RISC-V RV32I 기반 CPU Core 설계

오고은

목차

RISC-V 개요

- **ISA란?**

Instruction Set Architecture

CPU가 이해하고 실행할 수 있는 명령어들의 설계 규칙

프로그램이 실행되려면 컴파일러가 고급언어를 ISA에 맞는 명령어로 번역해야 한다.

-**ISA의 종류**

구분	**RISC (Reduced Instruction Set Computer)**	**CISC (Complex Instruction Set Computer)**
명령어 구조	단순한 명령어 다수 조합	한 명령어로 여러 작업 수행 가능
컴파일러 부담	높음 (여러 명령어로 나눠야 함)	낮음 (고급 언어 → 명령어 매핑 쉬움)
하드웨어 구조	단순함	복잡함
대표 예시	RISC-V, ARM, MIPS	x86, x86-64 (Intel, AMD)

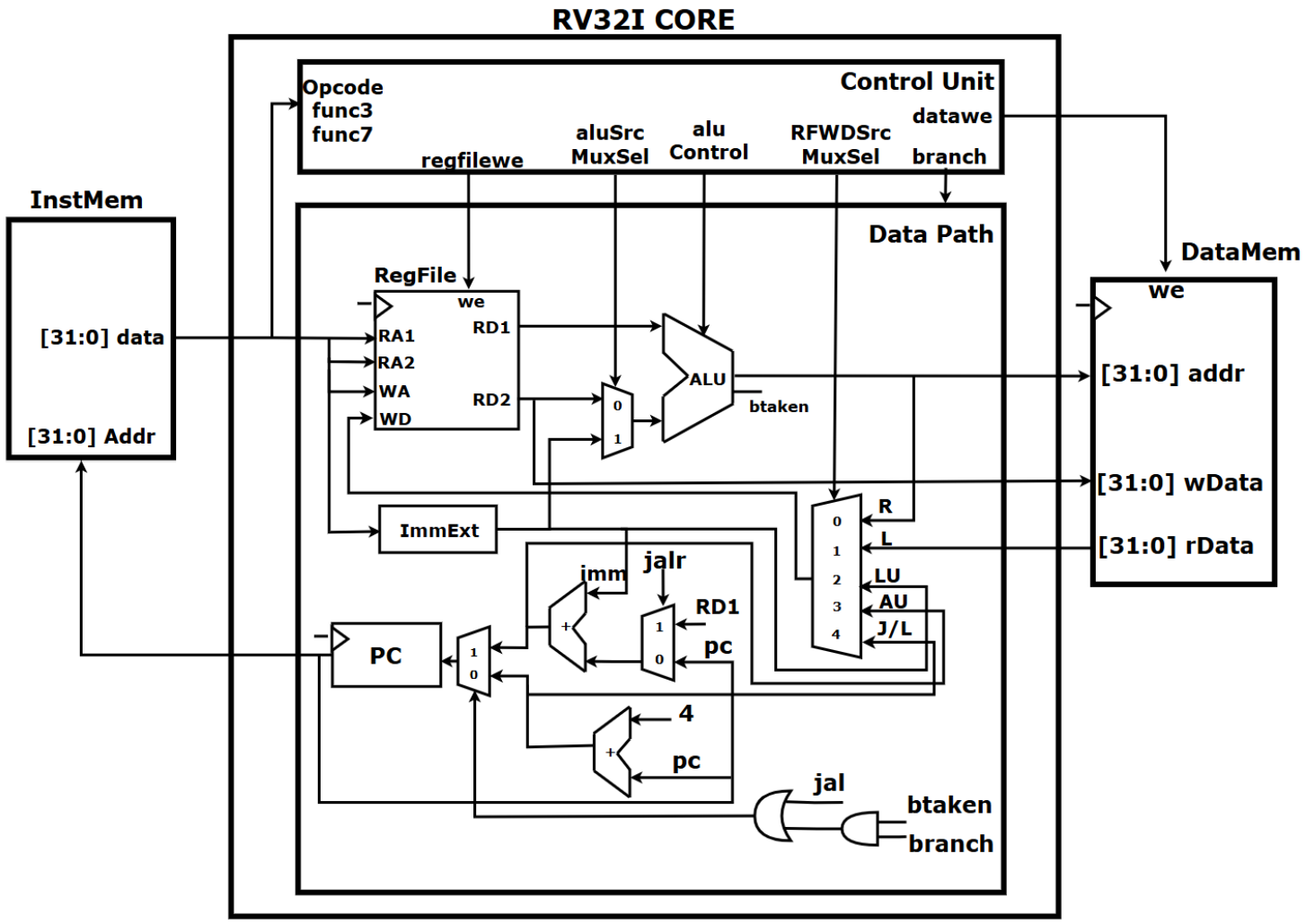
RISC-V는 RISC 기반의 오픈소스 ISA로 누구나 자유롭게 CPU를 설계하고 사용할 수 있도록 개발되었다.

구현환경

- 개발 언어
System Verilog

-Simulation Tool
Vivado

RISC-V CPU Core Block Diagram



RISC-V Type 정리

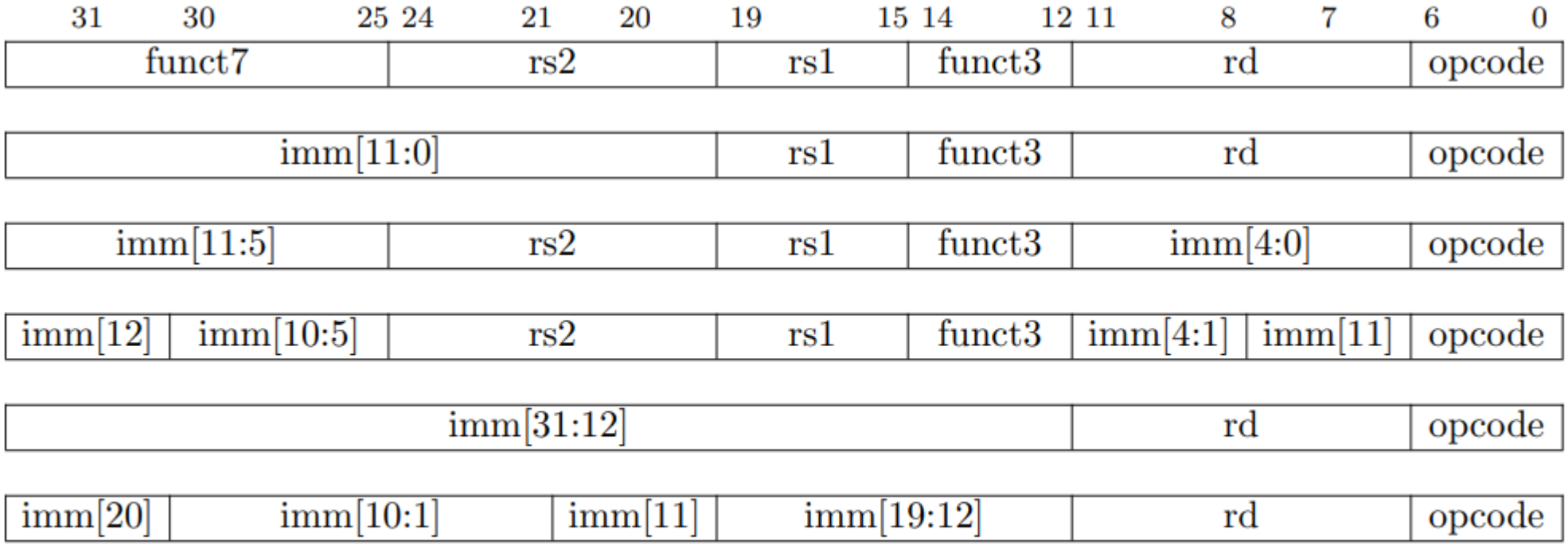


Figure 2.3: RISC-V base instruction formats showing immediate variants.

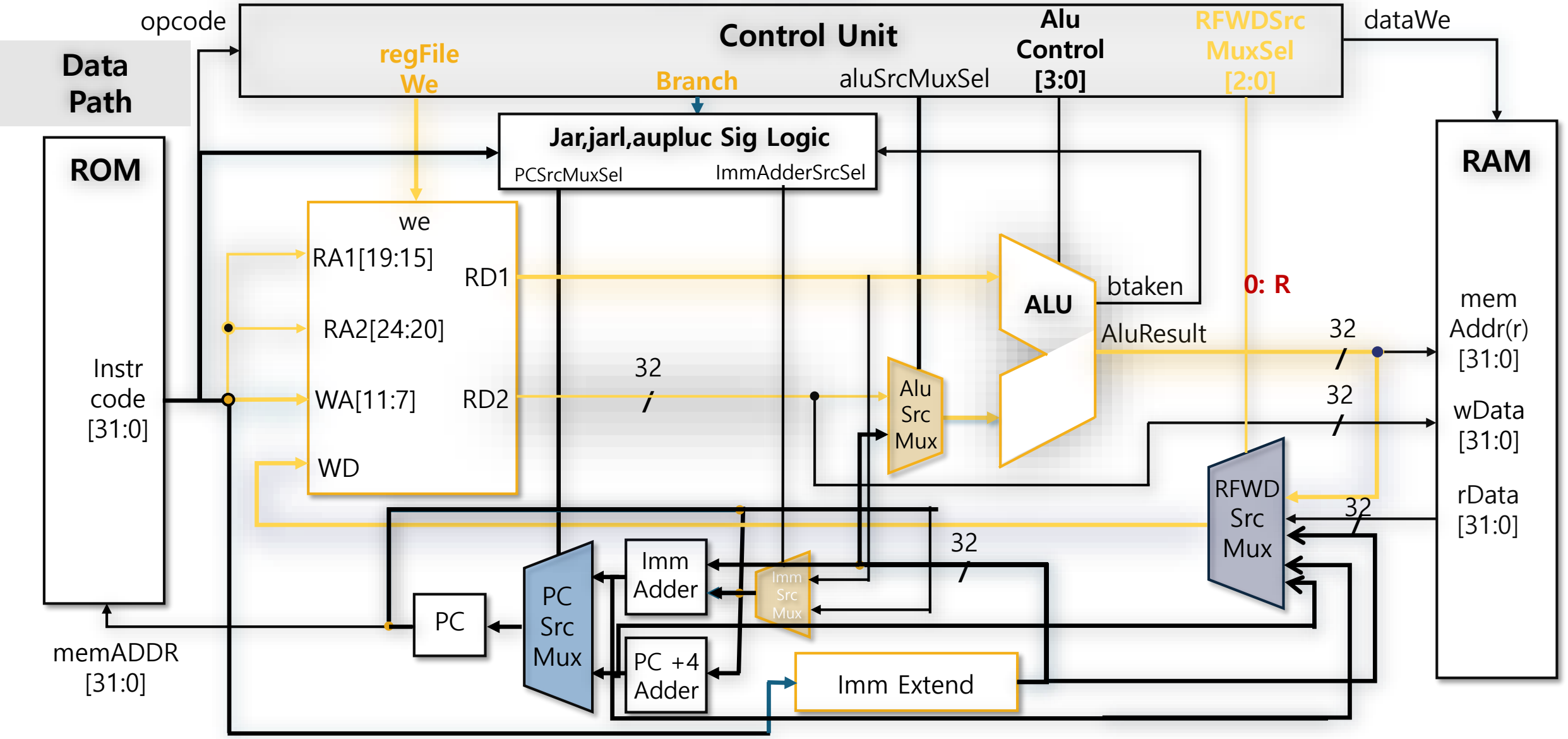
RISC-V Type 정리

R Type

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	TYPE	MNEMONIC	NAME	Descript	Note
Funct7 (7)							Regitster Source 2 (5)					Regitster Source 1 (5)					Funct3 (3)			Register Destination (5)					Opcode (7)												
0	0	0	0	0	0	0	rs2					rs1					0	0	0	rd					0	1	1	0	0	1	1	R-TYPE	ADD	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	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	Shift Left Logical	rd = rs1 << rs2	
0	0	0	0	0	0	0	rs2					rs1					1	0	1	rd					0	1	1	0	0	1	1		SRL	Shift Right Logical	rd = rs1 >> rs2	
0	1	0	0	0	0	0	rs2					rs1					1	0	1	rd					0	1	1	0	0	1	1		SRA	Shift Right Arith*	rd = rs1 >>> rs2	msb-extends
0	0	0	0	0	0	0	rs2					rs1					0	1	0	rd					0	1	1	0	0	1	1		SLT	Set Less Than	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	Set Less Than (U)	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	XOR	rd = rs1 ^ rs2	
0	0	0	0	0	0	0	rs2					rs1					1	1	0	rd					0	1	1	0	0	1	1		OR	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	AND	rd = rs1 & rs2	

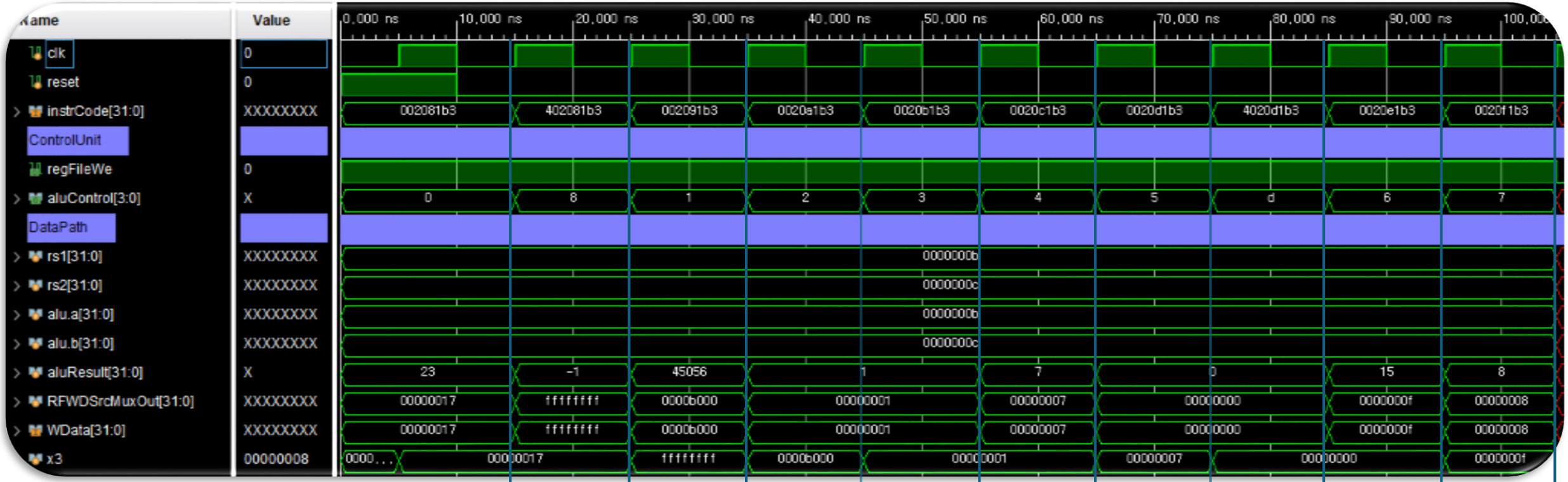
R-TYPE
OP Flow

ADD rd, rs1, rs2



Simulation

R-TYPE



add x3, x1, x2

Sll: x3, x1, x2

sltu x3, x1, x2

srl x3, x1, x2

or x3, x1, x2

Sub: x3, x1, x2

Slt: x3, x1, x2

xor x3, x1, x2

sra x3, x1, x2

and x3, x1, x2

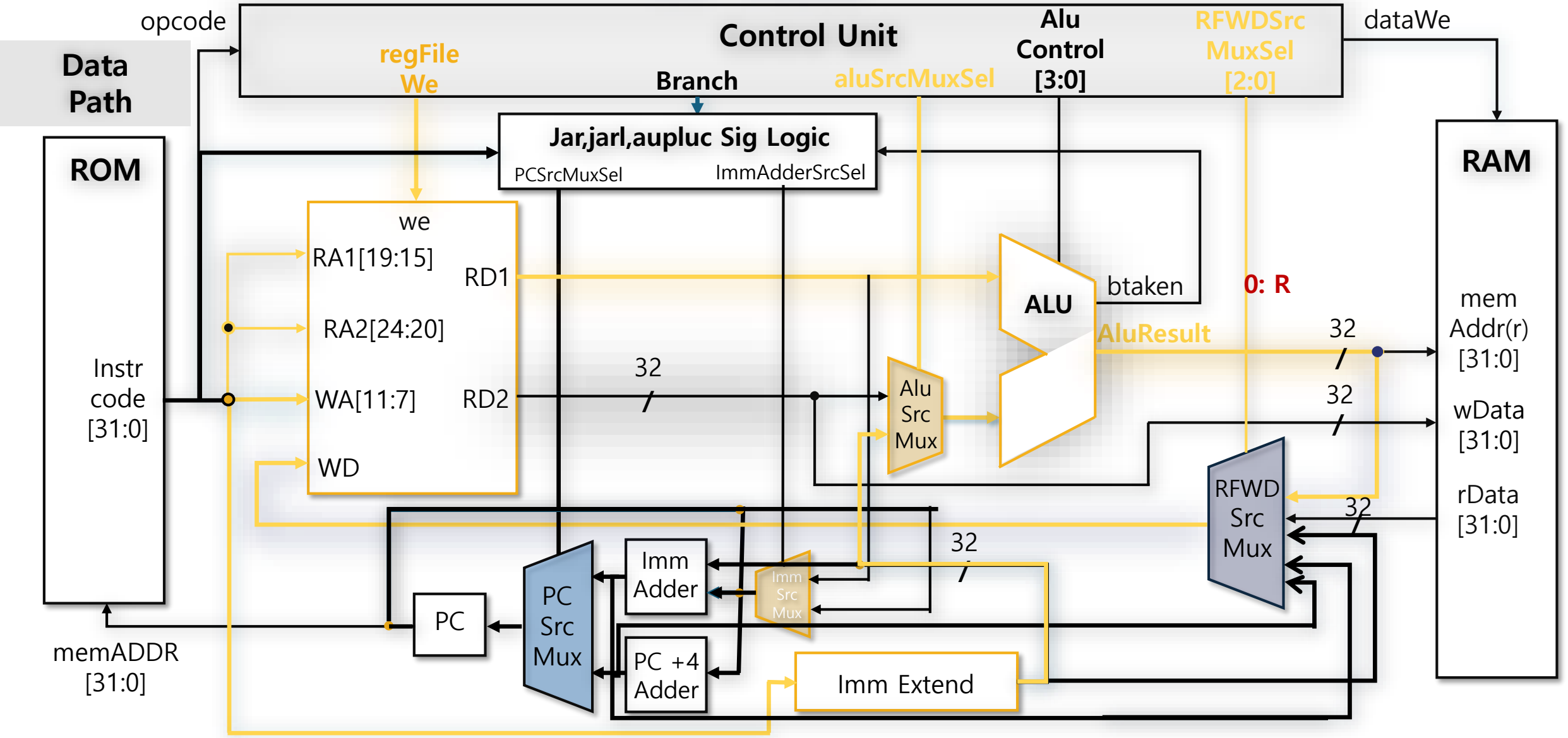
RISC-V Type 정리

I Type

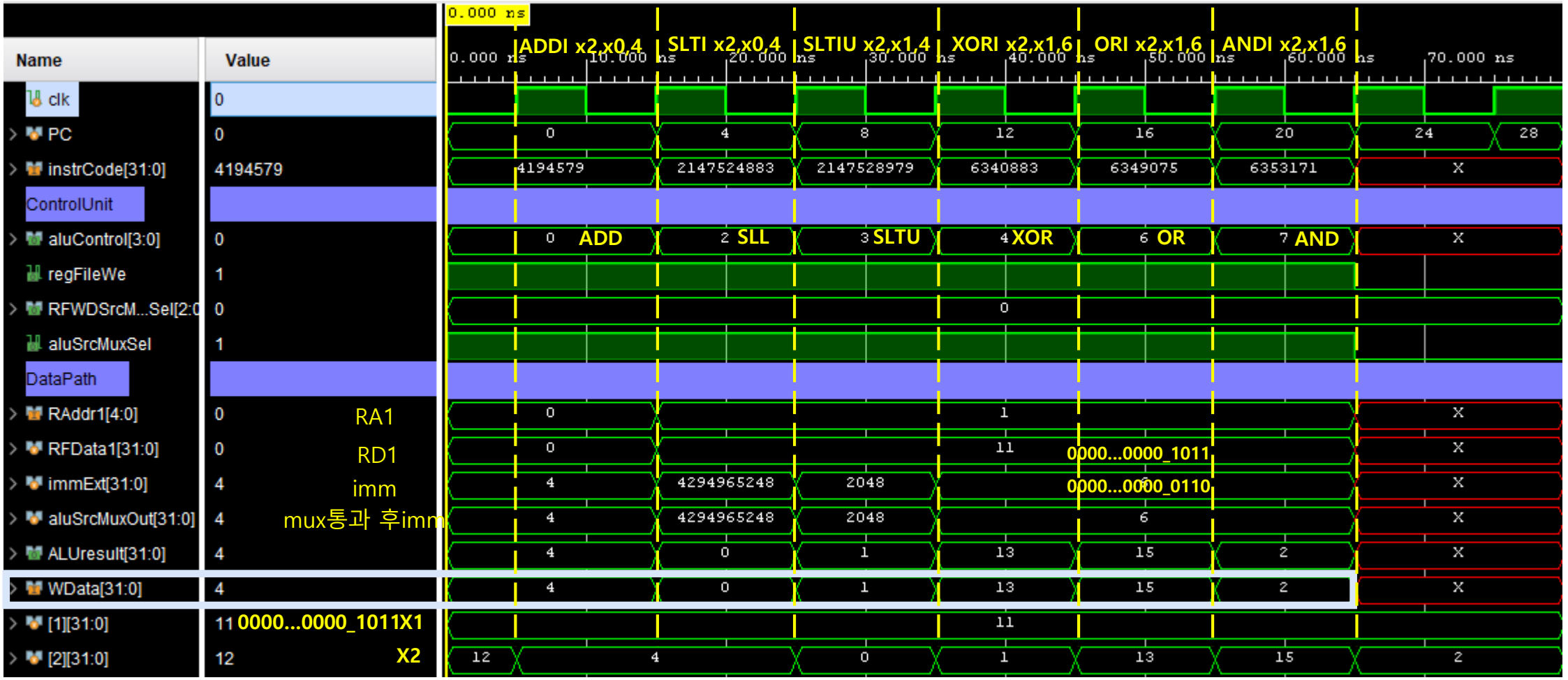
imm[11:0]								rs1	0	0	0	rd	0	0	1	0	0	1	1	I-TYPE	ADDI	ADD Immediate	rd = rs1 + imm	
imm[11:0]								rs1	0	1	0	rd	0	0	1	0	0	1	1		SLTI	Set Less Than Imm	rd = (rs1 < imm) ? 1 : 0	
imm[11:0]								rs1	0	1	1	rd	0	0	1	0	0	1	1		SLTIU	Set Less Than Imm (U)	rd = (rs1 < imm) ? 1 : 0	
imm[11:0]								rs1	1	0	0	rd	0	0	1	0	0	1	1		XORI	XOR Immediate	rd = rs1 ˆ imm	
imm[11:0]								rs1	1	1	0	rd	0	0	1	0	0	1	1		ORI	OR Immediate	rd = rs1 | imm	
imm[11:0]								rs1	1	1	1	rd	0	0	1	0	0	1	1		ANDI	AND Immediate	rd = rs1 & imm	
0	0	0	0	0	0	0	shamt	rs1	0	0	1	rd	0	0	1	0	0	1	1		SLLI	Shift Left Logical Imm	rd = rs1 << shamt[0:4]	
0	0	0	0	0	0	0	shamt	rs1	1	0	1	rd	0	0	1	0	0	1	1		SRLI	Shift Right Logical Imm	rd = rs1 >> shamt[0:4]	
0	1	0	0	0	0	0	shamt	rs1	1	0	1	rd	0	0	1	0	0	1	1		SRAI	Shift Right Arith Imm	rd = rs1 >>> shamt[0:4]	

I-TYPE
OP Flow

ADDI rd, rs1, imm



Simulation

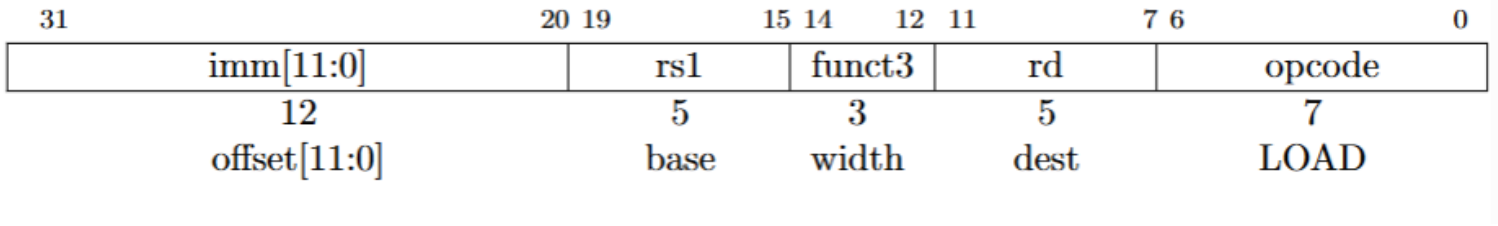


...0000_1101 ...0000_1111 ...0000_0010

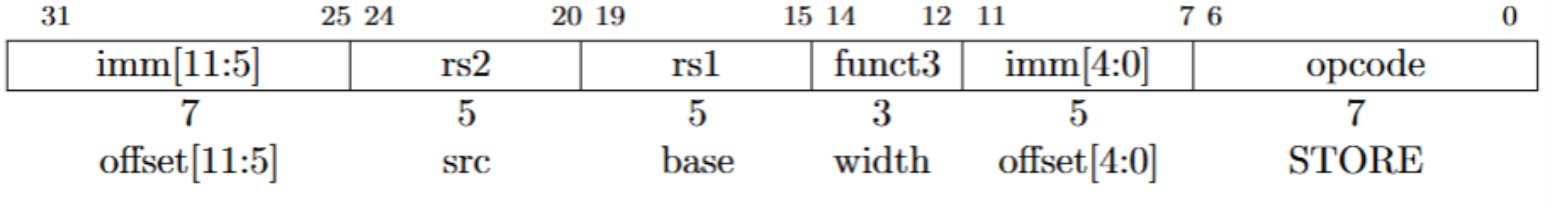
RISC-V Type

L, S Type

RV32I는 LW/SW 에서만 memory(RAM)에 access 가능하고, 이외 명령어는 CPU register에서만 동작한다.



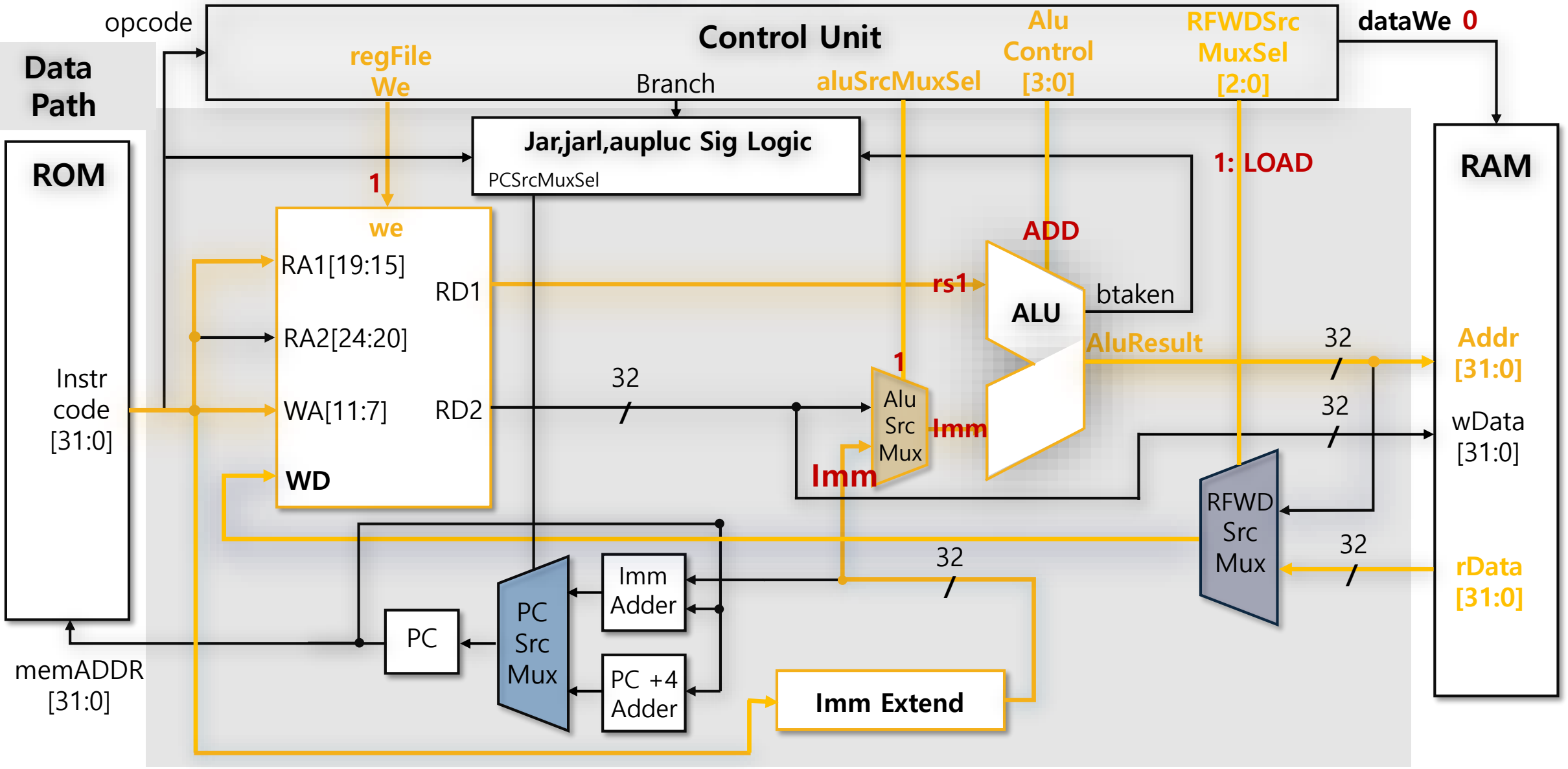
lw rd, offset(rs1) lw x3, 2(x0)



sw rs2, offset(rs1) sw x2, 8(x0)

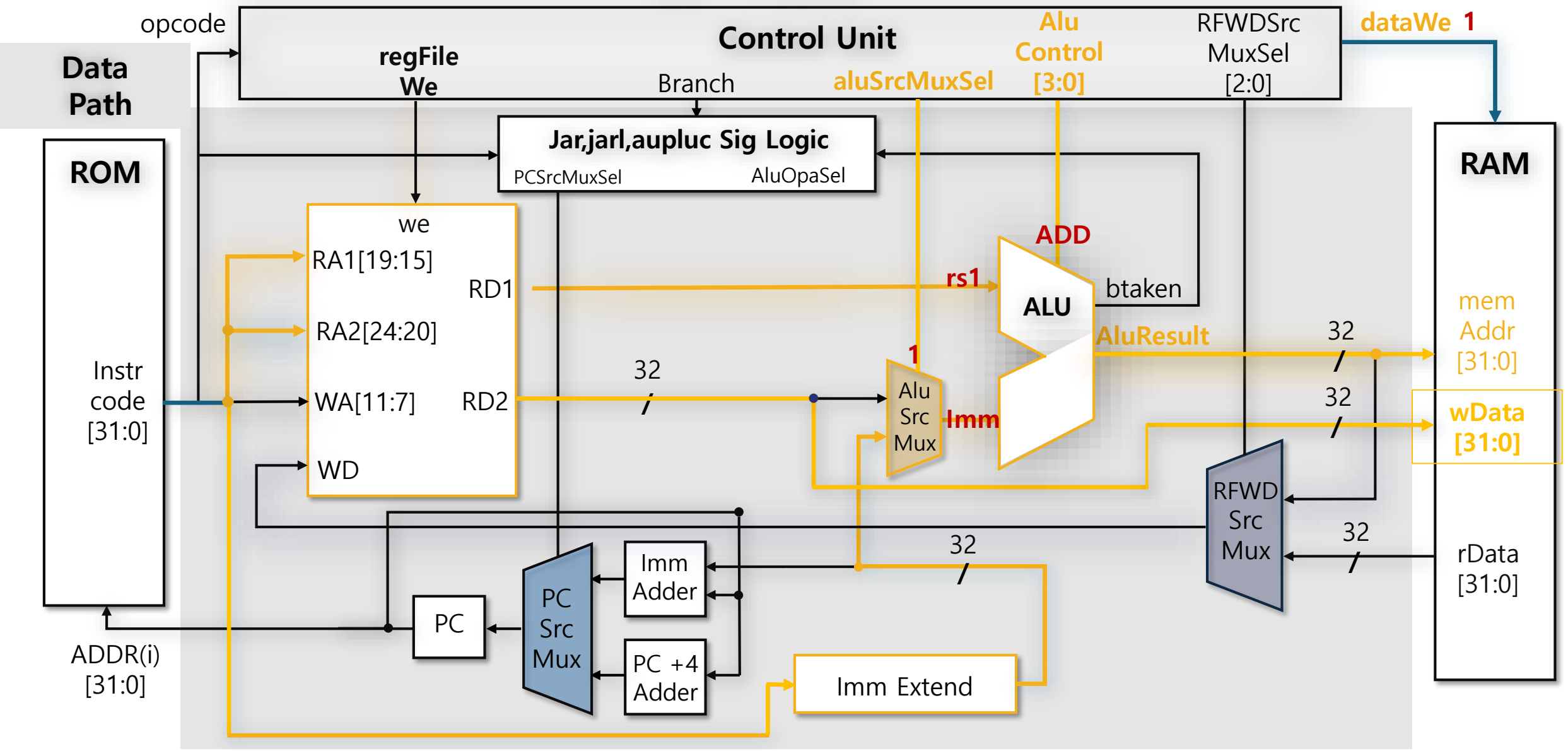
L-TYPE
OP Flow

LW rs2, rs1(imm)

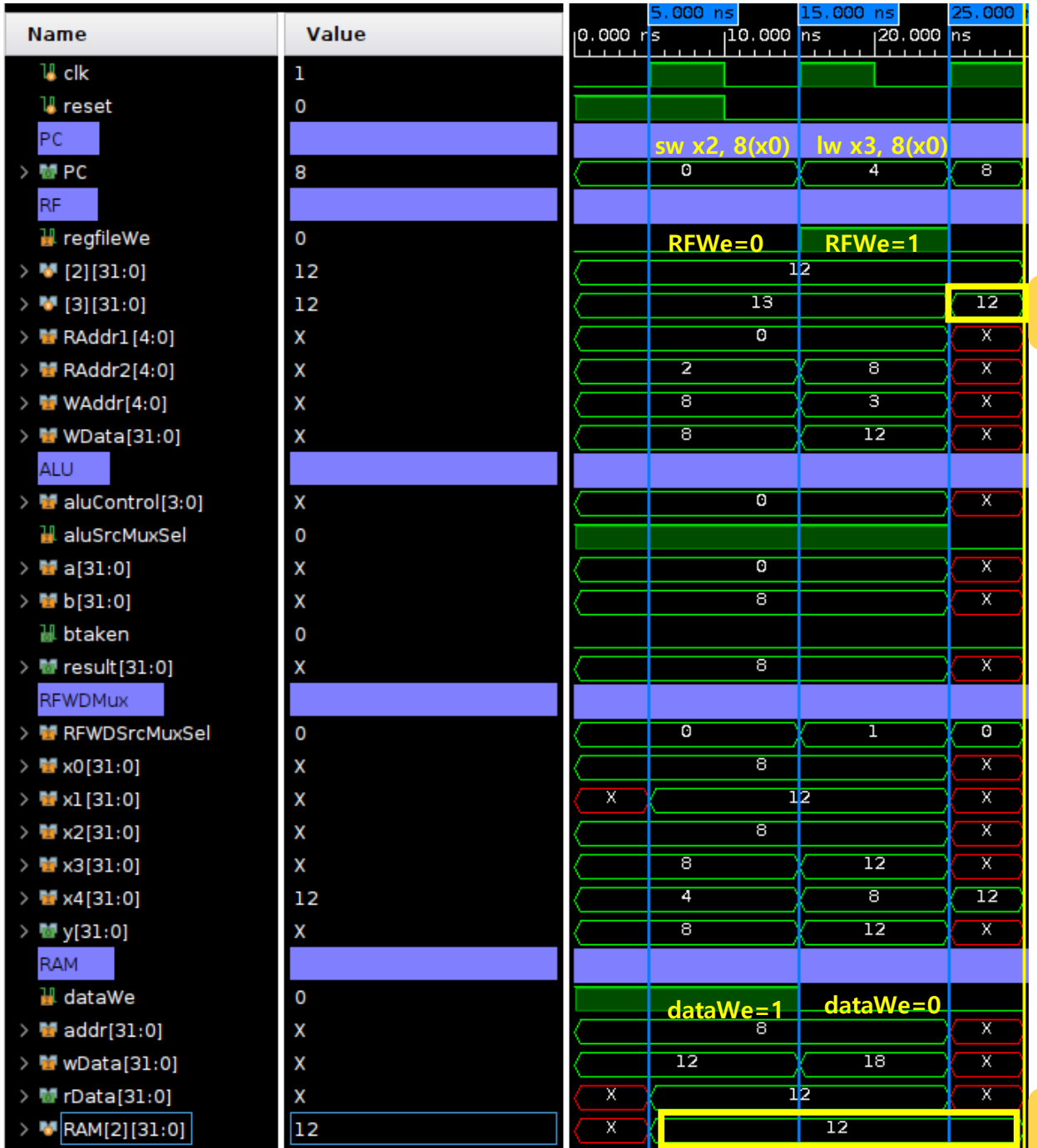


S-TYPE
OP Flow

SW rs2, rs1(imm)



Simulation



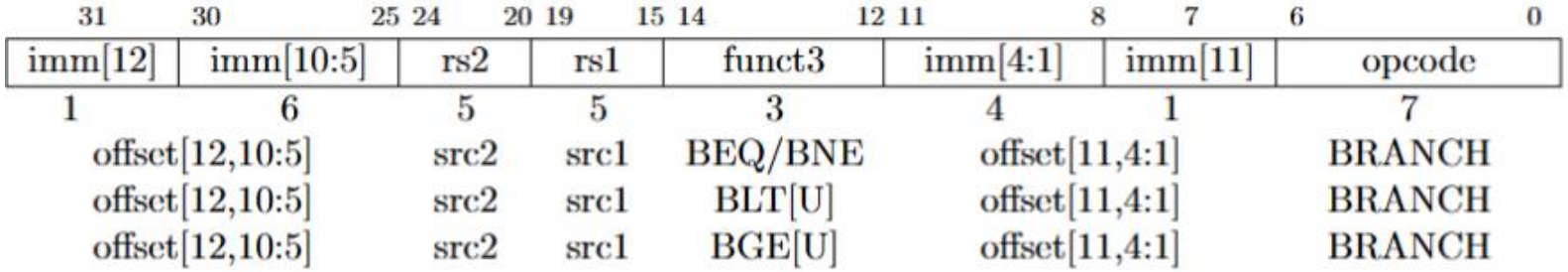
x3= RAM[2]

RAM[2] = x2

RISC-V Type

Control Transfer Instructions – B Type

1. Conditional Jumps (BXX)

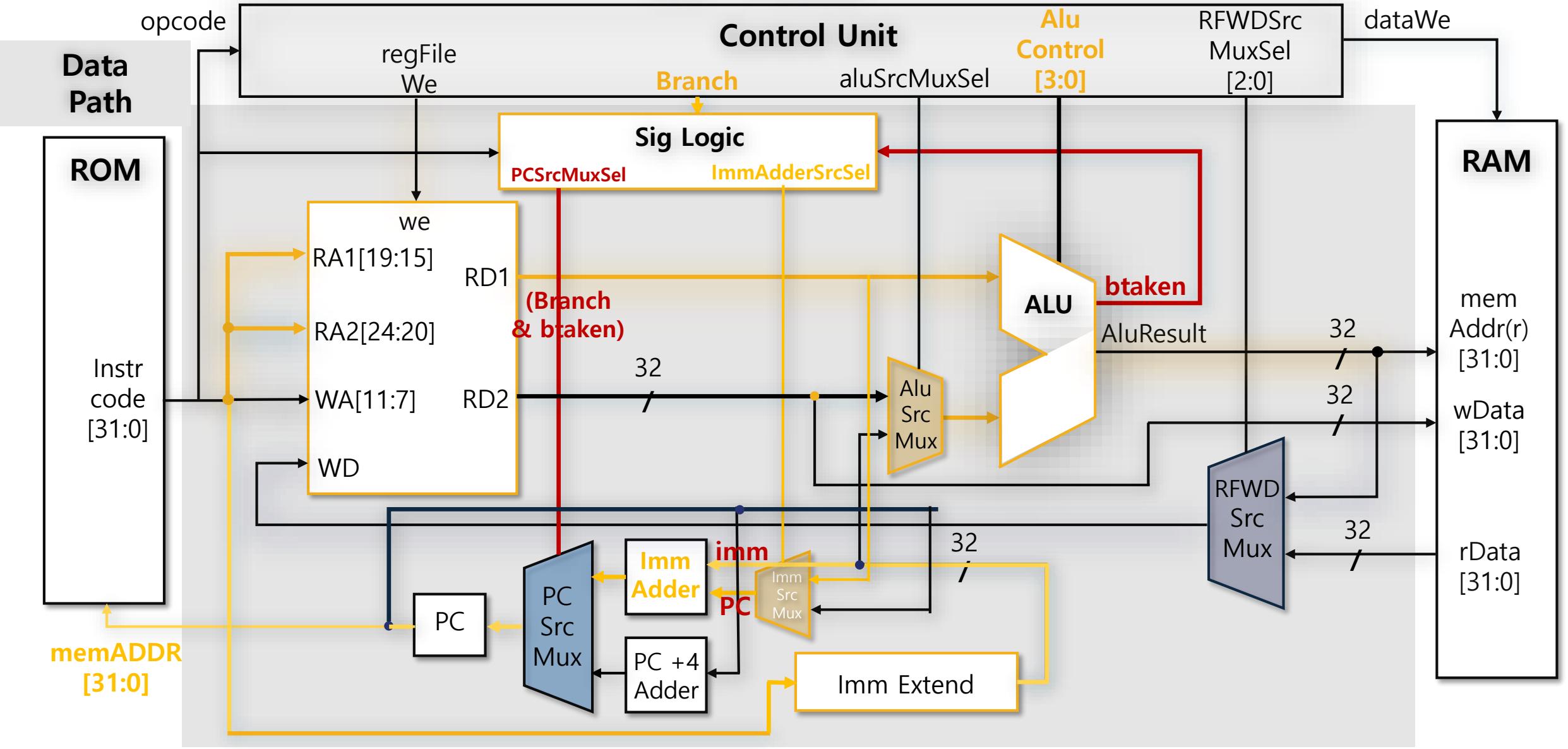


BEQ	rs1 == rs2
BNE	rs1 ≠ rs2
BLT	rs1 < rs2
BLTU	rs1 < rs2
BGE	rs1 ≥ rs2
BGEU	rs1 ≥ rs2

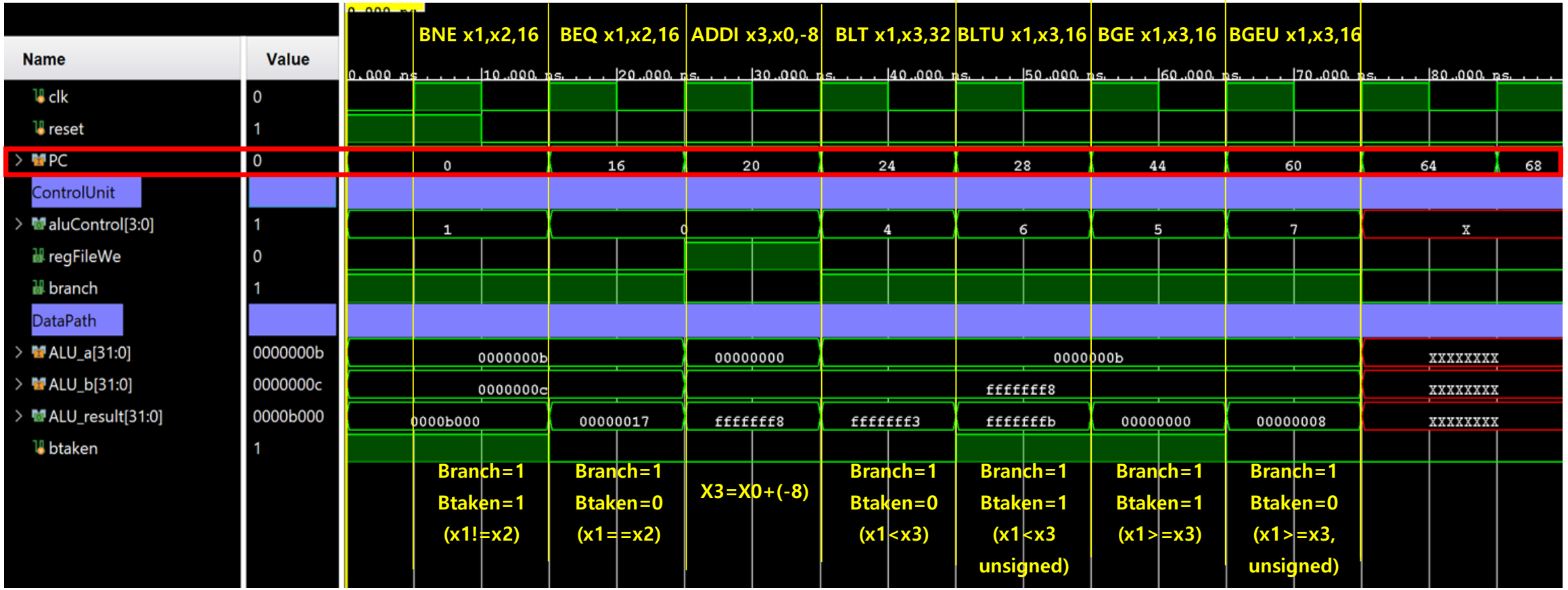
Pc+=imm

분기 주소 = PC + (un)sign-extended(offset << 1)

B-TYPE
OP Flow



Simulation

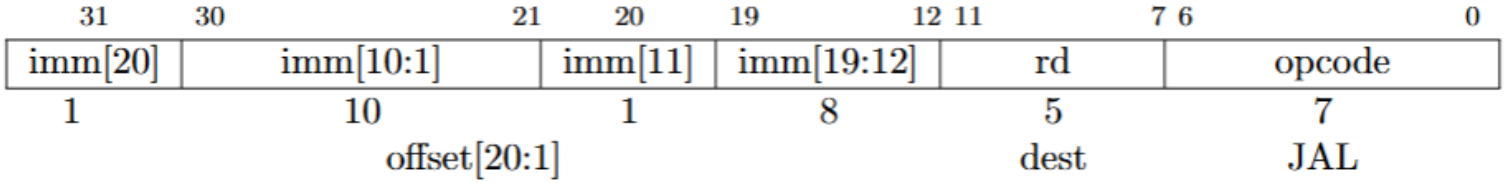

BNE x1,x2,16
BEQ x1,x2,16
ADDI x3,x0,-8
BLT x1,x3,32
BLTU x1,x3,16
BGE x1,x3,16
BGEU x1,x3,16
Name
Value
clk
reset
PC
ControlUnit
aluControl[3:0]
regFileWe
branch
DataPath
ALU_a[31:0]
ALU_b[31:0]
ALU_result[31:0]
btaken
0
1
0
1
1
0
1
0000000b
0000000c
0000b000
1
0
16
20
24
28
44
60
64
68
1
0
4
6
5
7
x
0000000b
00000000
0000000b
xxxxxxxx
0000000c
ffffffff8
xxxxxxxx
0000b000
00000017
ffffffff8
ffffffff3
ffffffffb
00000000
00000008
xxxxxxxx
Branch=1
Btaken=1
(x1!=x2)
Branch=1
Btaken=0
(x1==x2)
X3=X0+(-8)
Branch=1
Btaken=0
(x1<x3)
Branch=1
Btaken=1
(x1<x3
unsigned)
Branch=1
Btaken=1
(x1>=x3)
Branch=1
Btaken=0
(x1>=x3,
unsigned)

RISC-V Type 정리

Control Transfer Instructions – J Type

2. Unconditional Jumps (JAL, JALR)

JAL



rd= PC+4, PC+=imm

Immediate는 sign-extend 된 뒤, PC에 더해져서 점프할 대상 주소가 된다.

표준 RISC-V 소프트웨어 호출 규칙에서는 **x1** 레지스터를 return 주소용(ra)으로 사용하고
x5는 보조 링크 레지스터로 사용한다.

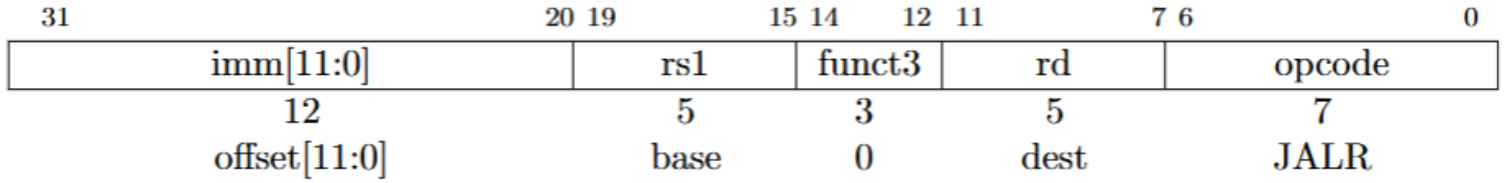
```
jal x1, function1   // function1 호출, 복귀 주소는 x1
jal x5, function2   // function2 호출, 기존 x1 값 보존
```

RISC-V Type 정리

Control Transfer Instructions

2. Unconditional Jumps (JAL, JALR)

JALR



Rd=PC+4, pc=rs1+imm

JALR 명령어는 간접 점프 명령어이고, I-type 형식을 사용해 인코딩됨
(간접 점프는 주소가 즉시값이 아니라 레지스터 + offset 형태인 경우)

복귀 주소(PC+4)를 rd 레지스터에 저장한다.

만약 복귀 주소를 저장할 필요가 없다면, rd를 **x0**(제로 레지스터)로 지정한다.

RISC-V Type 정리

U Type

Lui
rd ← imm << 12
큰 상수값을 로딩할 때 사용

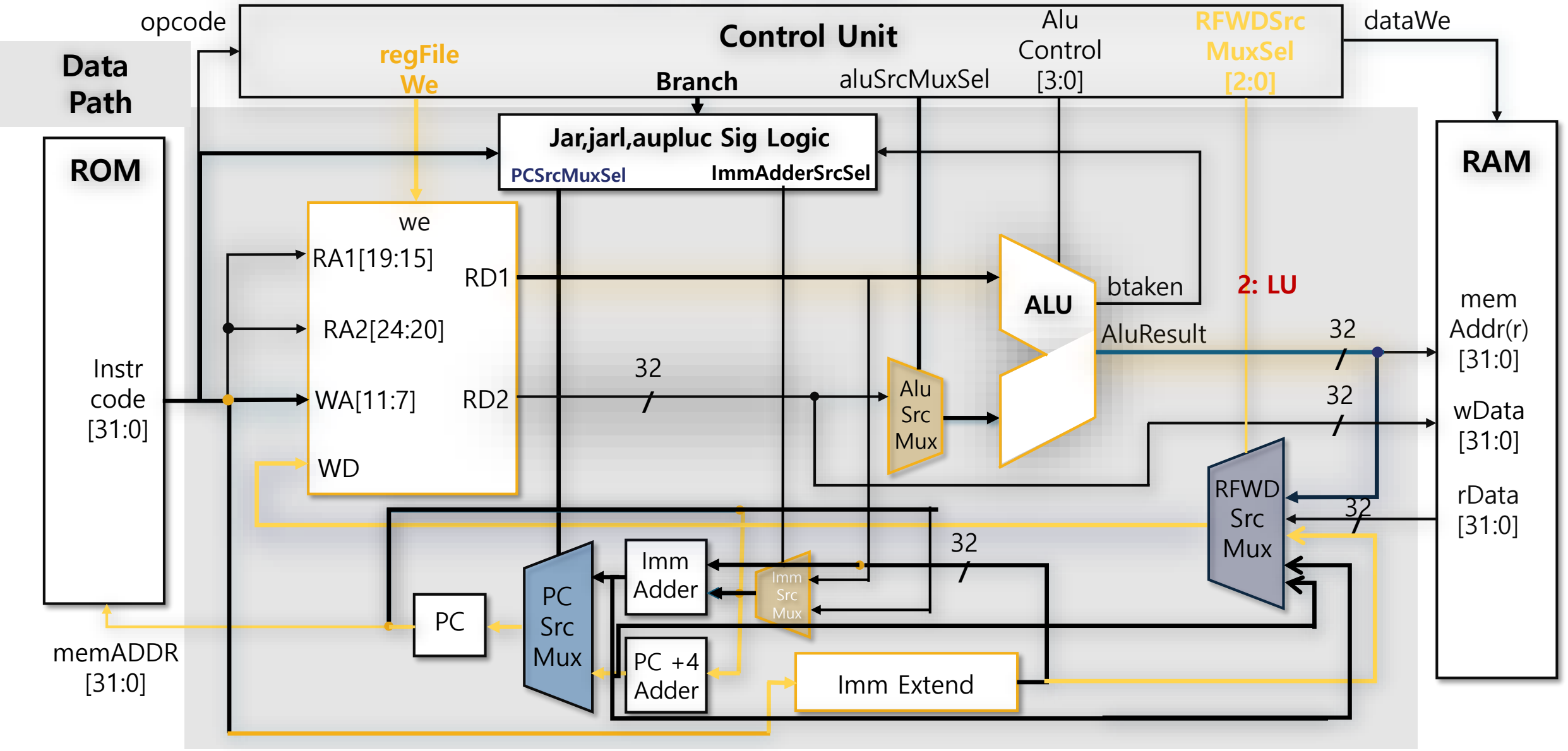
AUIPC (Add Upper Immediate to PC)

rd ← pc + (imm << 12)
현재 명령어 주소(PC)에 상위 20비트를 더한 값을 rd에 저장

U-TYPE

lui OP Flow

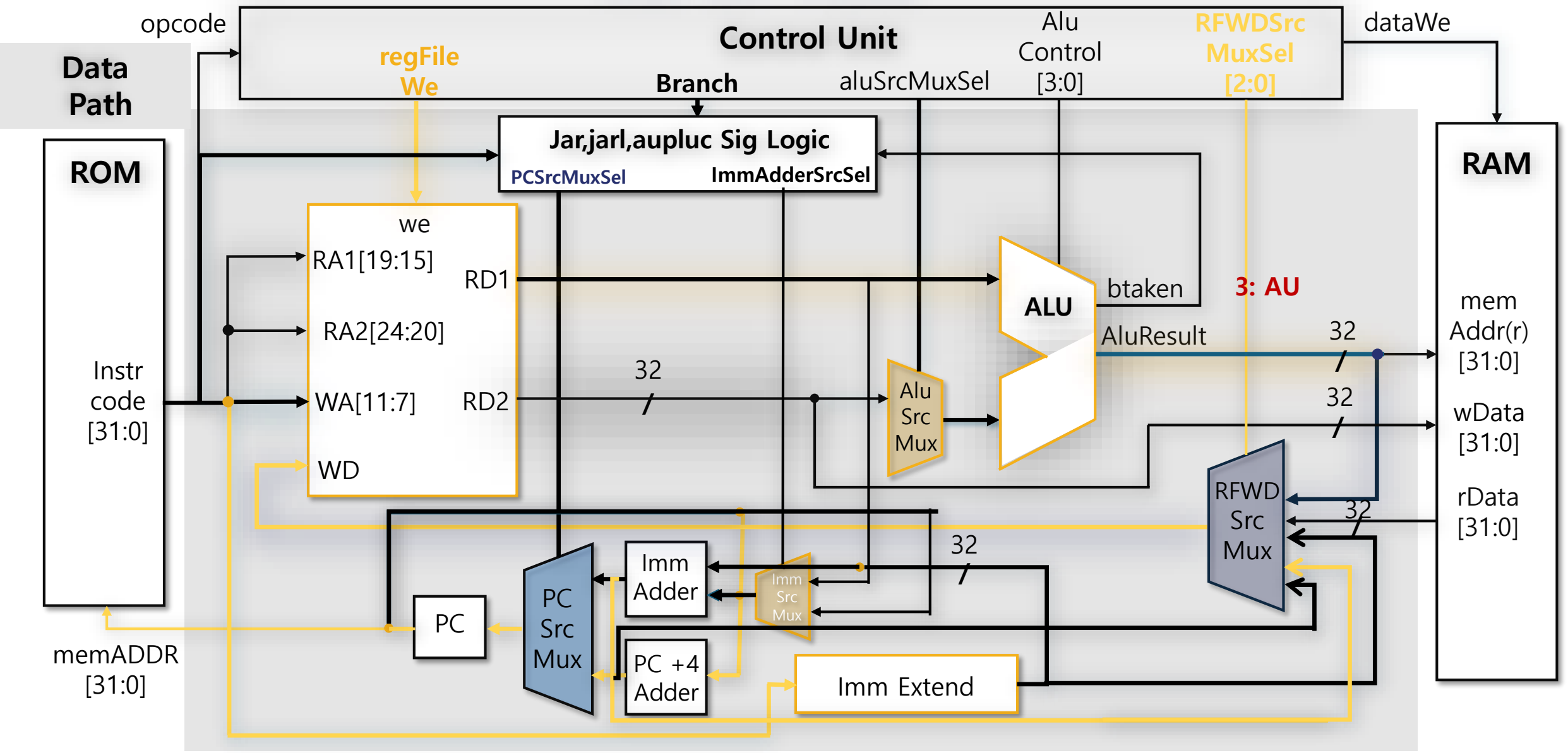
Lui rd, imm/ rd= imm <<12



U-TYPE

Aui OP Flow

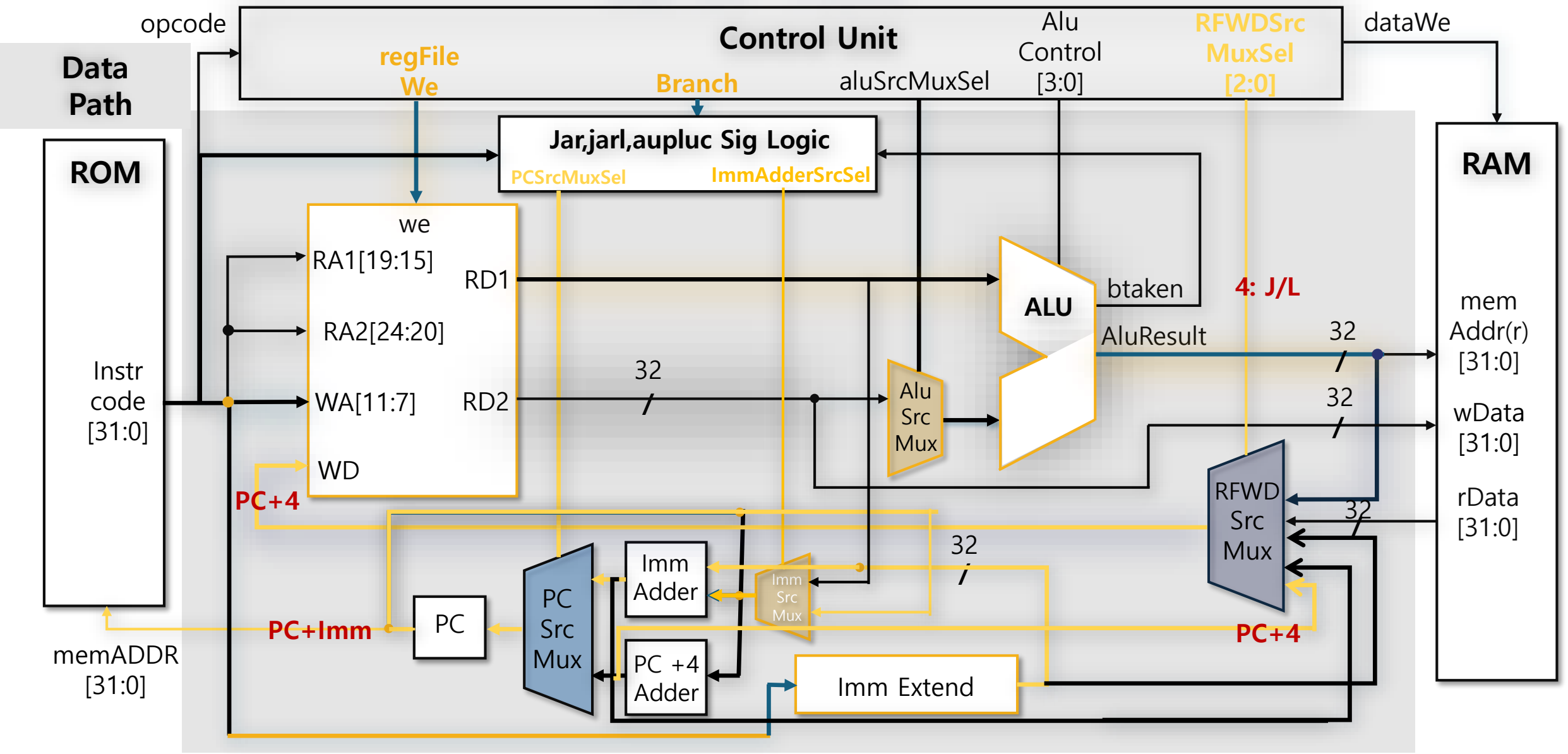
Aui rd, imm/ rd= PC+imm <<12



J-TYPE

Jal OP Flow

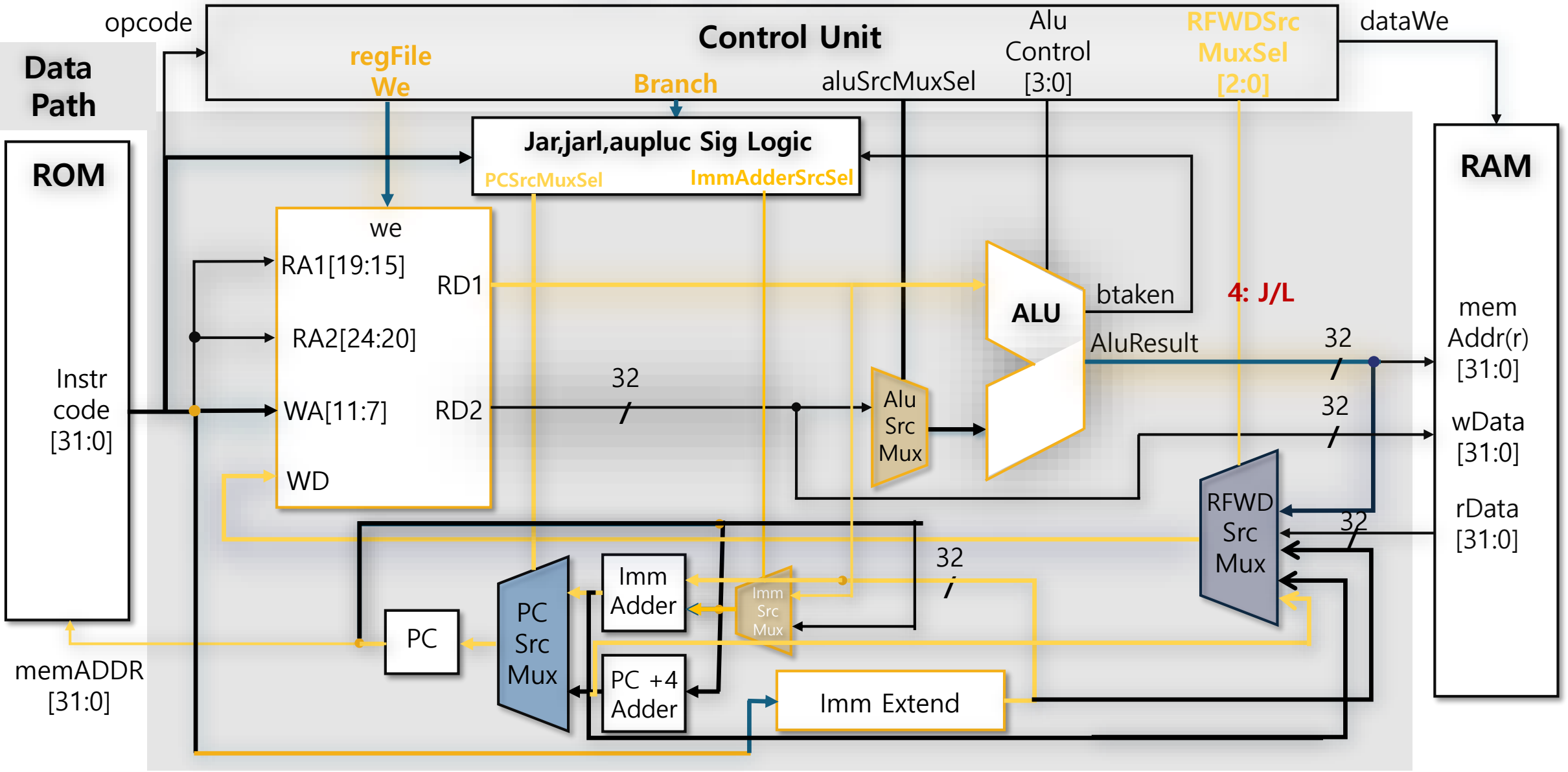
Jal rd, imm/ rd=PC+4, PC+=imm



J-TYPE

Jalr OP Flow

Jalr rd, rs1, imm/ rd=PC+4, PC=rs1+imm



Simulation


Name
Value
clk
reset
PC
PC
RF
we
RF[1][31:0]
RF[4][31:0]
RAddr1[4:0]
RAddr2[4:0]
WAddr[4:0]
WData[31:0]
RFWDMux
RFWDSrcMuxSel
x0[31:0]
x1[31:0]
x2[31:0]
x3[31:0]
x4[31:0]
PCSrcMux
sel
jal
btaken
branch
PCSrc0
PCSrc1
nextPC
PCImmAdderSrcMux
jalr
RD1
PC
o_PCImmAdderSrcMux
PCImmAdder
Imm
o_PCImmAdderSrcMux
PCSrc1
5.000 ns
15.000 ns
25.000 ns
35.000 ns
45.000 ns
lui x1, 0x1
au x1, 0x1
jal x1, 16
jalr x1, 16(x4)
LU
AU
J/L

느낀점

RV32I 명령어 set을 회로로 구현하는 과정을 통해, 명령어와 하드웨어 동작의 관계를 구조적으로 이해하게 됨

기능 단위로 분리된 설계를 구현하면서 연산 흐름뿐 아니라 제어 흐름의 중요성을 실감함