RISC-V RV32I 기반 CPU 설계

채민석

CONTENTS

1. 개 요

● ISA (Instruction Set Architecture)

- CPU가 인식하고 처리하는 명령어들의 집합 및 시스템 상태 변화를 정의
- 소프트웨어와 하드웨어 간의 표준화된 인터페이스를 제공하는 핵심 규격

● CISC (Complex Instruction Set Computer)

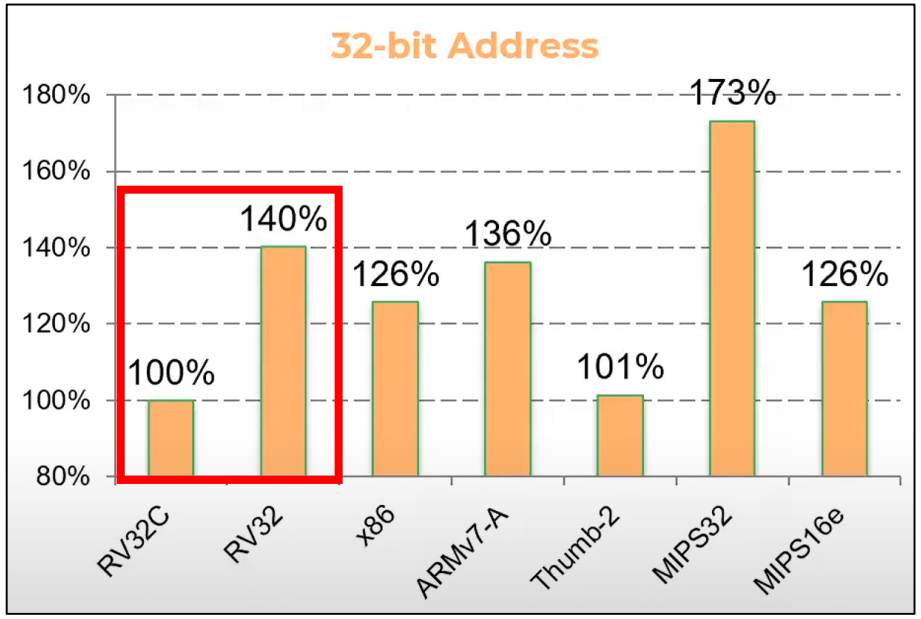
특 징 : Micro Processor에 대한 다양한 명령어 Set 포함
장 점 : 기능이 많아 하위 호환성이 좋음
단 점 : 전력소모↑, 가격↑, 속도↓
활 용 : 범용 컴퓨터의 CPU (PC, 서버)
예 시 : X86계열 (Intel, AMD)

● RISC (Reduced Instruction Set Computer)

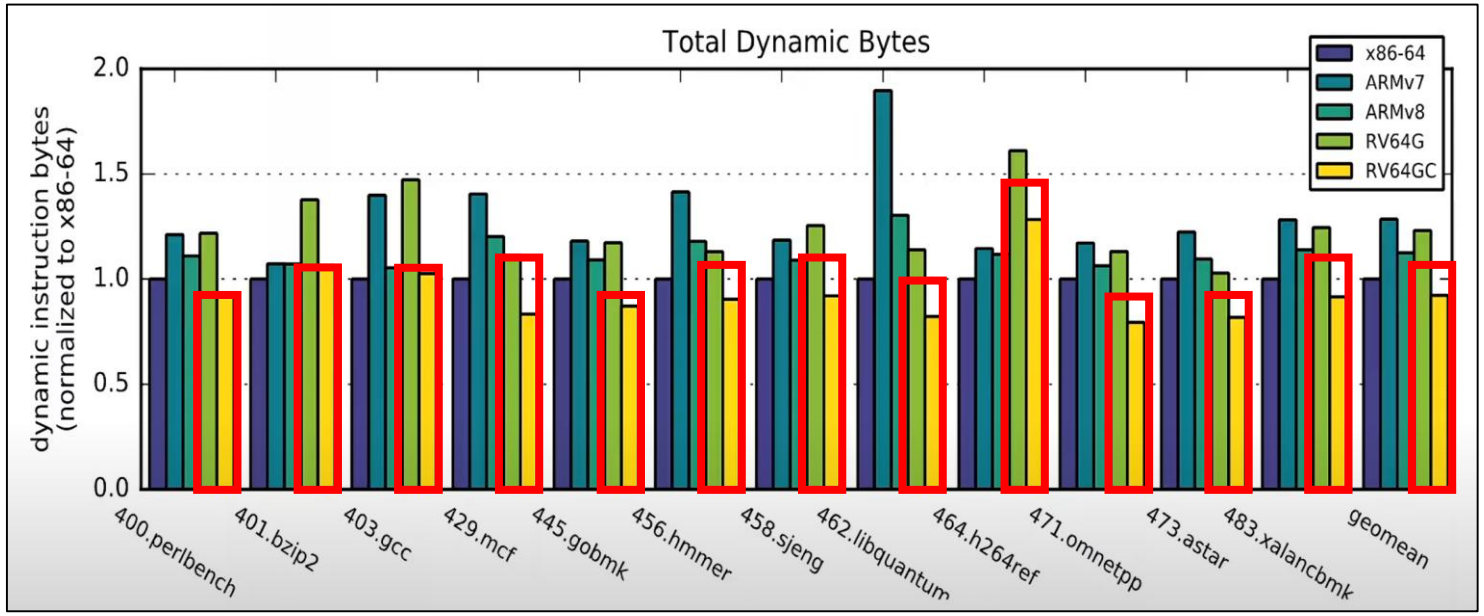
특 징 : 사용빈도가 높은 명령어로만 구성 (CISC 대비 약 20%)
장 점 : CISC보다 속도↑ 전력소모↓ 가격 ↓
단 점 : 하드웨어가 간단, 소프트웨어가 복잡, 하위 호환성 부족
활 용 : 발열과 전력 소모가 적어 임베디드 프로세서에서 사용
예 시 : ARM, MIPS, RISC-V 등

1. 개 요

● RISC-V 선정 이유



<Figure 1> RV32C 기준 ISA코드 크기 비교



<Figure 2> x86-64 기준 프로그램 동작 효율비교

[사진 출처] : 중소기업기술정보진흥원

2. 목표 및 개발 환경

● 프로젝트 목표

- RSIC-V의 RV32I에 대한 기본적인 CPU 구조를 설계한다
- 전반적인 CPU Data 흐름과 특징을 이해한다

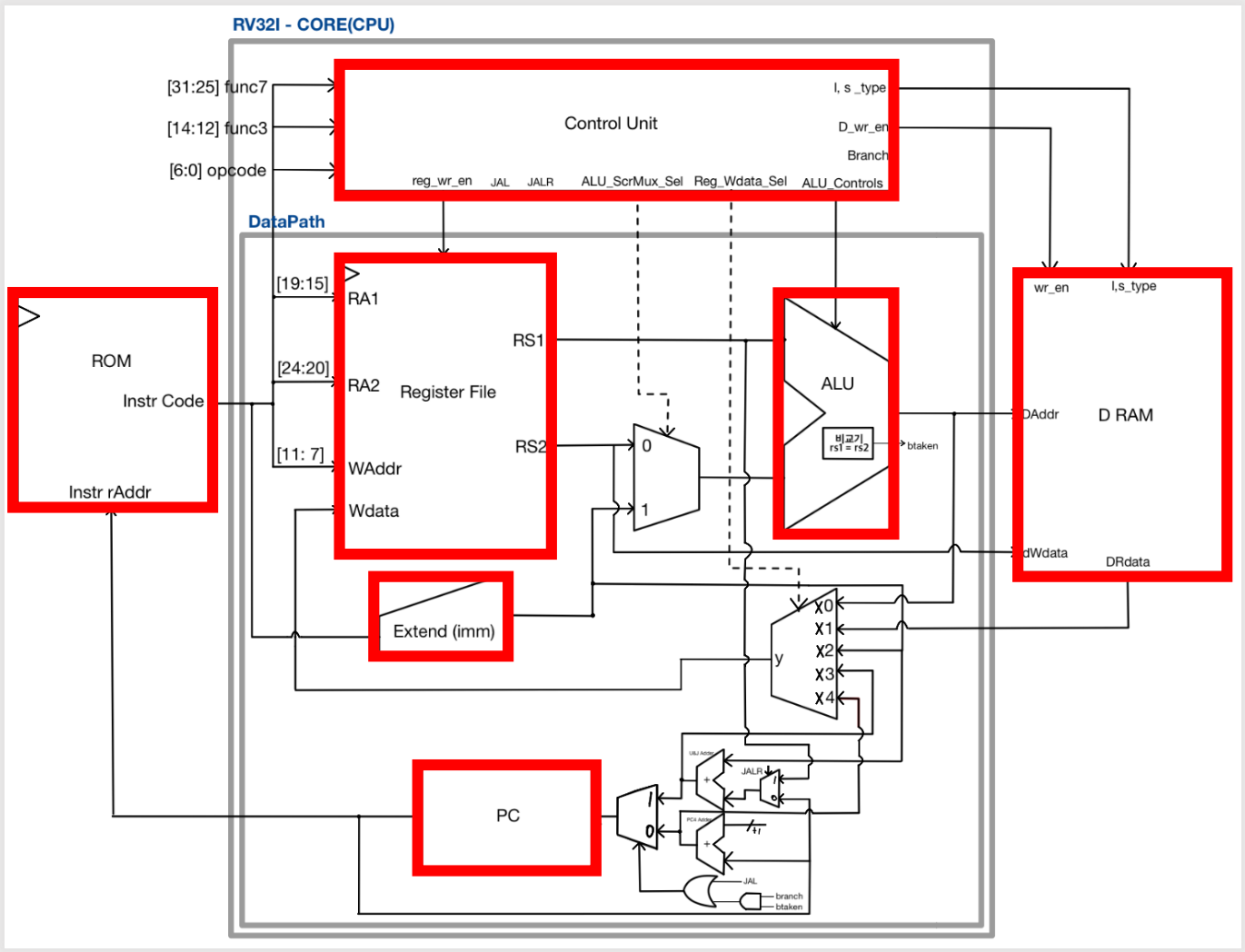
● 개발 환경

SystemVerilog

Vivado

RISC-V

3. RV32I



● RV32I 구조

Instruction Memory (ROM)

- 실행할 명령어 입력

Register File

- 레지스터 값 저장 및 읽기 (rs1, rs2 -> rd)

Arithmetic Logic Unit (ALU)

- Control Unit에서 나온 sel에 따른 연산 수행

Program Counter

- 명령어 주소 Counter (Word align, +1)

Extend

- 즉시값 추출 및 unsign, msb, 0 확장

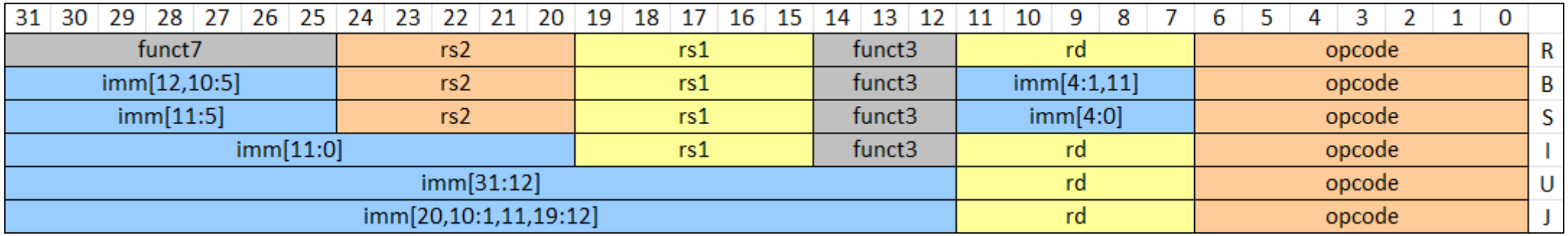
Control Unit

opcode, function 값을 통해 실행할 mux와 register, ALU 제어

Data Memory (RAM)

load, store 용 데이터 메모리 저장

3. RV32I



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							R
imm[12,10:5]							rs2					rs1					funct3			imm[4:1,11]					opcode							B
imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode							S
imm[11:0]												rs1					funct3			rd					opcode							I
imm[31:12]																				rd					opcode							U
imm[20,10:1,11,19:12]																				rd					opcode							J

● RV32I Type 종류

R type "Register"

주로 산술 및 논리 연산

S type "Store"

메모리에 값을 저장 연산

I type "Immediate"

연산에 상수 값 사용

B type "Branch"

레지스터 값 비교 후 특정 조건 만족 시 PC 분기

U type "Upper Immdeiate"

상수 값 로드 후 목적 레지스터에 저장

J type "Jump"

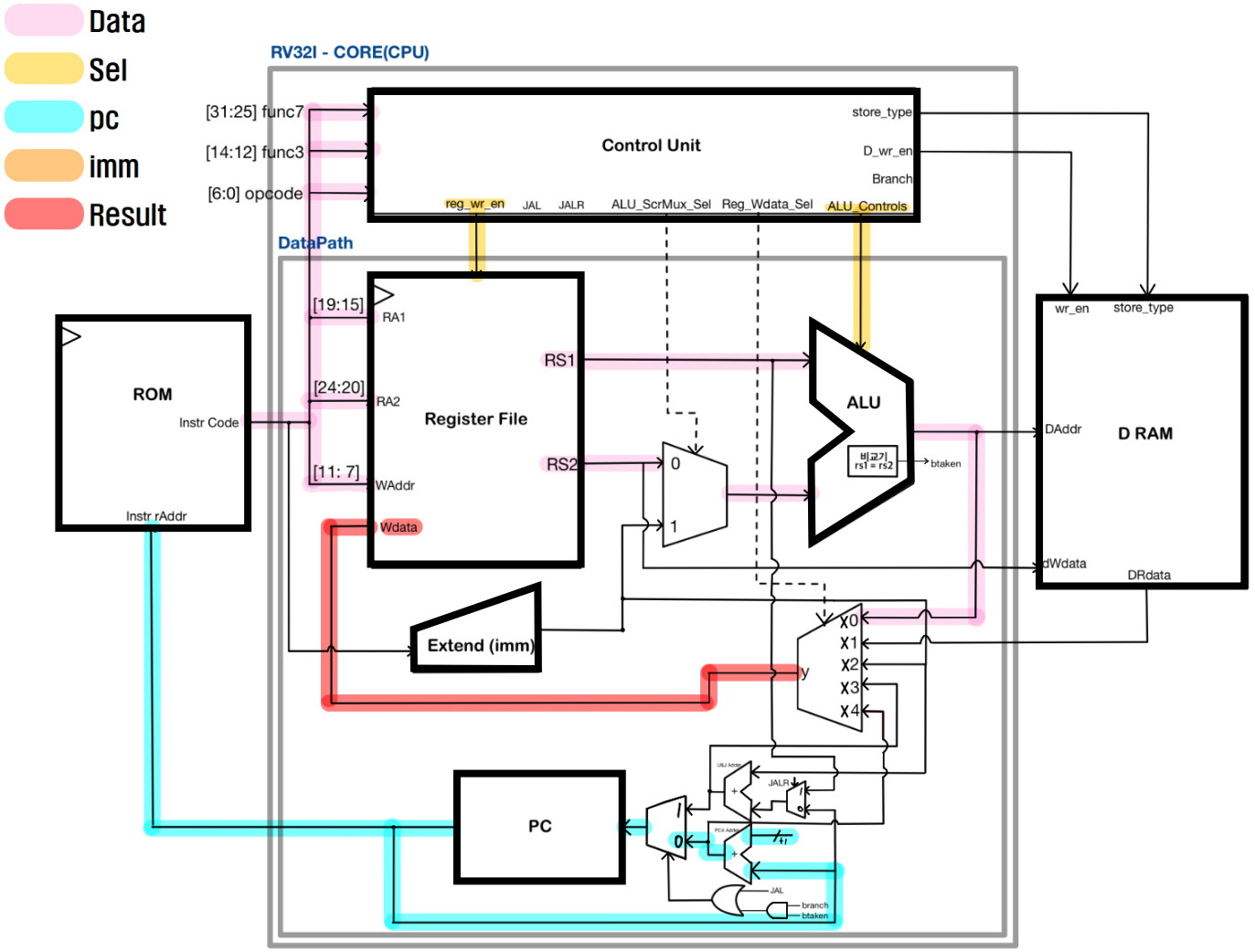
PC를 변경하여 흐름을 유연하게 제어

● Field

- **opcode** (7bits) : 명령어 Type 분류
- **funt3** (3bits) : 연산자 옵션 / 서브 코드 선택
- **funt7** (7bits) : funt3 + opcode가 같을 때 연산 세분화 코드
- **rs1 rs2** (5bits) : 소스 레지스터, 피연산자 or 베이스 주소
- **rd** (5bits) : 결과 저장 레지스터
- **Imm** : 즉시 상수 값 (offset)

4. 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	
funct7							rs2					rs1					funct3			rd					opcode							R



● R-Type

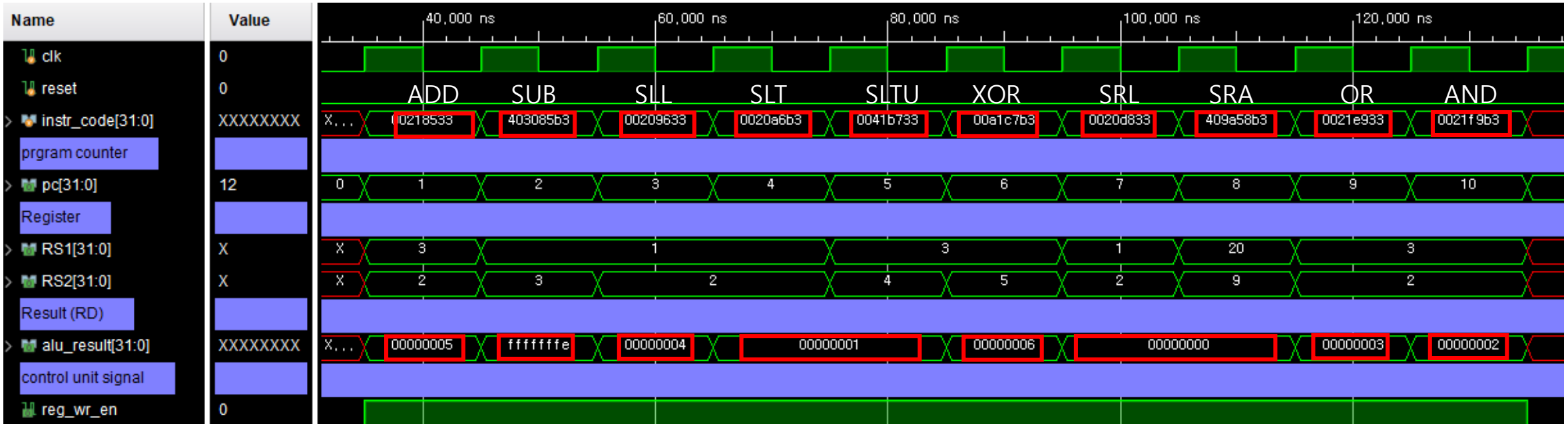
"Register"

모든 연산이 레지스터에서 발생

주로 산술 및 논리 연산 사용

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

4. R-Type Simulation

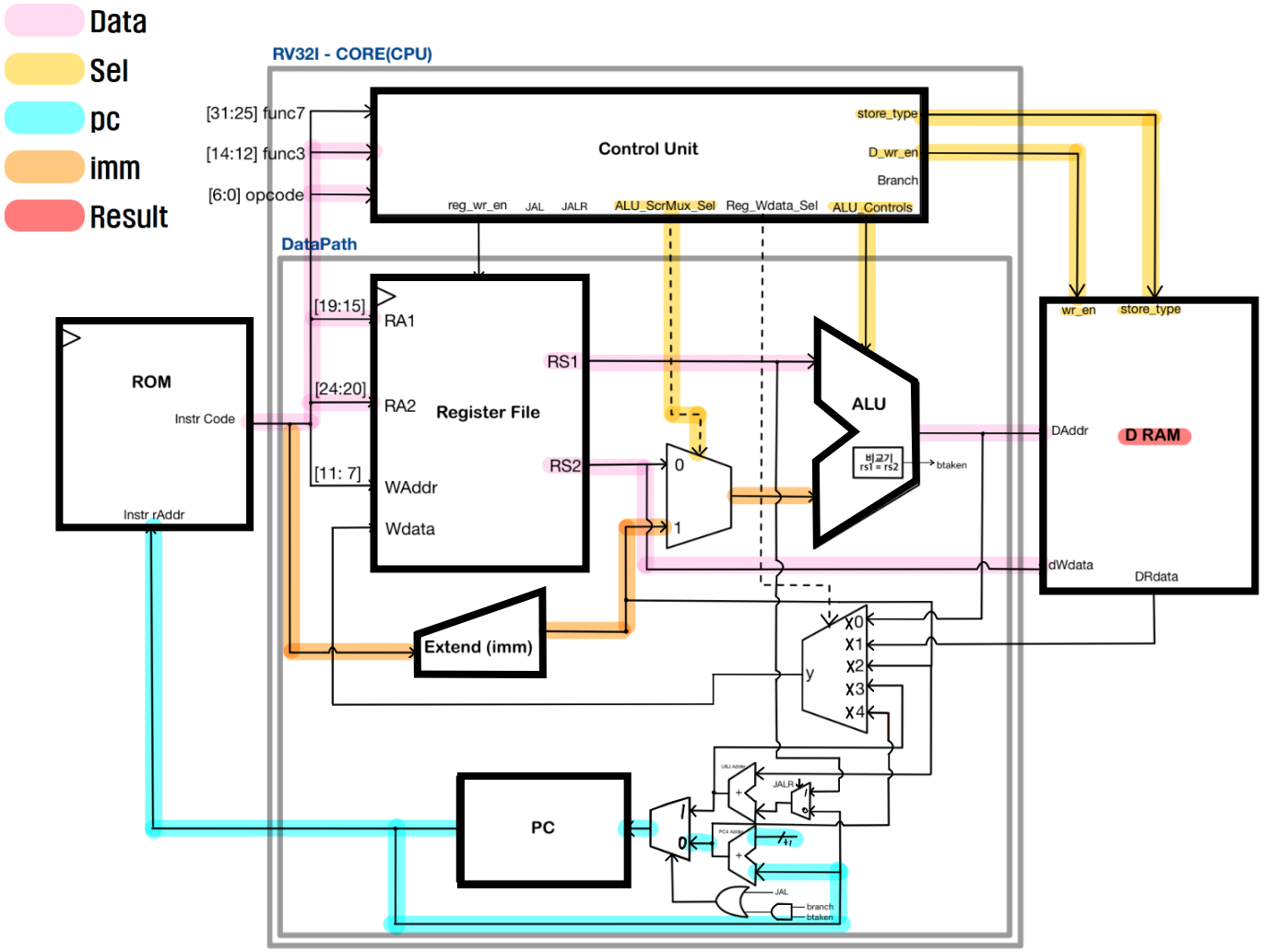


mnemoic	Semantics	Destination	result
ADD	rd = rs1 + rs2	x10, x2 x3	x0000_0005
SUB	rd = rs1 - rs2	x11 x3 x1	xFFFF_FFFE
SLL	rd = rs1 << rs2	x12 x2 x1	x0000_0004
SLT	rd ($signed(a)<$signed(b))?1:0	x13 x2 x1	x0000_0001
SLTU	rd = (rs1 < rs2)?1:0	x14 x4 x3	x0000_0001

mnemoic	Semantics	Destination	result
XOR	rd = rs1 ^ rs2	x15, x5 x3	x0000_0006
SRL	rd = rs1 >> rs2	x16 x2 x1	x0000_0000
SRA	rd = rs1 >>> rs2	x17 x9 x20	x0000_0000
OR	rd = rs1 | rs2	x18 x2 x3	x0000_0003
AND	rd = rs1 & rs2	x19 x2 x3	x0000_0002

4. S-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	
imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode							S



● S-Type

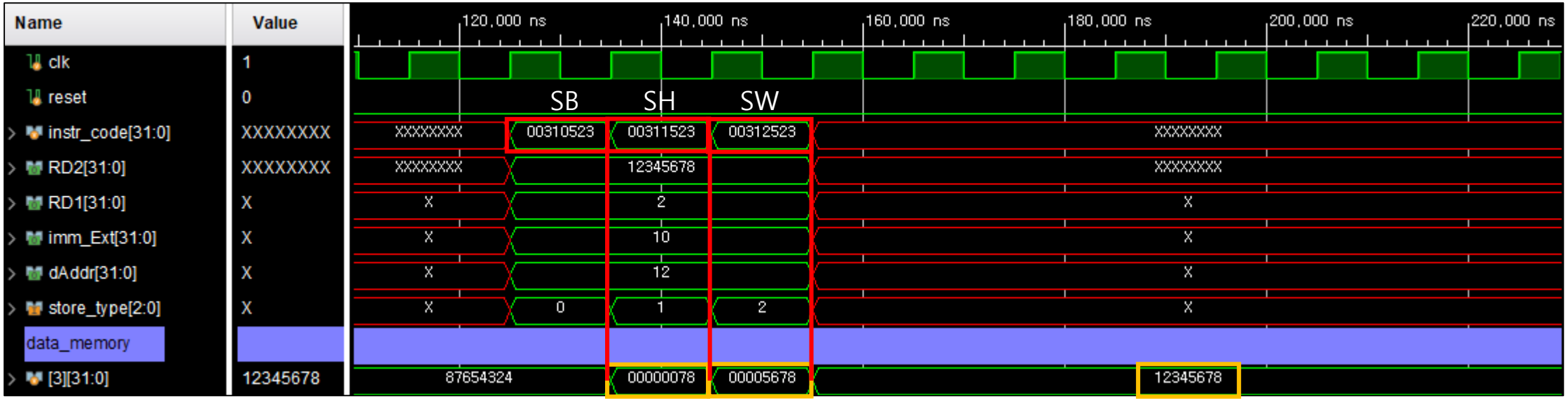
"Store"

레지스터 데이터 메모리에 값 저장

M[rs1 + imm] = rs2

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]

4. S-Type Simulation



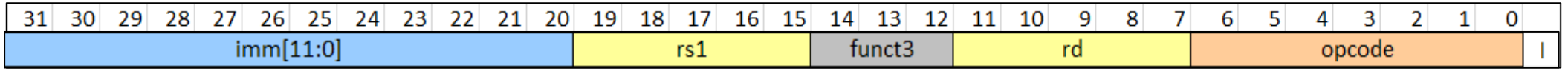

```
reg_file[3]  = 32'h12345678;
```

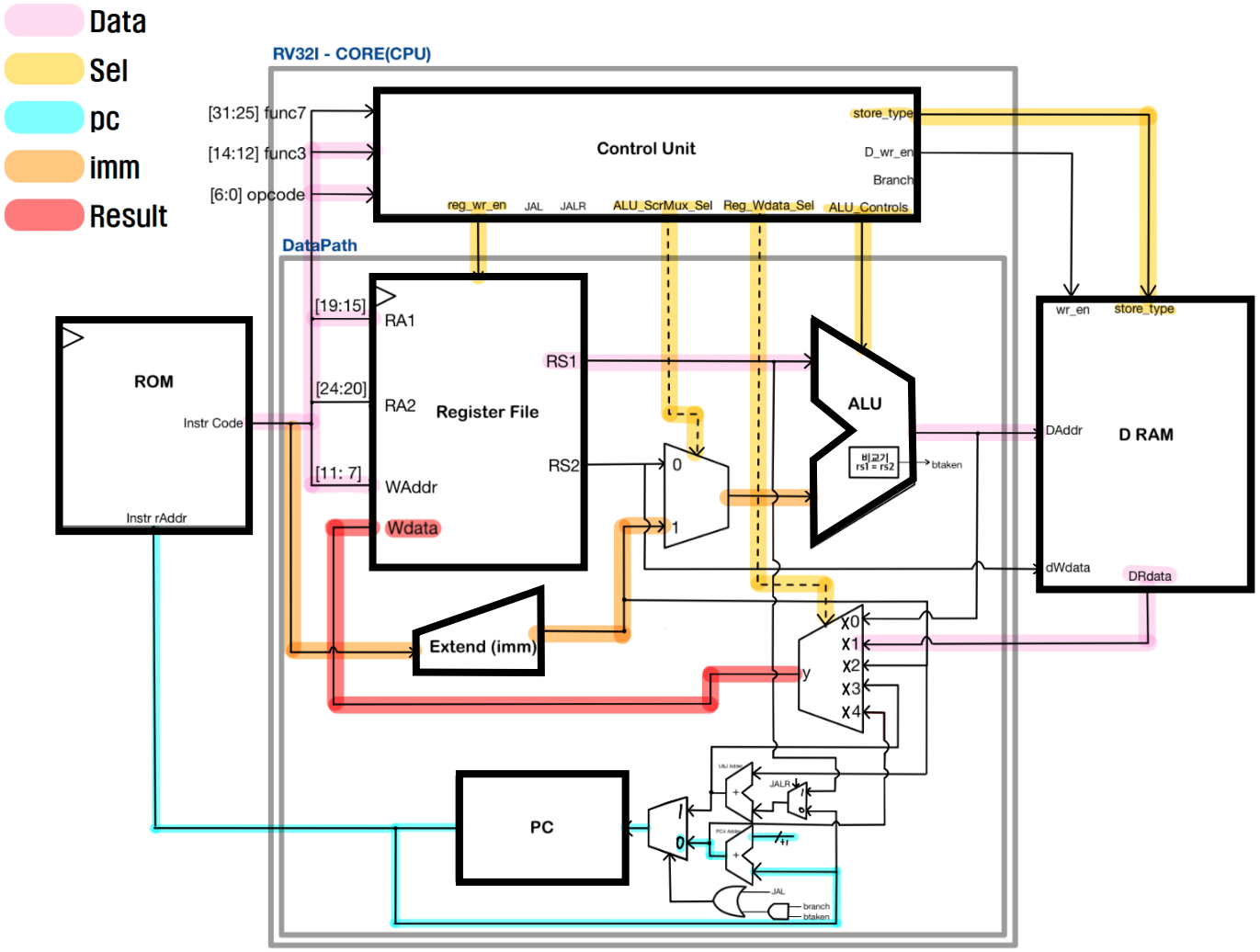
[SB] : addr = 1100 rs1 = 10 + imm = 1010 >> Data_Memory : 0000_0078
[SH] : addr = 1100 rs1 = 10 + imm = 1010 >> Data_Memory : 0000_5678
[SW] : addr = 1100 rs1 = 10 + imm = 1010 >> Data_Memory : 1234_5678

addr[31:2] 이므로 = 3
Ram[3]에 값 저장

```
always_ff @(posedge clk) begin
    if (d_wr_en) begin
    case (store_type)
        3'b000 :  data_mem[dAddr[31:2]] <= dWdata[ 7:0];
        3'b001 :  data_mem[dAddr[31:2]] <= dWdata[15:0];
        3'b010 :  data_mem[dAddr[31:2]] <= dWdata;
        default : data_mem[dAddr[31:2]] <= 32'dx;
    endcase
    end
end
```

4. IL-Type





● I-Type

"Immediate"

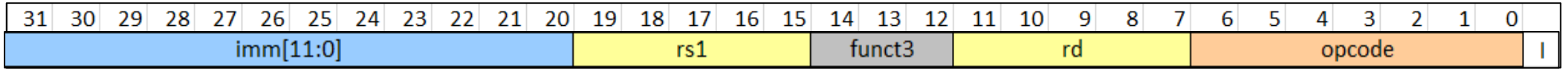
상수값으로 연산 수행

"I-Load"

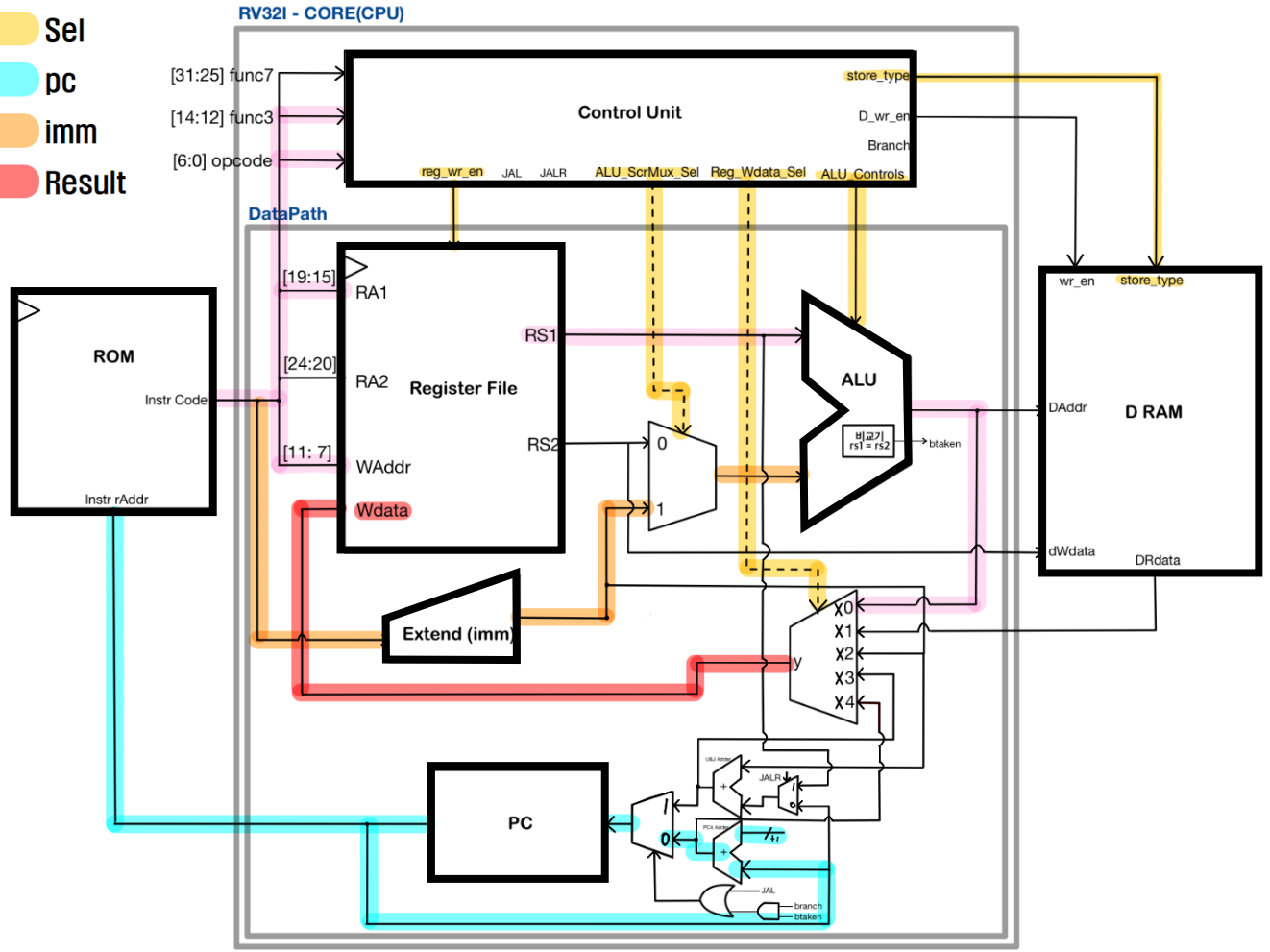
RAM에서 레지스터로 Data Load

I	LB	rd = M[rs1+imm][0:7]
I	LH	rd = M[rs1+imm][0:15]
I	LW	rd = M[rs1+imm][0:31]
I	LBU	rd = M[rs1+imm][0:7]
I	LHU	rd = M[rs1+imm][0:15]
I	ADDI	rd = rs1 + imm
I	SLTI	rd = (rs1 < imm)?1:0
I	SLTIU	rd = (rs1 < imm)?1:0
I	XORI	rd = rs1 ^ imm
I	ORI	rd = rs1 | imm
I	ANDI	rd = rs1 & imm
I	SLLI	rd = rs1 << imm[0:4]
I	SRLI	rd = rs1 >> imm[0:4]
I	SRAI	rd = rs1 >> imm[0:4]

4. I-Type



Data
Sel
pc
imm
Result



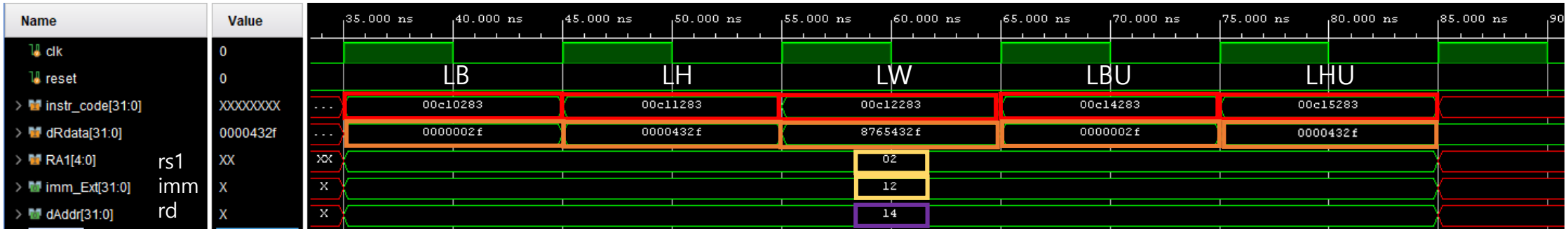
● I-Type

"Immediate"
상수값으로 연산 수행

"I-Load"
RAM에서 레지스터로 Data Load

I	LB	rd = M[rs1+imm][0:7]
I	LH	rd = M[rs1+imm][0:15]
I	LW	rd = M[rs1+imm][0:31]
I	LBU	rd = M[rs1+imm][0:7]
I	LHU	rd = M[rs1+imm][0:15]
I	ADDI	rd = rs1 + imm
I	SLTI	rd = (rs1 < imm)?1:0
I	SLTIU	rd = (rs1 < imm)?1:0
I	XORI	rd = rs1 ^ imm
I	ORI	rd = rs1 | imm
I	ANDI	rd = rs1 & imm
I	SLLI	rd = rs1 << imm[0:4]
I	SRLI	rd = rs1 >> imm[0:4]
I	SRAI	rd = rs1 >> imm[0:4]

4. IL-Type Simulation

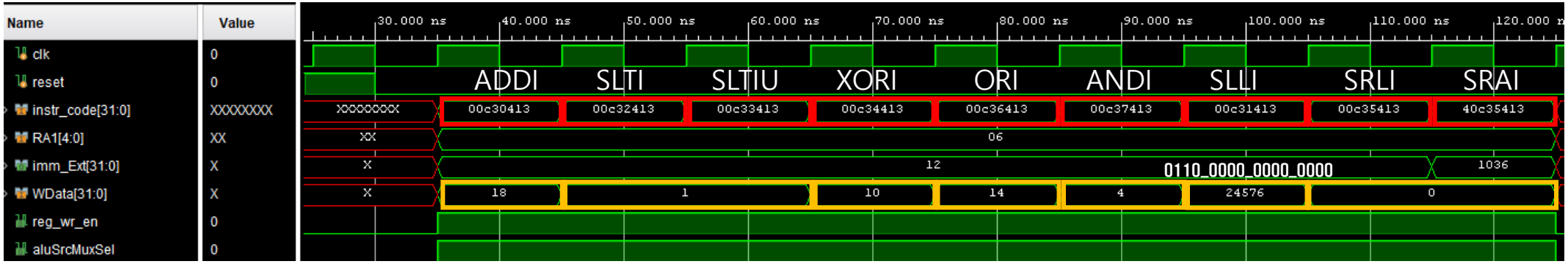


'Hx87654321 + E == 'Hx8765432F

mnemoic	Semantics	Format	Result
LB	rd = M[rs1+imm][0: 7]	x5, x2 x12	x0000_0005
LH	rd = M[rs1+imm][0:15]	x5, x2 x12	xFFFF_FFFE
LW	rd = M[rs1+imm][0:31]	x5, x2 x12	x0000_0004
LBU	rd = M[rs1+imm][0: 7] (zero)	x5, x2 x12	x0000_0001
LHU	rd = M[rs1+imm][0:15] (zero)	x5, x2 x12	x0000_0001

I	LB	rd = M[rs1+imm][0:7]
I	LH	rd = M[rs1+imm][0:15]
I	LW	rd = M[rs1+imm][0:31]
I	LBU	rd = M[rs1+imm][0:7]
I	LHU	rd = M[rs1+imm][0:15]

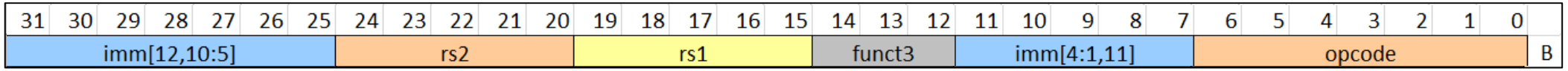
4. I-Type Simulation

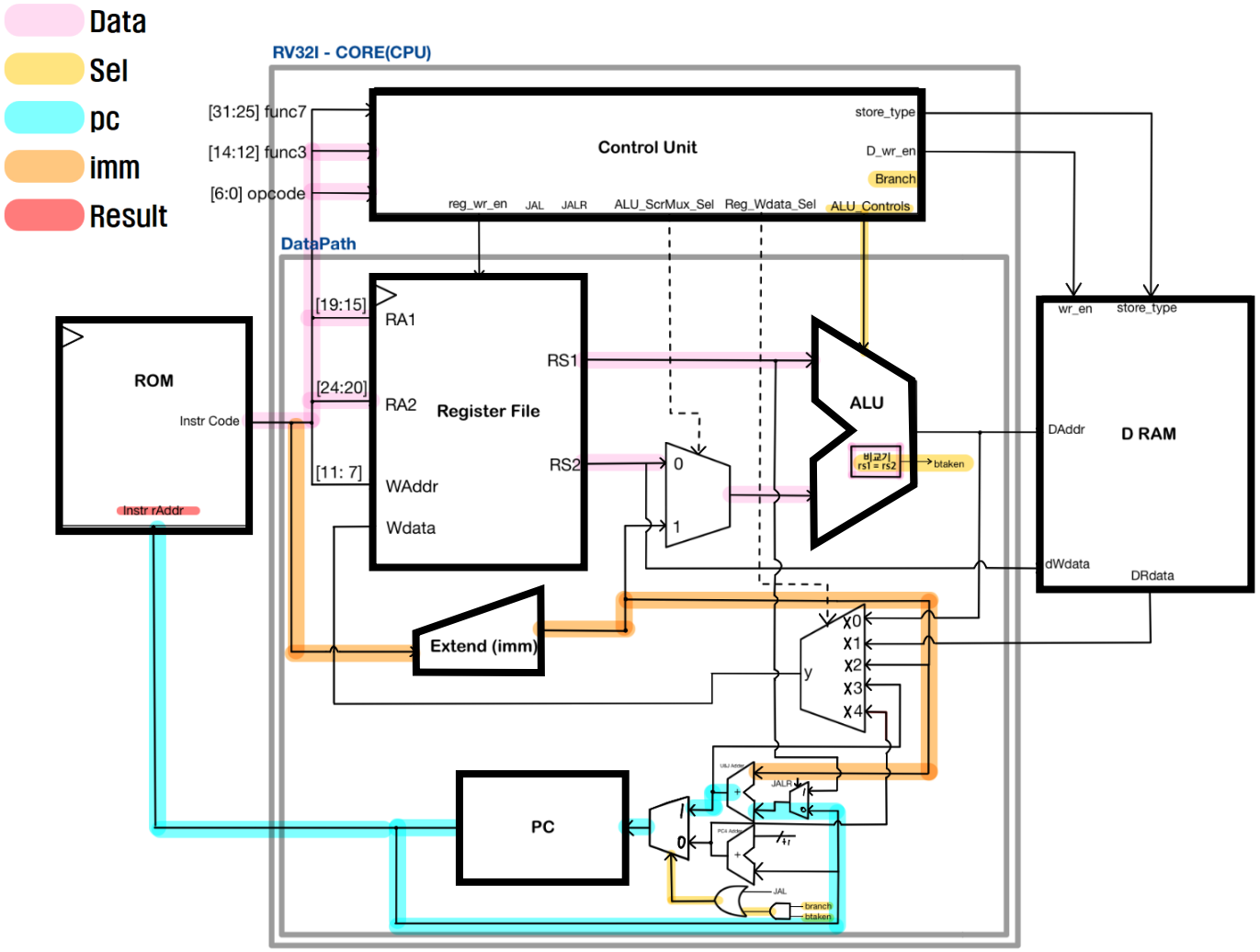


mnemoic	Semantics	Format	Result
ADDI	rd = rs1 + imm	x5, x6 x12	'dx18
SLTI	rd = (rs1 < imm)?1:0	x5, x6 x12	'dx1
SLTIU	rd = (rs1 < imm)?1:0	x5, x6 x12	'dx1
XORI	rd = rs1 ^ imm	x5, x6 x12	'dx10
ORI	rd = rs1 | imm	x5, x6 x12	'dx14

mnemoic	Semantics	Format	Result
ANDI	rd = rs1 & imm	x5, x6 x12	'dx4
SLLI	rd = rs1 << imm[0:4]	x5, x6 x12	'dx24576
SRLI	rd = M[rs1+imm][0:31]	x5, x6 x12	'dx0
SRAI	rd = rs1 >> imm[0:4]	x5, x6 x12	'dx0

4. B-Type





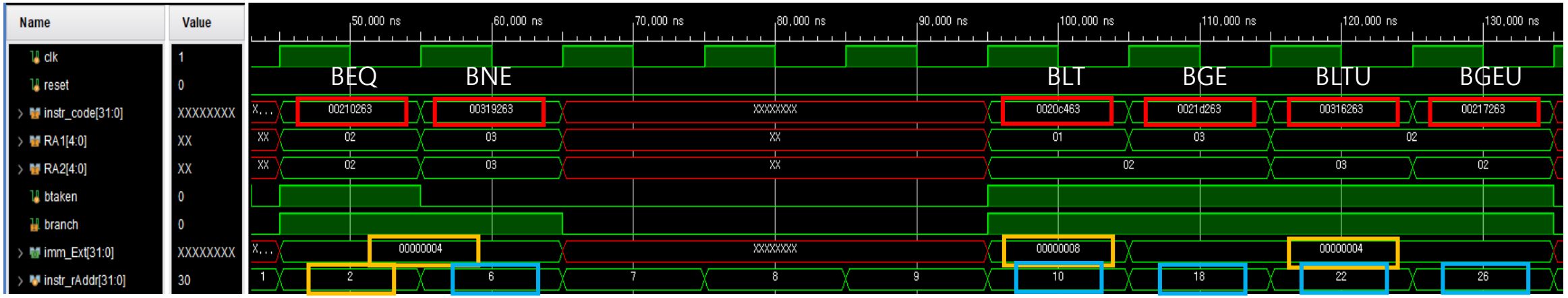
● B-Type

“Branch”

레지스터 값 비교 후 특정 조건 만족 시 PC 분기

B	BEQ	if(rs1 == rs2) PC += imm
B	BNE	if(rs1 != rs2) PC += imm
B	BLT	if(rs1 < rs2) PC += imm
B	BGE	if(rs1 >= rs2) PC += imm
B	BLTU	if(rs1 < rs2) PC += imm
B	BGEU	if(rs1 >= rs2) PC += imm

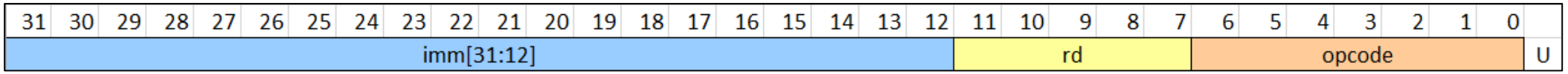
4. B-Type Simulation



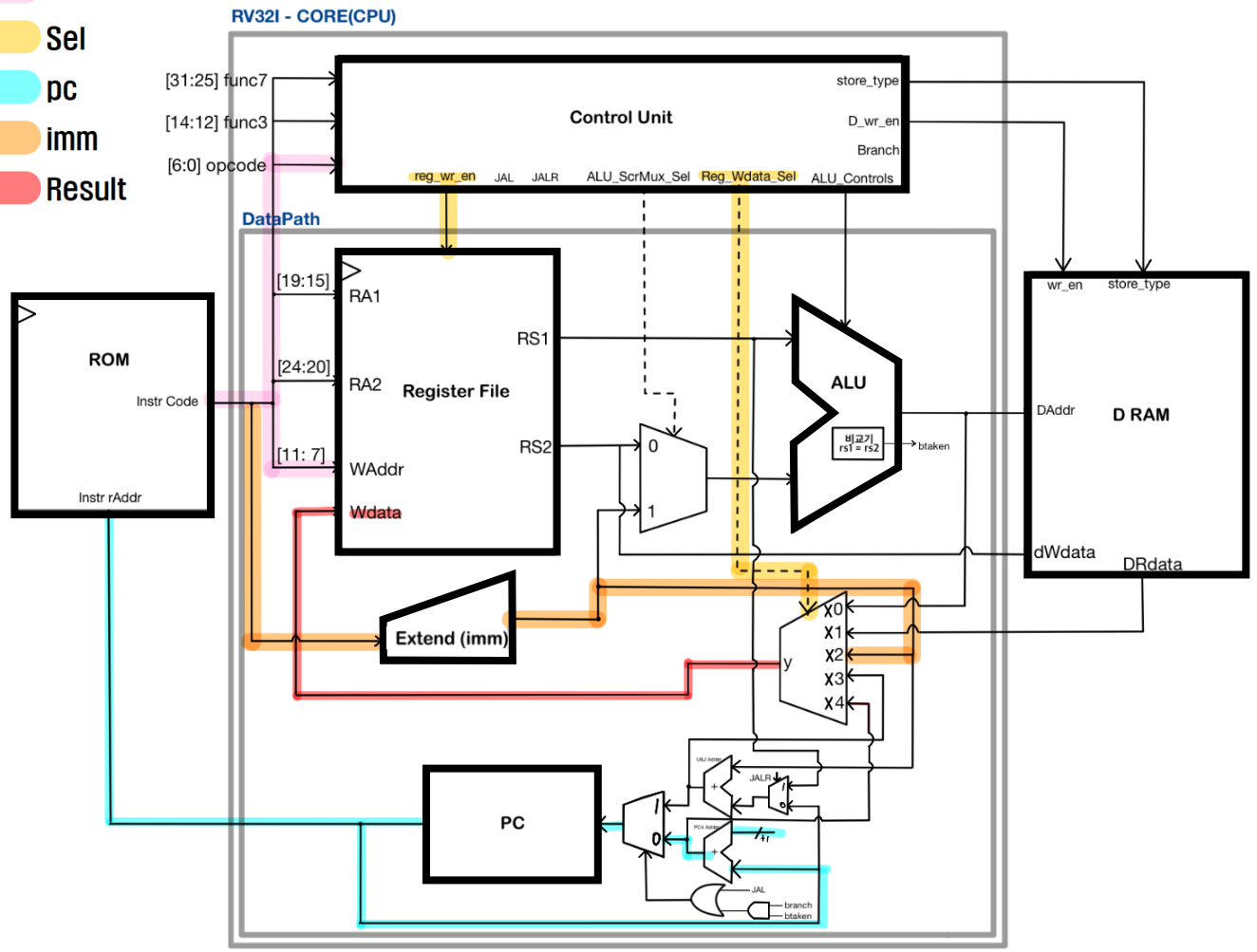
mnemoic	Semantics	Format
BEQ	if(rs1 == rs2) PC += imm	x2, x2, x4
BNE	if(rs1 != rs2) PC += imm	x2, x2, x4
BLT	if(rs1 < rs2) PC += imm	x51 x2, x8
BGE	if(rs1 >= rs2) PC += imm	x2, x2, x4
BLTU	if(rs1 < rs2) PC += imm	x2, x2, x4
BGEU	if(rs1 >= rs2) PC += imm	x2, x2, x4

BEQ : rs1 = 2 rs2 = 2 imm = 4
BNE : rs1 = 2 rs2 = 3 imm = 4 (FALSE)
BLT : rs1 = 1 rs2 = 2 imm = 8
BGE : rs1 = 3 rs2 = 2 imm = 4
BLTU : rs1 = 2 rs2 = 2 imm = 4
BGEU : rs1 = 2 rs2 = 2 imm = 4

4. U-Type (LUI)



Data
Sel
pc
imm
Result



● U-Type

"Upper-Immediate"

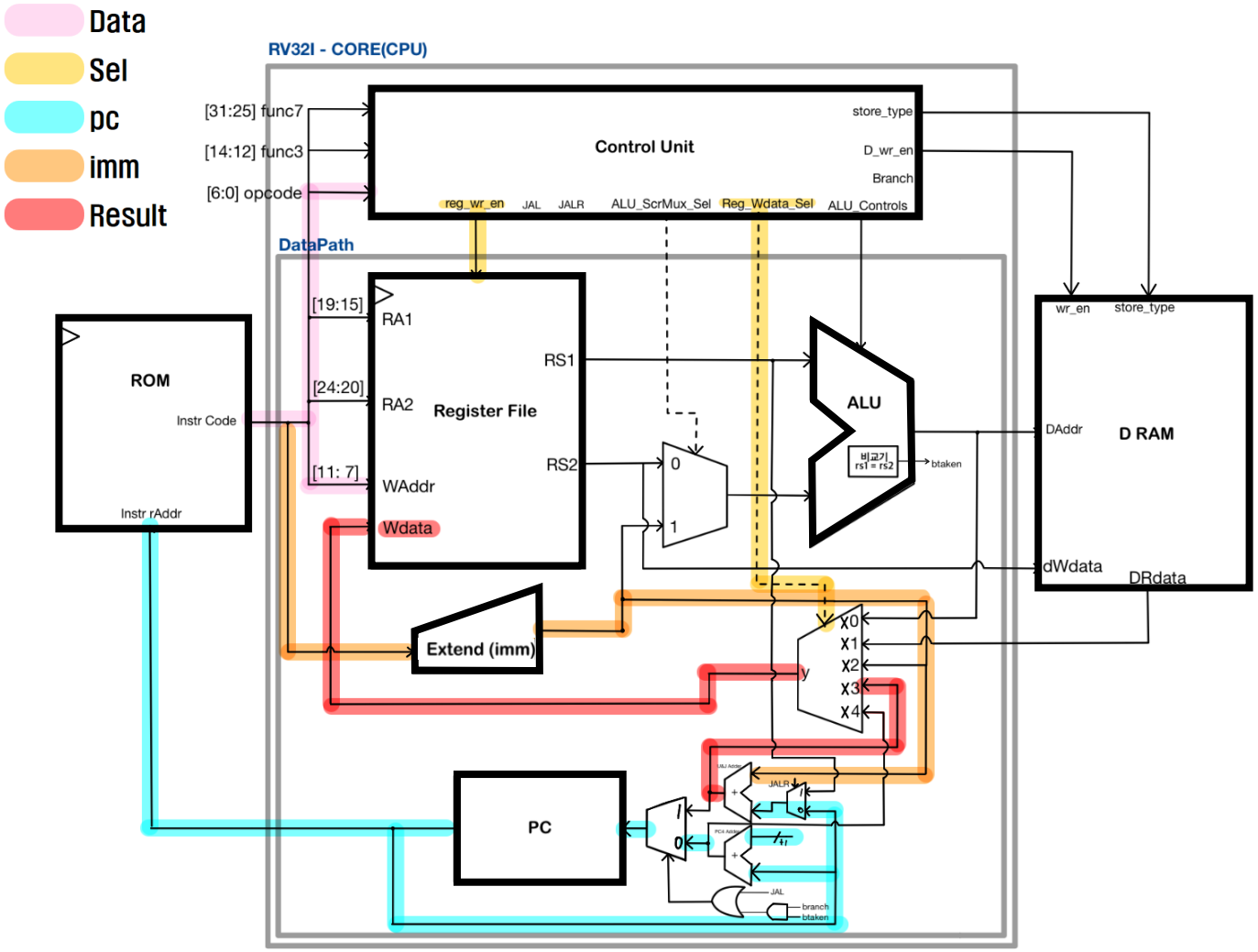
상수 값 로드 후 목적 레지스터에 저장

(메모리에서 안쓰고 바로 레지스터 보냄)

LUI	rd = imm
AUIPC	rd = PC + imm

4. U-Type (AUIPC)

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							U



● U-Type

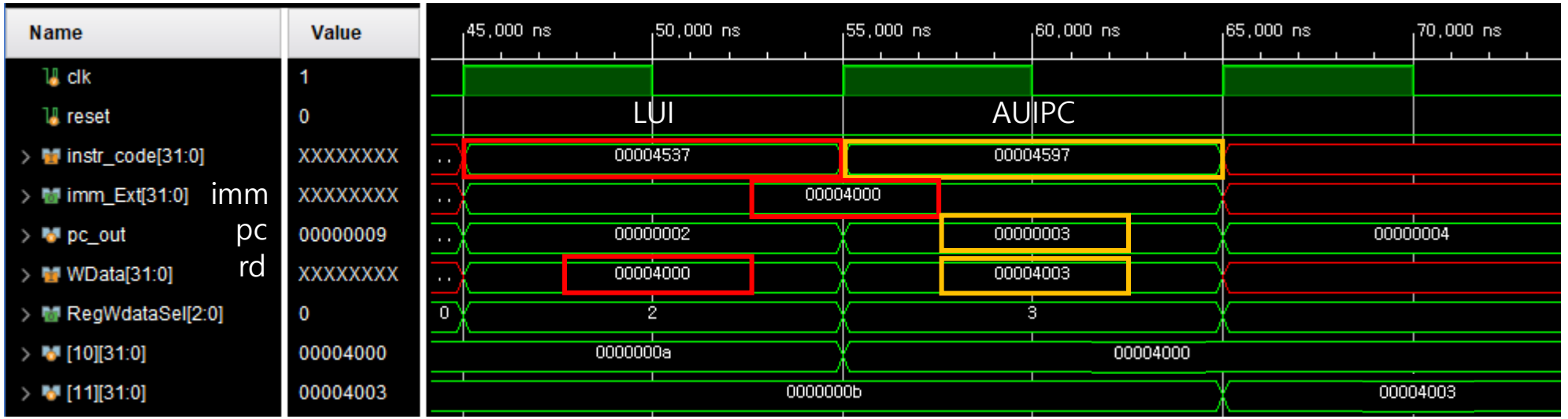
"Upper-Immediate"

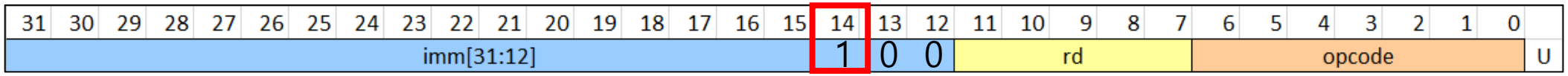
상수 값 로드 후 목적 레지스터에 저장

(메모리에서 안쓰고 바로 레지스터 보냄)

LUI	rd = imm
AUIPC	rd = PC + imm

4. U-Type Simulation





LUI	rd = imm
AUIPC	rd = PC + imm

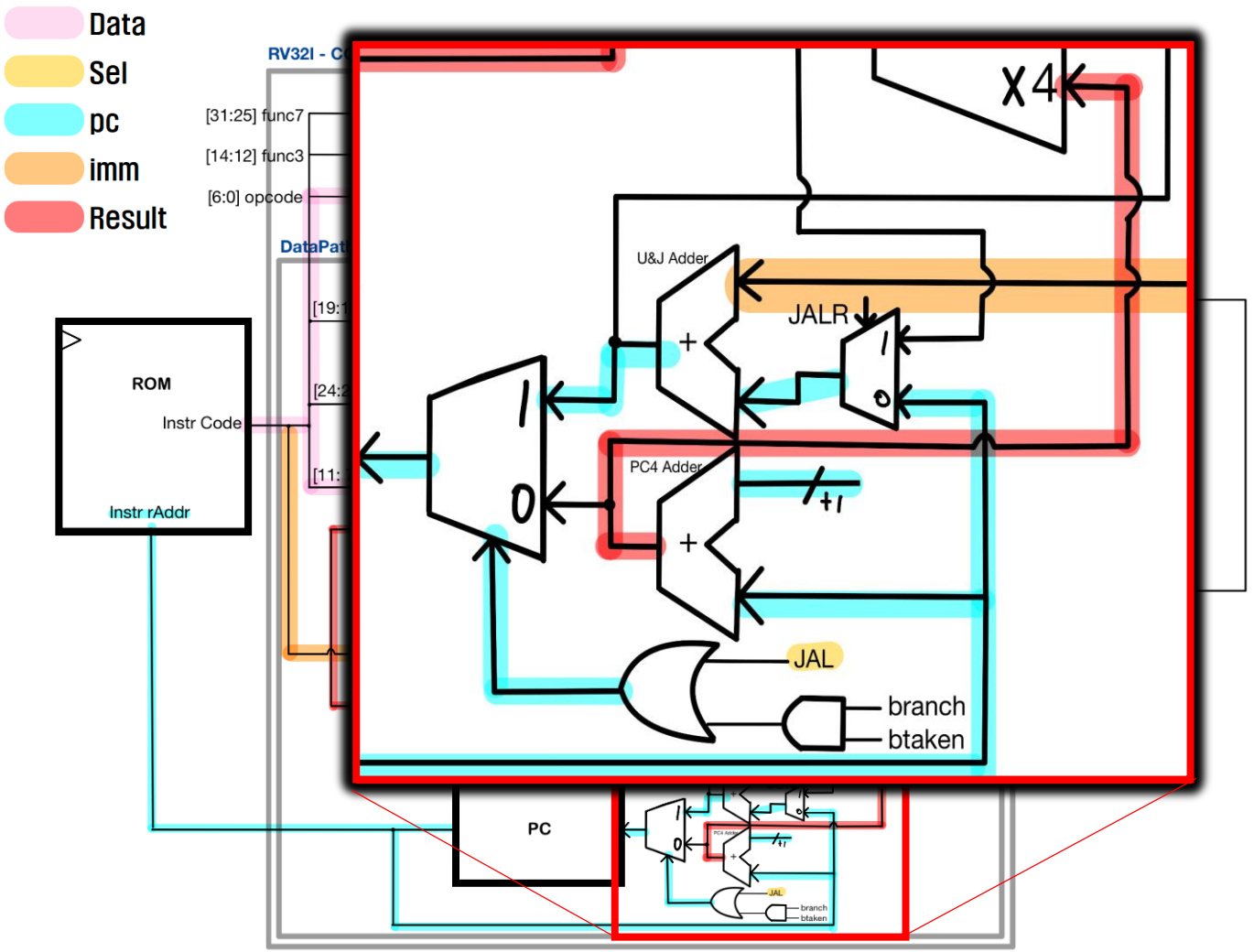
```
// lui                          imm=4  x10
rom[2] = 32'b0000000_00000_00000_100_01010_0110111;
// auipc                        imm=4  x11
rom[3] = 32'b0000000_00000_00000_100_01011_0010111;
```

LUI : imm = 0100_0000_0000_0000
rd= imm
hx4000 = 0100_0000_0000_0000

AUIPC : imm = 0100_0000_0000_0000 / pc = 3
rd= pc(3) + imm
hx4000 = 0100_0000_0000_0000

4. J-Type (JAL)

imm[20,10:1,11,19:12]					rd	1	1	0	1	1	1	1	J	JAL	rd = PC+4; PC += imm
imm[11:0]	rs1	0	0	0	rd	1	1	0	0	1	1	1	JL	JALR	rd = PC+4; PC = rs1 + imm



● J-Type

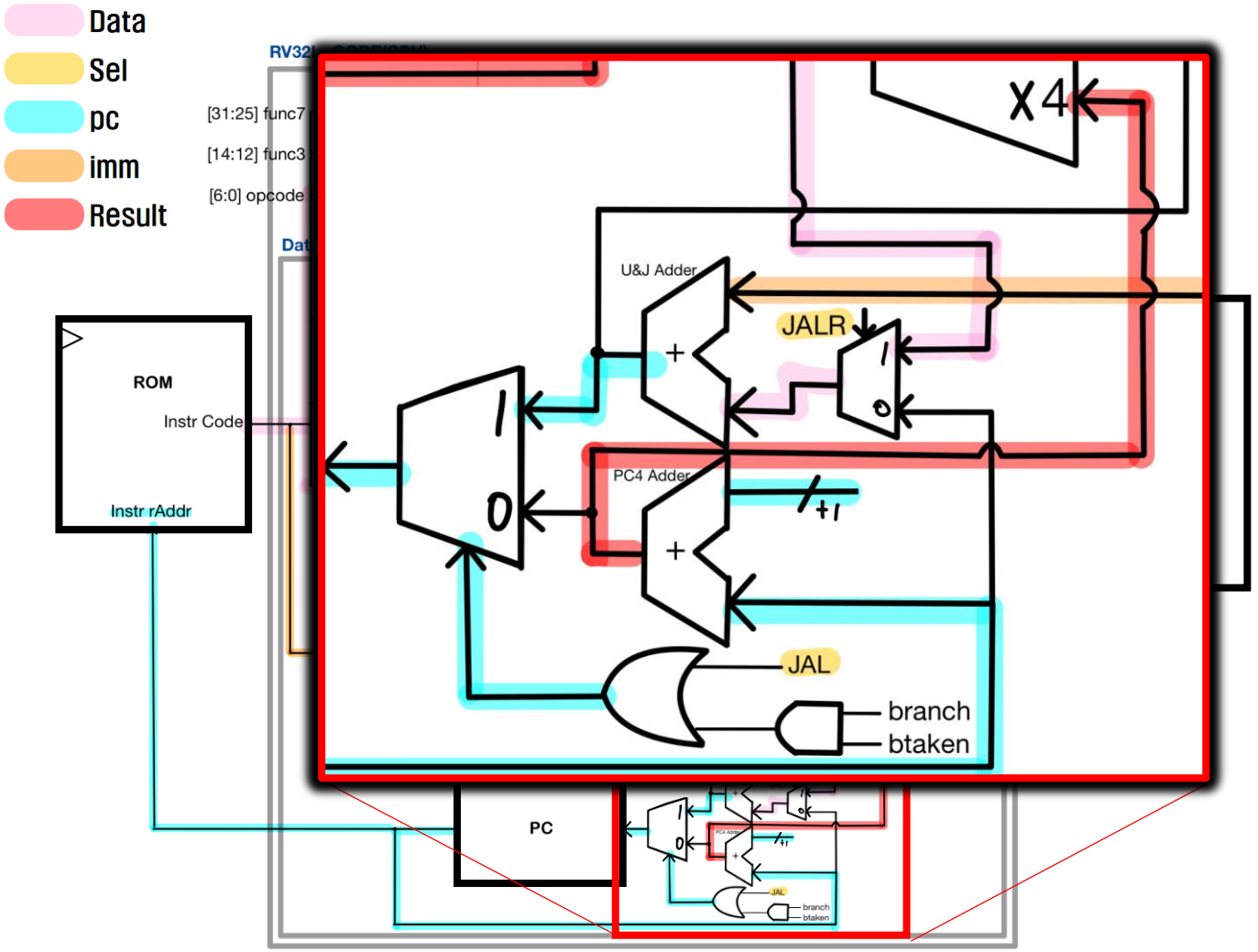
"Jump"

PC를 변경하여 흐름을 유연하게 제어

JAL	rd = PC+4; PC += imm
JALR	rd = PC+4; PC = rs1 + imm

4. JL-Type (JALR)

imm[20,10:1,11,19:12]					rd	1	1	0	1	1	1	1	J	JAL	rd = PC+4; PC += imm
imm[11:0]	rs1	0	0	0	rd	1	1	0	0	1	1	1	JL	JALR	rd = PC+4; PC = rs1 + imm



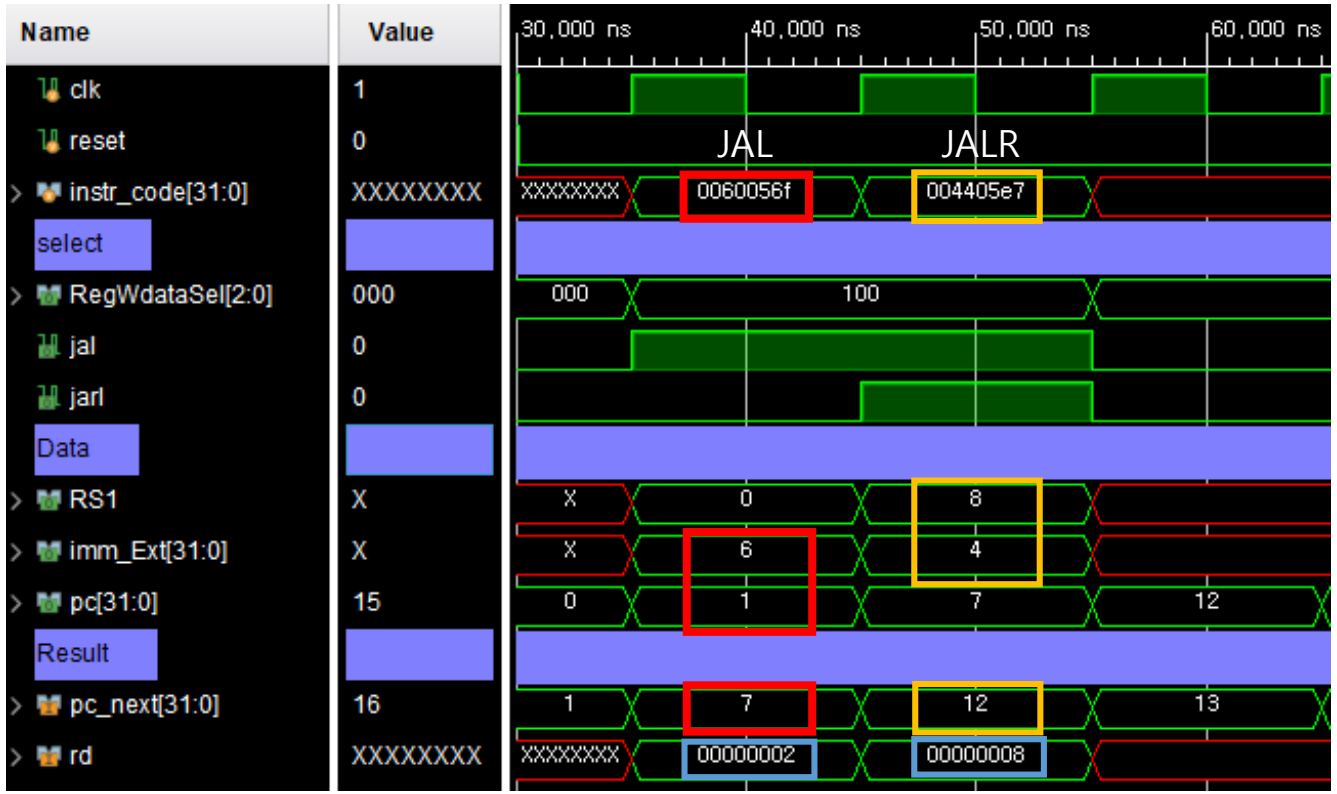
J-Type

"Jump"

PC를 변경하여 흐름을 유연하게 제어

JAL	rd = PC+4; PC += imm
JALR	rd = PC+4; PC = rs1 + imm

4. J-Type Simulation




```
// 32'h0060_056F JAL       IMM : 6    RD  :X10
rom[1] = 32'b000000000_11_0_0000_0000_01010_1101111;
// 32'h0044_05E7 JALR      RS1 : 8    RD: X11
rom[7] = 32'b0000_0000_0100_01000_000_01011_1100111;
```

JAL	rd = PC+4; PC += imm
JALR	rd = PC+4; PC = rs1 + imm

JAL: imm = 6
pc_next(7) = pc(1) + imm(6)
rd(2) = pc(1) + 1 (word index)

JALR: imm = 4 , rs1 = 8
pc_next(12) = imm(4) + rs1(8)
rd(8) = pc(7) + 1 (word index)

5. 고 찰

RV32I를 구현함으로써 하드웨어의 동작과 구조를 종합적으로 이해할 수 있었다
연산 방법뿐 아니라 제어 방법의 중요성도 확인할 수 있었다
다음에는 멀티사이클로 확장하고, 필요하면 파이프라인을 구현하여 CPI에 미치는 영향을 확인한다