System Verilog Project

RV32I Core – Single Cycle

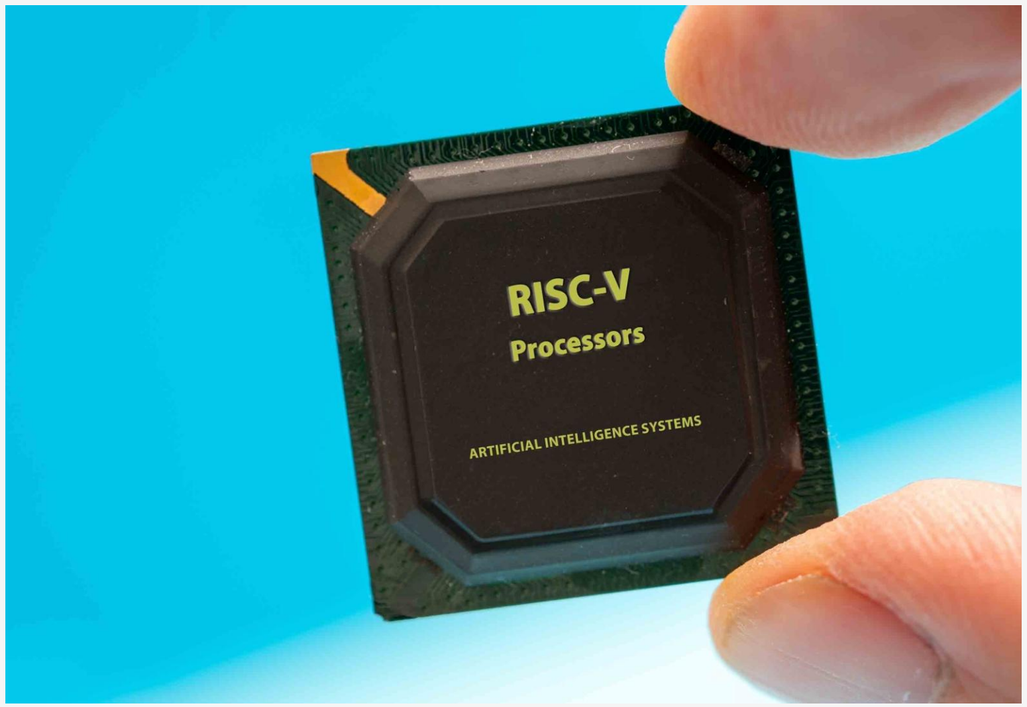
12조

임희주

2025.04.15

Contents

01 RISC – V 개요



RISC-V

1. RISC (Reduced Instruction Set Computer) 구조

- 기존의 CISC 보다 적은 수의 명령어
- 여러 Type의 명령어를 통한 CPU 제어
- 기본적으로 32bit 명령어 사용
- 회로 구성 단순

2. 오픈 소스 CPU 아키텍처

- 학술 / 연구용으로 적합 + 산업계 상용화 목표
- 다양한 설계 방식 존재

3. 특징

- 마이크로 컨트롤러, 임베디드, 모바일 등 다양한 분야에 적용
- 빠른 실행 속도, 높은 확장성

02 개발 환경 및 목적

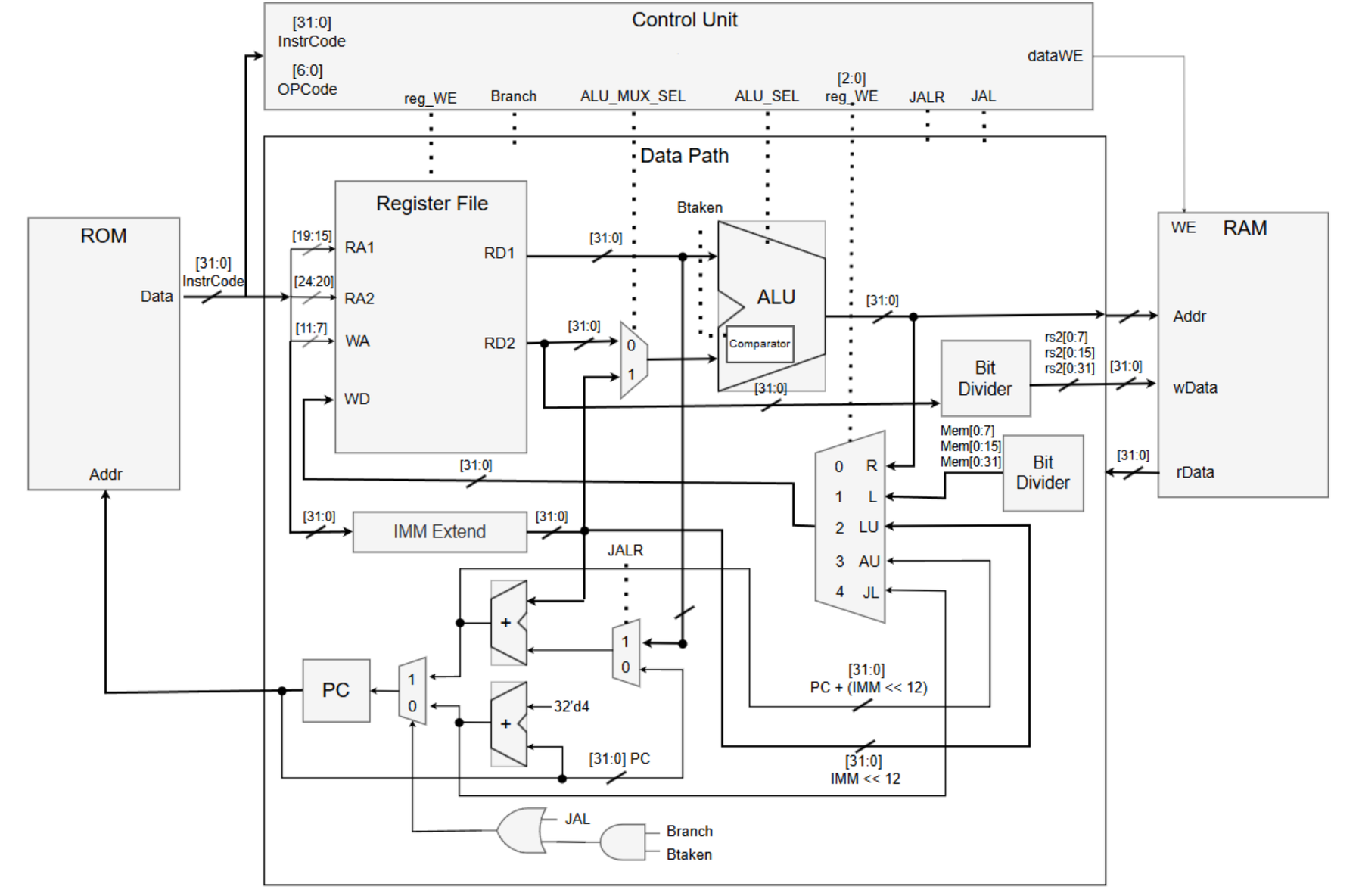
개발 환경

개발 목적

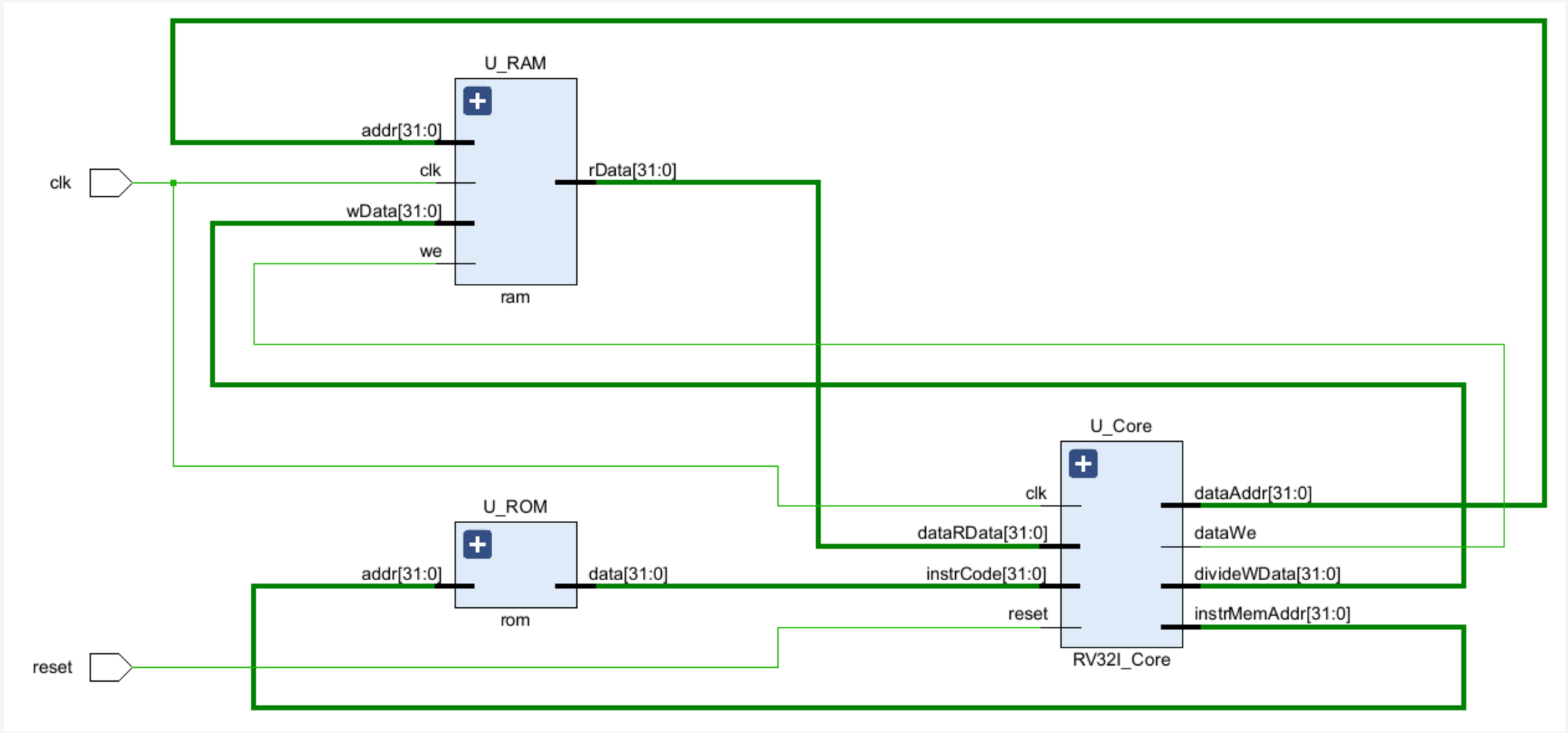
1. **SystemVerilog를 활용한 RV32I Core 구현**
   - 하버드 구조를 이루는 여러 모듈 구현 (ALU, MEM, PC)
   - RAM/ROM 등 메모리 구조의 이해
   - RISC–V의 기본 명령어 구현 및 이해

2. **RISC – V 구조의 이해**
   - Store / Load, Branch 등 명령어의 동작 이해
   - Multi-cycle, Pipelining 구조 구현을 위한 기본기
   - 최적화된 모듈 설계를 배움으로, MCU 개선 방법 학습

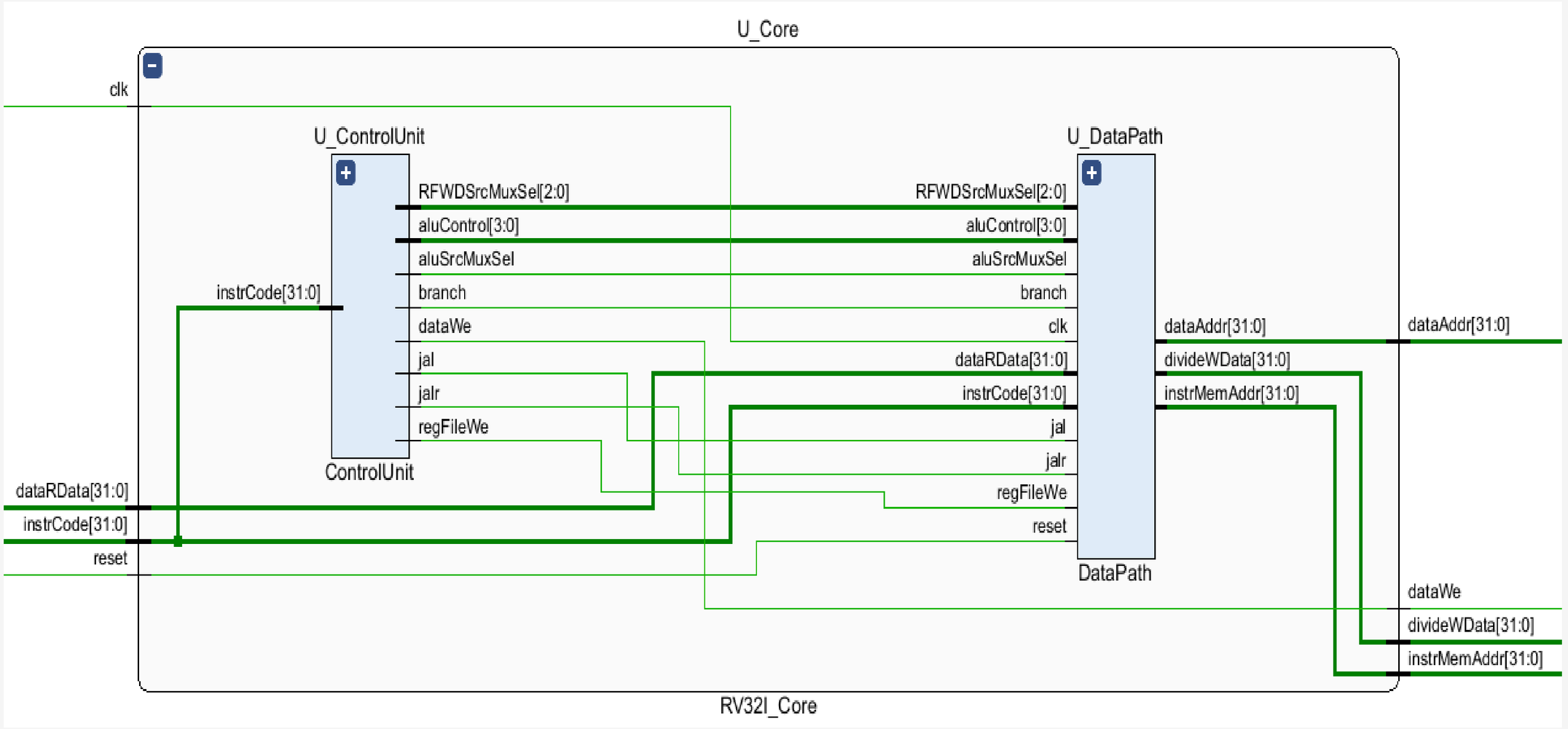
03 블록도 & 회로도


Control Unit
[31:0]
InstrCode
[6:0]
OPCode
dataWE
reg_WE
Branch
ALU_MUX_SEL
ALU_SEL
[2:0]
reg_WE
JALR
JAL
Data Path
Register File
RA1
RA2
WA
WD
RD1
RD2
[19:15]
[24:20]
[11:7]
[31:0]
ROM
Data
Addr
[31:0]
InstrCode
Btaken
ALU
Comparator
[31:0]
WE
RAM
Addr
wData
rData
rs2[0:7]
rs2[0:15]
rs2[0:31]
Bit
Divider
Mem[0:7]
Mem[0:15]
Mem[0:31]
0 R
1 L
2 LU
3 AU
4 JL
IMM Extend
JALR
PC
32'd4
[31:0] PC
[31:0]
PC + (IMM << 12)
[31:0]
IMM << 12
JAL
Branch
Btaken

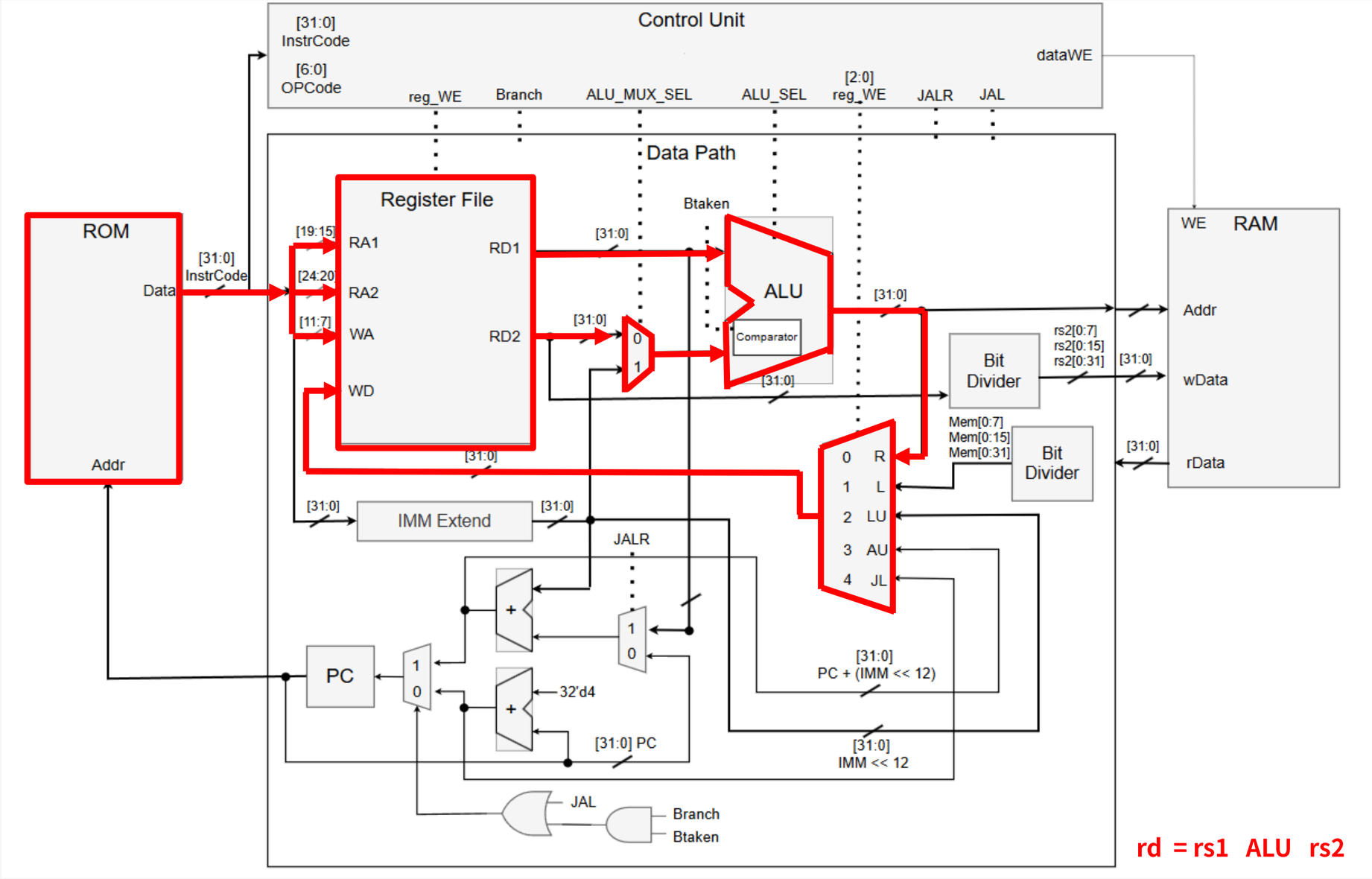
03 블록도 & 회로도


U_RAM
addr[31:0]
clk
wData[31:0]
we
rData[31:0]
ram
clk
U_ROM
addr[31:0]
data[31:0]
rom
reset
U_Core
clk
dataRData[31:0]
instrCode[31:0]
reset
dataAddr[31:0]
dataWe
divideWData[31:0]
instrMemAddr[31:0]
RV32I_Core

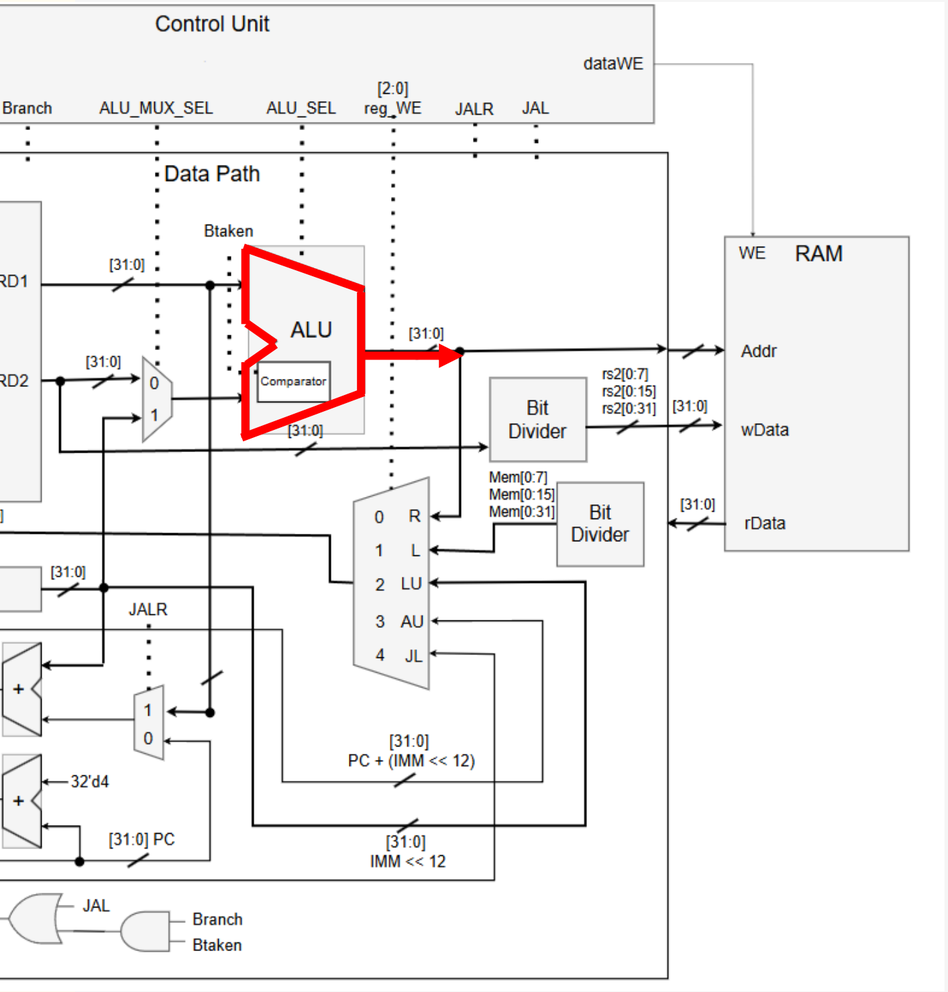
03 블록도 & 회로도


U_Core
clk
U_ControlUnit
RFWDSrcMuxSel[2:0]
aluControl[3:0]
aluSrcMuxSel
branch
dataWe
jal
jalr
regFileWe
instrCode[31:0]
ControlUnit
U_DataPath
RFWDSrcMuxSel[2:0]
aluControl[3:0]
aluSrcMuxSel
branch
clk
dataRData[31:0]
instrCode[31:0]
jal
jalr
regFileWe
reset
DataPath
dataAddr[31:0]
divideWData[31:0]
instrMemAddr[31:0]
dataAddr[31:0]
dataRData[31:0]
instrCode[31:0]
reset
dataWe
divideWData[31:0]
instrMemAddr[31:0]
RV32I_Core

04 Type 별 설명 (R-Type)



04 Type 별 설명 (R-Type)



ALU Function Control (Func3 & Func7[5])

[31:25]	[24:20]	[19:15]	[14:12]	[11:07]	[06:00]
Function7	RS2	RS1	Function3	RD	OPcode
0 0 0 0 0 0 0 0 1 0 0 0 0 0			[0 0 0] . . . [1 1 1]		0 1 1 0 0 1 1

```
`ADD:     result = a + b;
`SUB:     result = a - b;                    //func7[5] = "1"
`SLL:     result = a << b;
`SRL:     result = a >> b;
`SRA:     result = $signed(a) >>> b[4:0];    //func7[5] = "1"
`SLT:     result = ($signed(a) < $signed(b)) ? 1 : 0;
`SLTU:    result = (a < b) ? 1 : 0;
`XOR:     result = a ^ b;
`OR:      result = a | b;
`AND:     result = a & b;
default: result = 32'bx;
```

04 R-Type 시뮬레이션 설명

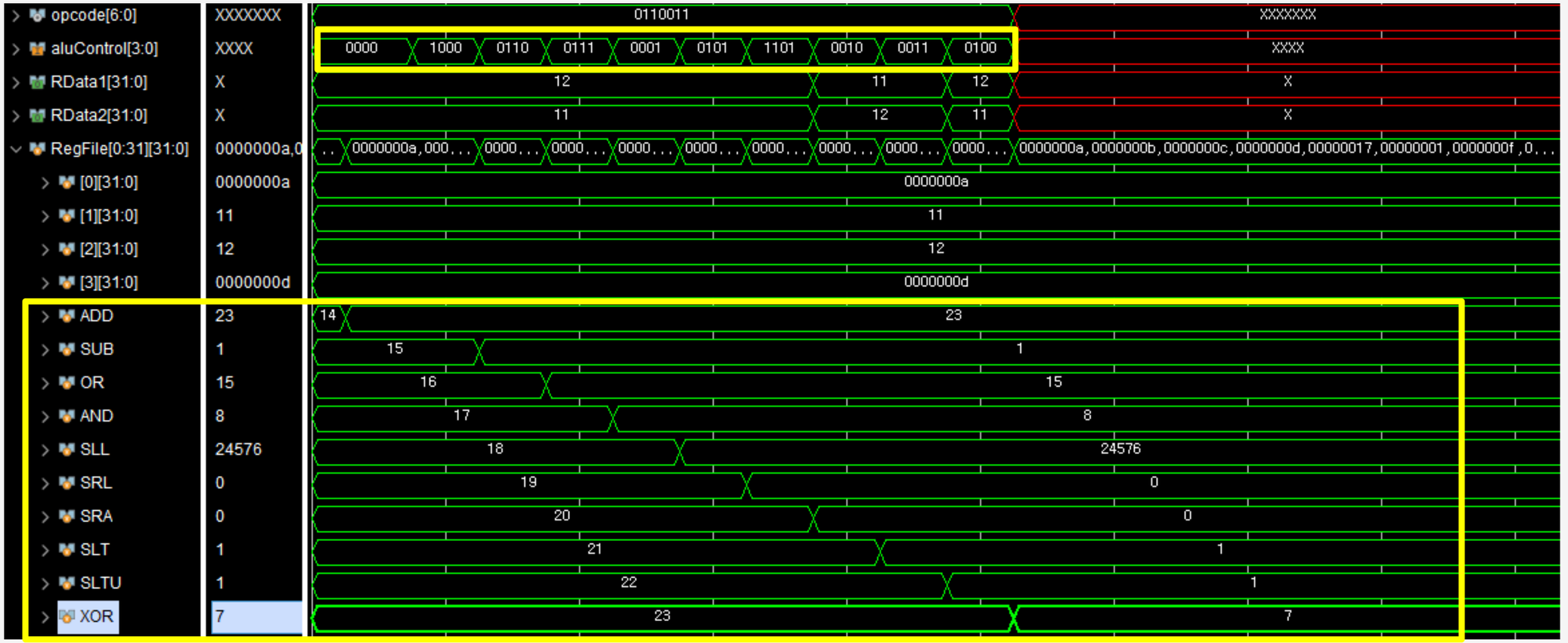
```
//rom[x] = 32'b func7 _ rs2 _ rs1 _ f3 _ rd _ opcode; // R-Type
rom[0] = 32'b0000000_00001_00010_000_00100_0110011;  // ADD  x4, x2, x1
rom[1] = 32'b0100000_00001_00010_000_00101_0110011;  // SUB  x5, x2, x1
rom[2] = 32'b0000000_00001_00010_110_00110_0110011;  // OR   x6, x2, x1
rom[3] = 32'b0000000_00001_00010_111_00111_0110011;  // AND  x7, x2, x1
rom[4] = 32'b0000000_00001_00010_001_01000_0110011;  // SLL  x8, x2, x1
rom[5] = 32'b0000000_00001_00010_101_01001_0110011;  // SRL  x9, x2, x1
rom[6] = 32'b0100000_00001_00010_101_01010_0110011;  // SRA  x10, x2, x1
rom[7] = 32'b0000000_00010_00001_010_01011_0110011;  // SLT  x11, x1, x2
rom[8] = 32'b0000000_00010_00001_011_01100_0110011;  // SLTU x12, x1, x2
rom[9] = 32'b0000000_00001_00010_100_01101_0110011;  // XOR  x13, x2, x1
```

Func7[5] & [2:0]Func3 = ALU Control

1. ADD : 11 + 12 = 23
2. SUB : 12 – 11 = 1
3. OR : 11 ^ 12 = 15
4. AND : 11 & 12 = 8
5. SLL : 32’d12 << 11 = 24576
6. SRL : 32’d12 >> 11 = 0
7. SRA : 32’d12 >> 11 (Arith = 0) = 0
8. SLT : 32’d11 < 32’d12 = 1
9. SLTU : 32’d11 < 32’d12 = 1
10. XOR : 11 XOR 12 = 7

※Signed

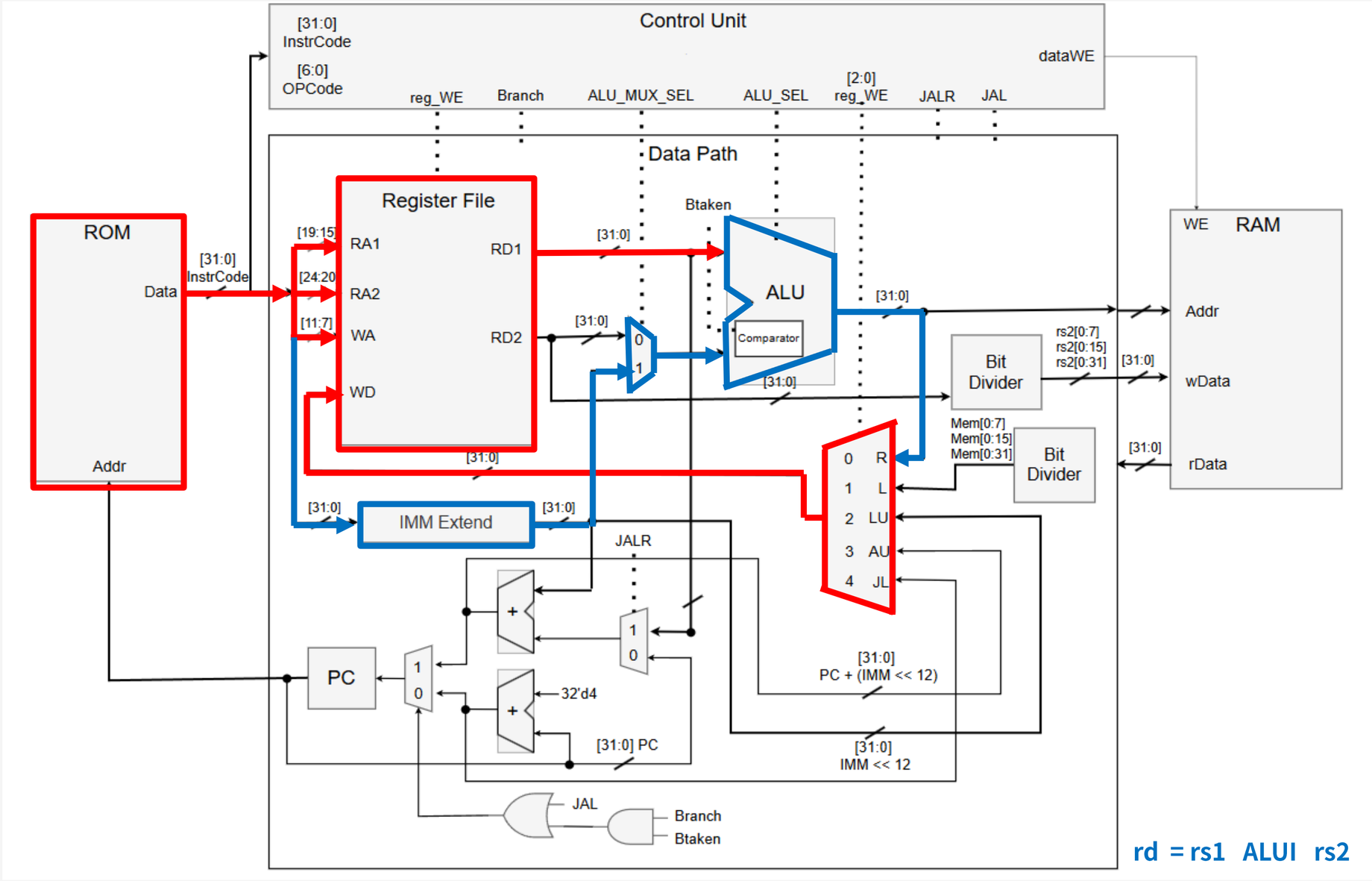
04 R-Type 시뮬레이션 설명



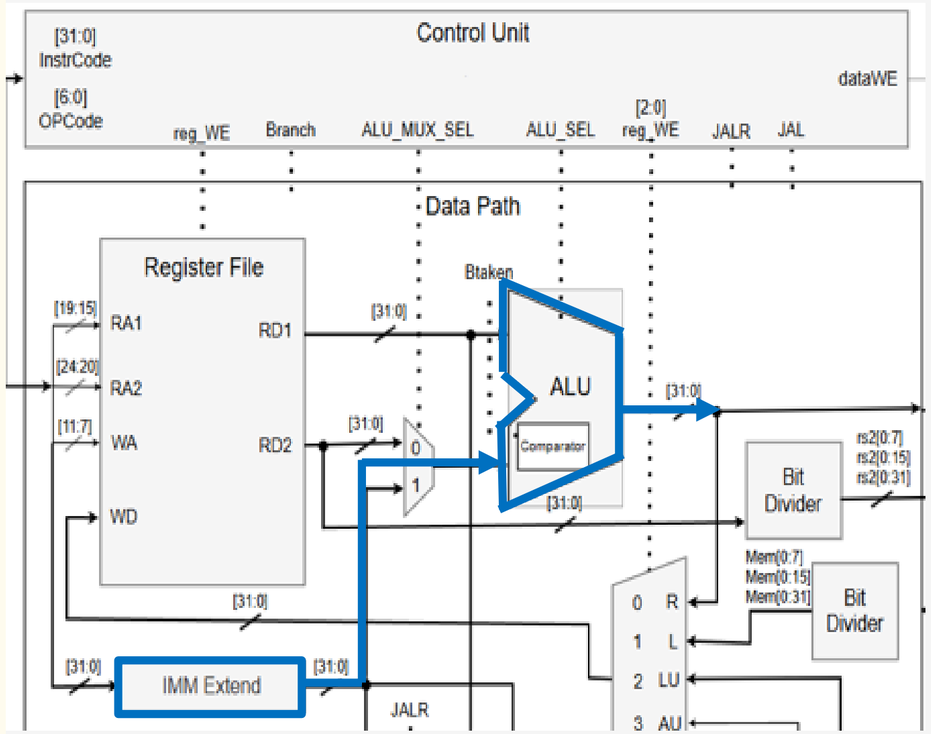
1. ADD : 11 + 12 = 23
2. SUB : 12 – 11 = 1
3. OR : 11 ^ 12 = 15
4. AND : 11 & 12 = 8
5. SLL : 32’d12 << 11 = 24576
6. SRL : 32’d12 >> 11 = 0
7. SRA : 32’d12 >> 11 (Arith = 0) = 0
8. SLT : 32’d11 < 32’d12 = 1
9. SLTU : 32’d11 < 32’d12 = 1
10. XOR : 11 XOR 12 = 7

※Signed

04 Type 별 설명 (I-Type)



04 Type 별 설명 (I-Type)



ALU Function Control [Imm]

[31:20]	[19:15]	[14:12]	[11:07]	[06:00]
Imm	RS1	Function3	RD	OPcode
		[0 0 0] . . . [1 1 1]		0 0 1 0 0 1 1

Shift Register (SLLI, SRLI, SRAI)

[31:25]	[24:20]	[19:15]	[14:12]	[11:07]	[06:00]
Function7	Shamt	RS1	Function3	RD	OPcode
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0			[0 0 1] [1 0 1] [1 0 1]		0 0 1 0 0 1 1

```
`ADD:     result = a + b;
`SUB:     result = a - b;                          //func7[5] = "1"
`SLL:     result = a << b;
`SRL:     result = a >> b;
`SRA:     result = $signed(a) >>> b[4:0];          //func7[5] = "1"
`SLT:     result = ($signed(a) < $signed(b)) ? 1 : 0;
`SLTU:    result = (a < b) ? 1 : 0;
`XOR:     result = a ^ b;
`OR:      result = a | b;
`AND:     result = a & b;
default: result = 32'bx;
```

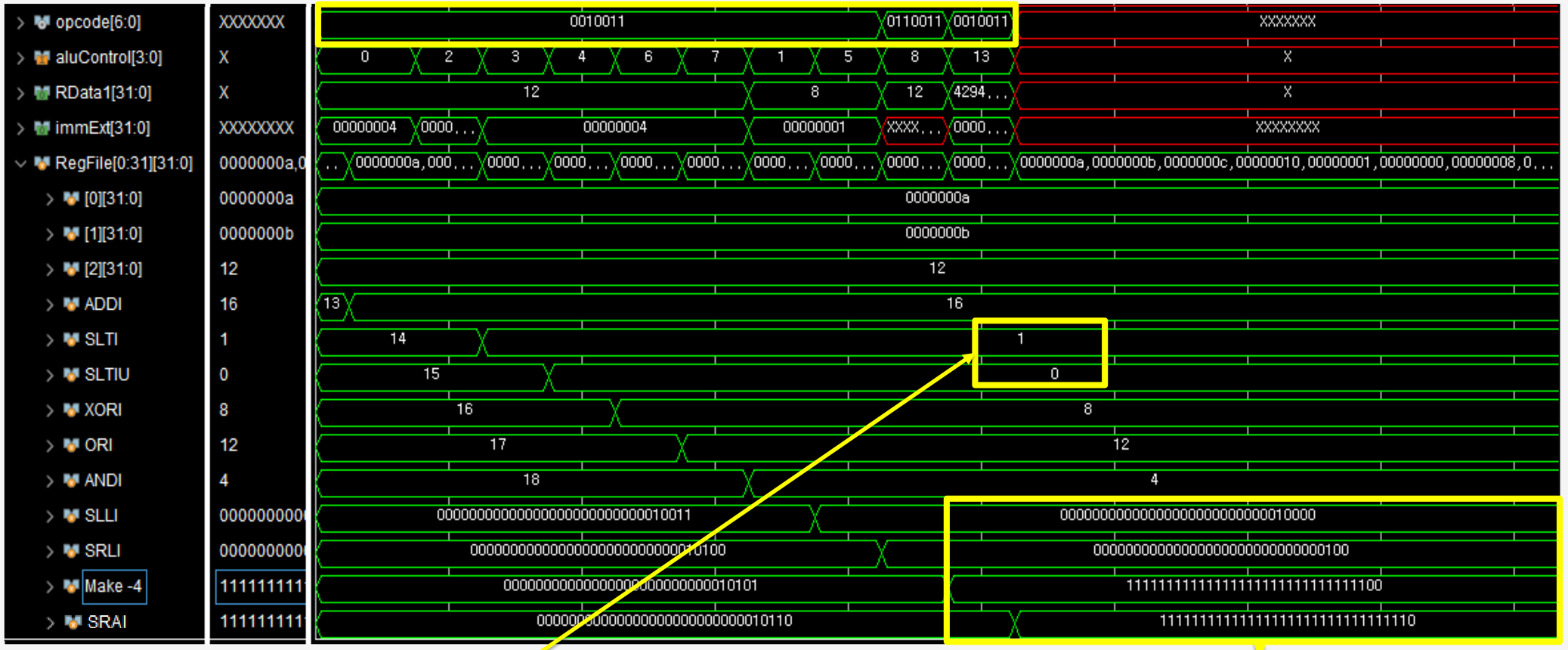
04 I-Type 시뮬레이션 설명

```
//rom[x]=32'b imm12      _ rs1 _ f3 _ rd _opcode   // I-Type
        rom[0] = 32'b000000000100_00010_000_00011_0010011; // ADDI  rd = x3, rs1 = 12, 4(x0)
        rom[1] = 32'b000000010000_00010_010_00100_0010011; // SLTI  rd = x4, rs1 = 12, 16(x0)
        rom[2] = 32'b000000000100_00010_011_00101_0010011; // SLTIU rd = x5, rs1 = 12, 4(x0)
        rom[3] = 32'b000000000100_00010_100_00110_0010011; // XORI  rd = x6, rs1 = 12, 4(x0)
        rom[4] = 32'b000000000100_00010_110_00111_0010011; // ORI   rd = x7, rs1 = 12, 4(x0)
        rom[5] = 32'b000000000100_00010_111_01000_0010011; // ANDI  rd = x8, rs1 = 12, 4(x0)

        //rom[x]=32'b fucn7 _ shamt _ rs1 _f3 _ rd  _opcode; // I-Type SLLI
        rom[6] = 32'b0000000_00001_00110_001_01001_0010011;  // SLLI rx = x9,  rs1 = 16, shamt = 00001
        rom[7] = 32'b0000000_00001_00110_101_01010_0010011;  // SRLI rx = x10, rs1 = 16, shamt = 00001
        rom[8] = 32'b0100000_00011_00010_000_01011_0110011;  // SUB  x11, x1, x3
        rom[9] = 32'b0100000_00001_01011_101_01100_0010011;  // SRAI rx = x12, rs1 = -2, shamt = 00001
```

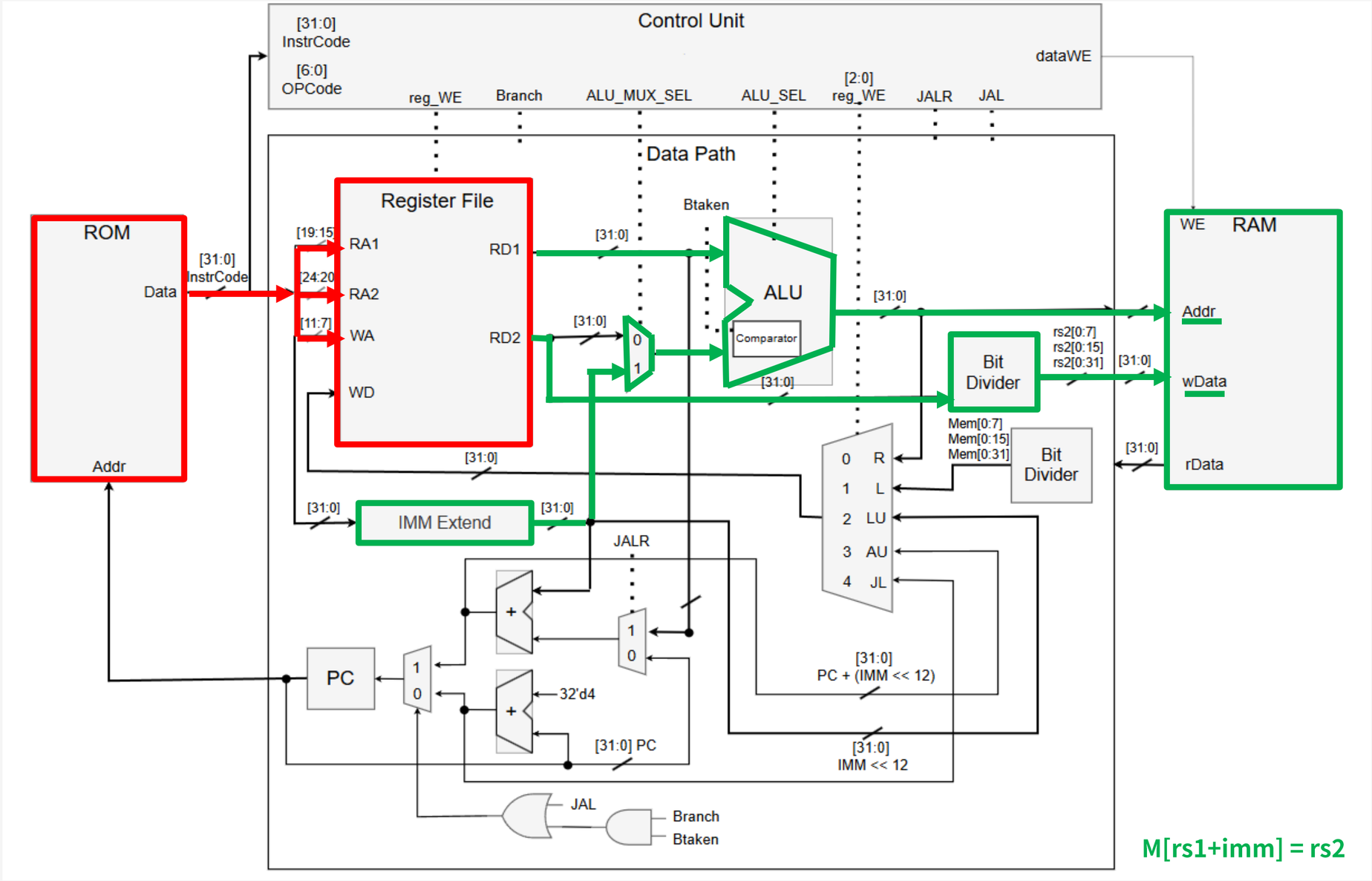
1. ADDI : 12 + 4(imm) = 16
2. SLTI : 12 < 16(imm) = 1
3. SLTIU : 12 < 4(imm) = 0
4. XORI : 12 XOR 4(imm) = 8
5. ORI : 12 OR 4(imm) = 12
6. ANDI : 12 AND 4(imm) = 4
7. SLLI : 16 << 32'd1 (shamt)
8. SRLI : 16 >> 32'd1 (shamt)
9. SRAI : 음수 Shift

04 I-Type 시뮬레이션 설명

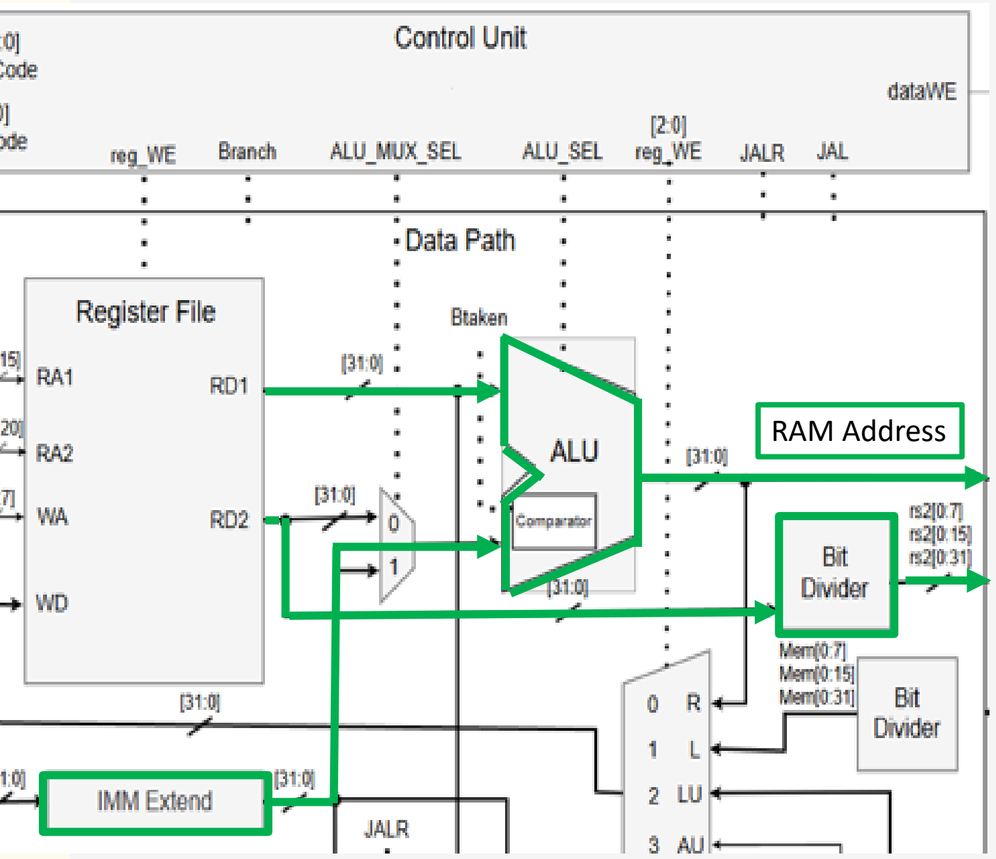


1. ADDI : 12 + 4(imm) = 16
2. SLTI : 12 < 16(imm) = 1
3. SLTIU : 12 < 4(imm) = 0
4. XORI : 12 XOR 4(imm) = 8
5. ORI : 12 OR 4(imm) = 12
6. ANDI : 12 AND 4(imm) = 4
7. SLLI : 16 << 32'd1 (shamt)
8. SRLI : 16 >> 32'd1 (shamt)
9. SRAI : 음수 Shift

04 Type 별 설명 (S-Type)



04 Type 별 설명 (S-Type)



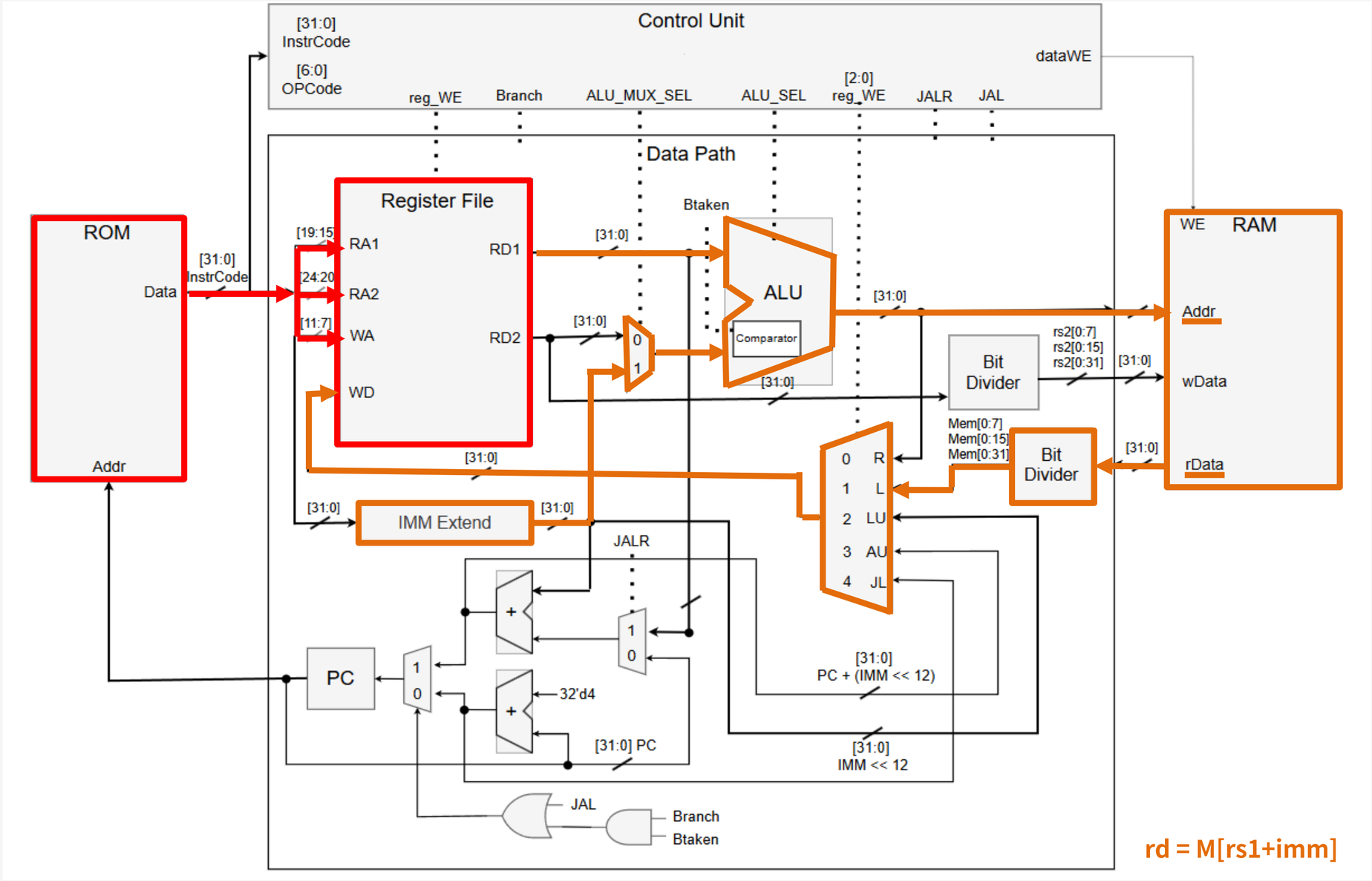
Store Data into RAM

[31:25]	[24:20]	[19:15]	[14:12]	[11:07]	[06:00]
Imm[11:5]	RS2	RS1	Function3	Imm[4:1][11]	OPcode
Byte Half Whole			[0 0 0] [0 0 1] [0 1 0]		0 1 0 0 0 1 1

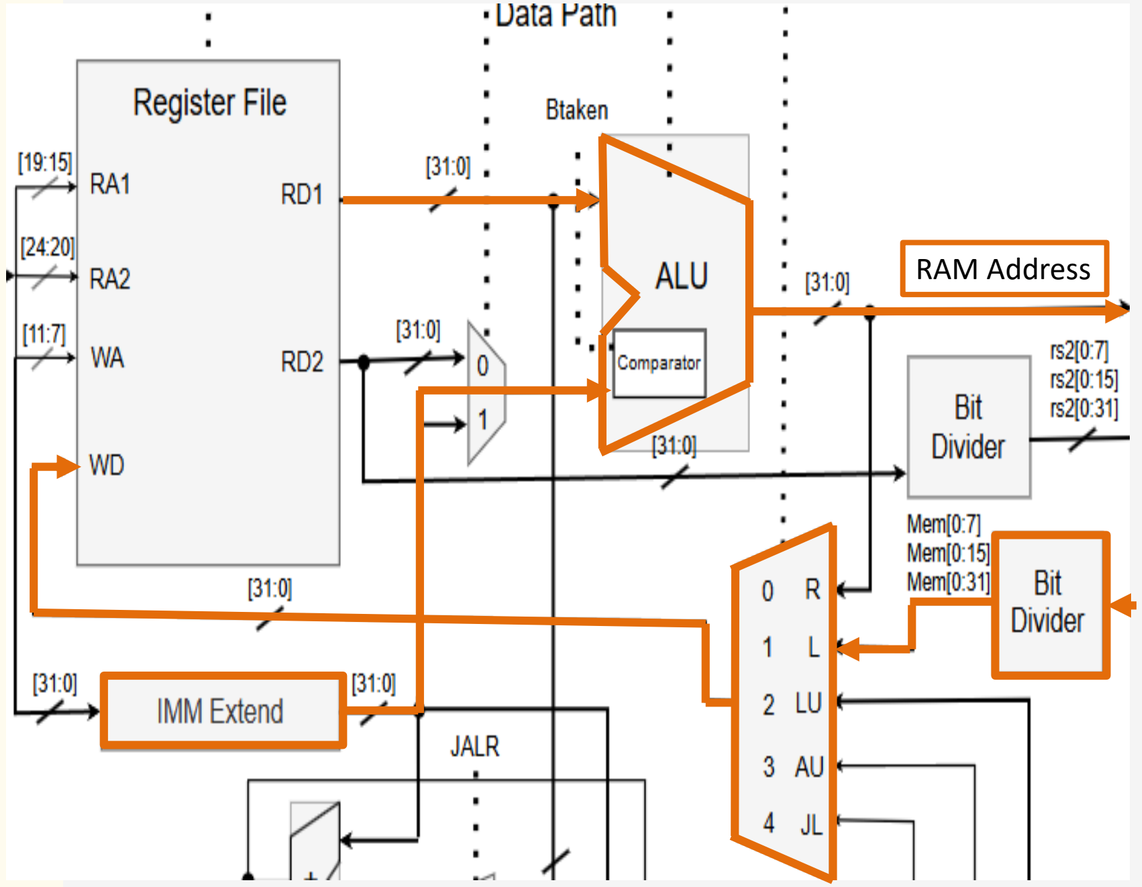
DATA Bit Divide

Byte	Addr : rs1 + Imm	Data : wData[0:7]
Half	Addr : rs1 + Imm	Data : wData[0:15]
Whole	Addr : rs1 + Imm	Data : wData[0:31]

04 Type 별 설명 (L-Type)



04 Type 별 설명 (L-Type)



Read Data From RAM

[31:20]	[19:15]	[14:12]	[11:07]	[06:00]
Imm[11:0	RS1	Function3	RD	OPcode
		[0 0 0] . . . [1 1 1]		0 0 1 0 0 1 1

Read Divided Data

Byte	Data : rData[0:7]
Half	Data : rData[0:15]
Whole	Data : rData[0:31]

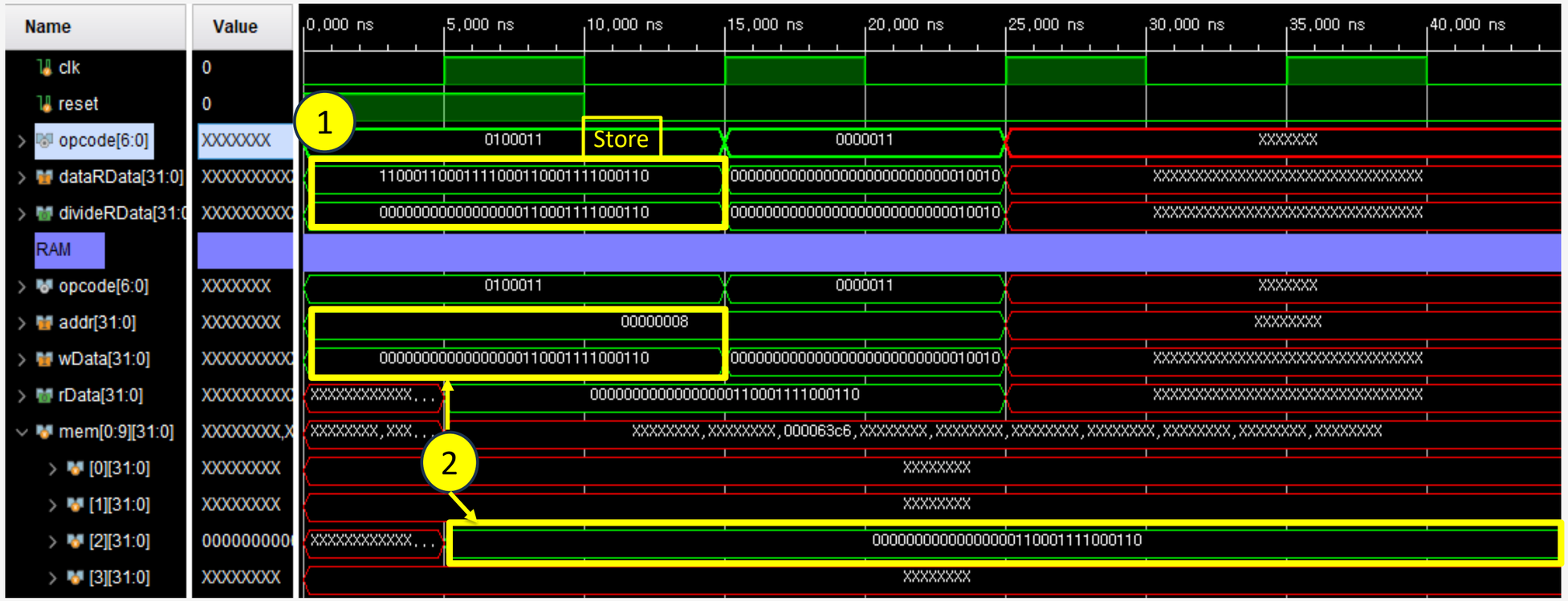
04 S & L-Type 시뮬레이션 설명

```
//rom[x]=32'b imm7   _ rs2 _ rs1 _f3 _ imm5_ opcode;  // S-Type
 rom[0] = 32'b0000000_11111_00000_001_01000_0100011;  // sw x2, 8(x0), divide into 15bit & put RAM
 //rom[x]=32'b imm12       _ rs1 _f3 _ rd  _ opcode;   // L-Type
 rom[1] = 32'b000000001000_00000_000_00011_0000011;   // LB x3, 8(x0); divide into 7bit & put REG
```

※ rs2 = 32’b1100_0110_0011_1100_0110_0011_1100_0110

1. Store rs2[0:15] to RAM Address [2]
2. Load rs2[0:7] Arith(1) to RegFile Address[3]

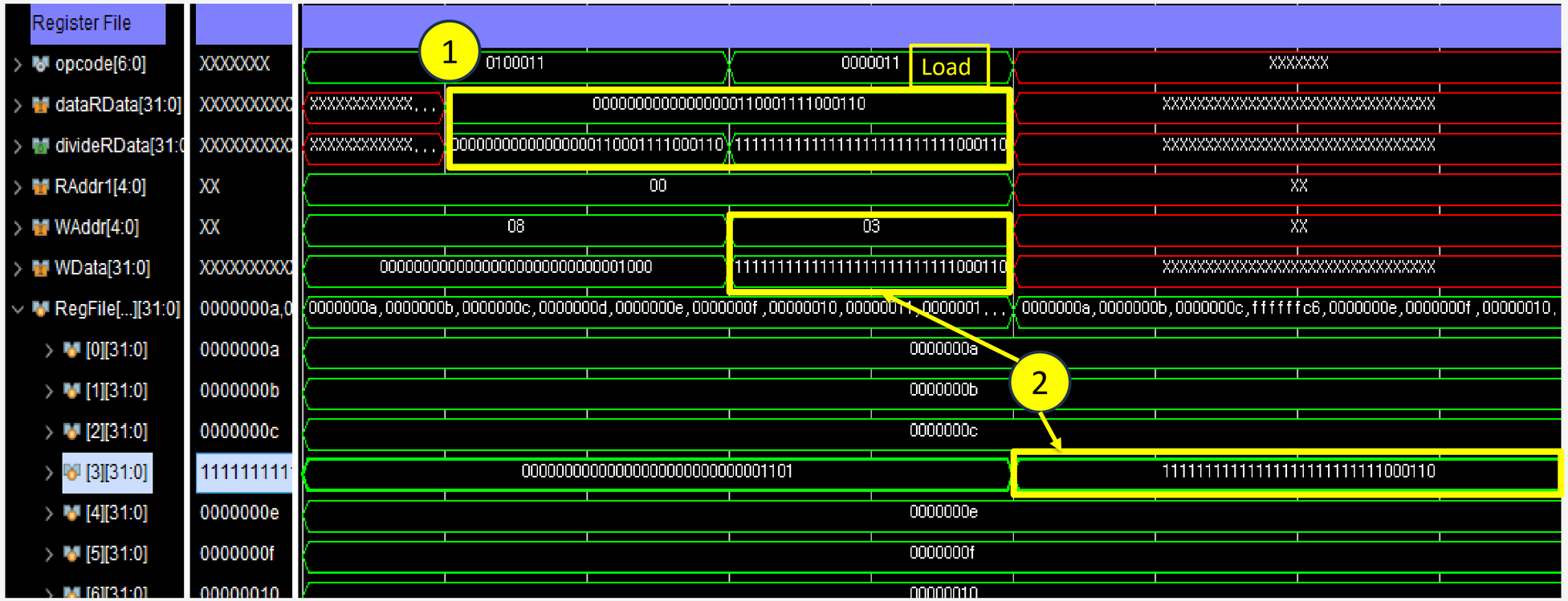
04 S & L-Type 시뮬레이션 설명



※ rs2 = 32’b1100_0110_0011_1100_0110_0011_1100_0110

1. Divide 32bit rs2 -> rs2[0:15]
2. Store rs2[0:15] to RAM Address [2]

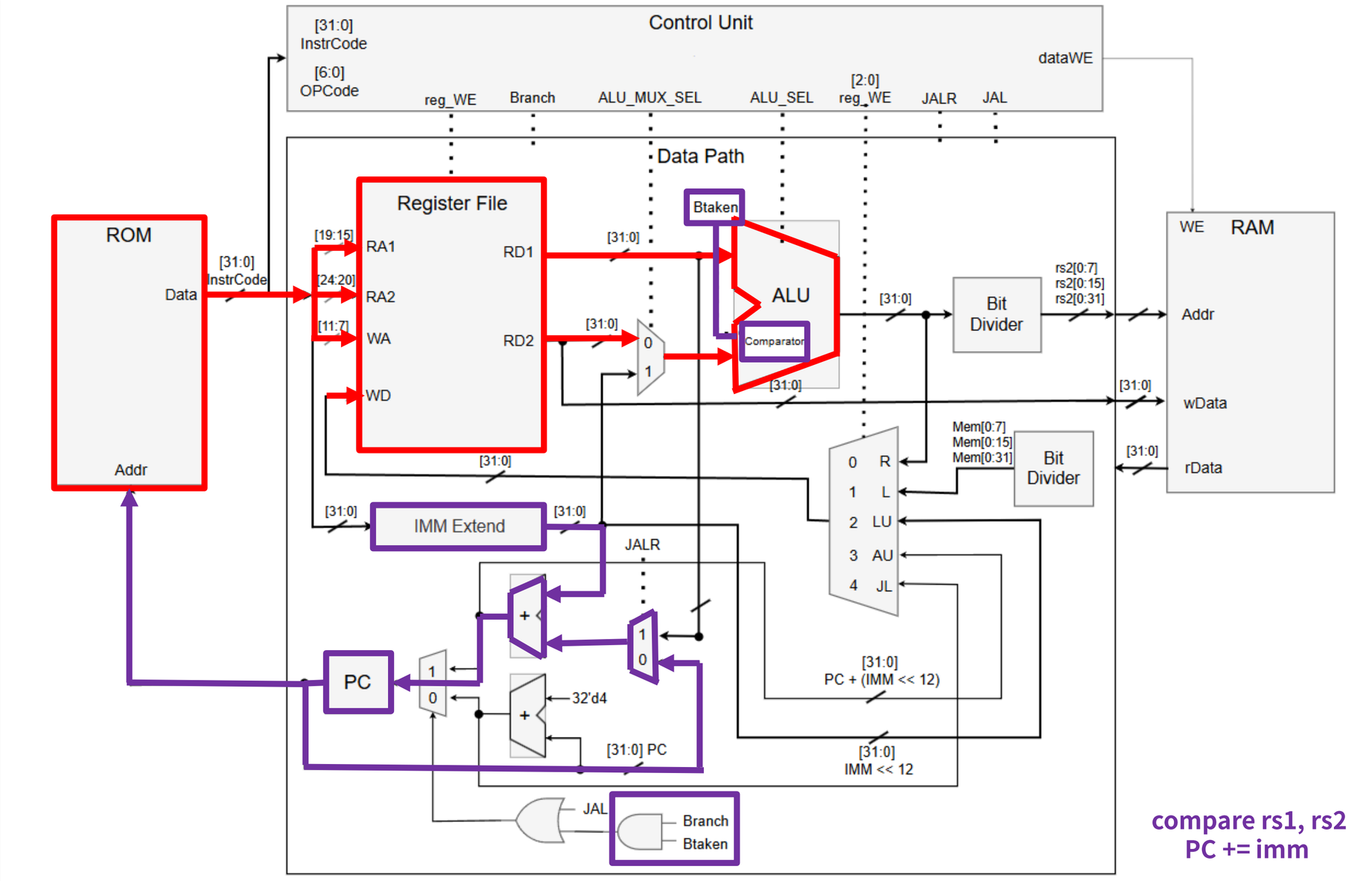
04 S & L-Type 시뮬레이션 설명



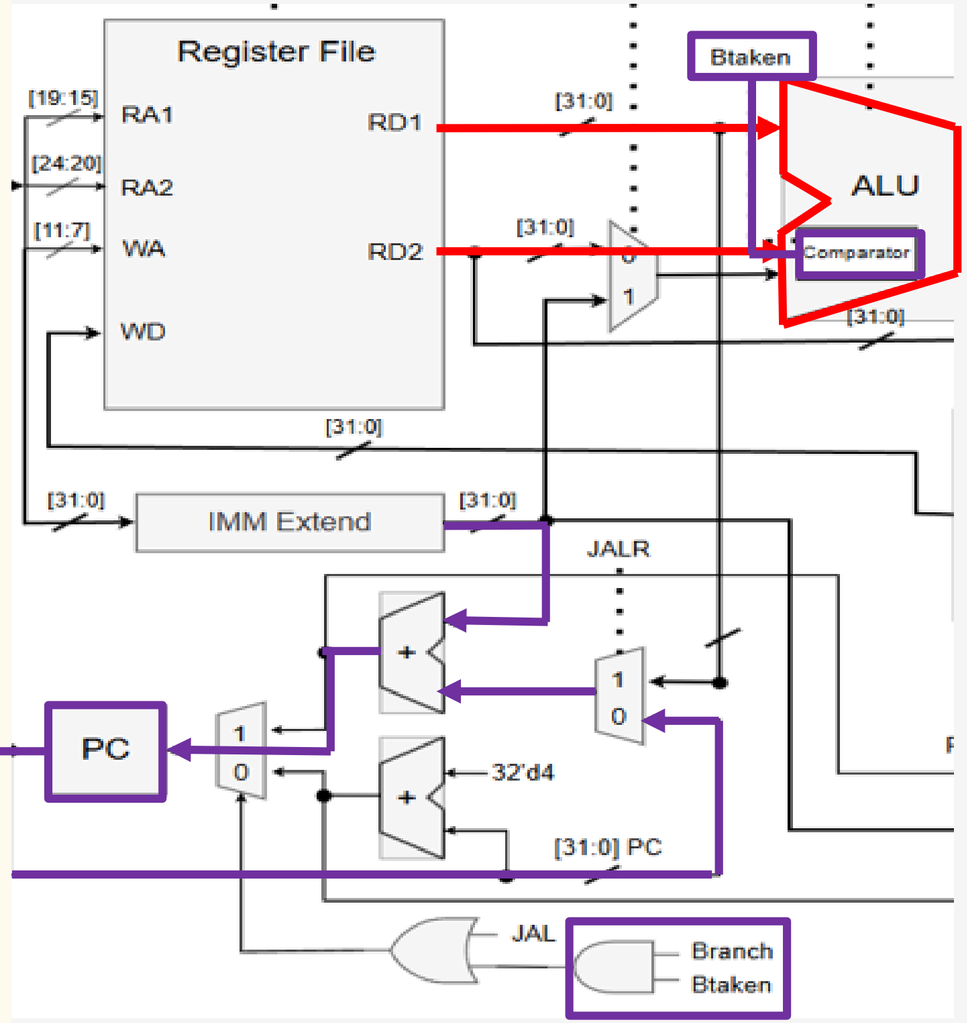
※ rs2 = 32’b1100_0110_0011_1100_0110_0011_1100_0110

1. (LB) Divide 15bit rs2 -> rs2[0:7] + Airth(1)
2. Store rs2[0:7] to RegFile Address [3]

04 Type 별 설명 (B-Type)



04 Type 별 설명 (B-Type)



Compare Data & Make Branch (PC += Imm)

[31:25]	[24:20]	[19:15]	[14:12]	[11:07]	[06:00]
Imm[12][10:5]	RS2	RS1	Function3	Imm[4:1][11]	OPcode
			[0 0 0] . . . [1 1 1]		1 1 0 0 0 1 1

```
`BEQ:    btaken = (a == b);
`BNE:    btaken = (a != b);
`BLT:    btaken = ($signed(a) < $signed(b));
`BGE:    btaken = ($signed(a) >= $signed(b));
`BLTU:   btaken = (a < b);
`BGEU:   btaken = (a >= b);
default: btaken = 1'b0;
```

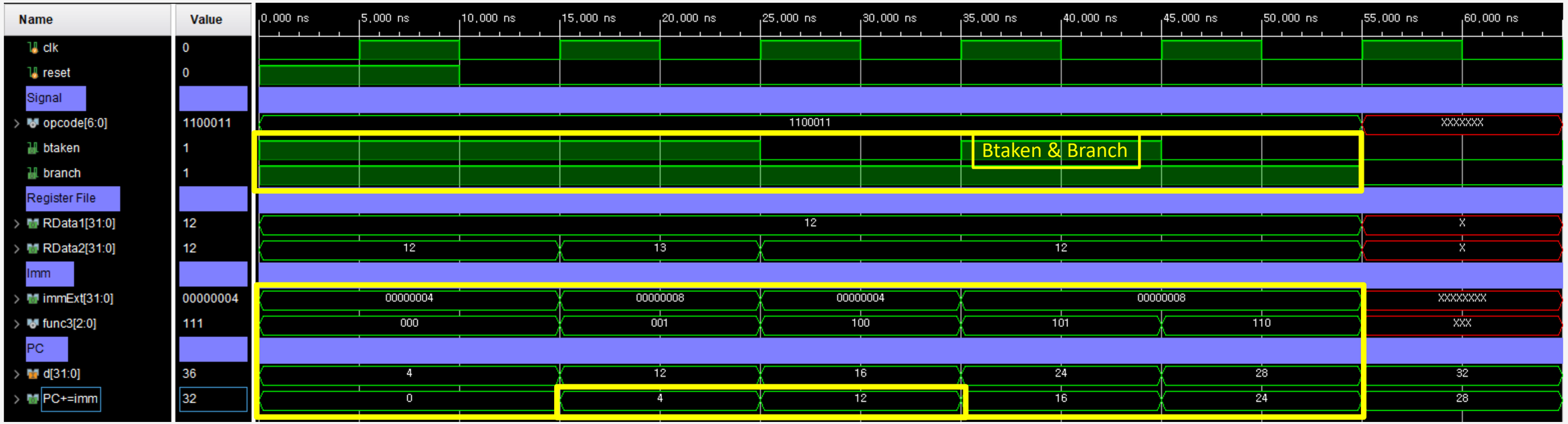
04 B-Type 시뮬레이션 설명

```
//rom[x]=32'b imm7  _ rs2 _ rs1 _f3 _ imm5_ opcode; // B-Type
rom[0] = 32'b0000000_00010_00010_000_00100_1100011; // BEQ  x2, x2, 4 True
rom[1] = 32'b0000000_00011_00010_001_01000_1100011; // BNE  x2, x3, 8 True
rom[3] = 32'b0000000_00010_00010_100_00100_1100011; // BLT  x2, x2, 4 False
rom[4] = 32'b0000000_00010_00010_101_01000_1100011; // BGE  x2, x2, 8 True
rom[6] = 32'b0000000_00011_00010_110_00100_1100011; // BLTU x2, x3, 4 True
rom[7] = 32'b0000000_00010_00010_111_00100_1100011; // BGEU x2, x2, 4 True
```

비교 연산 값이 1(True)라면 PC += imm

1. BEQ : (rs1 == rs2) = ‘1’ -> 4
2. BNE : (rs1 != rs2) = ‘0’ -> 8
3. BLT : (rs1 < rs2) = ‘0’ -> 4
4. BGE : (rs1 >= rs2) = ‘1’ -> 8
5. BLTU : (rs1 < rs2) = ‘0’ -> 4
6. BGEU : (rs1 >= rs2) = ‘1’ -> 4

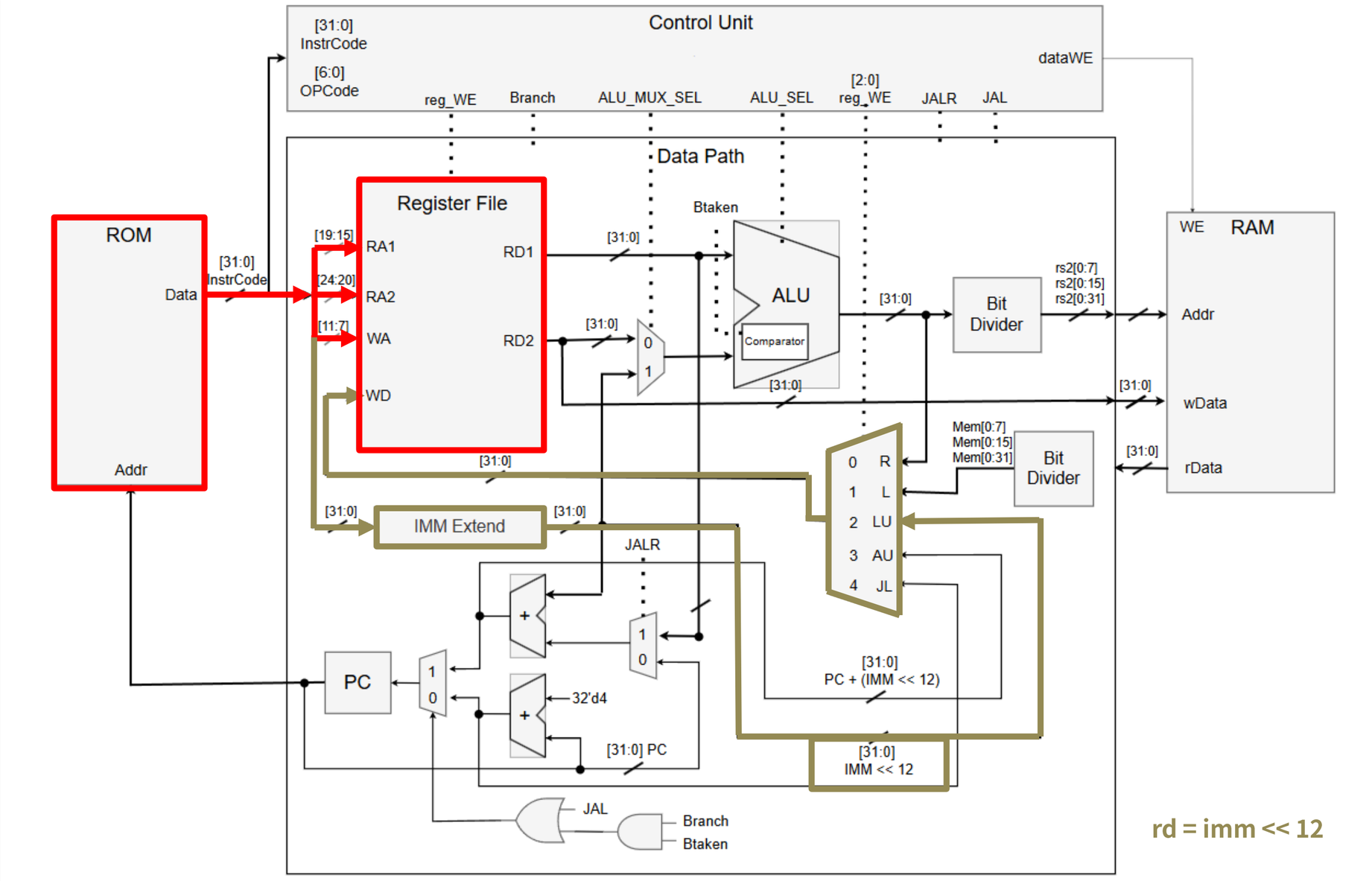
04 B-Type 시뮬레이션 설명



비교 연산 값이 1(True)라면 PC += imm

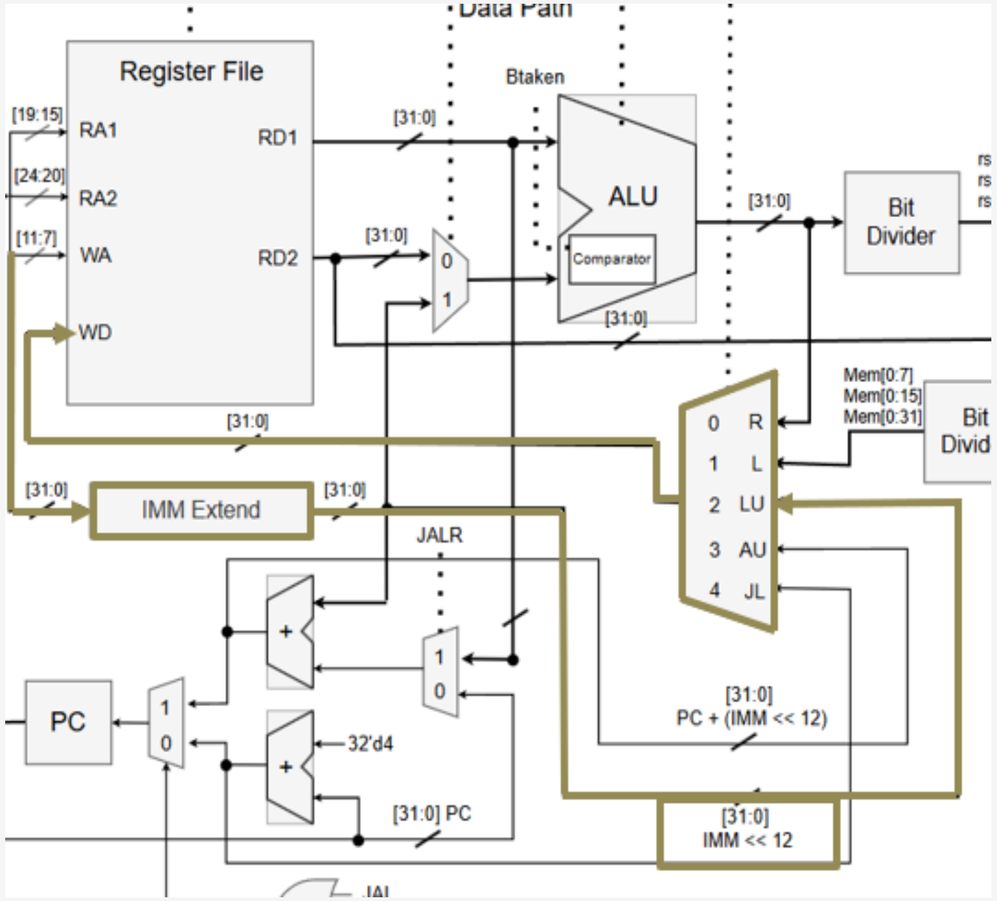
1. BEQ : (rs1 == rs2) = '1' -> 4
2. BNE : (rs1 != rs2) = '1' -> 8
3. BLT : (rs1 < rs2) = '0' -> 4
4. BGE : (rs1 >= rs2) = '1' -> 8
5. BLTU : (rs1 < rs2) = '0' -> 4
6. BGEU : (rs1 >= rs2) = '1' -> 4

04 Type 별 설명 (LU-Type)



rd = imm << 12

04 Type 별 설명 (LU-Type)

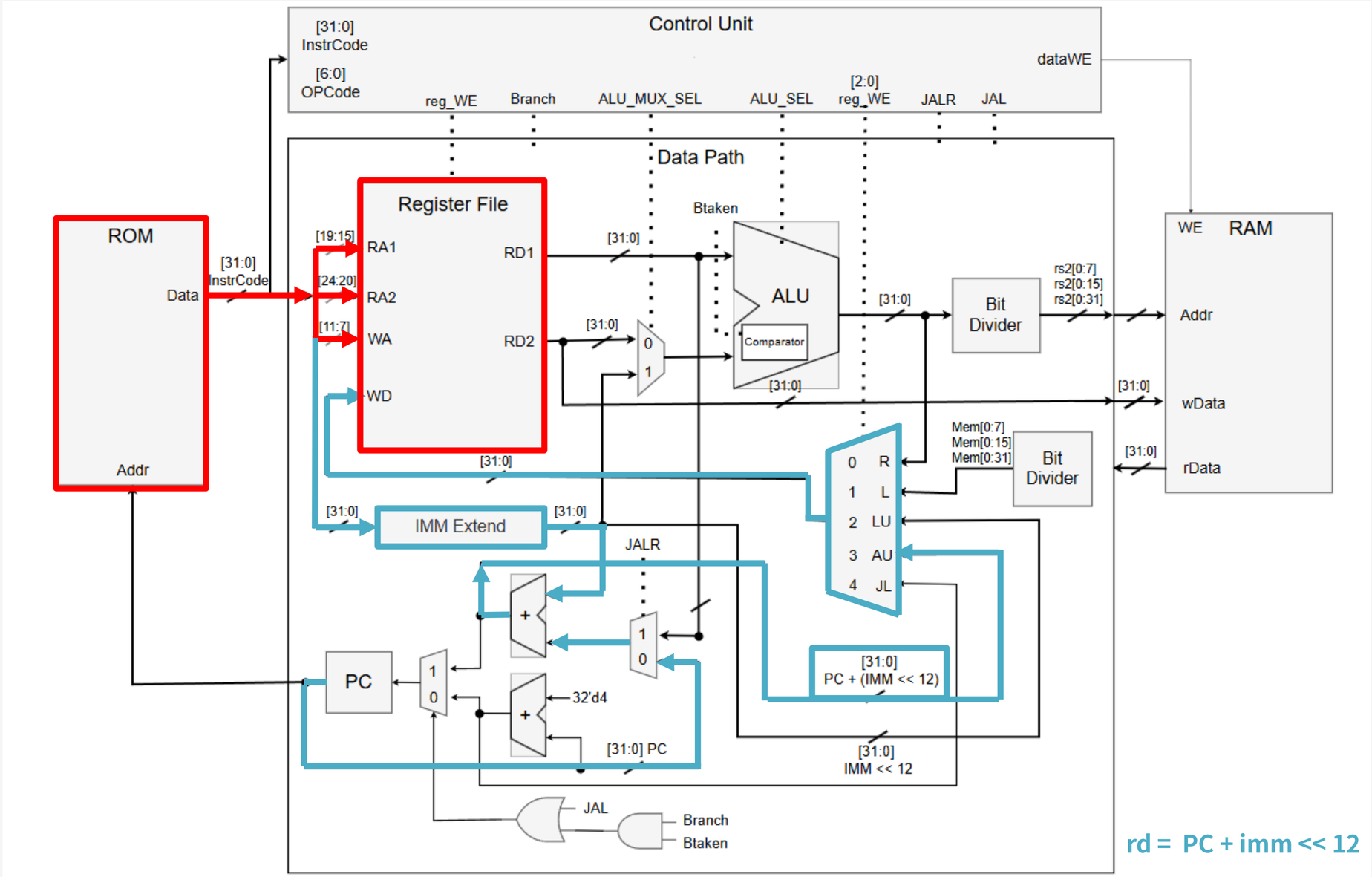


Address Shift / Jump (imm << 12)

Register 직접 저장

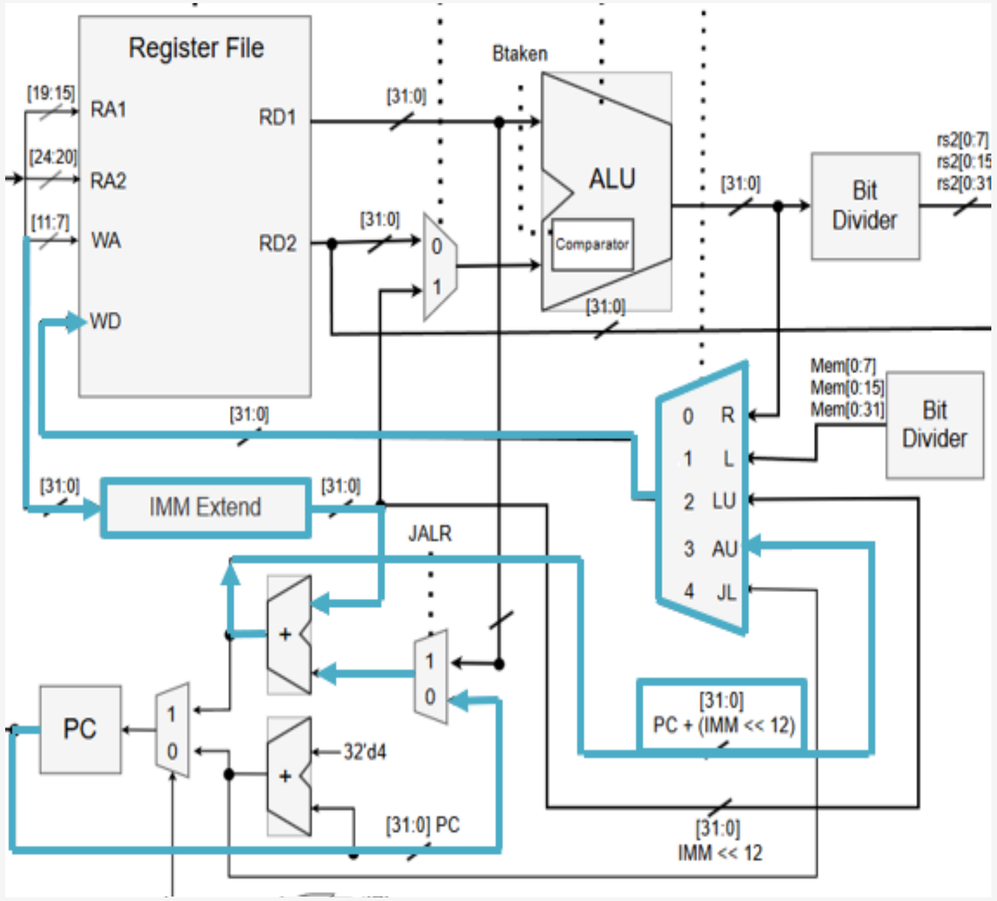
[31:12]	[11:07]	[06:00]
Imm[31:12]	RD	OPcode
		0 1 1 0 1 1 1

04 Type 별 설명 (AU-Type)



rd = PC + imm << 12

04 Type 별 설명 (AU-Type)



Address Shift / Jump PC + (imm << 12)

PC에 연산 결과 더함

[31:12]	[11:07]	[06:00]
Imm[31:12]	RD	OPcode
		0 0 1 0 1 1 1

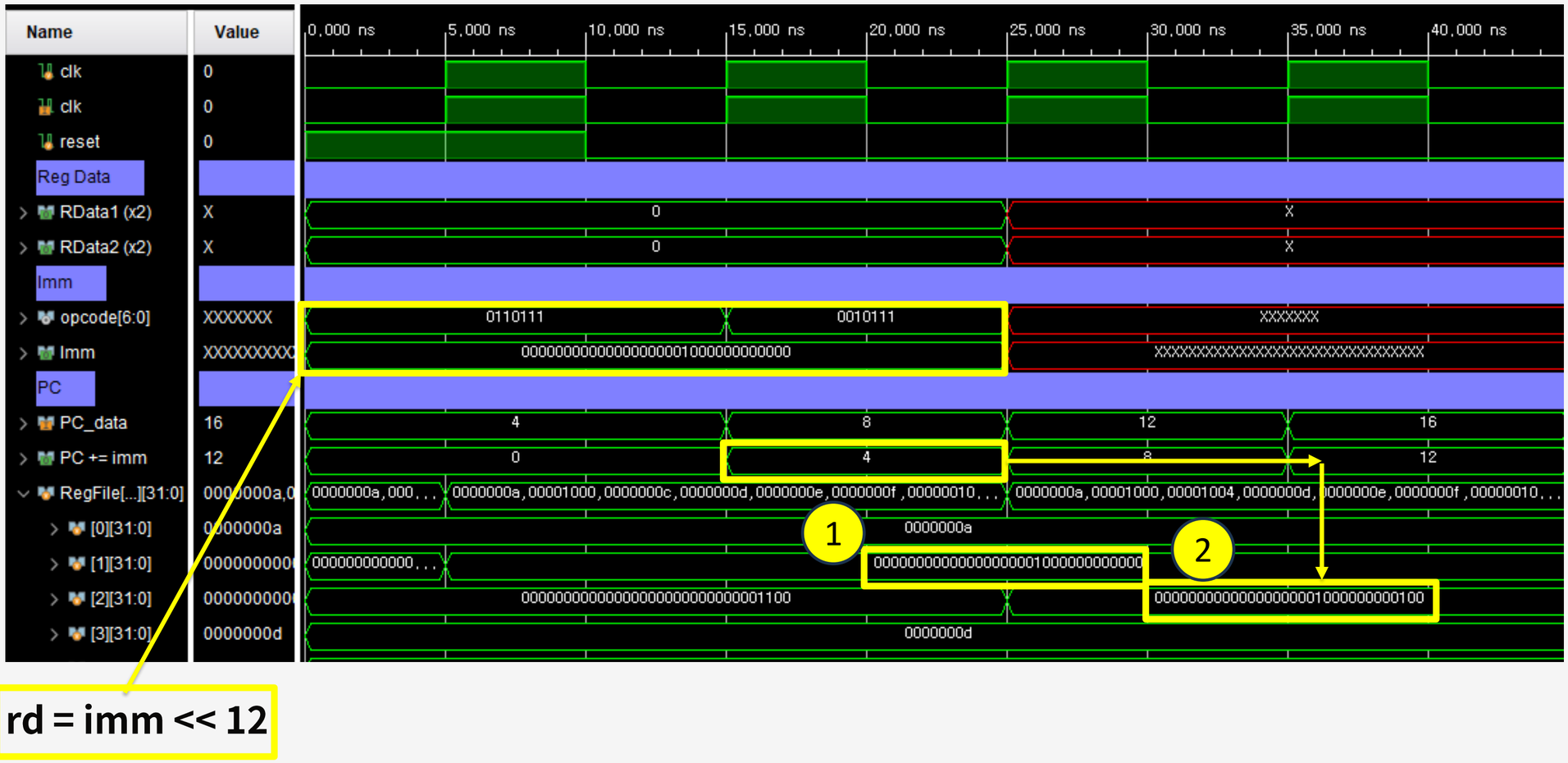
04 LU & AU-Type 시뮬레이션 설명

```
//rom[x]=32'b imm20                _ rd  _ opcode; // LU-Type
rom[0] = 32'b00000000000000000001_00001_0110111; // LUI   x1, imm1
//rom[x]=32'b imm20                _ rd  _ opcode; // AU-Type
rom[1] = 32'b00000000000000000001_00010_0010111; // AUIPC x2, imm1
```

rd = imm << 12

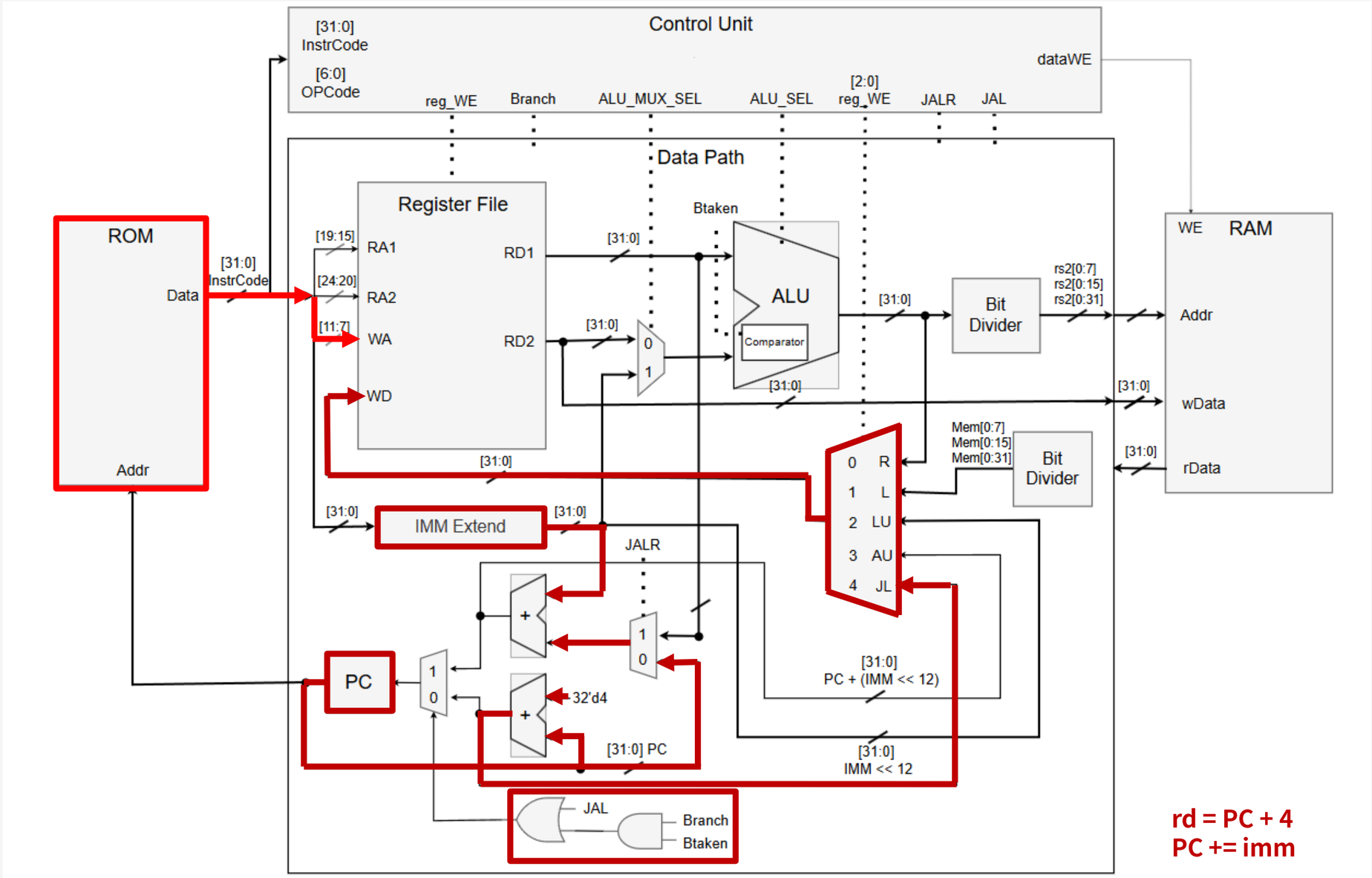
1. **LUI : rd = imm(1) << 12**
2. **AUIPC : rd = PC + imm(1) << 12**

04 LU & AU-Type 시뮬레이션 설명

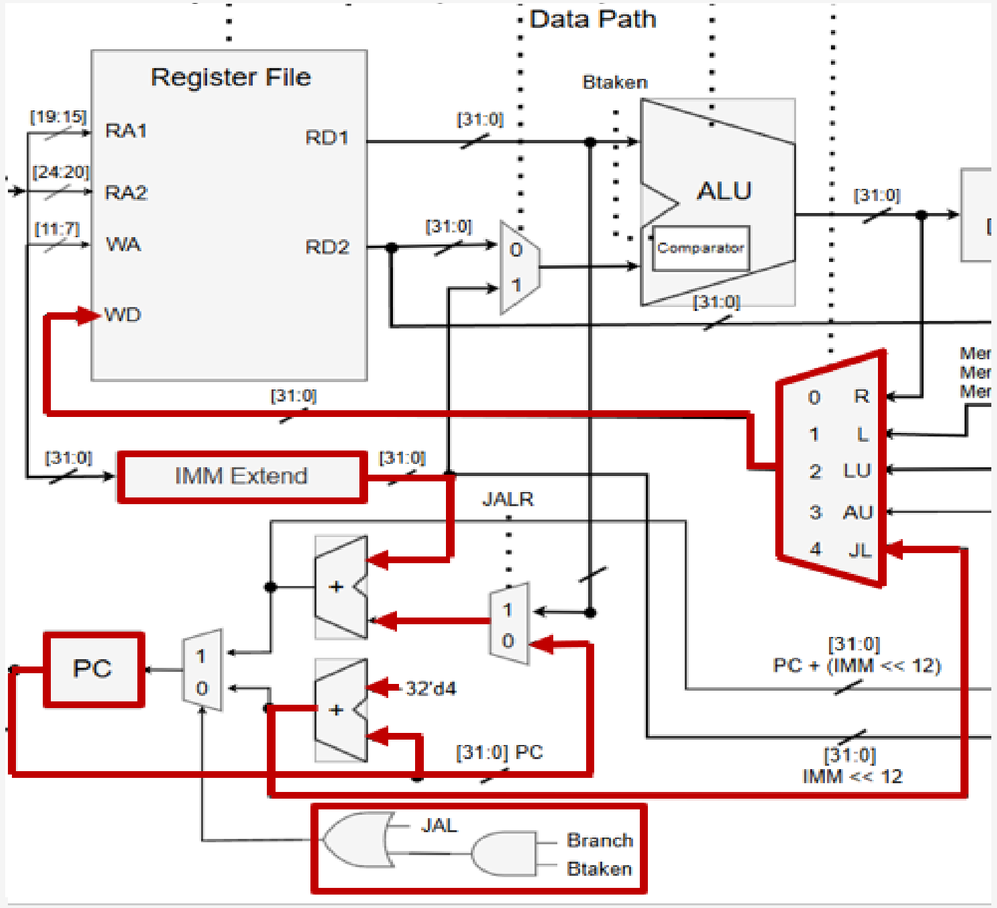


1. LUI : rd = imm(1) << 12
2. AUIPC : rd = PC + imm(1) << 12

04 Type 별 설명 (J-Type)



04 Type 별 설명 (J-Type)



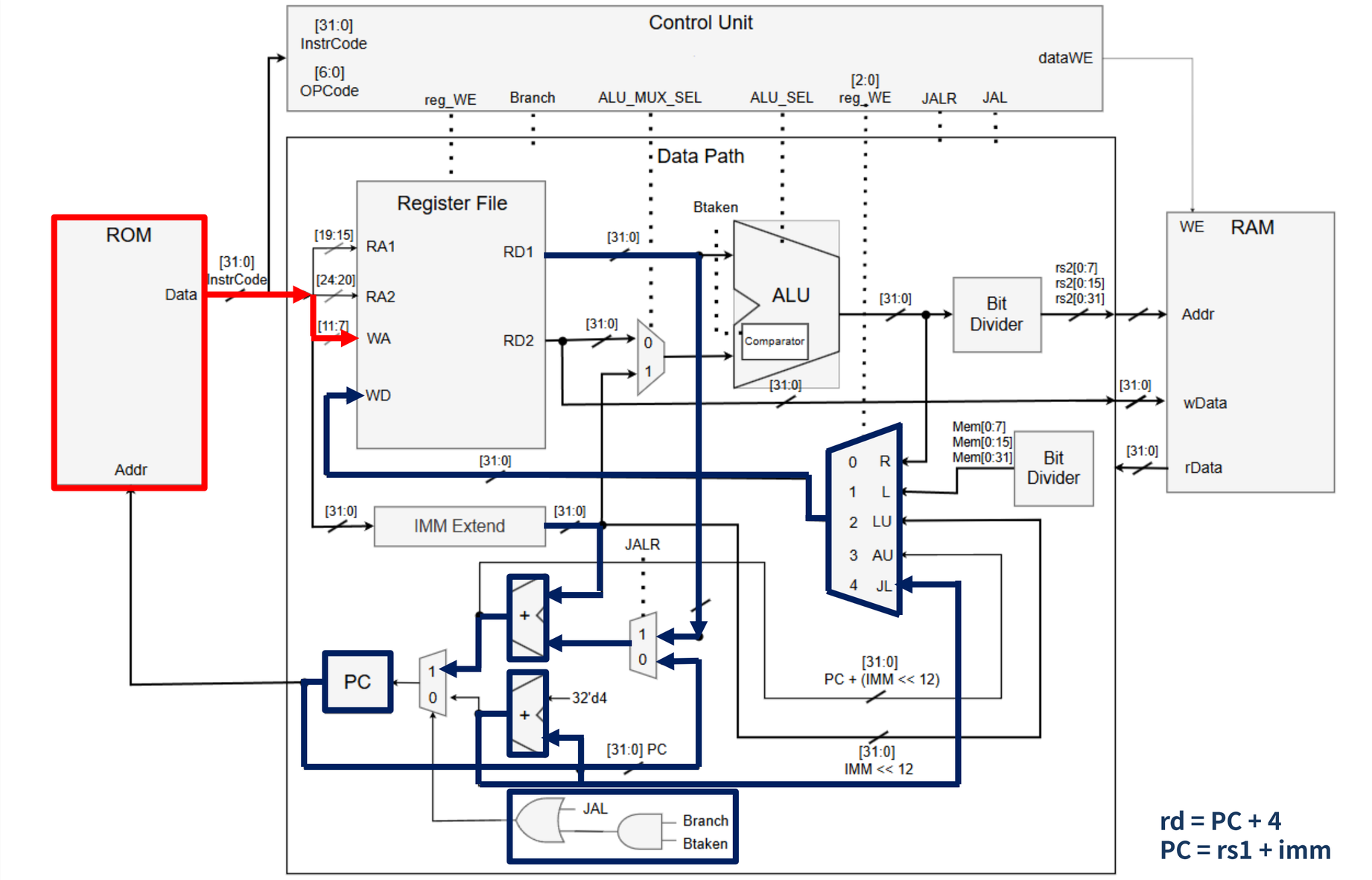
Address Shift / Jump + Read (인터럽트에 활용)

RD = PC +4 ;

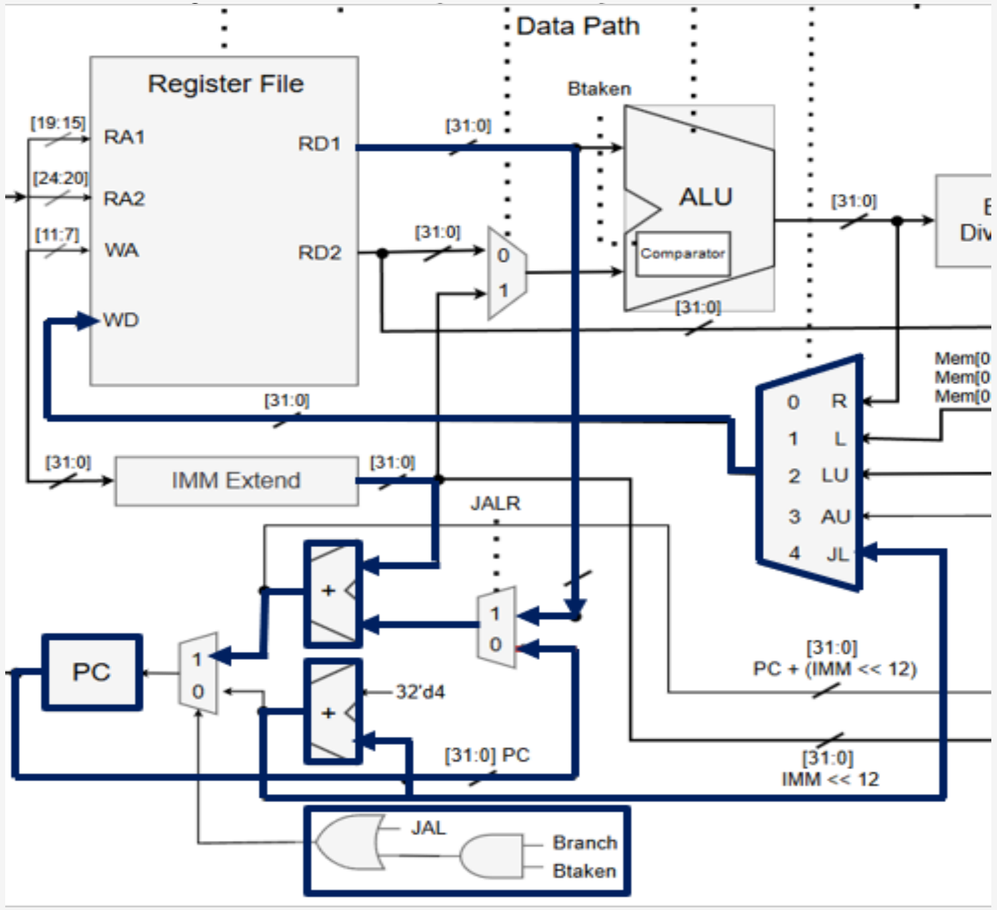
PC += imm ;

[31:12]	[11:07]	[06:00]
Imm[20:1]	RD	OPcode
Imm[20][10:1][11][19:12]		1 1 0 1 1 1 1

04 Type 별 설명 (JL-Type)



04 Type 별 설명 (JL-Type)



Address Shift / Jump + Read (인터럽트에 활용)

RD = PC +4 ;

PC = RS1 + imm ;

[31:20]	[19:15]	[14:12]	[11:07]	[06:00]
Imm[11:0]	RS1	Function3	RD	OPcode
		[0 0 0]		1 1 0 0 1 1 1

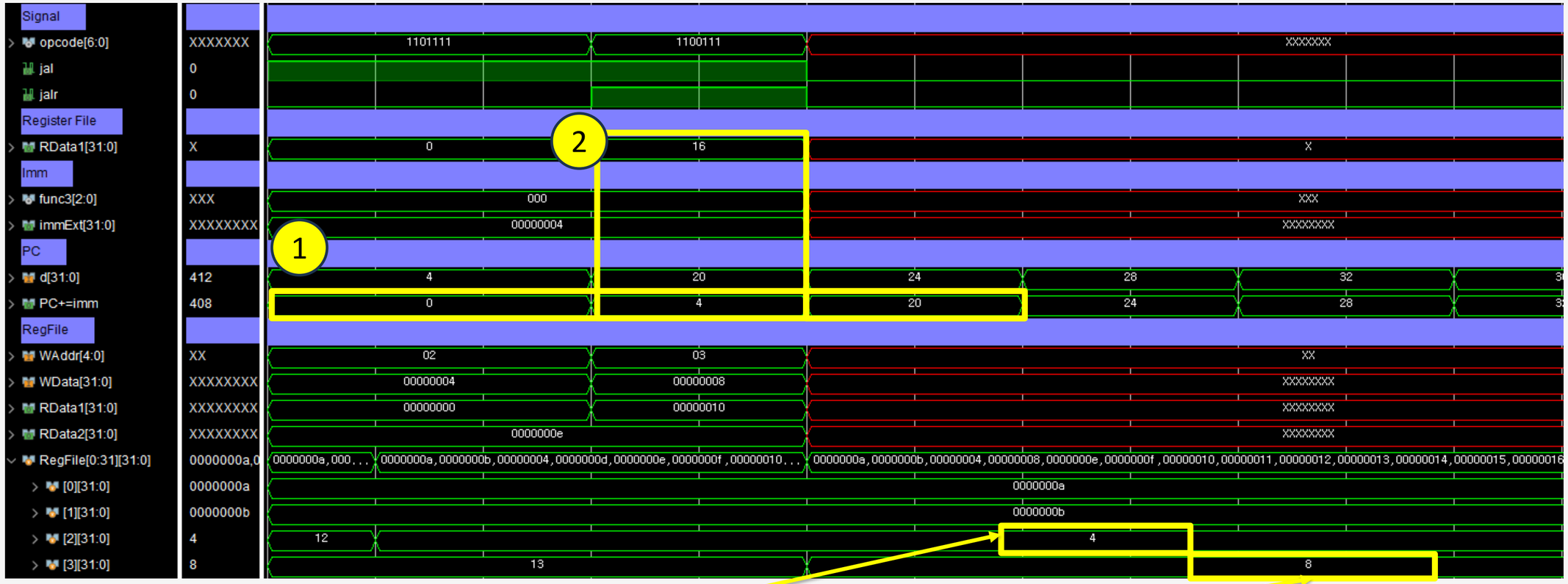
04 J & JL-Type 시뮬레이션 설명

```
//rom[x]=32'b imm20              _ rd  _ opcode;    // J-Type
rom[0] = 32'b00000000010000000000_00010_1101111;    // JAL rd = x2 , imm4
//rom[x]=32'b imm12       _ rs1 _000_ rd _ opcode;  // JL-Type
rom[1] = 32'b000000000100_00110_000_00011_1100111;  // JALR rd = 3 rs1 = x6, imm4
```

rd = PC + 4; PC += imm

1. **JAL : rd = PC + 4 ; PC += imm;**
2. **JALR : rd = PC + 4 ; PC = rs1 + imm;**

04 J & JL-Type 시뮬레이션 설명

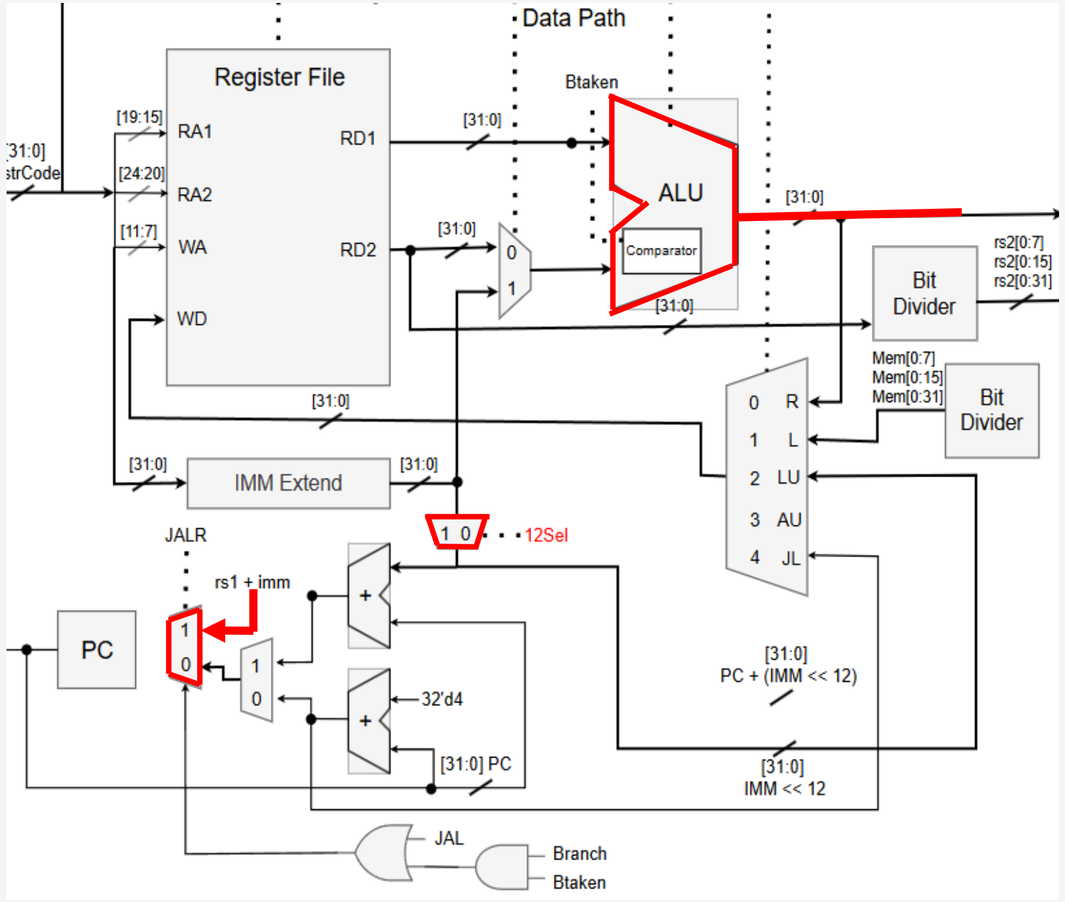


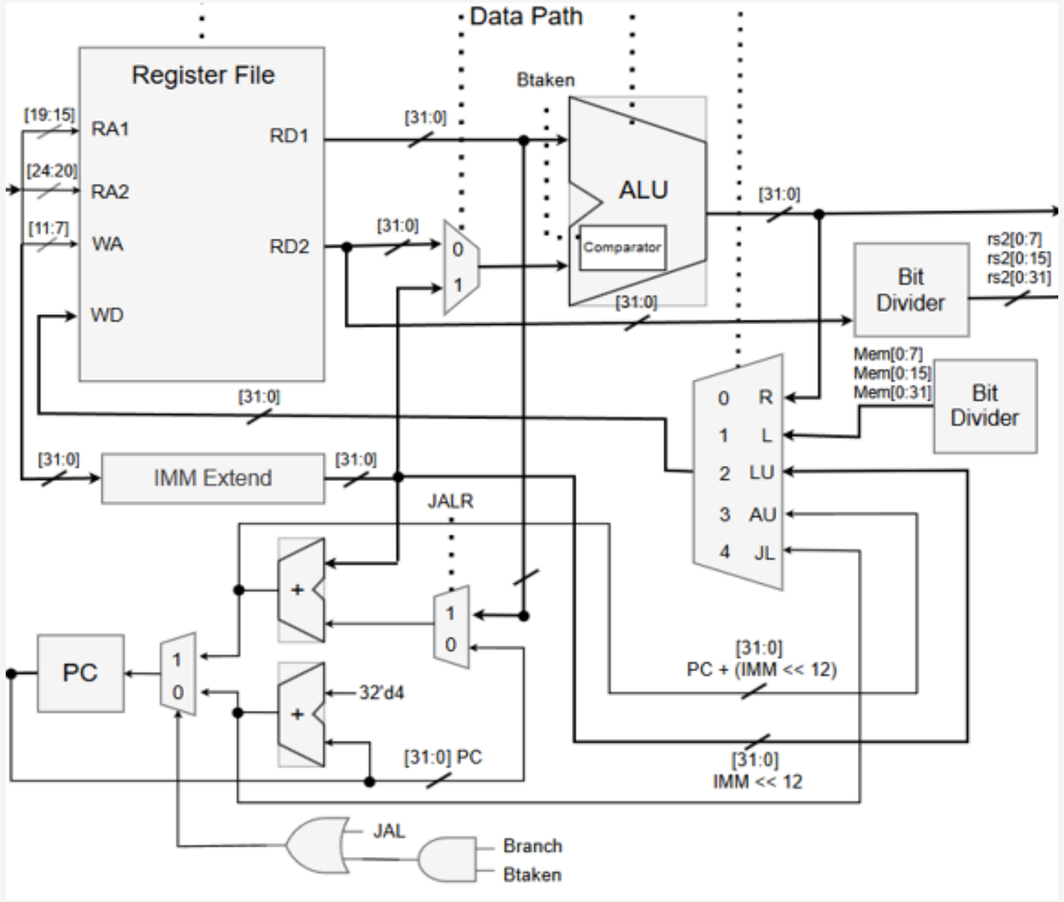
rd = PC + 4; PC += imm

1. **JAL : rd = PC + 4 ; PC += imm;**
2. **JALR : rd = PC + 4 ; PC = rs1 + imm;**

05 고찰

다양한 방법을 통한 최적화된 설계

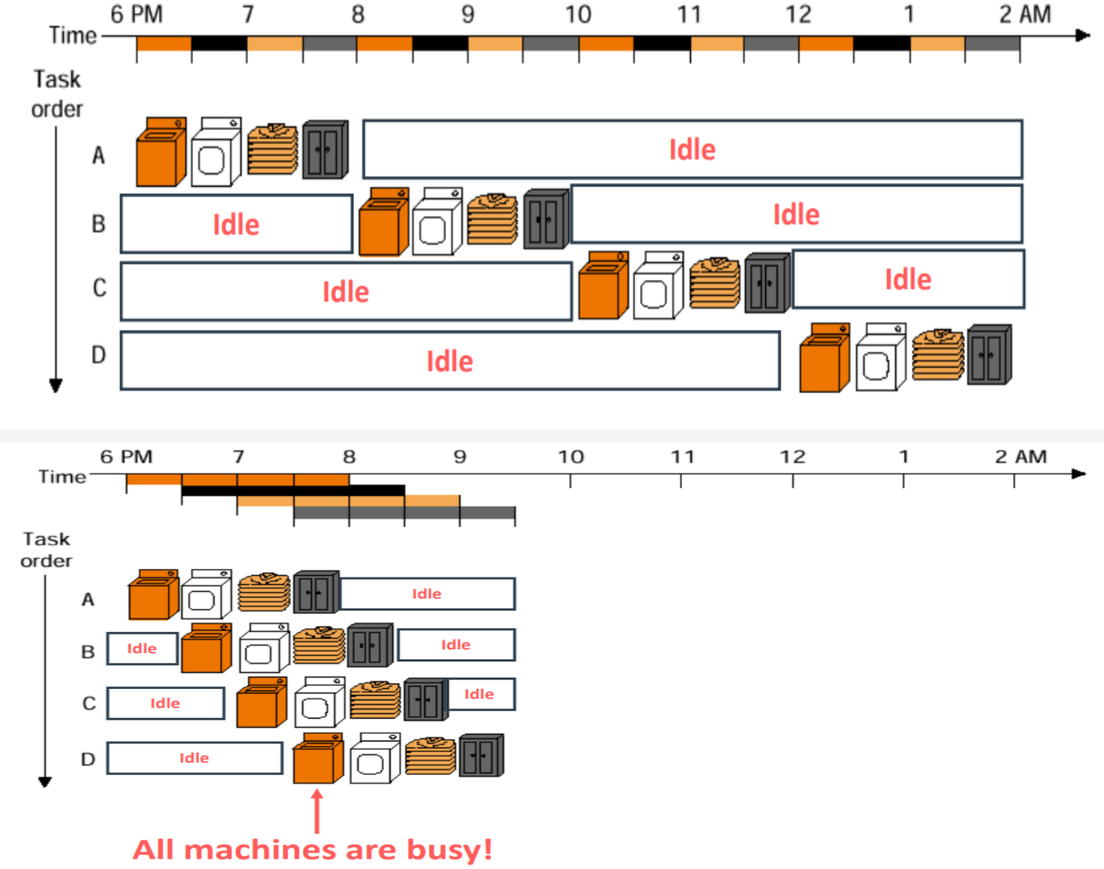




ALU 연산 필요 X, 추가된 MUX 및 제어신호 제거 → MCU 최적화

05 고찰

Single-Cycle에 대한 이해와 기능 확장의 기대



Multi-Cycle / Pipelining으로 업그레이드

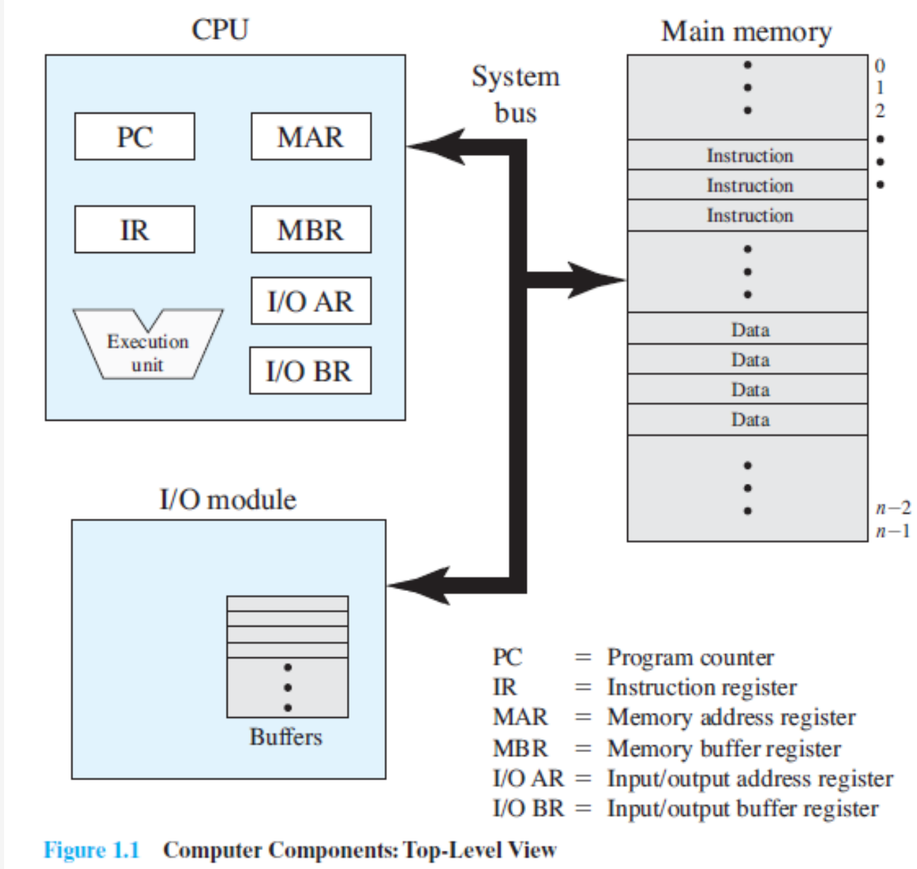


Figure 1.1 Computer Components: Top-Level View

Data Bus 확장

THANK YOU