AI 시스템반도체설계 2기
RISC-V CPU

2025-08-25

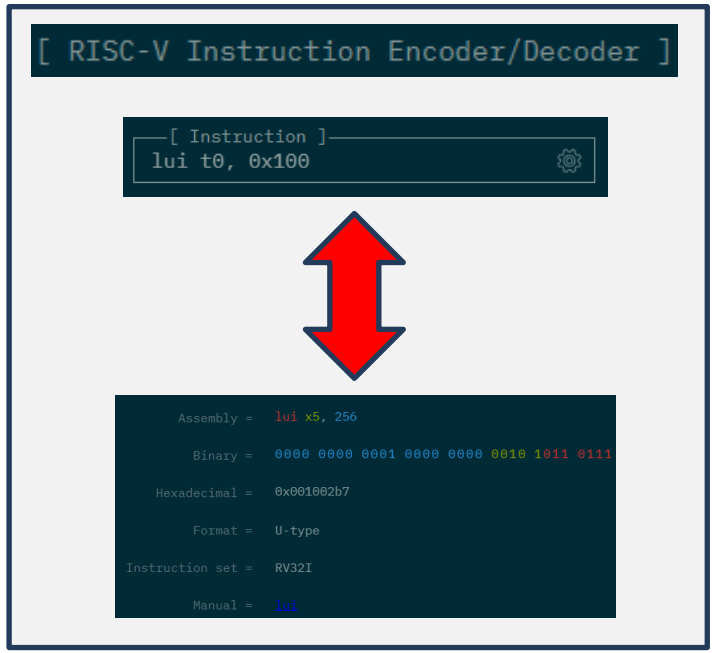
JONG WAN KO (JONGWANKO@GMAIL.COM)

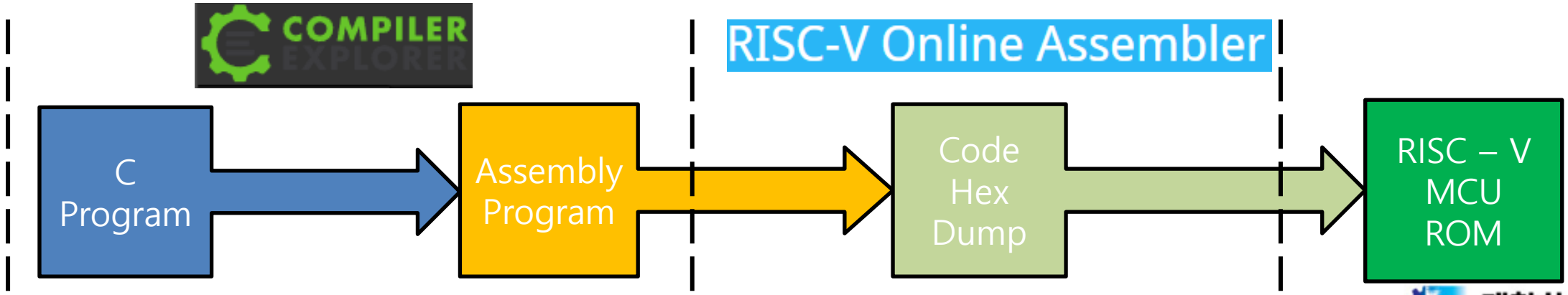
Contents

- Development Environment
- RISC – V Introduction
- Data Path (R, L, I, S, B, LU, AU, J, JL)
- Type Timing (R, L, I, S, B, LU, AU, J, JL)
- Test Program : Bubble Sort
- Conclusion
- Trouble Shooting
- Insights after finishing the project

Development Environment

- **HDL :** System Verilog
  - Utilized for the structural and behavioral design of the RISC-V core.
- **Software** : RV32I Assembly , C
  - Employed to implement test programs and validate the functionality of the RISC-V core.
- **EDA :** AMD Xilinx Vivado 2020.2
  - Adopted for use of RTL simulation.
- **IDE:** Visual Studio Code
- **Supporting Tools** : Online RISC-V Assembler, RVCodec.js, Compiler Explorer (Godbolt)





RISC – V Introduction

	Content
Designer	University of California, Berkeley
Bits	32, 64, 128
Introduced	6 August 2014 (11 years ago)
Design	RISC
Endianness	Little
Extensions	M, A, F, D, Q, etc.

➢ Details of RISC – V Table

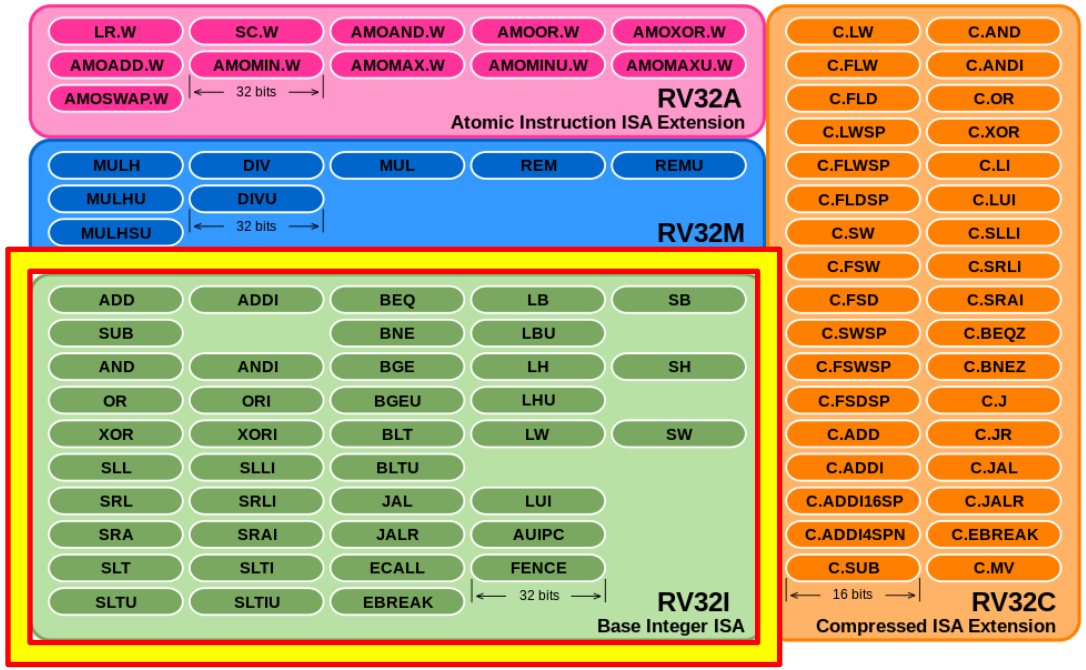
Simple Instruction Set

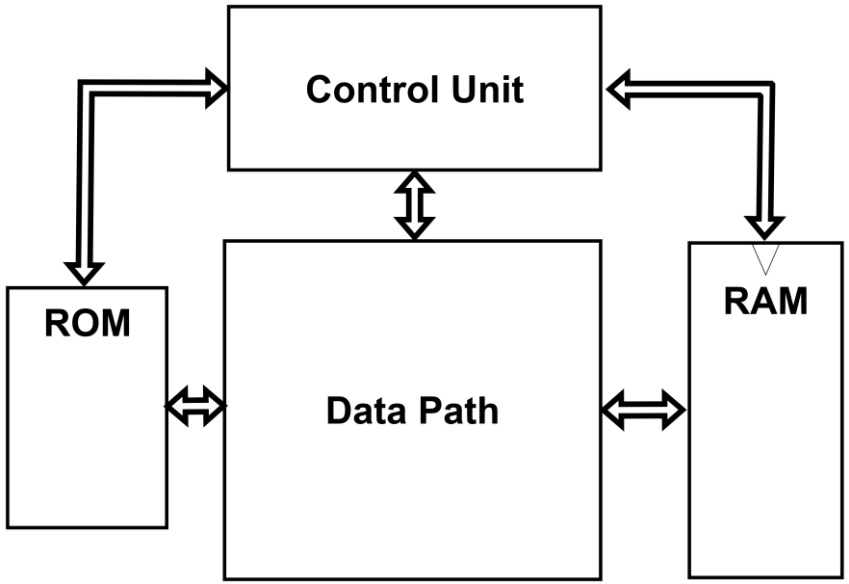
Free and Open Source

Flexible ISA Structure

➢ 3 Important details of RISC – V Architecture

RISC – V Introduction (Cont.)





- Instruction Set of RV32I
- RISC-V Core Implemented in Harvard Architecture

Simple Instruction Set
=>**RISC Instruction**

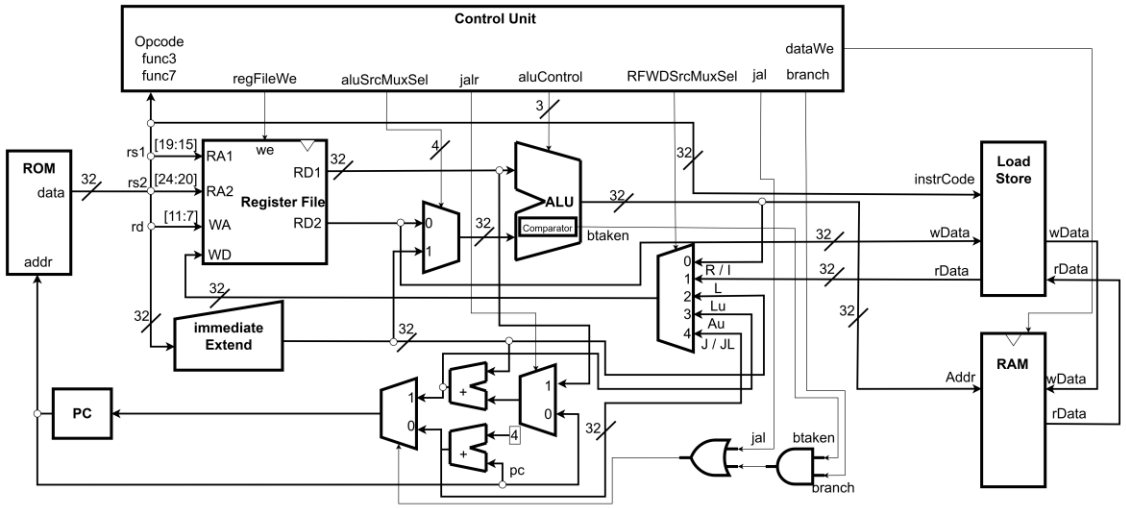
Flexible ISA Structure
=> **RV32I**

Free and Open Source
=> **Anyone can design**

RISC – V Introduction (Cont.)

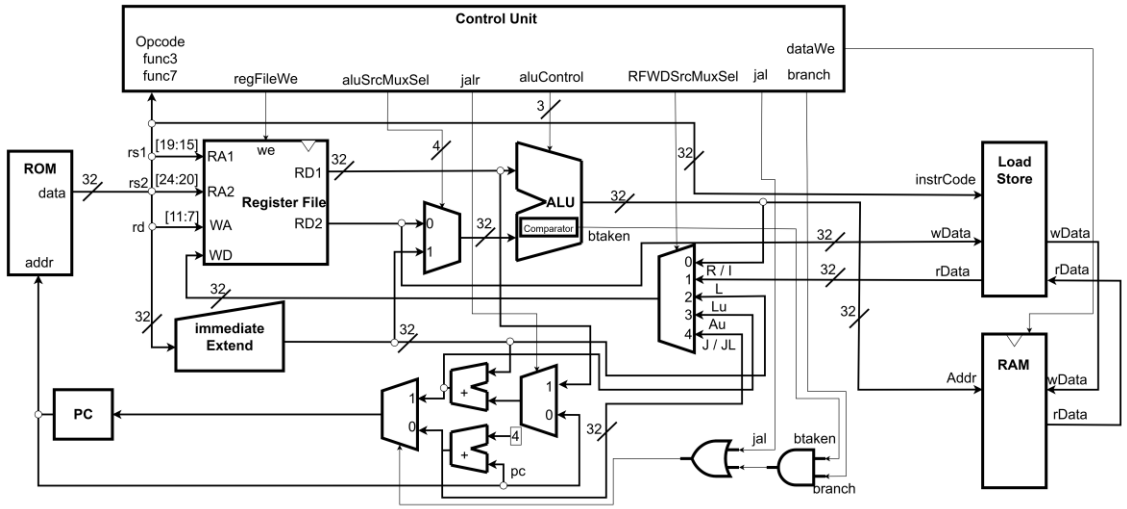
Type	Instructions
R - Type	ADD, SUB, SLL, SRL, SRA, SLTU, XOR, AND
L - Type	LB, LH, LW, LBU, LHU
I - Type	ADDI, SLTI, XORI, ORI, ANDI, SLLI, SRLI, SRAI
S - Type	SB, SH, SW
B - Type	BEQ, BNE, BGE, BLTU, BGEU
LU - Type	LUI
AU - Type	AUIPC
J - Type	JAL
JL - Type	JALR

➢ Selected Instruction Set of RV32I

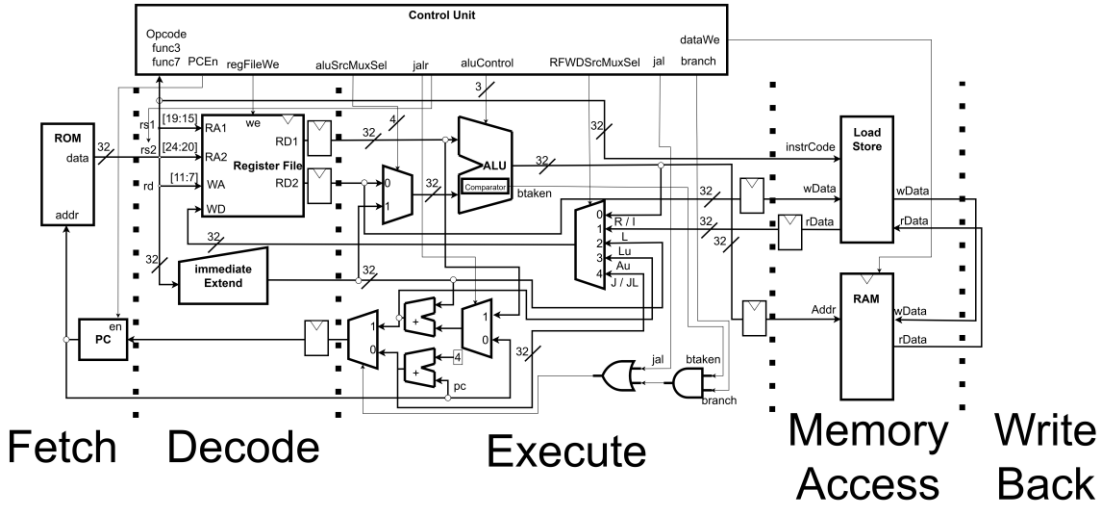


➢ Single Cycle RISC-V Core

RISC – V Introduction (Cont.)



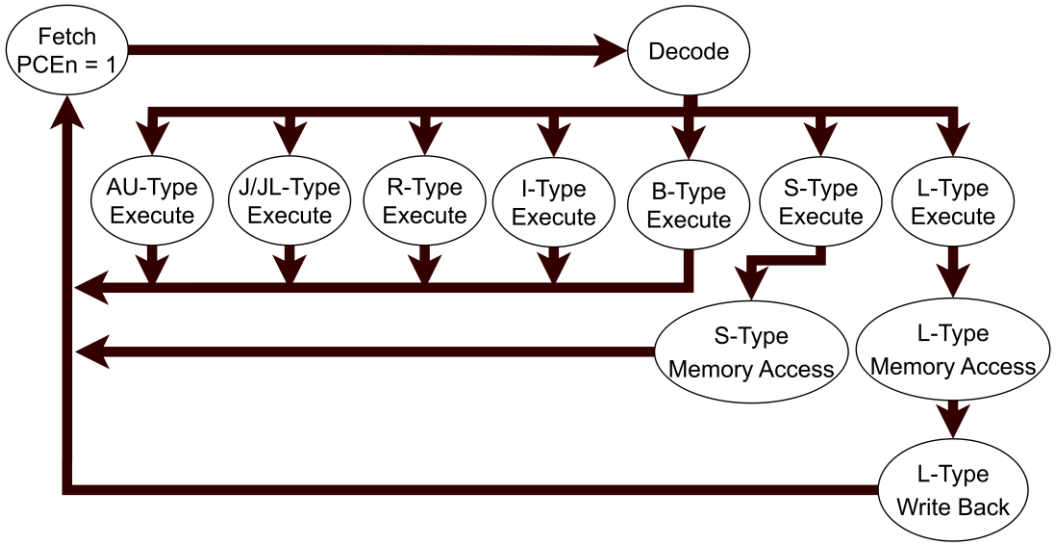
➢ Single Cycle RISC-V Core



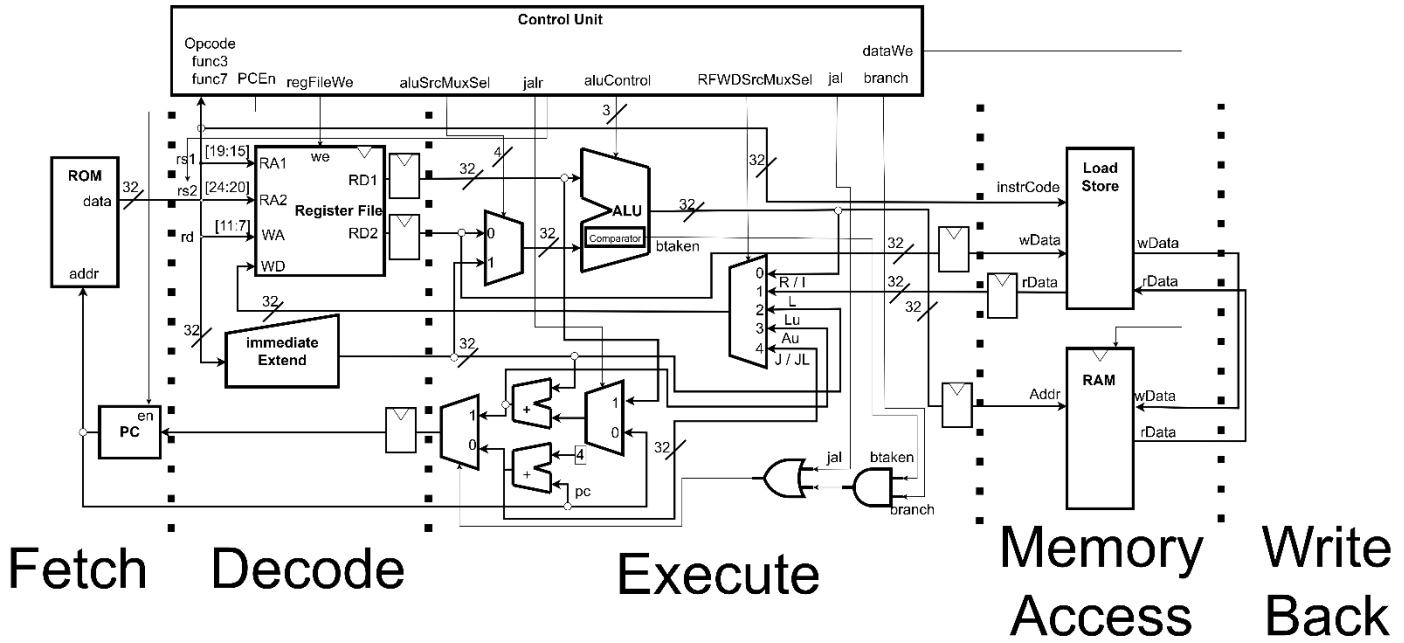
➢ Multi Cycle RISC-V Core

Advantage of Multi Cycle Core	
Improved Performance	Each instruction takes only as many cycles as needed. **=> Simple instructions finish faster than complex ones.**
Higher Clock Frequency	Since each cycle handles only a portion of the instruction, the critical path is shorter. **=> Faster clock design.**

RISC – V Introduction (Cont.)



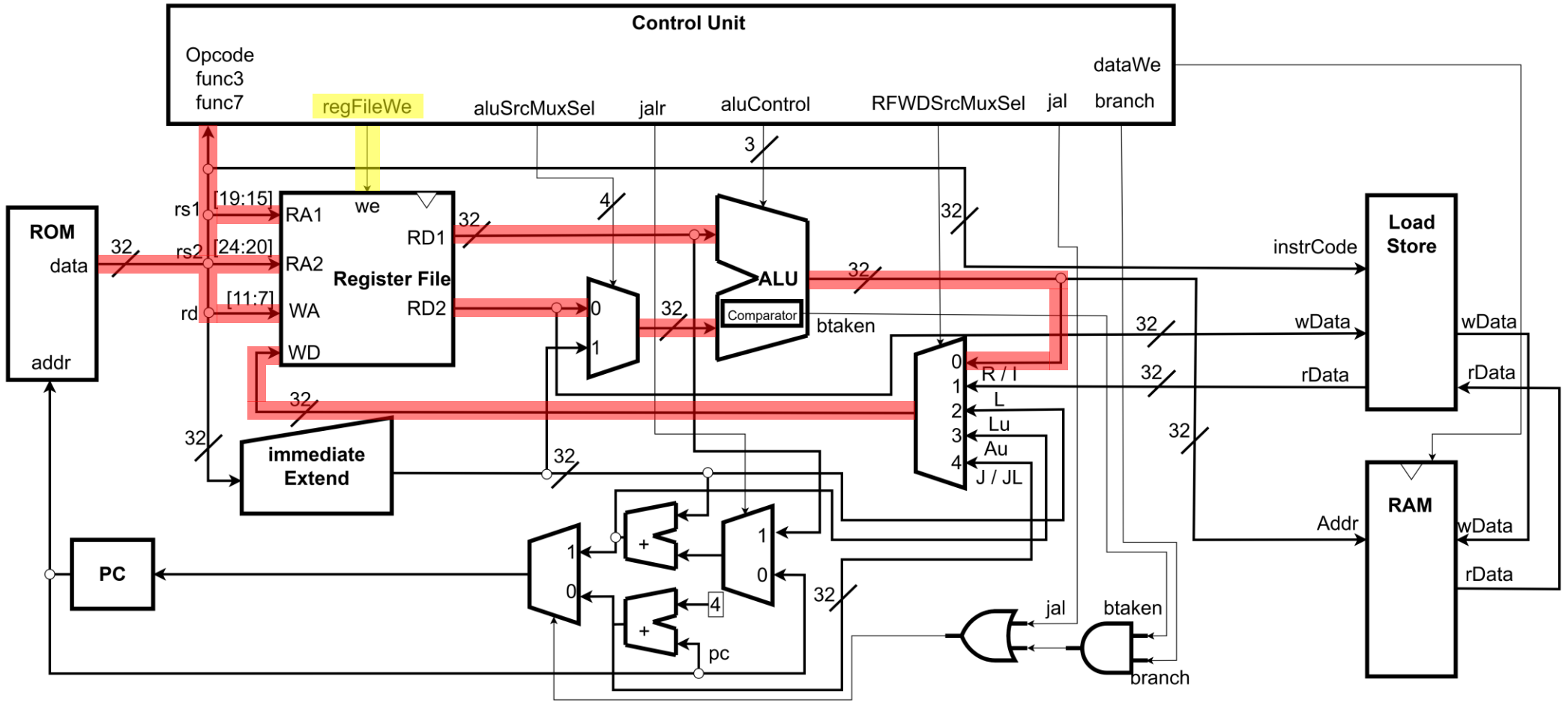
- Multi Cycle FSM



- Multi Cycle RISC-V Block Diagram

R – Type Data Path

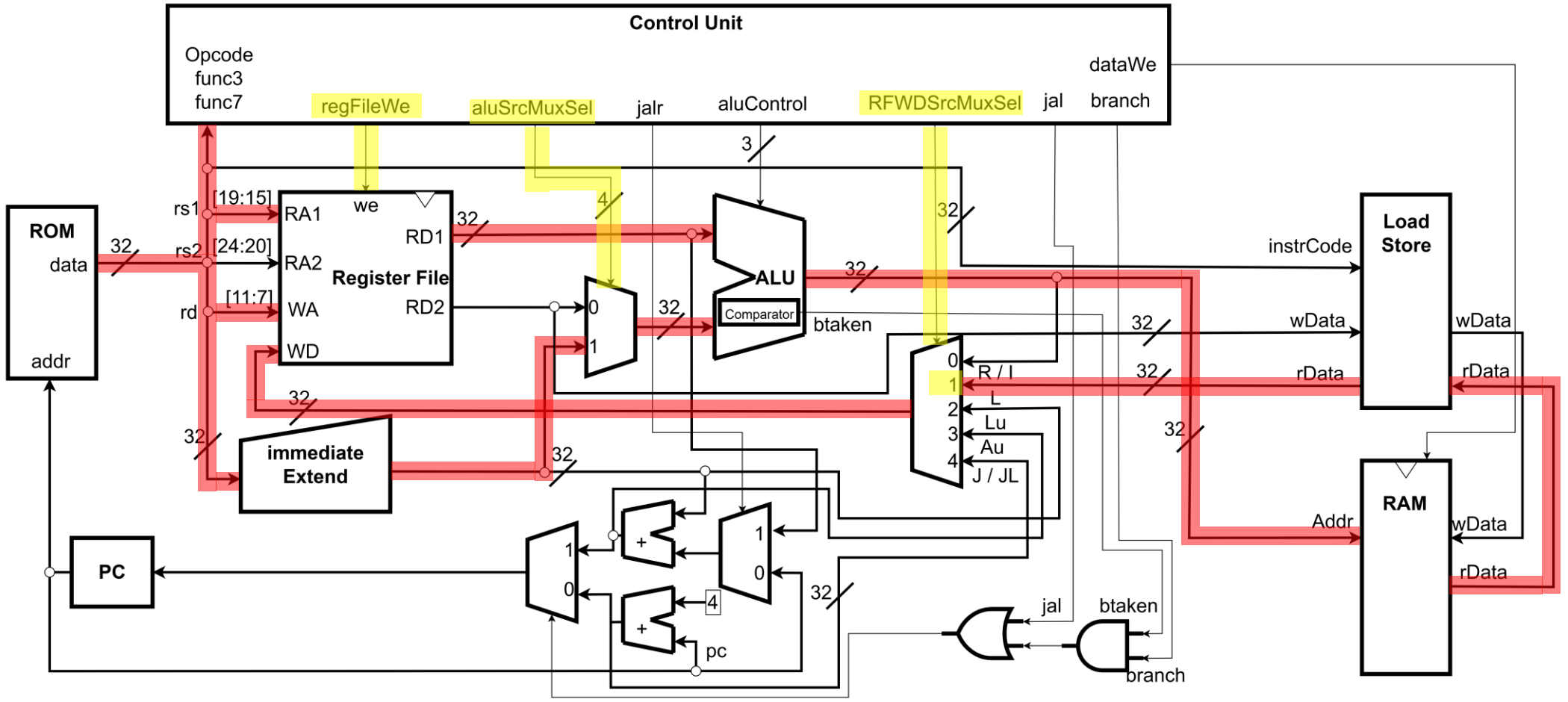
Signals	regFileWe	aluSrcMuxSel	busWe	RFWDSrcMuxSel	branch	jal	jalr
0 or 1	1	0	0	000	0	0	0



➢ R –Type Data Path

L – Type Data Path

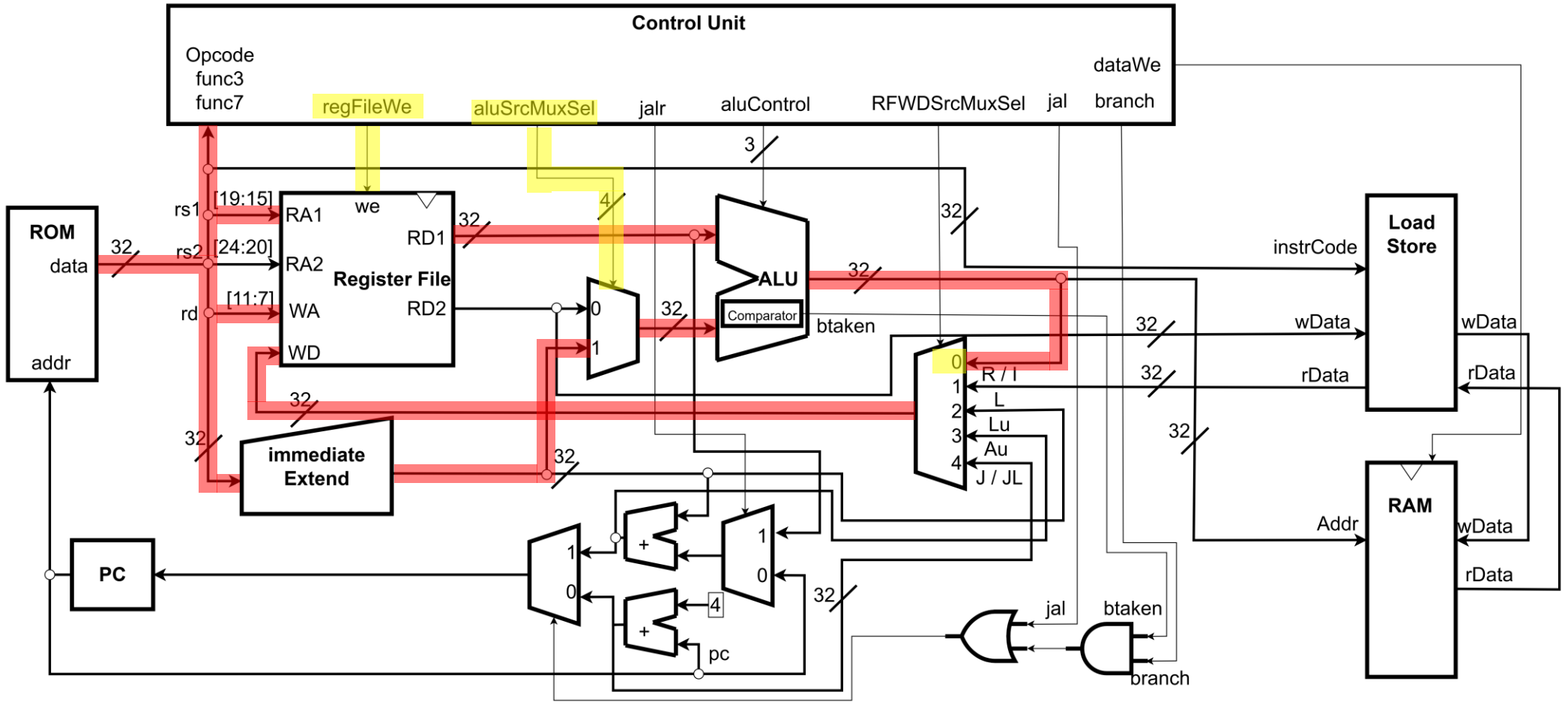
Signals	regFileWe	aluSrcMuxSel	busWe	RFWDSrcMuxSel	branch	jal	jalr
0 or 1	1	1	0	001	0	0	0



➢ L –Type Data Path

I – Type Data Path

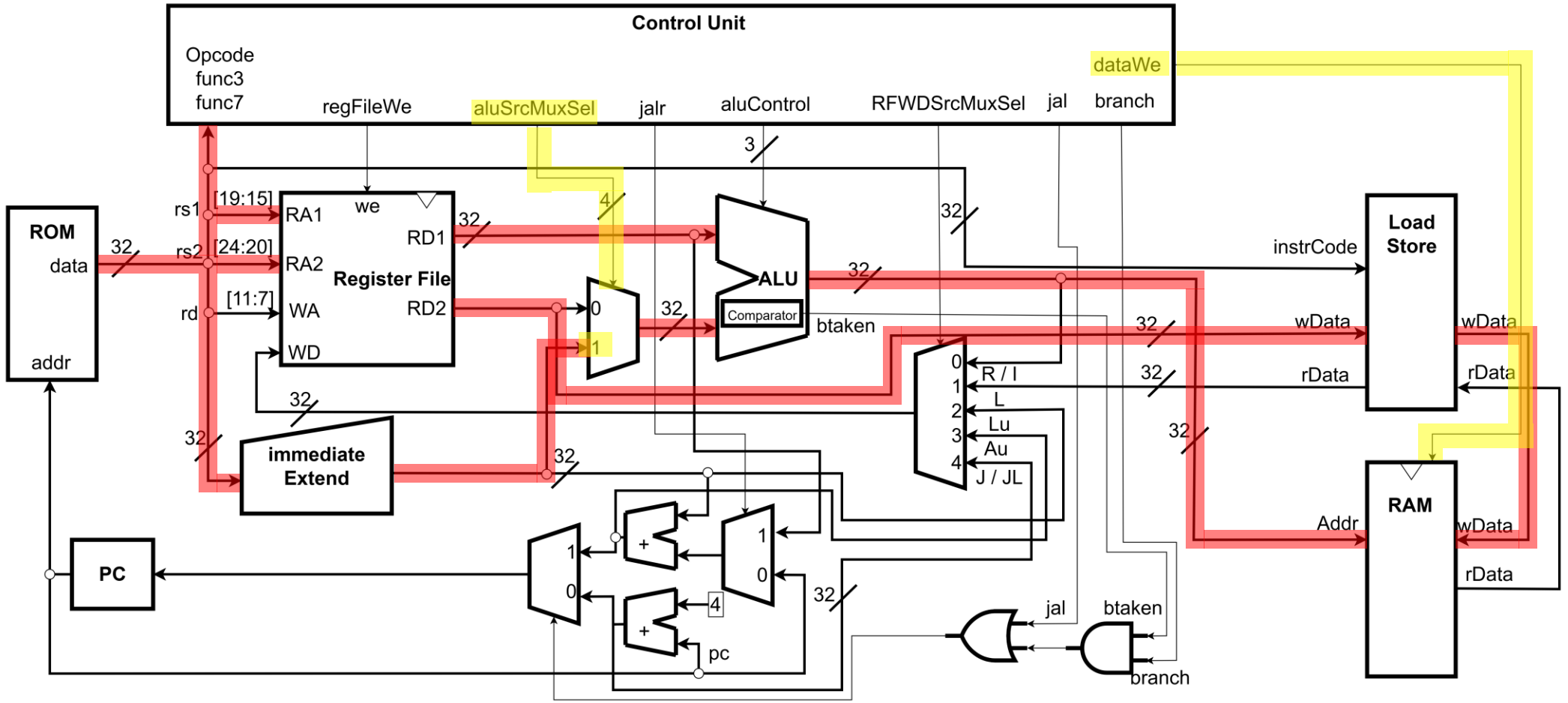
Signals	regFileWe	aluSrcMuxSel	busWe	RFWDSrcMuxSel	branch	jal	jalr
0 or 1	1	1	0	000	0	0	0



➢ I –Type Data Path

S – Type Data Path

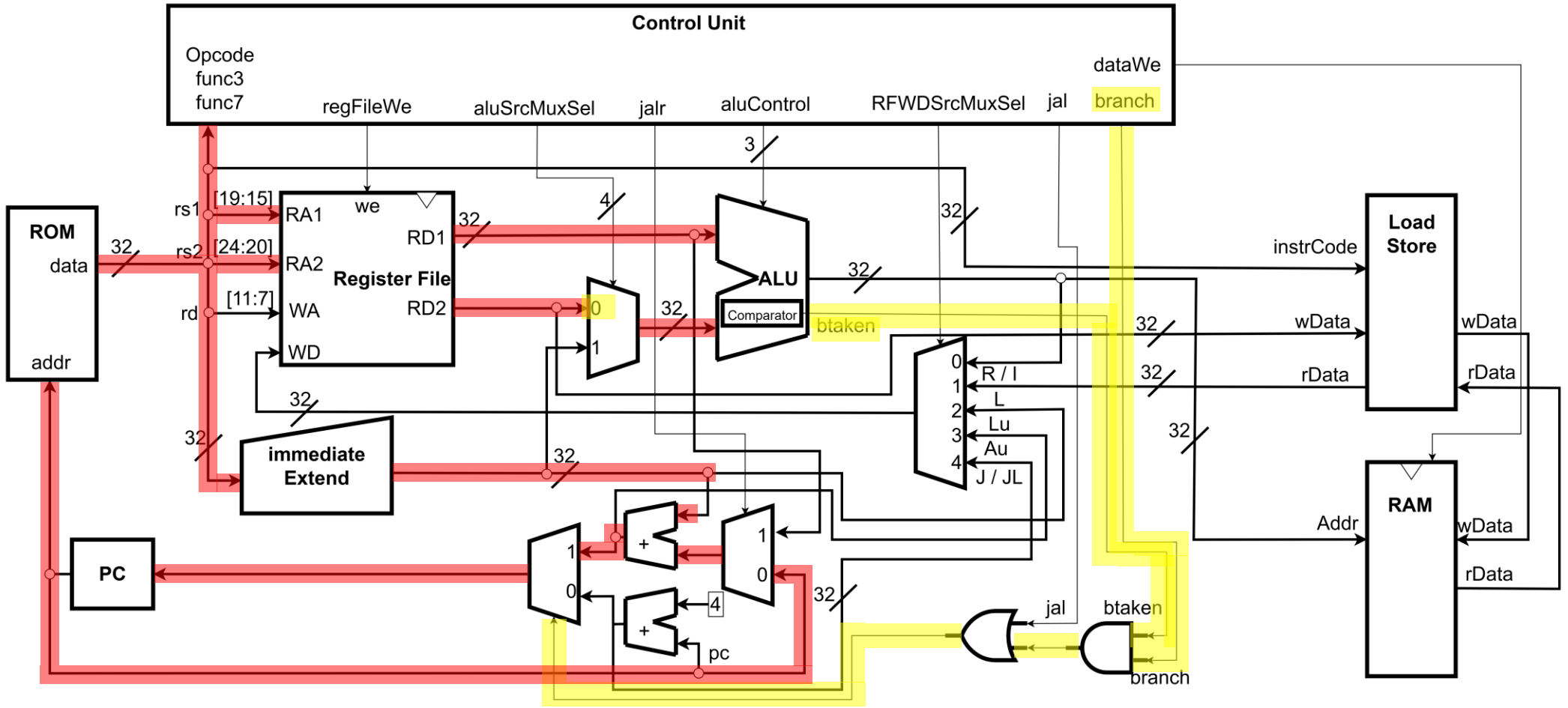
Signals	regFileWe	aluSrcMuxSel	busWe	RFWDSrcMuxSel	branch	jal	jalr
0 or 1	0	1	1	000	0	0	0



➢ S –Type Data Path

B – Type Data Path

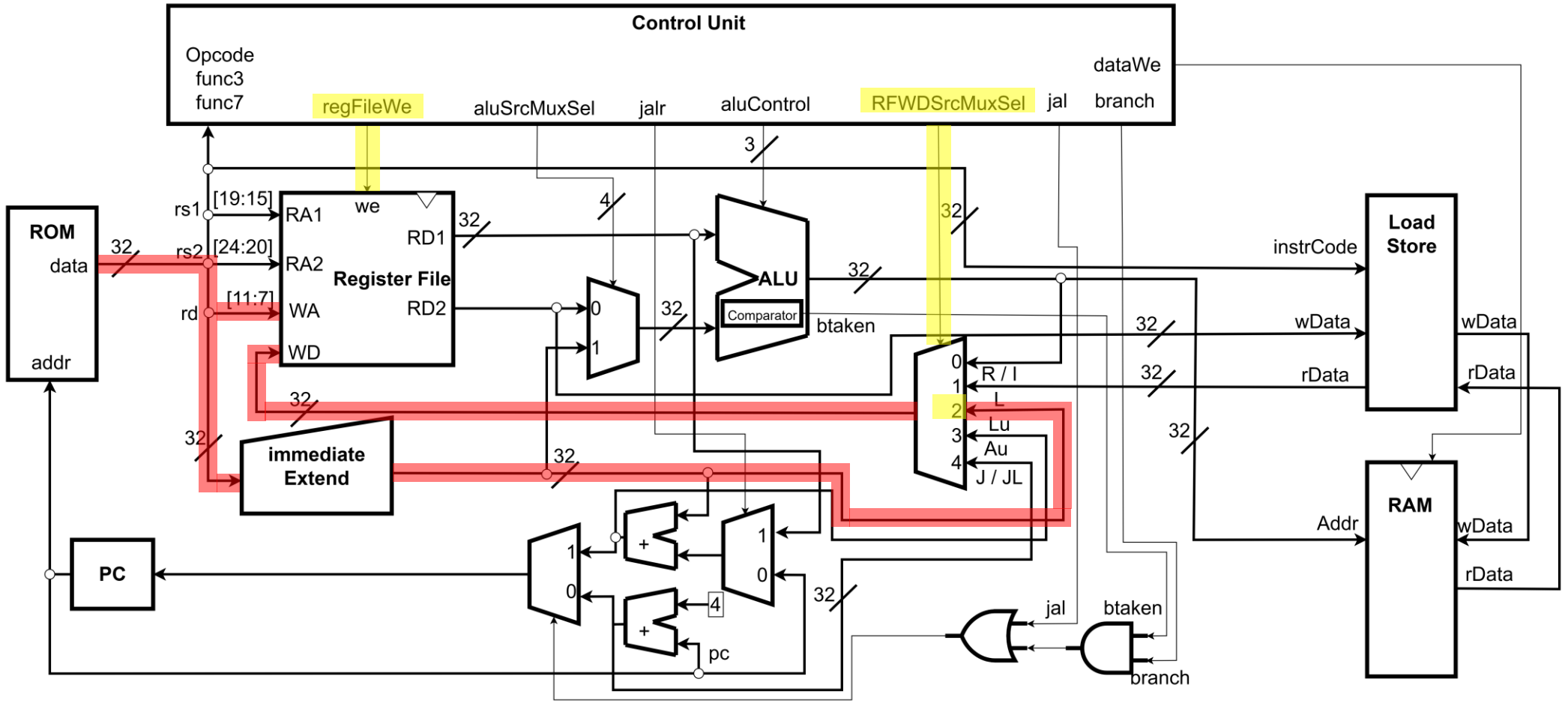
Signals	regFileWe	aluSrcMuxSel	busWe	RFWDSrcMuxSel	branch	jal	jalr
0 or 1	0	0	0	000	1	0	0



➢ B –Type Data Path

LU – Type Data Path

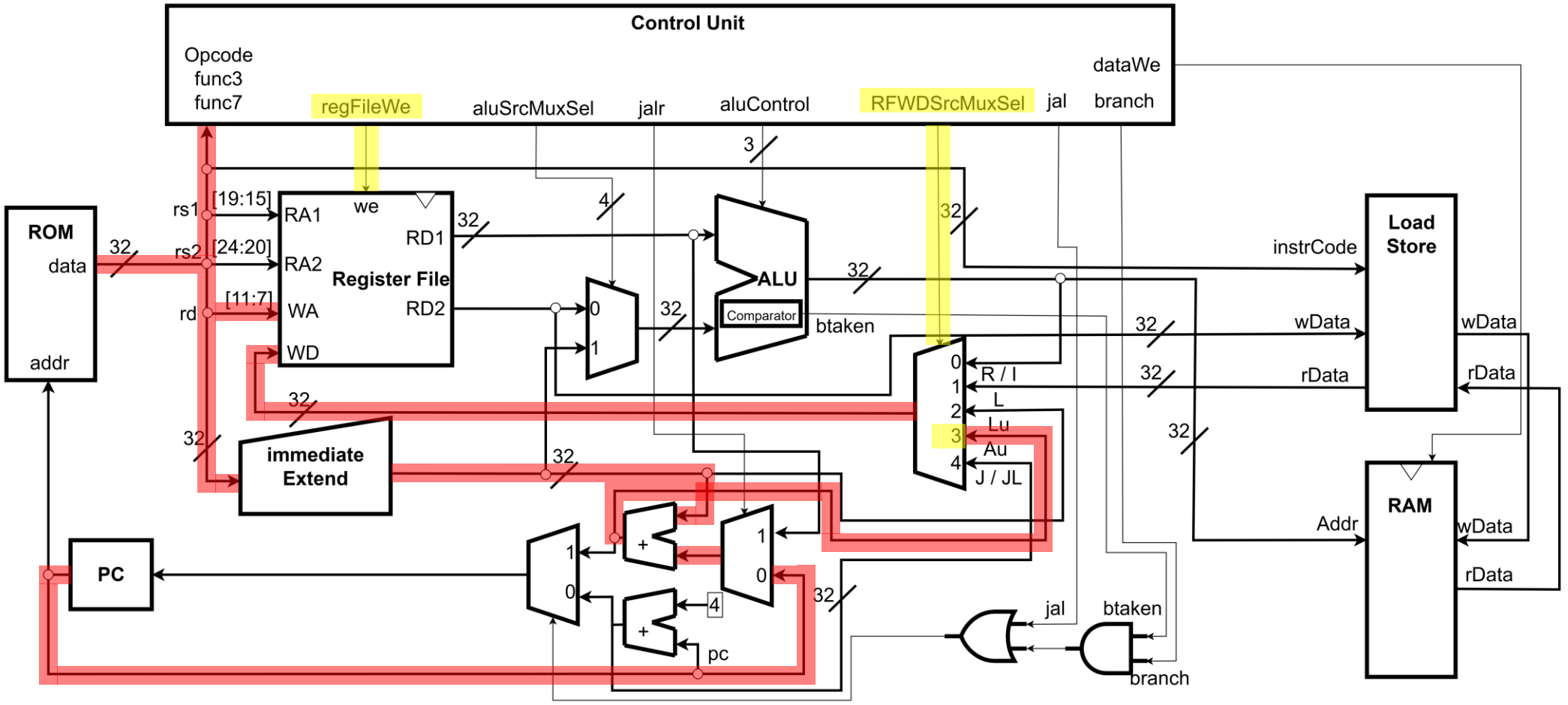
Signals	regFileWe	aluSrcMuxSel	busWe	RFWDSrcMuxSel	branch	jal	jalr
0 or 1	1	0	0	010	0	0	0



➢ LU –Type Data Path

AU – Type Data Path

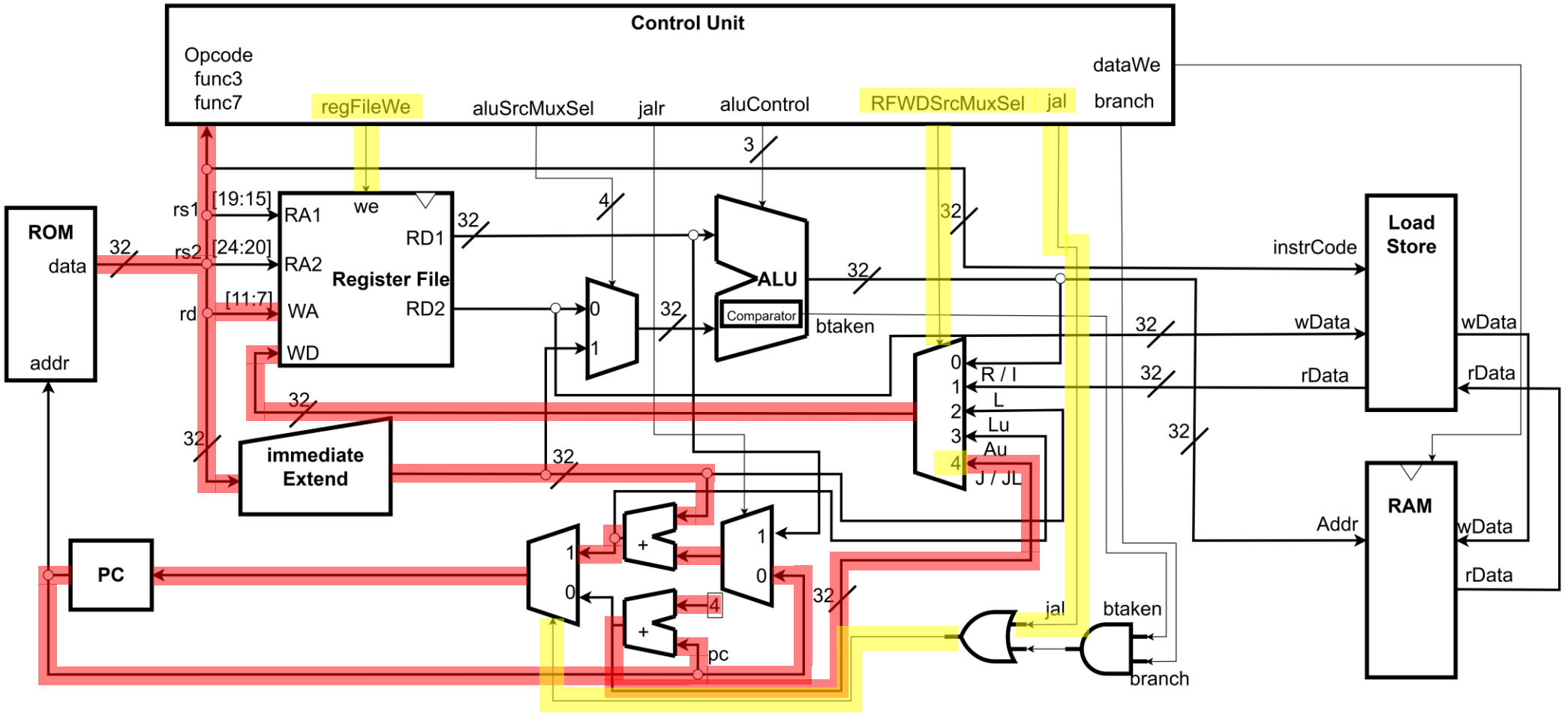
Signals	regFileWe	aluSrcMuxSel	busWe	RFWDSrcMuxSel	branch	jal	jalr
0 or 1	1	0	0	011	0	0	0



➢ AU –Type Data Path

J – Type Data Path

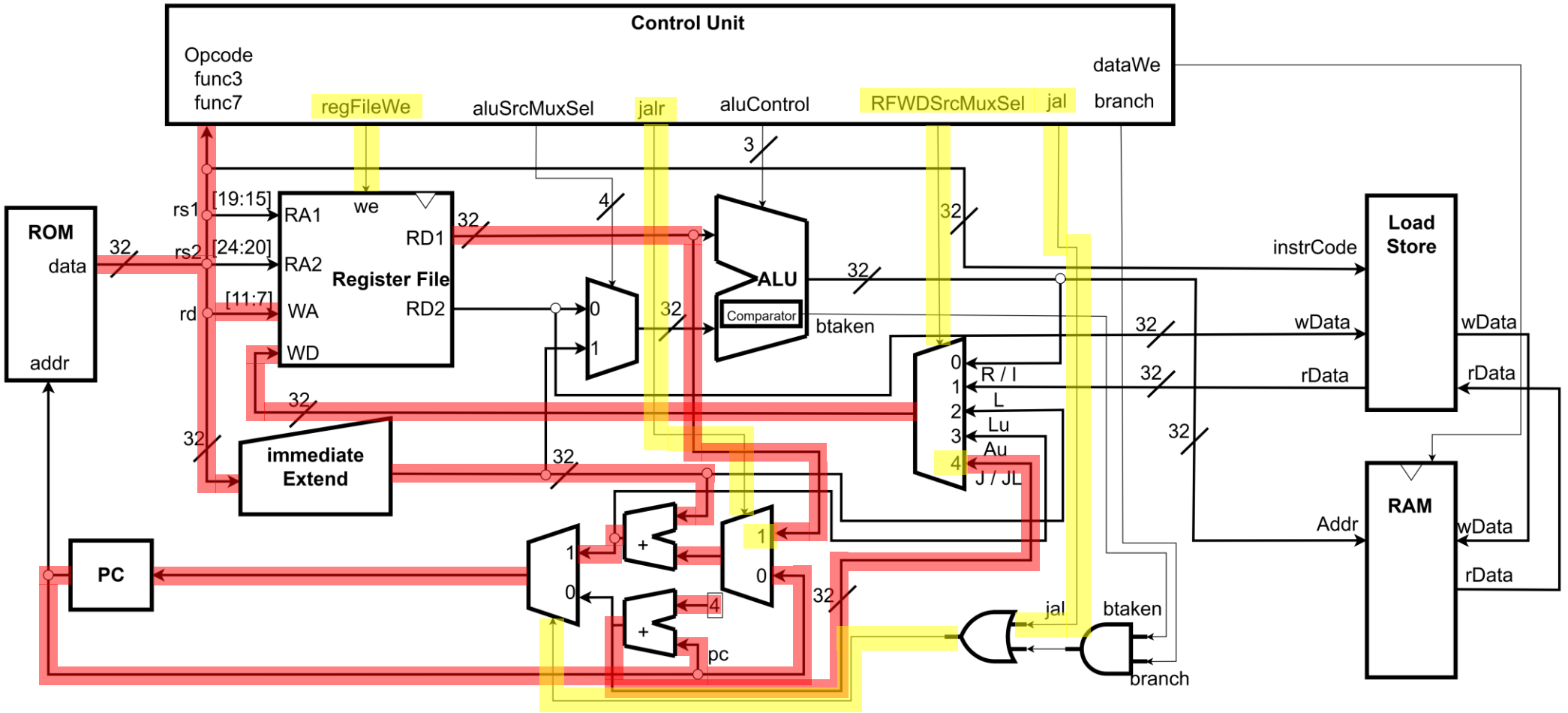
Signals	regFileWe	aluSrcMuxSel	busWe	RFWDSrcMuxSel	branch	jal	jalr
0 or 1	1	0	0	100	0	1	0



➢ J –Type Data Path

JL – Type Data Path

Signals	regFileWe	aluSrcMuxSel	busWe	RFWDSrcMuxSel	branch	jal	jalr
0 or 1	1	0	0	100	0	1	1



➢ JL –Type Data Path

R– Type Timing

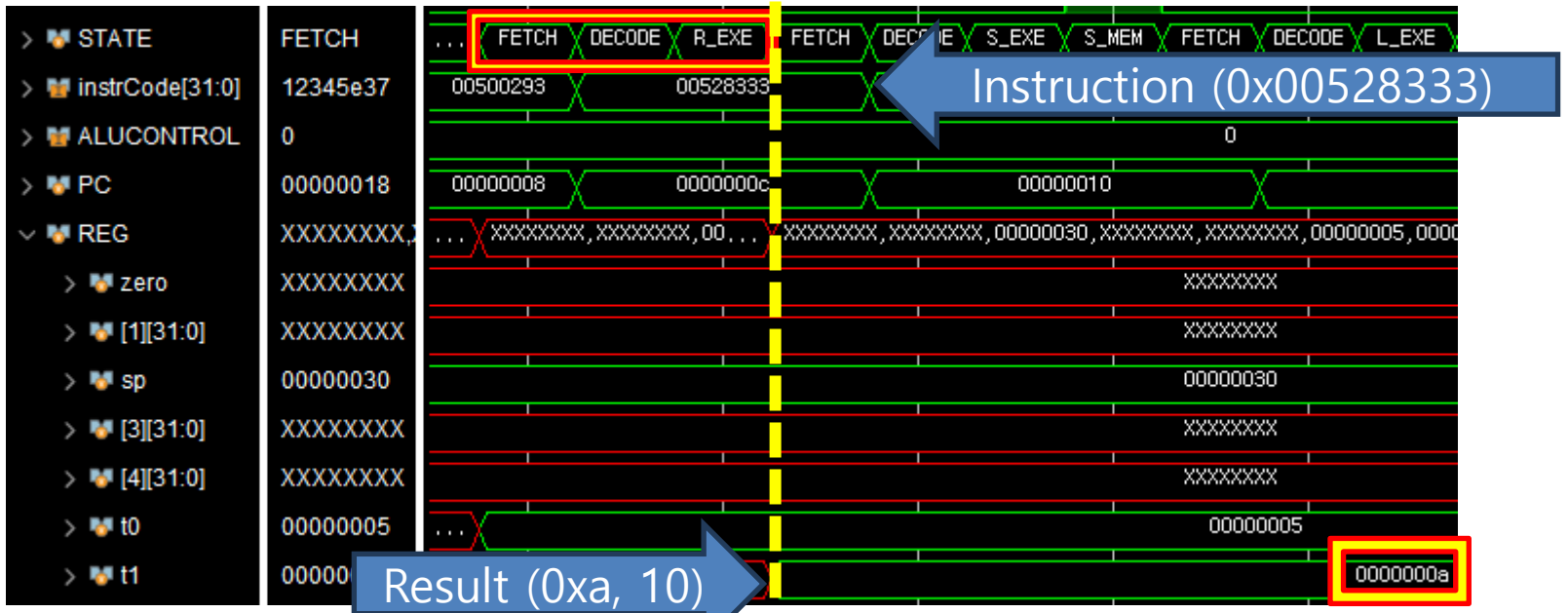
```
add   t1, t0, t0
```

```
Assembly =   add x6, x5, x5

  Binary =   0000 0000 0101 0010 1000 0011 0011 0011

Hexadecimal = 0x00528333

  Format =   R-type
```

Operation	Expression	Decimal Result
	t0 = 5	
ADDI	t1 = t0 + t0	**10**

➢ Assembly Code Test Program for I - Type

➢ Test Program expected results Table



➢ R – Type Timming

L – Type Timing

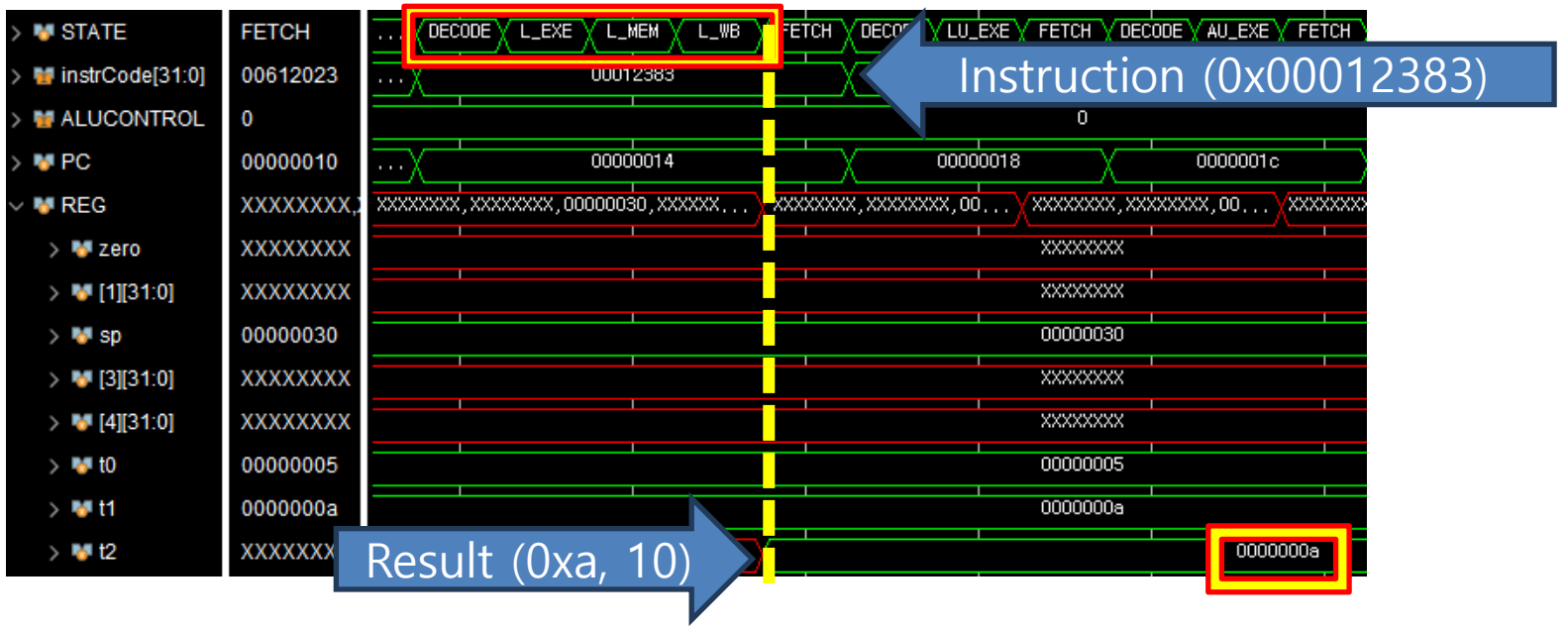
```
lw    t2, 0(sp)
```

```
Assembly =   sw x6, 0(x2)

  Binary =   0000 0000 0110 0001 0010 0000 0010 0011

Hexadecimal =  0x00612023

  Format =   S-type
```

➢ Assembly Code Test Program for L - Type

Operation	Expression	Decimal Result
REG t2 = mem[0x30] = mem[12]		
L	t2 = 10	**10**

➢ Test Program expected results Table



➢ L – Type Timming

I – Type Timing

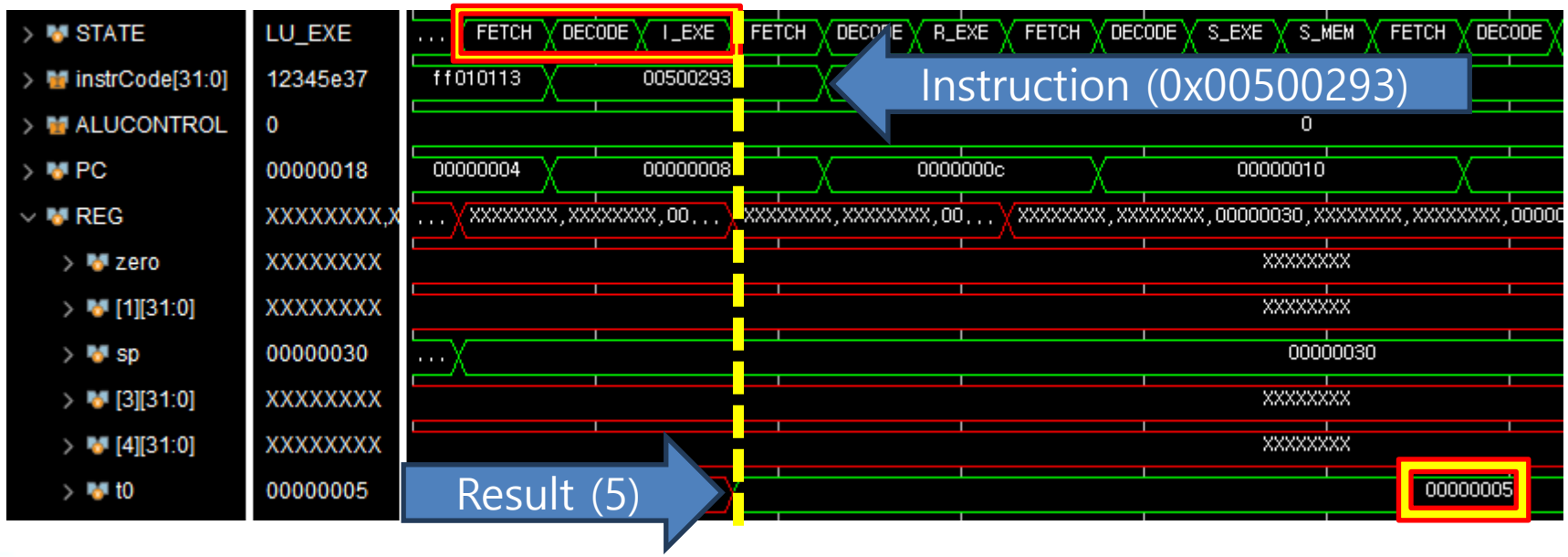
```
addi  t0, zero, 5
```

```
Assembly =   addi x5, x0, 5

Binary =     0000 0000 0101 0000 0000 0010 1001 0011

Hexadecimal = 0x00500293

Format =     I-type
```

Operation	Expression	Decimal Result
	Immediate = 5	
ADDI	t0 = 5	**5**

➢ Assembly Code Test Program for I - Type

➢ Test Program expected results Table



➢ I – Type Timming

S – Type Timing

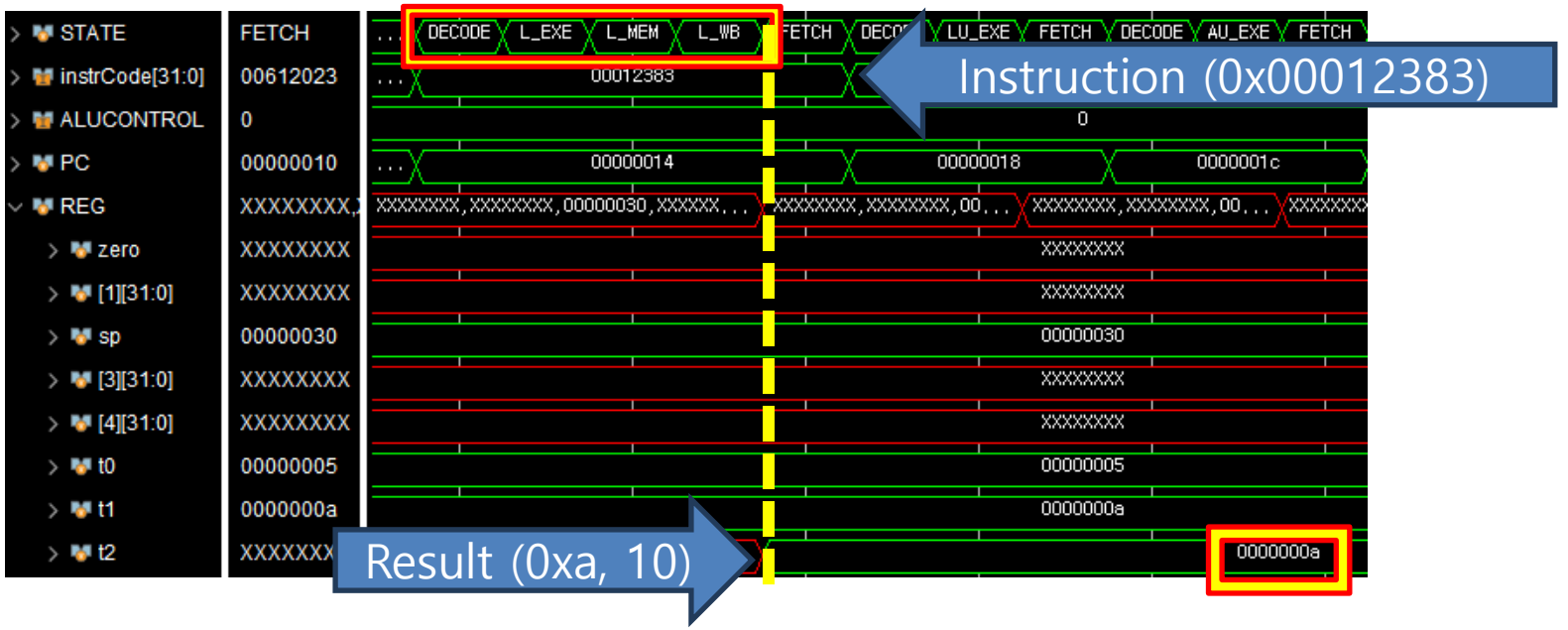
```
sw    t1, 0(sp)
```

```
Assembly =    sw x6, 0(x2)

  Binary =    0000 0000 0110 0001 0010 0000 0010 0011

Hexadecimal = 0x00612023

  Format =    S-type
```

➢ Assembly Code Test Program for S - Type

Operation	Expression	Decimal Result
RAM mem[sp+0] = mem[0x30] = mem[12]		
LU	RAM mem[sp+0] = t1	**10**

➢ Test Program expected results Table



➢ L – Type Timming

LU – Type Timing

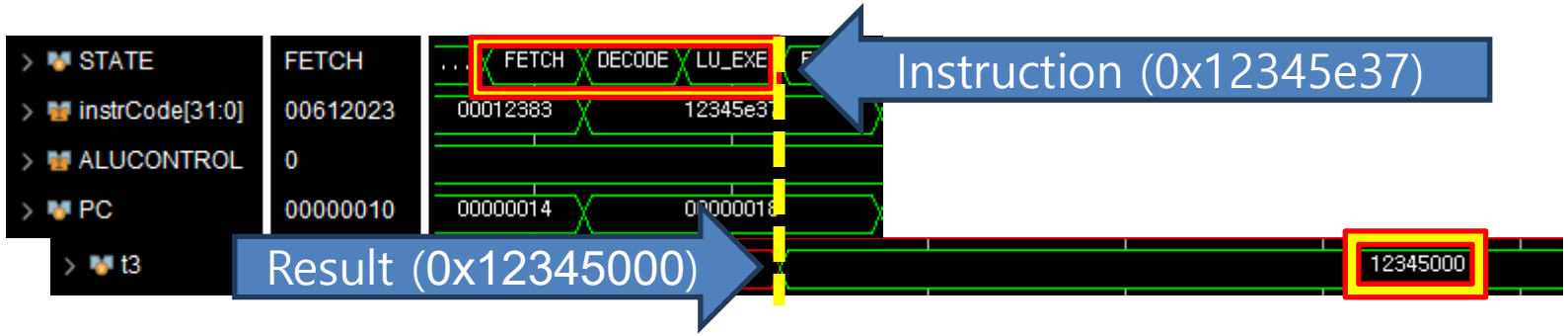
```
lui   t3, 0x12345
```

```
Assembly =   lui x28, 74565
Binary =     0001 0010 0011 0100 0101 1110 0011 0111
Hexadecimal = 0x12345e37
Format =     U-type
```

➢ Assembly Code Test Program for LU - Type

Operation	Expression	Decimal Result
Immediate = 0x12345		
LU	t3 = mem[12]	**0x12345000**

➢ Test Program expected results Table



➢ LU – Type Timming

AU – Type Timing

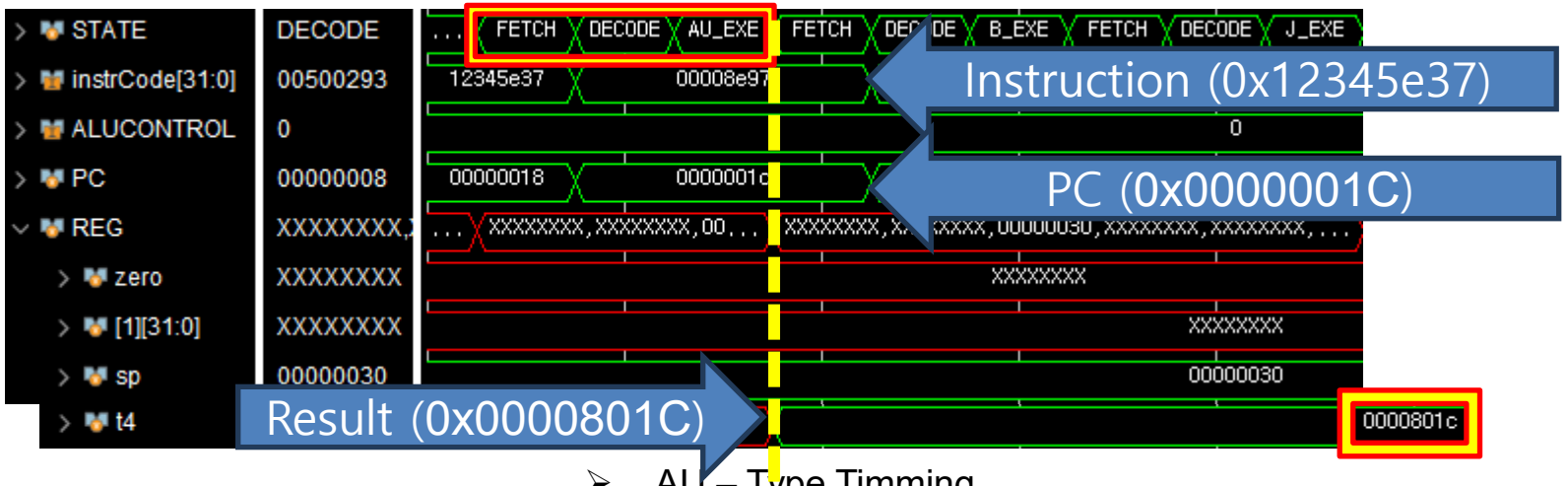
```
auipc t4, 8
```

```
Assembly =   auipc x29, 8
Binary =     0000 0000 0000 0000 1000 1110 1001 0111
Hexadecimal = 0x00008e97
Format =     U-type
```

➢ Assembly Code Test Program for AU - Type

Operation	Expression	Decimal Result
PC = 0x0001C		
AU	t4 = PC + upper_immed	0x0000801C

➢ Test Program expected results Table



➢ AU – Type Timming

B – Type Timing

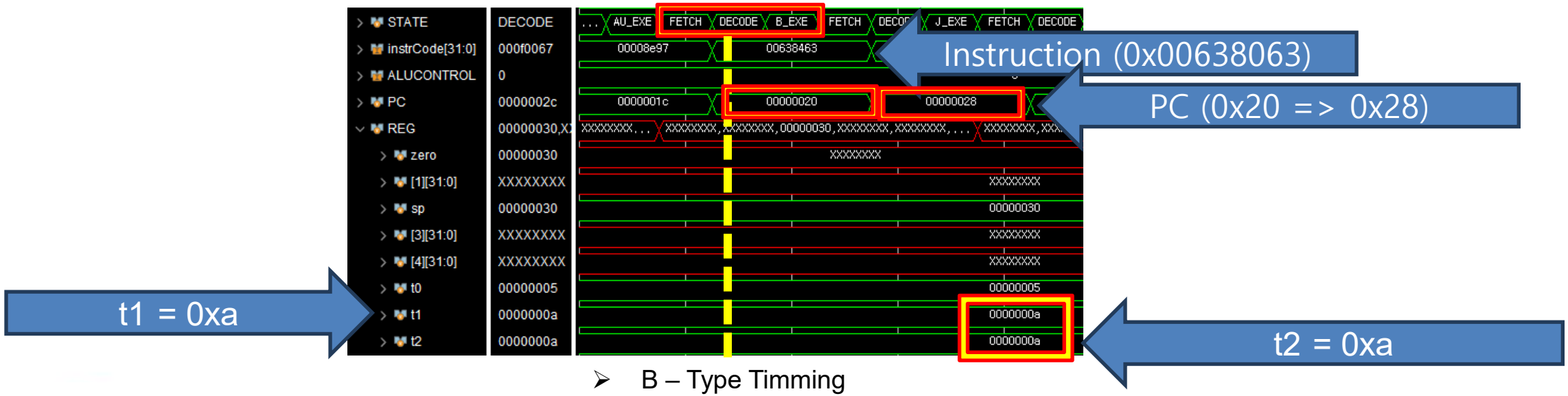
```
beq    t2, t1, do_jal
```

```
Assembly =    beq x7, x6, 0

Binary =      0000 0000 0110 0011 1000 0000 0110 0011

Hexadecimal = 0x00638063

Format =      B-type
```

➢ Assembly Code Test Program for B - Type

Operation	Expression	Decimal Result
PC = do_jal		
AU	(t2 == t1) => PC = do_jal	0x00000028

➢ Test Program expected results Table



➢ B – Type Timming

JAL – Type Timing

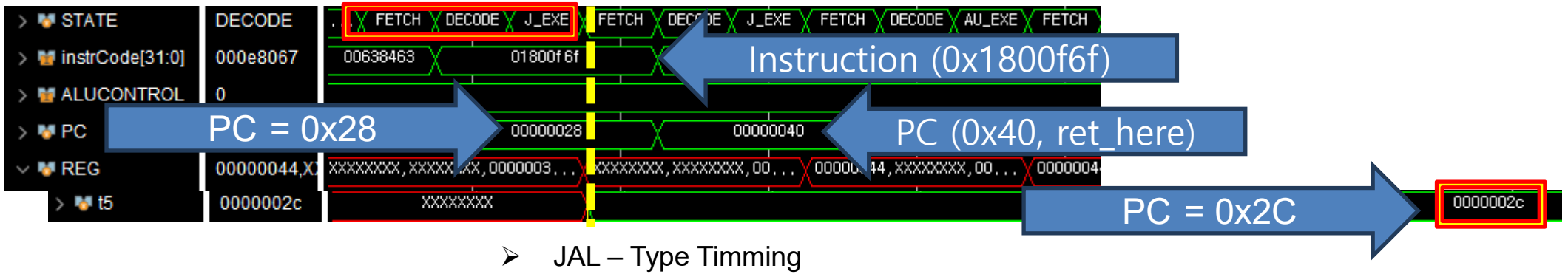
```
do_jal:
    jal    t5, ret_here
```

```
Assembly =    jal x30, 0

  Binary =    0000 0000 0000 0000 0000 1111 0110 1111

Hexadecimal =  0x00000f6f

  Format =    J-type
```

➢ Assembly Code Test Program for JAL - Type

Operation	Expression	Decimal Result
	Immediate = ret_here	
JAL	t5 = PC+ 4, PC += ret_here	ret_here

➢ Test Program expected results Table



➢ JAL – Type Timming

JALR – Type Timing

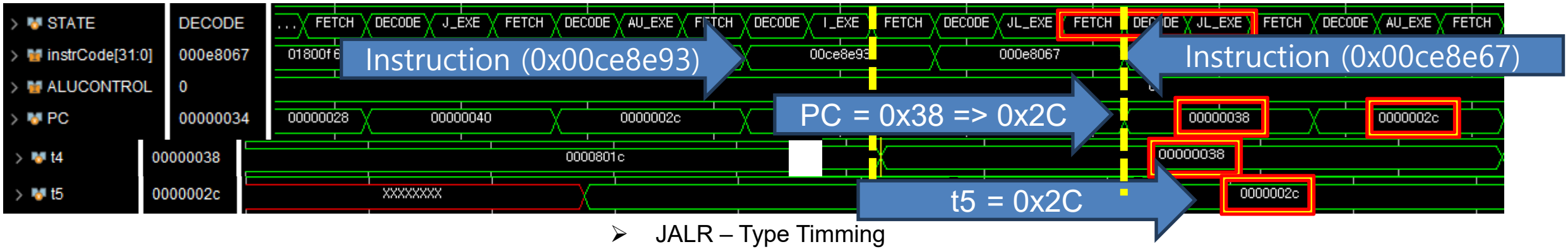
```
link_back:
    auipc t4, 0              # t4 = PC_of_auipc
    addi  t4, t4, 12         # t4 = PC + 12
    jalr  x0, t4, 0          # PC ← t4
cont:
    jalr  x0, t5, 0          # PC ← t5
```

```
Assembly =   jalr x0, 0(x0)

Binary =     0000 0000 0000 0000 0000 0000 0110 0111

Hexadecimal = 0x00000067

Format =     I-type
```

➢ Assembly Code Test Program for JALR - Type

Operation	Expression	Decimal Result
JALR 0x34 => 0x3C		
JALR	PC += t4 => PC = t5 = 0x2C	0x000002C

➢ Test Program expected results Table



➢ JALR – Type Timming

R – Type Test Program

```c
#include <stdio.h>

int main() {
    int a = 0;
    int b = 0;
    int c = 0;
    a = 20;
    b = 5;

    c = a + b;                      // ADD
    c = a - b;                      // SUB
    c = a << b;                     // SLL
    c = a >> b;                     // SRA
    c = (unsigned)a >> b;           // SRL
    c = (a < b);                    // SLT
    c = (unsigned)a < (unsigned)b;  // SLTU
    c = a ^ b;                      // XOR
    c = a | b;                      // OR
    c = a & b;                      // AND

    return 0;
}
```

➢ C Code Test Program for R - Type

Operation	Expression	Decimal Result
	* a = 20, b = 5	
ADD	a + b	**25**
SUB	a - b	**15**
SLL	a << b	**640**
SRA	a >> b	**0**
SRL	(unsigned)a >> b	**0**
SLT	(a < b)	**0**
SLTU	(unsigned)a < (unsigned)b	**0**
XOR	a ^ b	**17**
OR	a | b	**21**
AND	a & b	**4**

➢ Test Program expected results Table

➢ R – Type Simulation Result (Register File Mem ,Unsigned Decimal)

L – Type Test Program

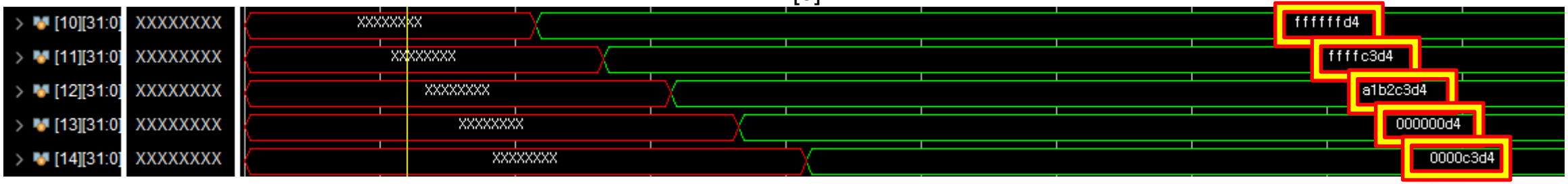
```
main:
    addi  sp, x0, 16

    addi  sp, sp, -16

    lui   t1, 0xA1B2C          # x6 = 0xA1B2C000
    addi  t1, t1, 0x000003D4   # x6 = 0xA1B2C3D4

    sw    t1, 0(sp)            #  mem[0]

    lb    a0, 0(sp)           # a0=x10 : LB
    lh    a1, 0(sp)           # a1=x11 : LH
    lw    a2, 0(sp)           # a2=x12 : LW
    lbu   a3, 0(sp)           # a3=x13 : LBU
    lhu   a4, 0(sp)           # a4=x14 : LHU

halt:
    jal   zero, halt
```

Operation	Expression	Decimal Result
* x6 = 0xA1B2C3D4, RAM=> mem[0]		
LB	mem[10] = 0xFFFFFFD4	**0xFFFFFFD4**
LH	mem[11] = 0xFFFFC3D4	**0xFFFFC3D4**
LW	mem[12] = 0xA1B2C3D4	**0xA1B2C3D4**
LBU	mem[13] =0x000000D4	**0x000000D4**
LHU	mem[14] =0x0000C3D4	**0x0000C3D4**

➢ Test Program expected results Table

➢ Assembly Code Test Program for L - Type

➢ RAM mem[0] Data



➢ L – Type Simulation Result (Register File Mem ,Hexadecimal)

I – Type Test Program

```
main:
    addi  sp, zero, 16
    addi  sp, sp, -16

    # t0 = 100; t1 = 100 + (-50) = 50
    addi  t0, zero, 100
    addi  t1, t0, -50

    # SLTI: (-1 < 0) => 1
    addi  t0, zero, -1
    slti  t1, t0, 0

    # t0 = 0x0FFF; SLTIU: (4095 < 1) => 0
    lui   t0, 0x1           # 0x00001000
    addi  t0, t0, -1        # 0x00000FFF
    sltiu t1, t0, 1
```

➢ Assembly Code Test Program for L - Type

Operation	Expression	Decimal Result
ADDI	100 + (-50)	**0x32**
SLTI	-1 < 0	**1**
SLTIU	4095 < 1	**0**

➢ Test Program expected results Table

➢ L – Type Simulation Result (Register File Mem ,Hexadecimal)

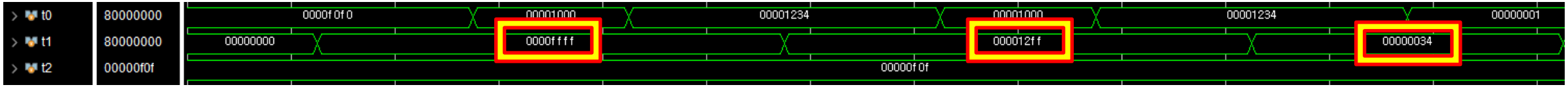
I – Type Test Program (Cont.)

```
# XOR: 0xF0F0 ^ 0x0F0F = 0xFFFF
   lui   t0, 0xF           # 0x0000F000
   addi  t0, t0, 0x0F0     # 0x0000F0F0
   lui   t2, 0x1           # 0x00001000
   addi  t2, t2, -241      # 0x00000F0F
   xor   t1, t0, t2        # -> 0x0000FFFF

   # ORI/ANDI with 0x00FF
   lui   t0, 0x1
   addi  t0, t0, 0x234     # 0x00001234
   ori   t1, t0, 0x0FF     # -> 0x000012FF

   lui   t0, 0x1
   addi  t0, t0, 0x234     # 0x00001234
   andi  t1, t0, 0x0FF     # -> 0x00000034
```

➢ Assembly Code Test Program for L - Type

Operation	Expression	Decimal Result
XOR	0xF0F0 ^ 0x0F0F	**0xFFFF**
ORI	0x00001234 | 0x00FF	**0x000012FF**
ANDI	0x00001234 | 0x00FF	**0x00000034**

➢ Test Program expected results Table



➢ L – Type Simulation Result (Register File Mem ,Hexadecimal)

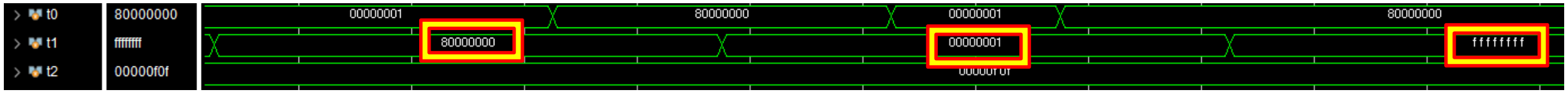
I – Type Test Program (Cont.)

```
# Shift 1 Left 31 times -> use 0x8000_0000
addi  t0, zero, 1
slli  t1, t0, 31        # -> 0x80000000

addi  t0, zero, 1
slli  t0, t0, 31        # t0 = 0x80000000
srli  t1, t0, 31        # -> 0x00000001

addi  t0, zero, 1
slli  t0, t0, 31        # t0 = 0x80000000
srai  t1, t0, 31        # -> 0xFFFFFFFF
```

➢ Assembly Code Test Program for L - Type

Operation	Expression	Decimal Result
SLLI	1 << 31	**0x80000000**
SRLI	0x80000000 >> 31	**0x00000001**
SRAI	0x80000000 >>> 31	**0xFFFFFFFF**

➢ Test Program expected results Table



➢ L – Type Simulation Result (Register File Mem ,Hexadecimal)

S – Type Test Program

```
main :
    addi  sp, zero, 16
    addi  sp, sp, -16

    lui   t0, 0xA1B2C
    addi  t1, t0, 0x3D4  #t1 = 0xA1B2C3D4


    sb   t1, 0(sp)      # 0xD4
    sh   t1, 4(sp)      # 0xC3D4
    sw   t1, 8(sp)      # 0xA1B2C3D4

halt:
    jal   zero, halt
```

➢ Assembly Code Test Program for S - Type

Operation	Expression	Decimal Result
SB	t1[7:0]	**0xD4**
SH	t1[15:0]	**0xC3D4**
SW	t1[31:0]	**0xA1B2C3D4**

➢ Test Program expected results Table

➢ S – Type Simulation Result (RAM Mem ,Hexadecimal)

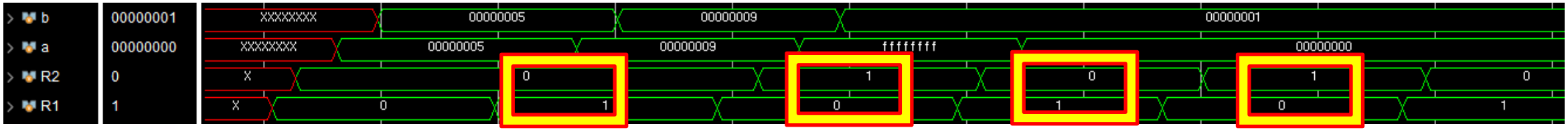
B – Type Test Program

```c
#include <stdint.h>

int main(void) {
    volatile int      t0 = 0;   // T 1 : F 0
    volatile int      t1 = 0;   // T 0 : F 1
    volatile int32_t  a, b;     // signed
    volatile uint32_t ua, ub;   // unsigned
    // 1) BEQ: 5 == 5
    a = 5; b = 5;
    if (a == b) { t0 = 1; t1 = 0; } else { t0 = 0; t1 = 1; }
    // 2) BNE: 9 != 9
    a = 9; b = 9;
    if (a != b) { t0 = 1; t1 = 0; } else { t0 = 0; t1 = 1; }
    // 3) BLT (signed): -1 < 1
    a = -1; b = 1;
    if (a < b) { t0 = 1; t1 = 0; } else { t0 = 0; t1 = 1; }
    // 4) BGE (signed): 0 >= 1
    a = 0; b = 1;
    if (a >= b) { t0 = 1; t1 = 0; } else { t0 = 0; t1 = 1; }
                            …
}
```

Operation	Expression	Decimal Result
State True: R1 : 1 R2 : 0, State False R1 : 0, R2 : 1		
BEQ	5 == 5	**R1 : 1 R2 : 0**
BNE	9 != 9	**R1 : 0 R2 : 1**
BLT	-1 < 1	**R1 : 1 R2 : 0**
BGE	0 >= 1	**R1 : 0 R2 : 1**

➢ Test Program expected results Table

➢ C Code Test Program for B - Type



➢ B – Type Simulation Result (RAM Mem ,Hexadecimal)

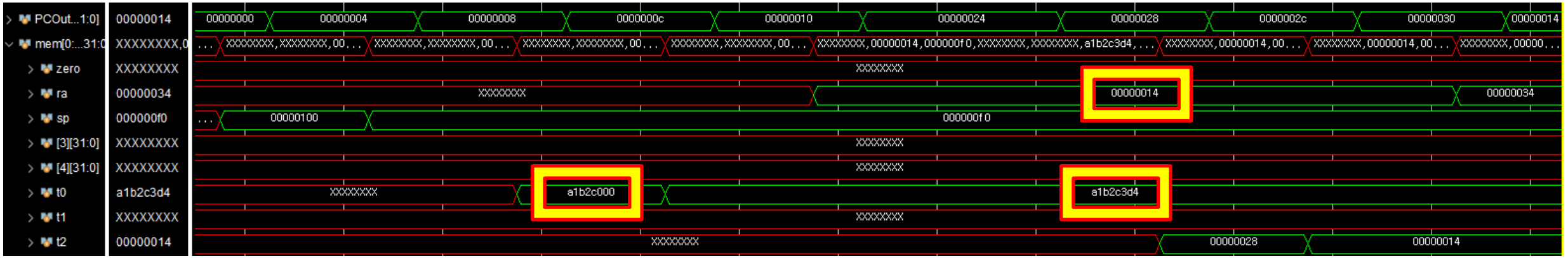
LU, AU, JAL, JALR – Type Test Program (Cont.)

```
_start:
    addi  sp, zero, 0x100
    addi  sp, sp,   -16
    lui   t0, 0xA1B2C
    addi  t0, t0, 0x3D4
    jal   ra, _jump_test
after_jal:
```

Operation	Expression	Decimal Result
LU	t0 = 0xA1B2C	**0xA1B2C**
AUIPC		
J		
JL	ra = 0x14	**0x14**

➢ Test Program expected results Table

➢ C Code Test Program for LU, AU, JAL, JALR - Type



➢ LU, AU, JAL, JALR – Type Simulation Result (RAM Mem ,Hexadecimal)

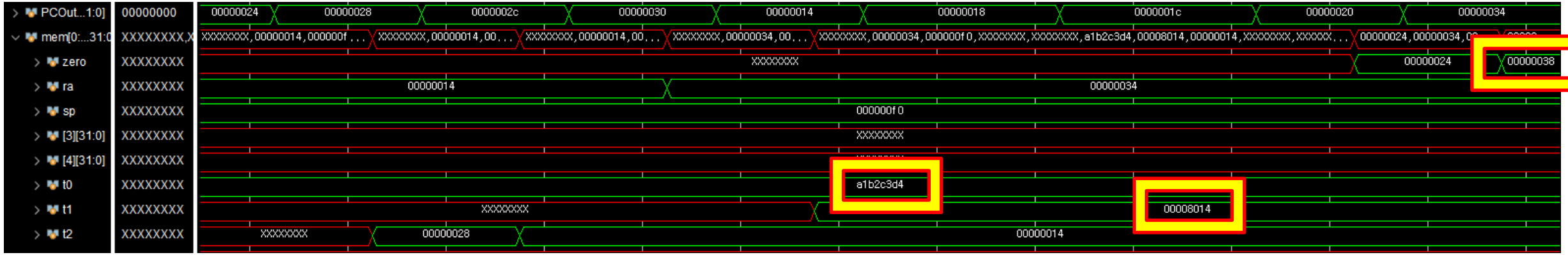
LU, AU, JAL, JALR – Type Test Program (Cont.)

```
_auicp:
    auipc t1, 0x8
    sw    t1, 0(sp)
    sw    ra, 8(sp)
    j     halt

_jump_test:
    sw    ra, 4(sp)
    la    t2, _auicp
    jalr  ra, 0(t2)

halt:
    j     halt
```

➢ Assembly Test Program for LU, AU, JAL, JALR - Type

Operation	Expression	Decimal Result
LU	t0 = 0xA1B2C	**0xA1B2C**
AUIPC	t1 = pc + 0x8000	**0x8014**
J	ra = 0x38	**0x38**
JL	ra = 0x14	**0x14**

➢ Test Program expected results Table



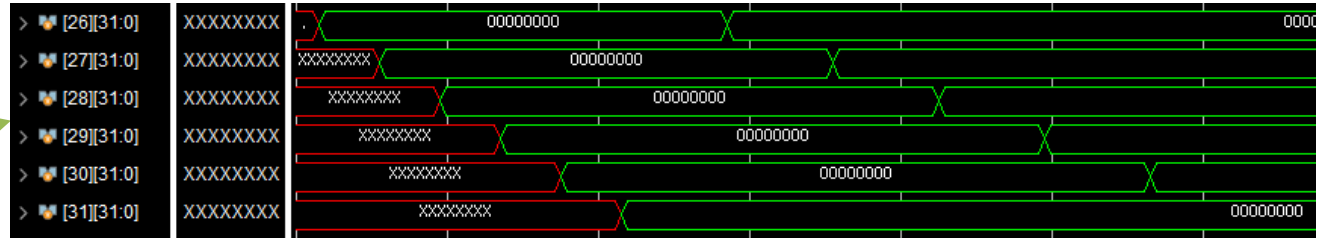
➢ LU, AU, JAL, JALR – Type Simulation Result (RAM Mem ,Hexadecimal)

Test Program : Bubble Sort

```
        li      sp, 0x90
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
```

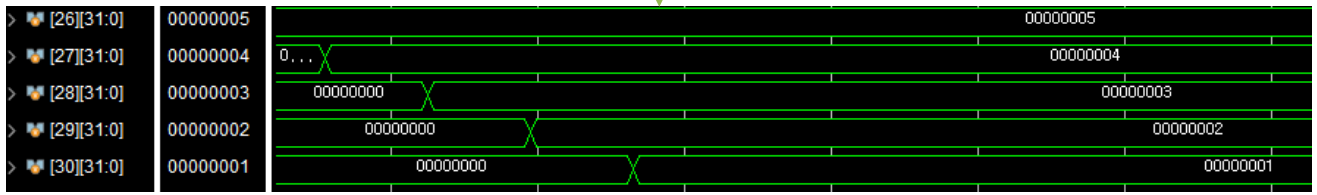
➢ Assembly Test Program

➢ S0 => 0x90



➢ RAM에 Zero 초기화

➢ a5 => REG mem[15]



➢ Register load immediate => Ram Store word

➢ Pseudo Code

➢ jal ra, sort

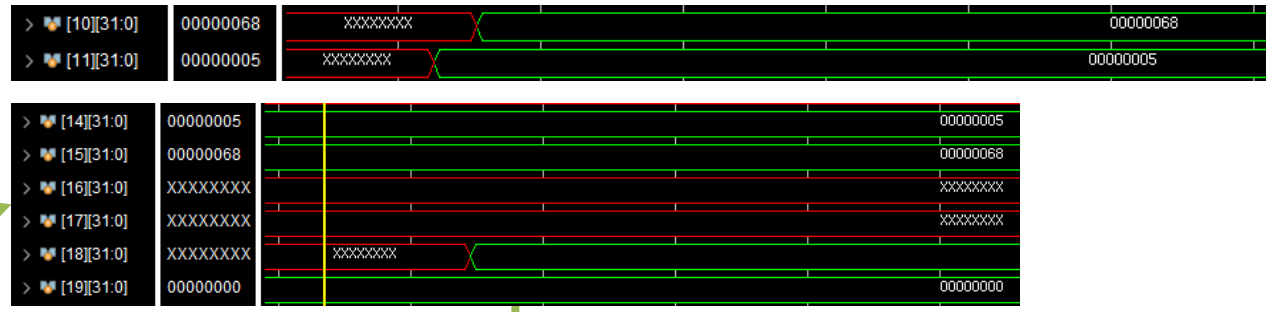
LI, SW, ADDI

Test Program : Bubble Sort (Cont.)

```
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)
        j       .L4

.L4:
        lw      a4,-20(s0)
        lw      a5,-40(s0)
        blt     a4,a5,.L8
        nop
        nop
        lw      ra,44(sp)
        lw      s0,40(sp)
        addi    sp,sp,48
        jr      ra

.L8:
        sw      zero,-24(s0)
        j       .L5
```

- Assembly Test Program

- array[0] address = 0x68, length = 5



- Load 0 and 5

```
sort(arData,5);
```

- 0 < 5 ?
  - Yes : Jump to .L8

```
for(int i=0;i<size;i++)
```

- Jump to .L5

ADDI, SW, J, JR, BLT

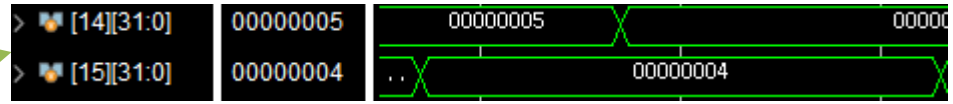
Test Program : Bubble Sort (Cont.)

```
.L5:
        lw      a4,-40(s0)
        lw      a5,-20(s0)
        sub     a5,a4,a5
        addi    a5,a5,-1
        lw      a4,-24(s0)
        blt     a4,a5,.L7
        lw      a5,-20(s0)
        addi    a5,a5,1
        sw      a5,-20(s0)
```

➢ Assembly Test Program

➢ a4 = 5, a5 = 0

➢ n = n – 1 = 4



➢ 0 < 4 ?

➢ Yes : L7

```
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]);
            }
        }
    }
```

(0 < 4) ? Yes : L7

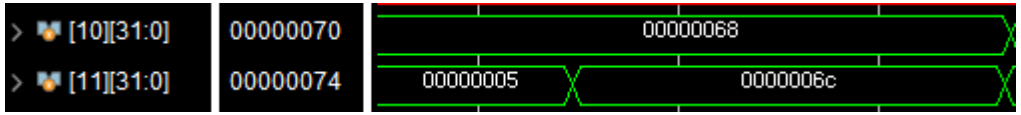
BLT, LW, SUB, ADDI

Test Program : Bubble Sort (Cont.)

```
.L7:
        lw      a5,-24(s0)
        slli    a5,a5,2
        lw      a4,-36(s0)
        add     a5,a4,a5
        lw      a4,0(a5)
        lw      a5,-24(s0)
        addi    a5,a5,1
        slli    a5,a5,2
        lw      a3,-36(s0)
        add     a5,a3,a5
        lw      a5,0(a5)
        ble     a4,a5,.L6
        lw      a5,-24(s0)
        slli    a5,a5,2
        lw      a4,-36(s0)
        add     a3,a4,a5
        lw      a5,-24(s0)
        addi    a5,a5,1
        slli    a5,a5,2
        lw      a4,-36(s0)
        add     a5,a4,a5
        mv      a1,a5
        mv      a0,a3
        call    swap
```

- a4 = **5**

- a5 = **4**
- (5 <= 4) ? Ignore .L6



- a0 = 0x68 (mem[26])
- a1 = 0x6C (mem[27])

- Assembly Test Program

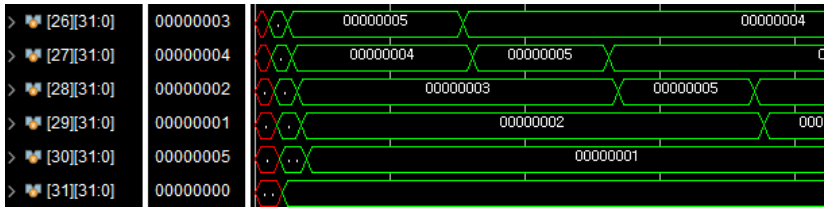
SLLI, ADD, LW, BLE

Test Program : Bubble Sort (Cont.)

```
swap:
        addi    sp,sp,-48
        sw      ra,44(sp)
        sw      s0,40(sp)
        addi    s0,sp,48
        sw      a0,-36(s0)
        sw      a1,-40(s0)
        lw      a5,-36(s0)
        lw      a5,0(a5)
        sw      a5,-20(s0)
        lw      a5,-40(s0)
        lw      a4,0(a5)
        lw      a5,-36(s0)
        sw      a4,0(a5)
        lw      a5,-40(s0)
        lw      a4,-20(s0)
        sw      a4,0(a5)
        nop
        lw      ra,44(sp)
        lw      s0,40(sp)
        addi    sp,sp,48
        jr      ra
```

- Assembly Test Program

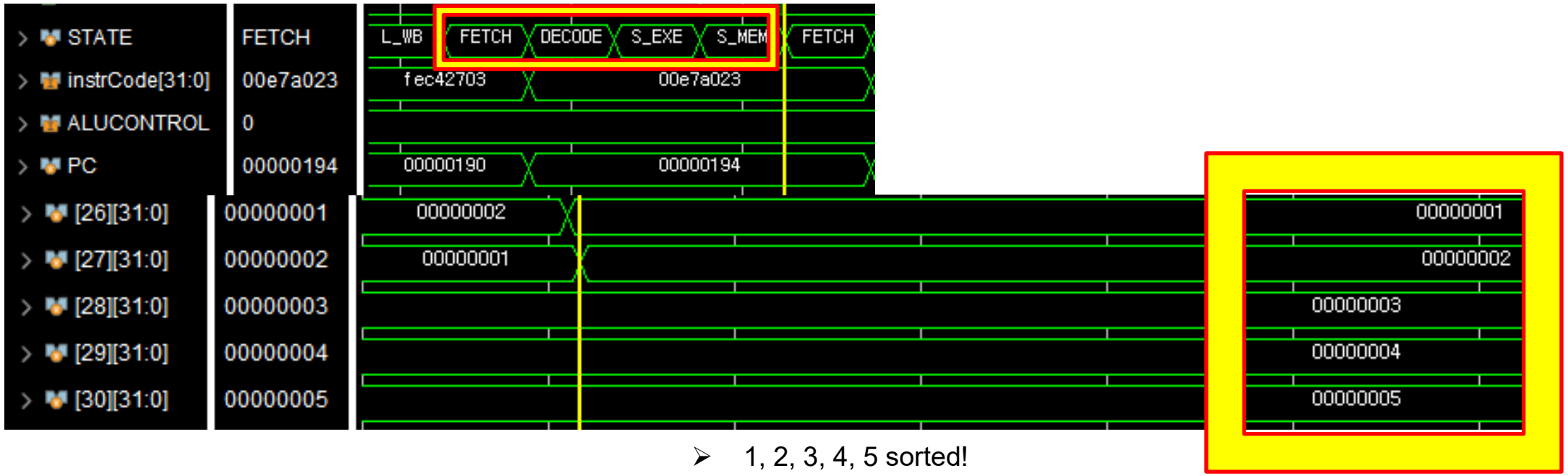
- Store pointers
  - mem[3] ← 0x68 (arr[j] address)
  - mem[2] ← 0x6C (arr[j+1] address)
- temp ← *a0 (arr[j])
  - [0x30-0x14=0x1C] = mem[7] ← 5 (temp=5)
- a4 ← *a1 (arr[j+1])
  - [0x6C] = mem[27] = 4 (arr[j+1])
- *a0 ← a4 (arr[j] = arr[j+1] = 4)
  - [0x68] = mem[26] ← 4
- *a1 ← temp (arr[j+1] = 5)
  - [0x6C] = mem[27] ← 5
- Return to sort...



- Swapped 4 and 5

Swapped 4 and 5

Test Program : Bubble Sort (Cont.)



- 1, 2, 3, 4, 5 sorted!

Program works well

Conclusion

- **Objectives and Achieved Results**

Instruction Set	Objective	Implemented
R - Type	10	10
L - Type	5	5
I - Type	9	9
S - Type	3	3
B - Type	6	6
LU,AU,JAL,JL - Type	4	4

  - **All the assigned tasks have been successfully implemented.**

- **Limitations and Future Development**
  - Pipelining
    - Future work includes introducing pipelining (2–5 stages such as IF/ID/EX/MEM/WB).This will improve instruction throughput and overall execution efficiency by overlapping stages.
  - M Extension
    - By implementing the RISC-V M extension, instructions like MUL, DIV, and REM will be supported. This adds frequently used arithmetic operations, enhancing the CPU's functionality and usability.

Trouble Shooting

- **Issue**
  - While writing assembly programs, the stack pointer (sp) often pointed to an invalid or overlapping region.
  - When adding new instruction types, it was difficult to determine how to modify the datapath.

- **Cause**
  - The RAM size was not fully considered, and the sp value was assigned arbitrarily without checking memory boundaries.
  - It was unclear where to insert MUXes and which data paths should be used.

- **Fix**
  - Investigated the available RAM size first, then calculated an appropriate initial sp value within valid memory space.
  - Adjusted the datapath by carefully deciding MUX positions and data routes, ensuring both data flow correctness and control timing.

- **Lesson Learned**
  - Always check RAM capacity before assigning sp, and reserve safe stack space to prevent memory corruption.
  - Careful consideration of data flow and control timing is essential when extending the datapath with new instructions.

Insights after finishing the project

- **From Single-Cycle to FSM**
  - At first, I could not fully understand how instructions execute in a single-cycle CPU. After restructuring the design into FSM stages (Fetch, Decode, Execute, Memory, Writeback), I gained clarity. Writing test programs and verifying each instruction step by step helped me truly grasp the CPU operation.

- **Future Expectations**
  - This project will help me better understand microcontrollers (MCUs) and their internal structures. It will also make writing low-level programs, such as assembly or hardware control code, easier. I expect these skills to greatly benefit my future studies and practical projects.

Q & A