計算機系統設計

無條件跳躍指令

Mao-Hsu Yen
yenmh@mail.ntou.edu.tw

RISC-V RV32IM INSTRUCTION SET

Unconditional Jump Instruction

Mnemonic, Operands	Description	Implementation
JAL rd, offset	Jump to address and place return address in *rd*.	x[rd] = pc + 4 pc += sext(offset)
JALR rd, rs1, offset	Jump to address and place return address in *rd*.	x[rd] = pc + 4 pc = (x[rs1] + sext(offset)) & 0xFFFFFFFE

※Register x0 can be used as the destination if the result is not required.

※The and 0xFFFFFFFE in the JALR instruction ensures the target address is 2-byte aligned, as required by RISC-V. This prevents jumps to invalid or misaligned addresses, ensuring correct program execution.

Integer Register-Immediate Instruction

Mnemonic, Operands	Description	Implementation
LUI rd, imm	Build 32-bit constants and uses the U-type format. LUI places the U-immediate value in the top 20 bits of the destination register *rd*, filling in the lowest 12 bits with zeros.	x[rd] = sext(immediate) << 12
AUIPC rd, imm	Build pc-relative addresses and uses the U-type format. AUIPC forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the pc, then places the result in register *rd*.	x[rd] = pc + sext(immediate) << 12

RISC-V RV32IM INSTRUCTION SET

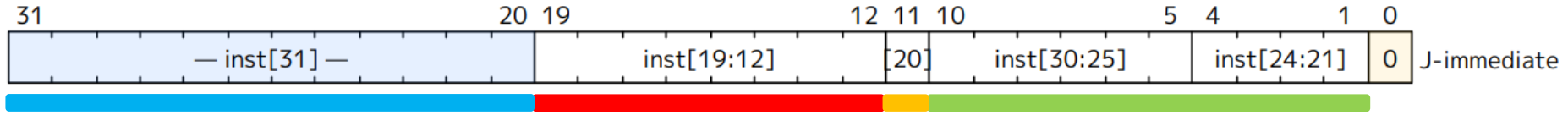
Pseudo-Instruction (假指令)

Mnemonic, Operands	Actual Instruction	Description
J offset	JAL x0, offset	Used for unconditional jumps to a target address.
JR rs1	JALR x0, rs1, 0	Used for unconditional jumps to the address specified by register *rs1*.
RET	JALR x0, ra, 0	Used to return from a subroutine. The default register for *ra* is *x1*.
CALL offset	AUIPC ra, offset[31:12] JALR ra, ra, offset[11:0]	The CALL pseudo-instruction is used to call a subroutine and stores the return address in the *ra* register (default: *x1*). It is implemented using two instructions: AUIPC and JALR. First, AUIPC calculates the upper base address of the target, and then JALR combines this base with the lower offset, performs the jump, and saves the return address in *x1*. This approach enables long-distance jumps while ensuring the program can return to its original flow.
LI rd, imm	ADDI rd, x0, imm	Used to load an immediate value into the register *rd*. When the immediate value exceeds 12 bits, it is split into a series of LUI and ADDI instructions.
	LUI rd, imm[31:12] ADDI rd, rd, imm[11:0]	

RISC-V RV32IM INSTRUCTION SET

Unconditional Jump Instruction

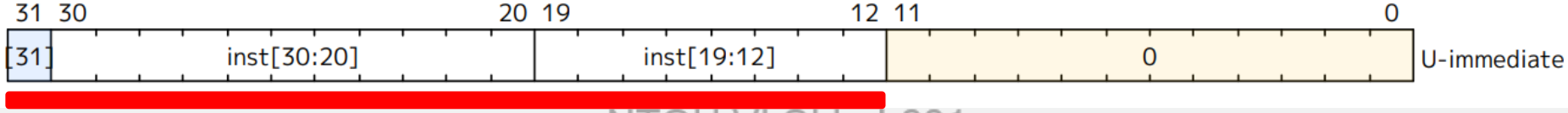
Immediate value	Destination registers	Opcode	Instruction
	rd	1101111	JAL



Immediate value	Source registers 1	Funct3	Destination registers	Opcode	Instruction
imm[11:0]	rs1	000	Rd	1100111	JALR

Integer Register-Immediate Instruction

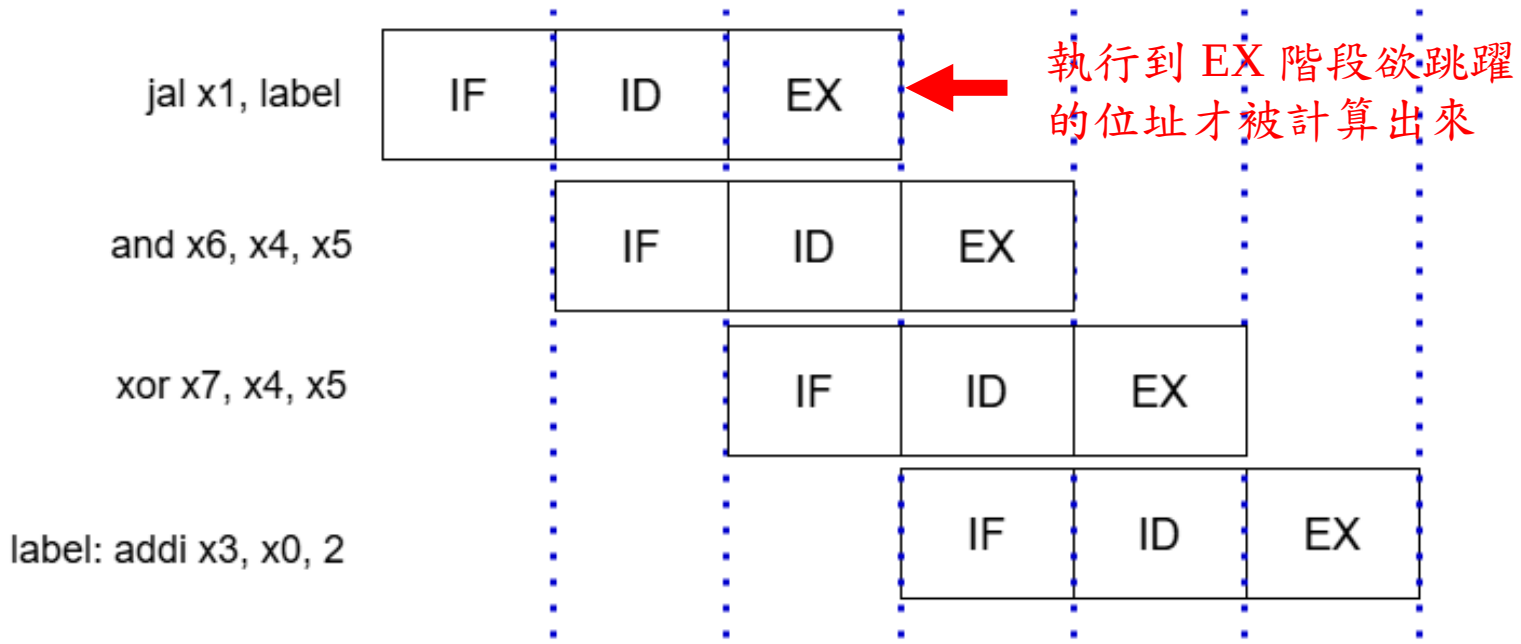
Immediate value	Destination registers	Opcode	Instruction
imm[31:12]	rd	0110111	LUI
imm[31:12]	rd	0010111	AUIPC



控制危障 (Control Hazards)

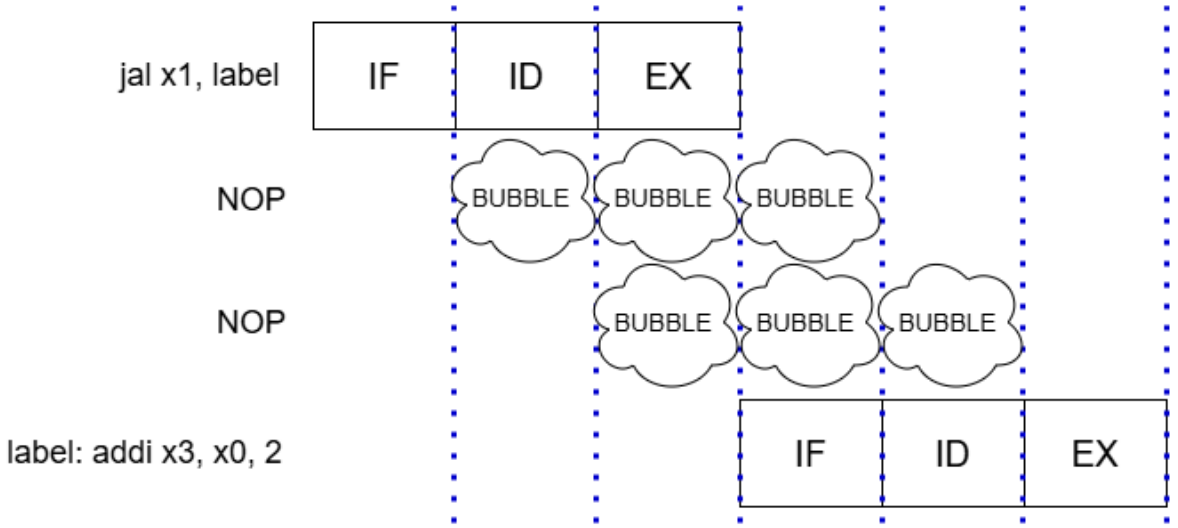
- **Control Hazards**
  - 由跳躍相關指令所引發的危障，如下圖所示，第一條指令 (jal x2, label) 為無條件跳躍指令，正常來說下一個執行的指令應該是要位於 label 位址的，但由於管線化的關係第二第三筆指令已經被截取出來，造成指令執行的順序錯誤。

指令位址	指令
0x10	jal x1, label
0x14	and x6, x4, x5
0x18	xor x7, x4, x5
0x1C	andi x8, x4, 1
. . .	正常的程式執行順序
0xEC	label: andi x3, x0, 2
0xF0	Jalr x0, x1, 0

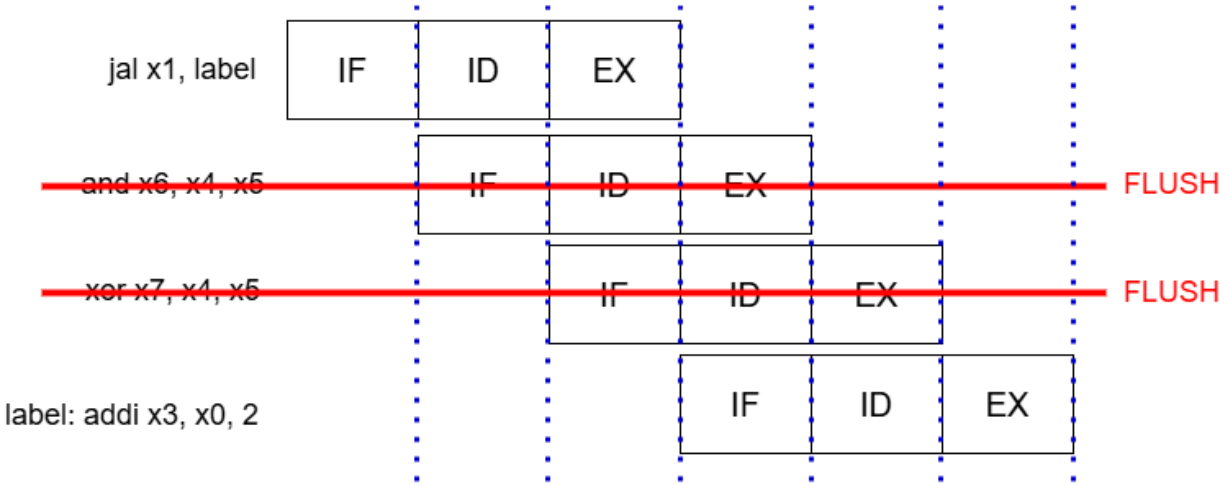


控制危障解決方法

- **軟體方面**
  - 在跳躍相關指令後加入兩個 NOP 指令達成兩個時脈延遲的效果。
- **硬體方面**
  - 透過清除 (flsuh) 兩個管線內的資料使其變成 NOP 指令，以此達成兩個時脈延遲的效果。



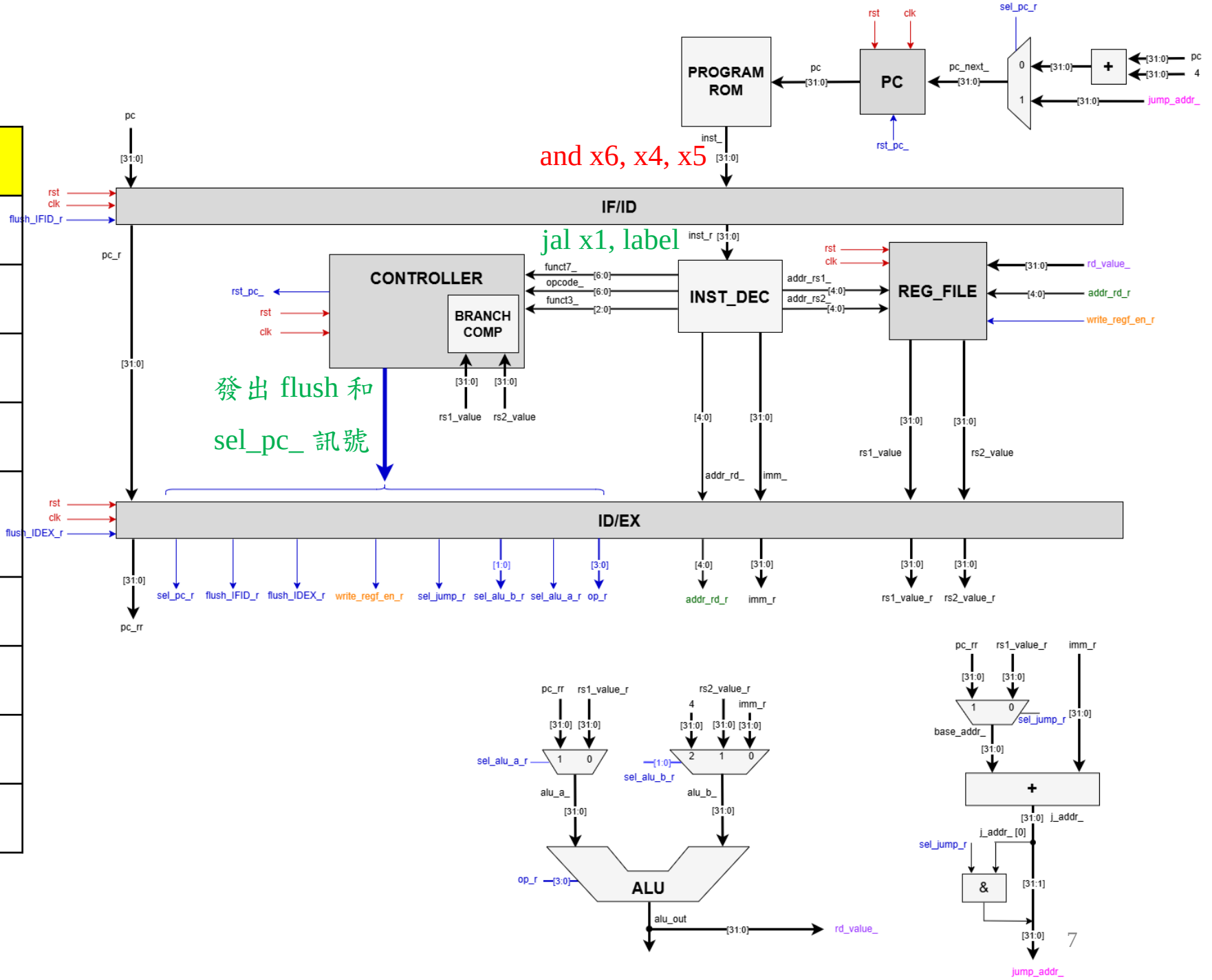
軟體方面



硬體方面

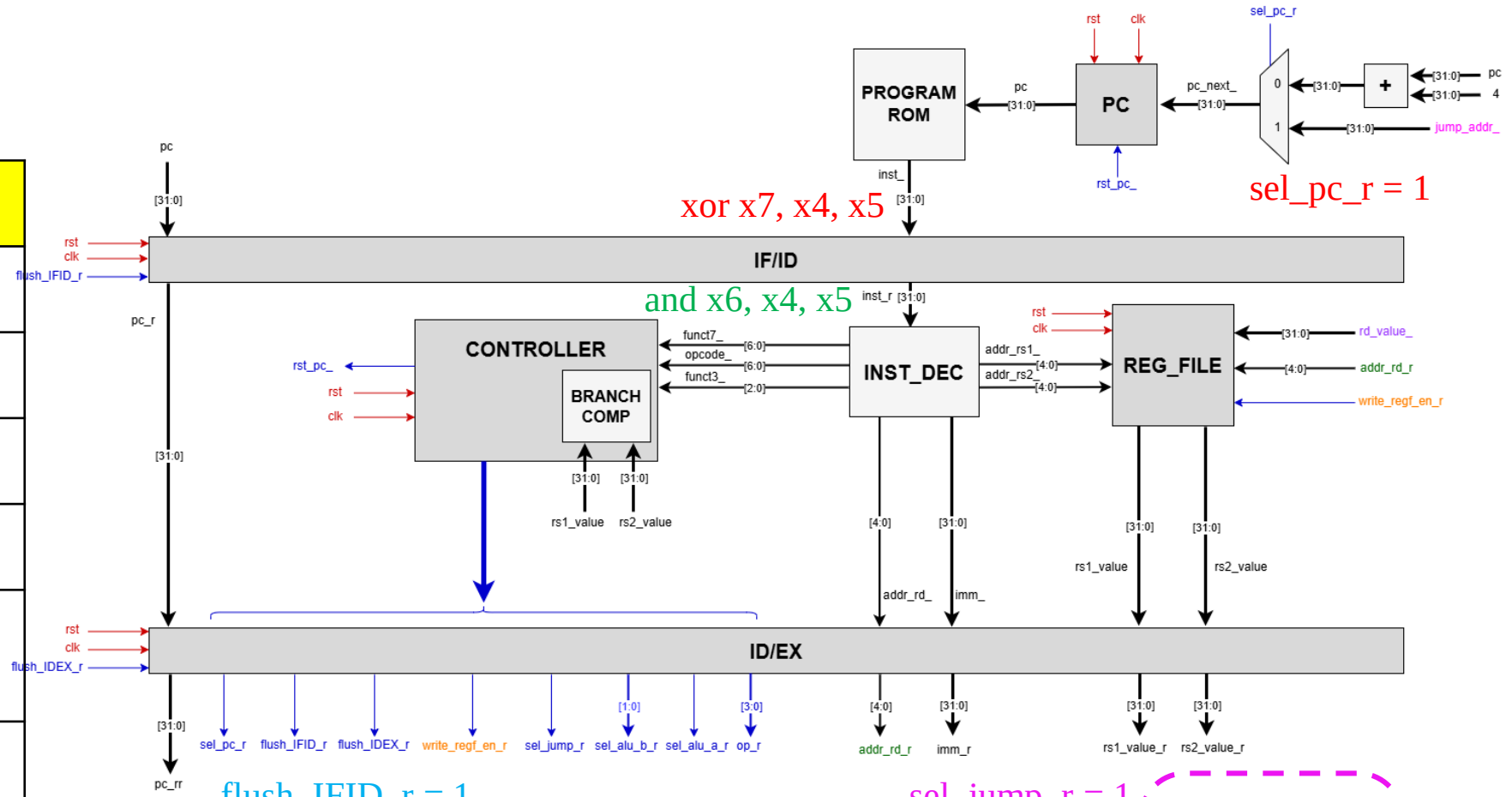
範例(1/7)

指令位址	指令
0x10	jal x1, label
0x14	and x6, x4, x5
0x18	xor x7, x4, x5
0x1C	andi x8, x4, 1
. . .	
0xEC	label: andi x3, x0, 2
0xF0	Jalr x0, x1, 0
0xF4	xor x7, x4, x5
0xF8	andi x8, x4, 1



範例(2/7)

指令位址	指令
0x10	jal x1, label
0x14	and x6, x4, x5
0x18	xor x7, x4, x5
0x1C	andi x8, x4, 1
. . .	
0xEC	label: andi x3, x0, 2
0xF0	Jalr x0, x1, 0
0xF4	xor x7, x4, x5
0xF8	andi x8, x4, 1



flush_IFID_r = 1

flush_IDEX_r = 1

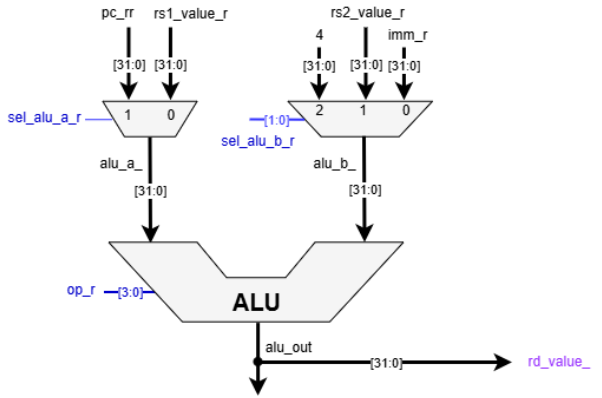
sel_pc_r = 1

sel_alu_a_r = 1

sel_alu_b_r = 2

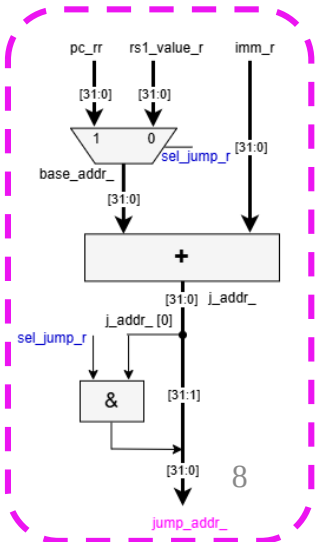
op_r = `ALUOP_ADD

write_regf_en_r = 1



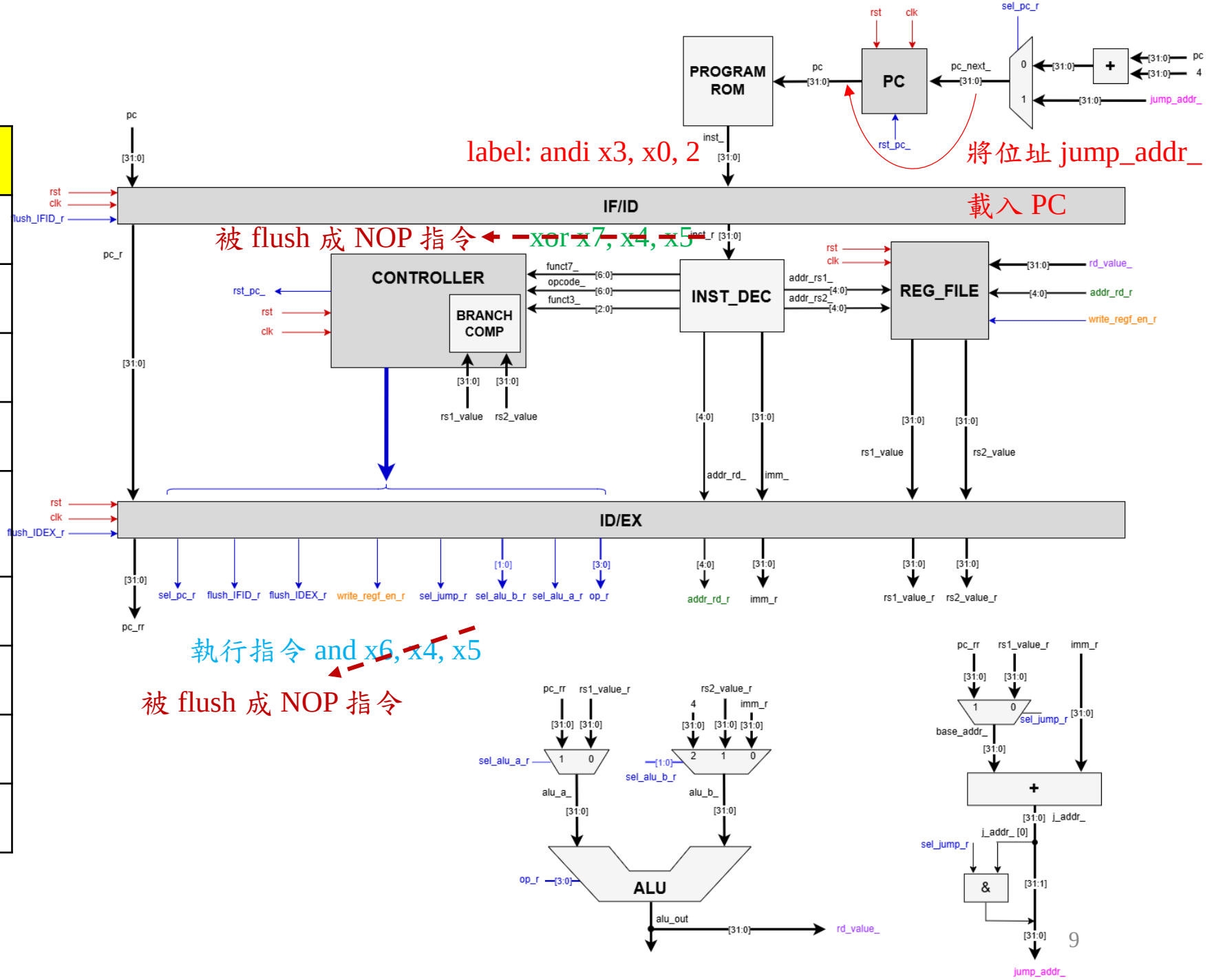
執行指令 jal x1, label

sel_jump_r = 1



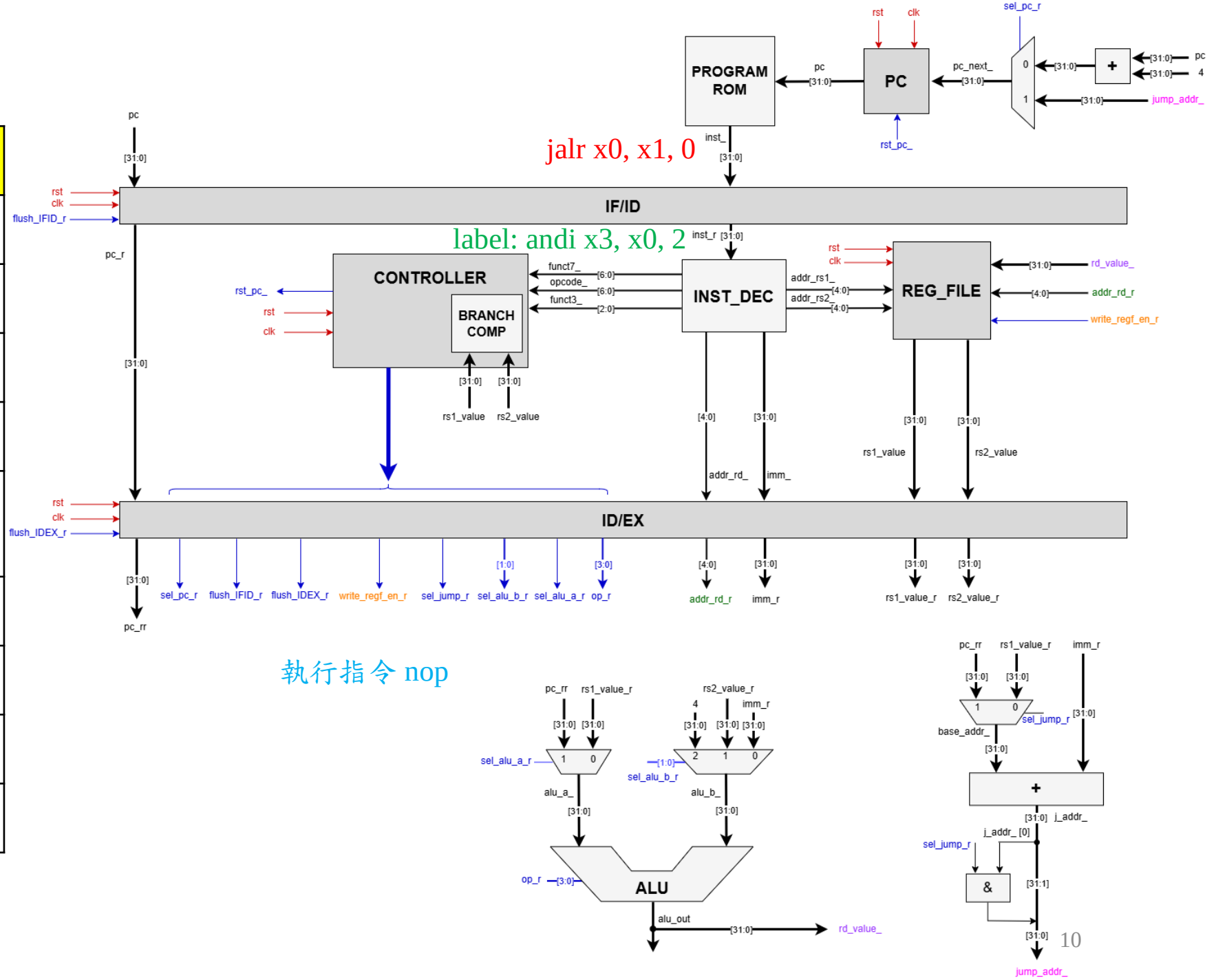
範例(3/7)

指令位址	指令
0x10	jal x1, label
0x14	and x6, x4, x5
0x18	xor x7, x4, x5
0x1C	andi x8, x4, 1
. . .	
0xEC	label: andi x3, x0, 2
0xF0	Jalr x0, x1, 0
0xF4	xor x7, x4, x5
0xF8	andi x8, x4, 1



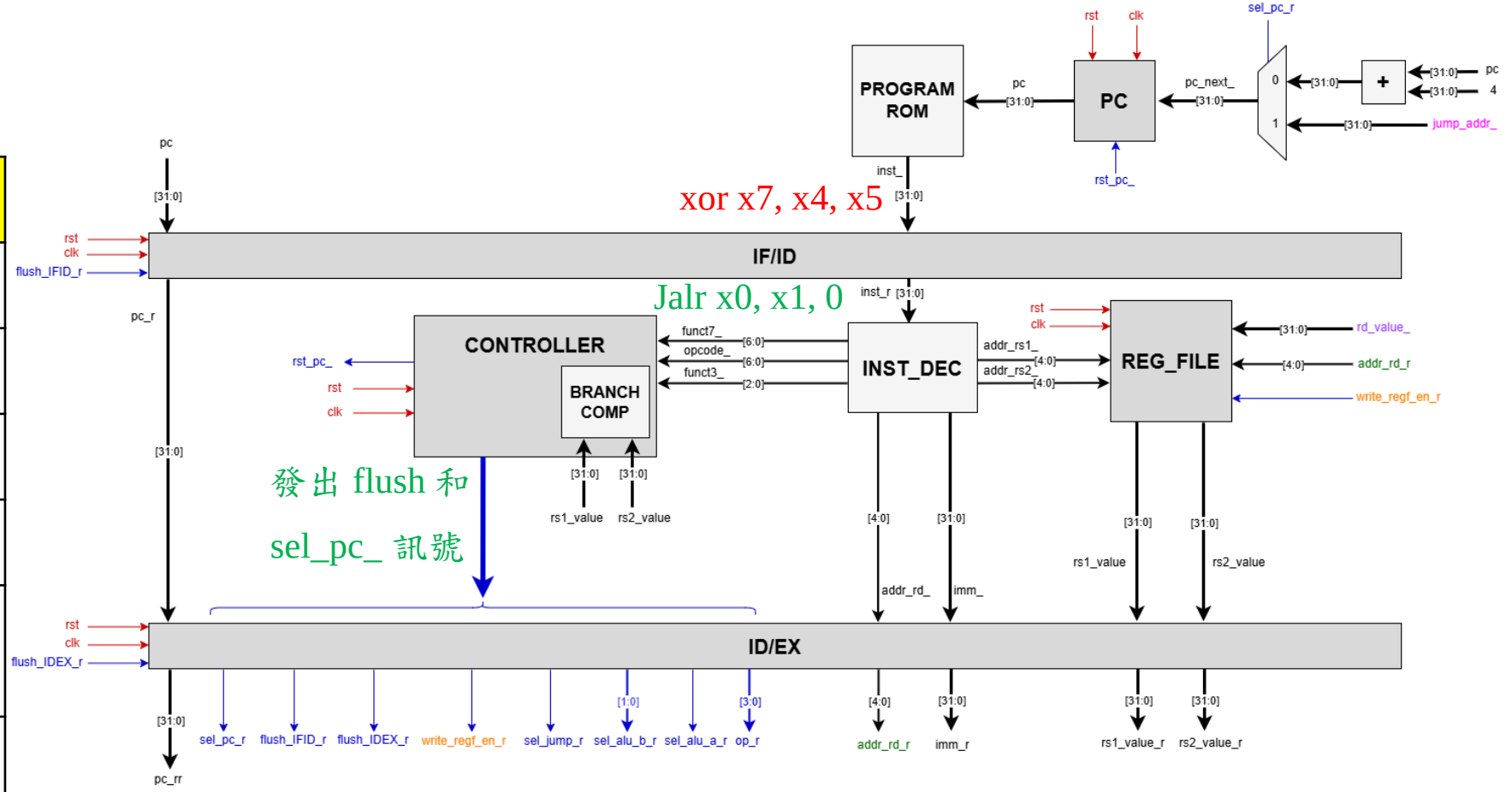
範例(4/7)

指令位址	指令
0x10	jal x1, label
0x14	and x6, x4, x5
0x18	xor x7, x4, x5
0x1C	andi x8, x4, 1
. . .	
0xEC	label: andi x3, x0, 2
0xF0	Jalr x0, x1, 0
0xF4	xor x7, x4, x5
0xF8	andi x8, x4, 1

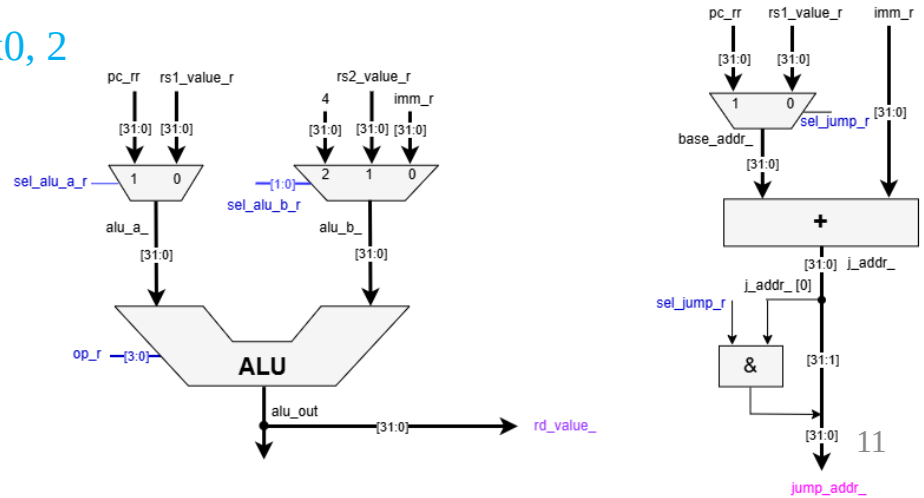


範例(5/7)

指令位址	指令
0x10	jal x1, label
0x14	and x6, x4, x5
0x18	xor x7, x4, x5
0x1C	andi x8, x4, 1
. . .	
0xEC	label: andi x3, x0, 2
0xF0	Jalr x0, x1, 0
0xF4	xor x7, x4, x5
0xF8	andi x8, x4, 1

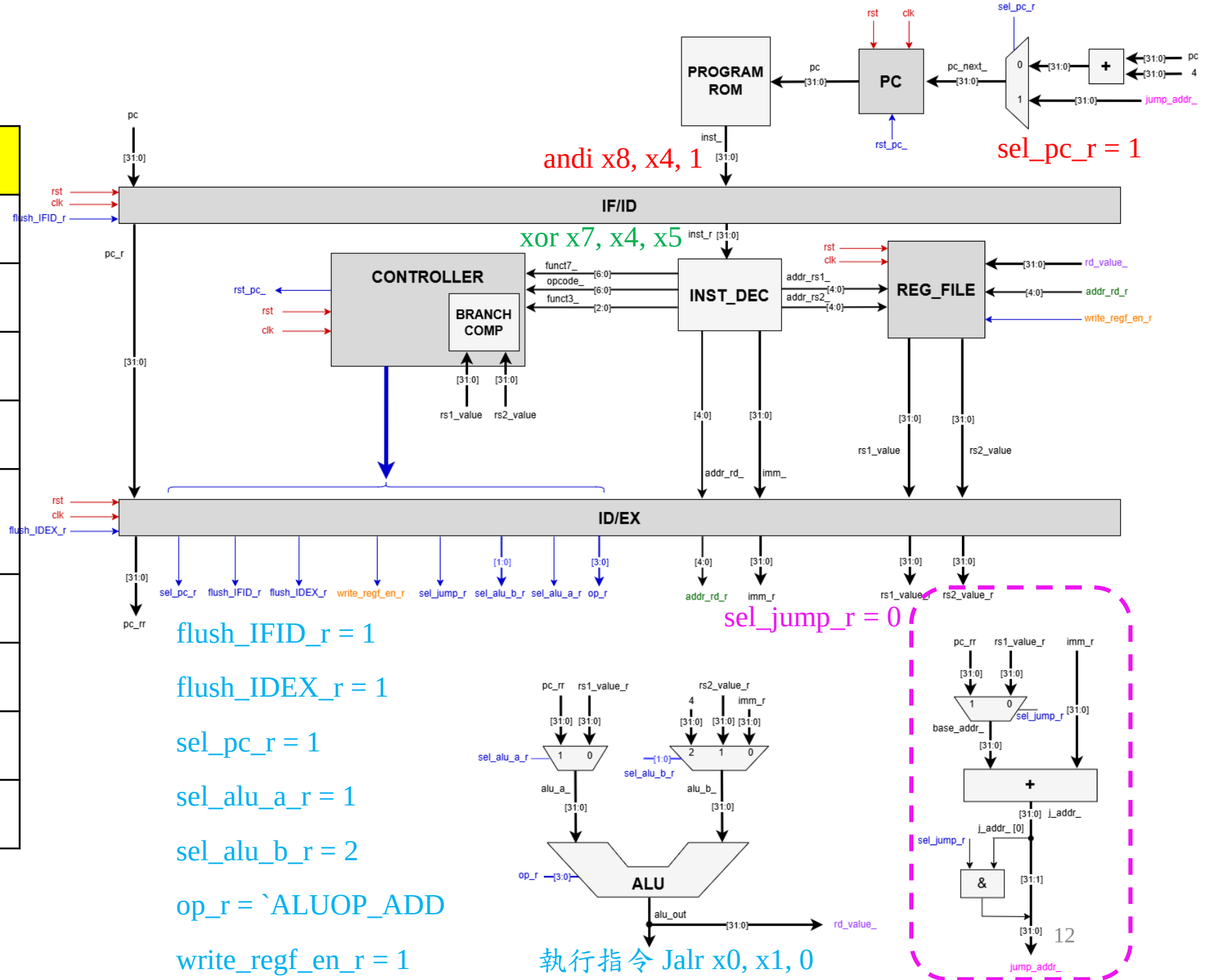


執行指令 label: andi x3, x0, 2



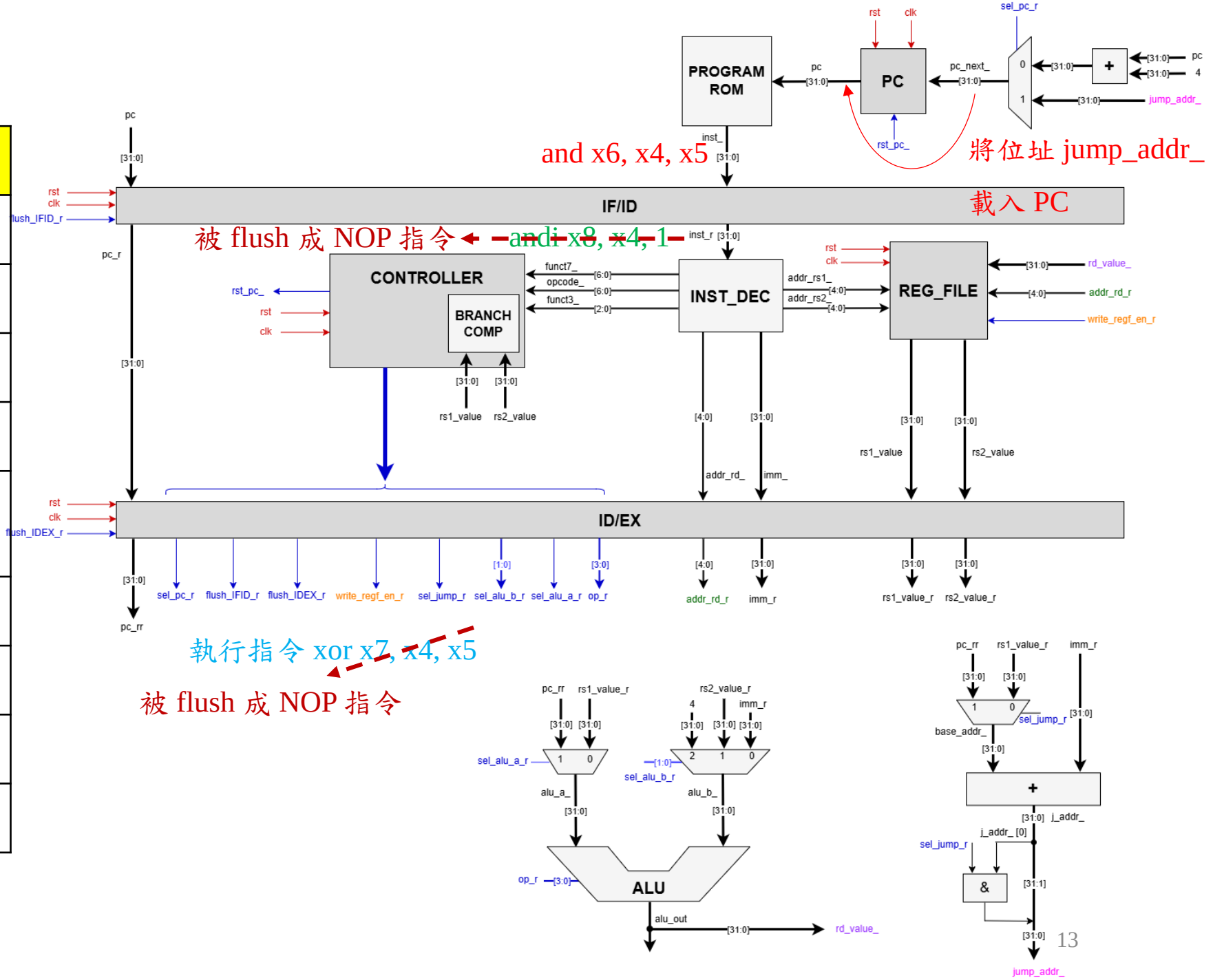
範例(6/7)

指令位址	指令
0x10	jal x1, label
0x14	and x6, x4, x5
0x18	xor x7, x4, x5
0x1C	andi x8, x4, 1
. . .	
0xEC	label: andi x3, x0, 2
0xF0	Jalr x0, x1, 0
0xF4	xor x7, x4, x5
0xF8	andi x8, x4, 1



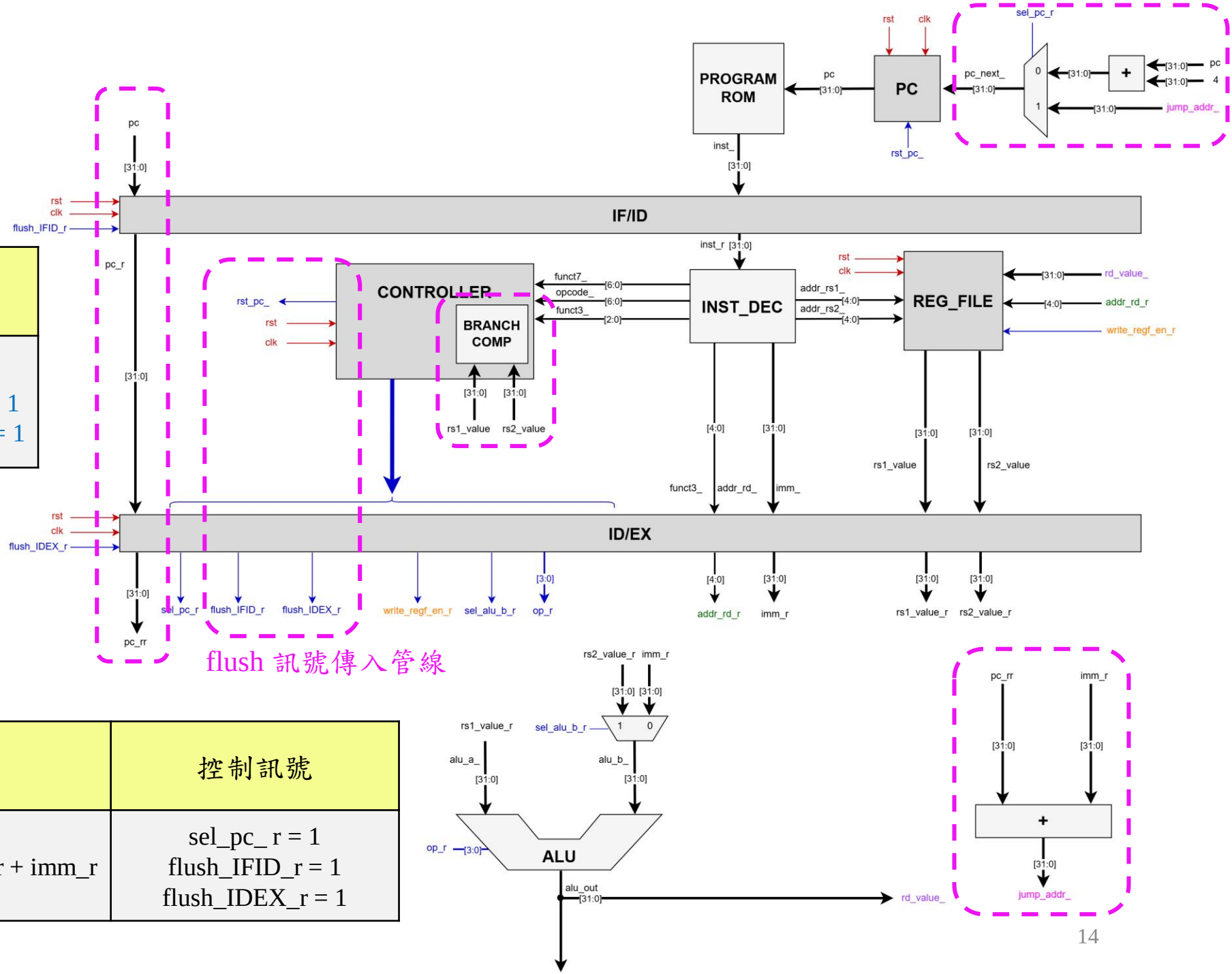
範例(7/7)

指令位址	指令
0x10	jal x1, label
0x14	and x6, x4, x5
0x18	xor x7, x4, x5
0x1C	andi x8, x4, 1
. . .	
0xEC	label: andi x3, x0, 2
0xF0	Jalr x0, x1, 0
0xF4	xor x7, x4, x5
0xF8	andi x8, x4, 1



上周完成之架構

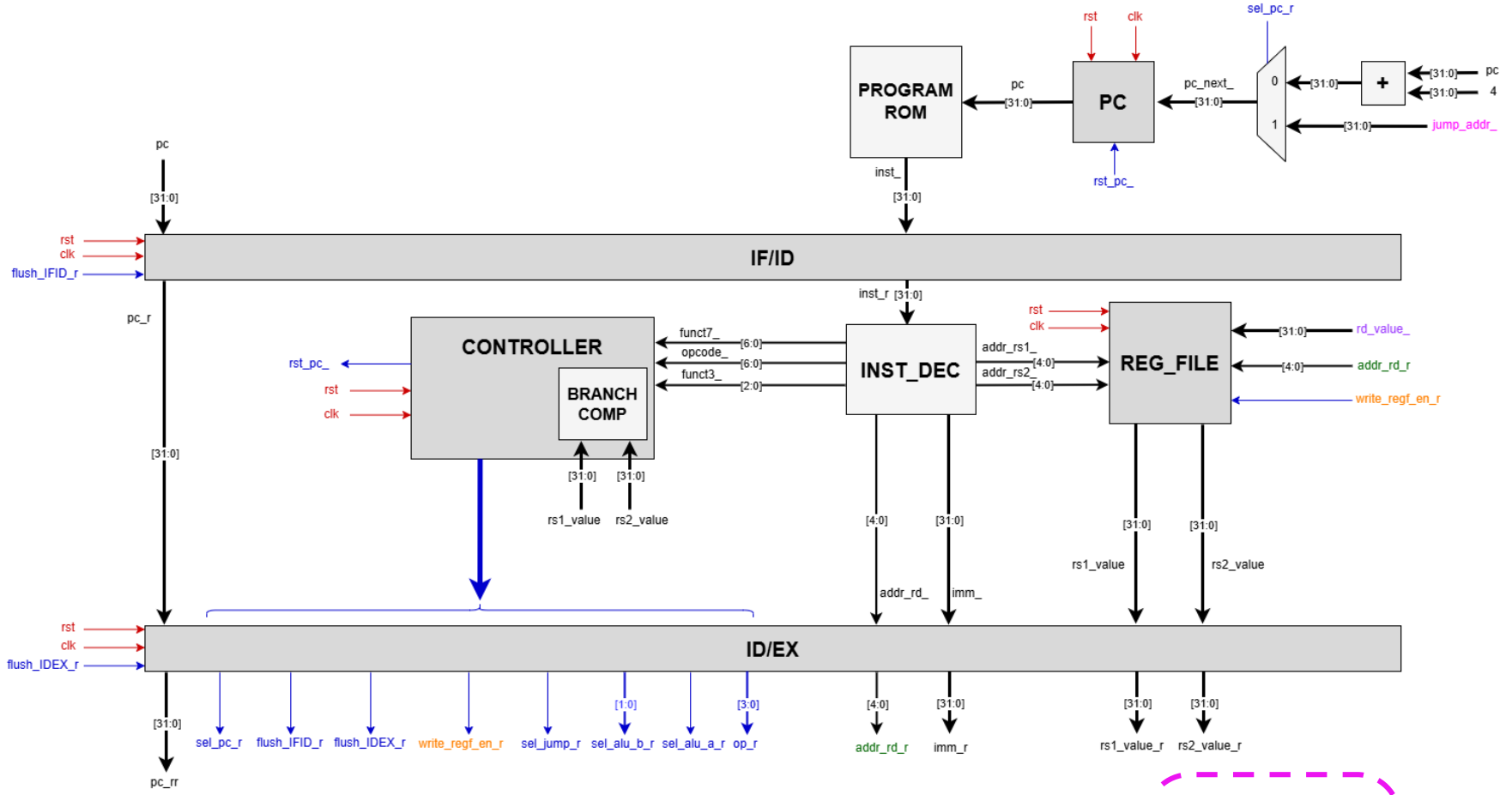
動作	控制訊號
if ((opcode_ == `Opcode_B) && (funct3_ == `F_BEQ) && BEQ_FLAG) 發出 BEQ 的控制訊號	sel_pc_ = 1 flush_IFID_ = 1 flush_IDEX_ = 1



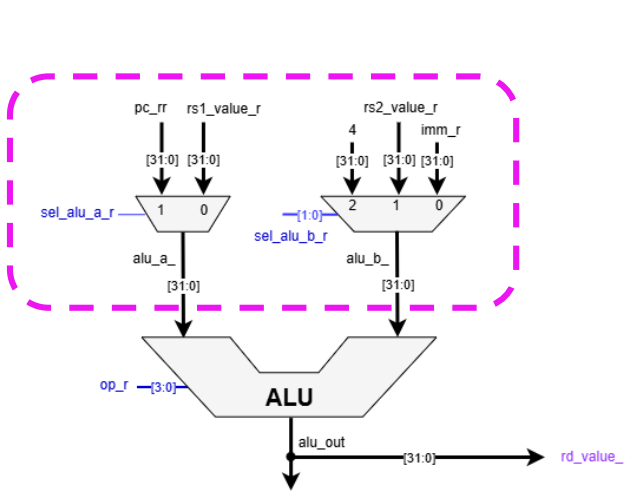
動作	控制訊號
pc_next_ ← pc_rr + imm_r	sel_pc_ r = 1 flush_IFID_r = 1 flush_IDEX_r = 1

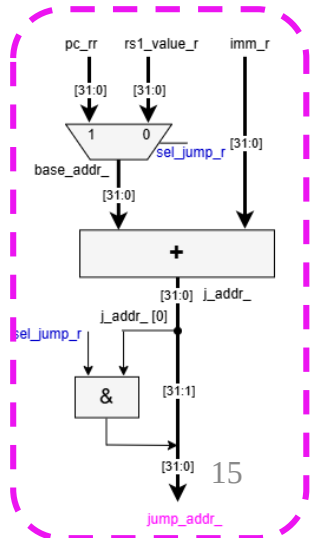
新架構

動作	控制訊號
if (opcode_ == `Opcode_JAL) 發出 JAL 的控制訊號	sel_pc_ = 1 flush_IFID_ = 1 flush_IDEX_ = 1 sel_alu_a_ = 1 sel_alu_b_ = 2 sel_jump_ = 1 op_ = `ALUOP_ADD write_regf_en_ = 1

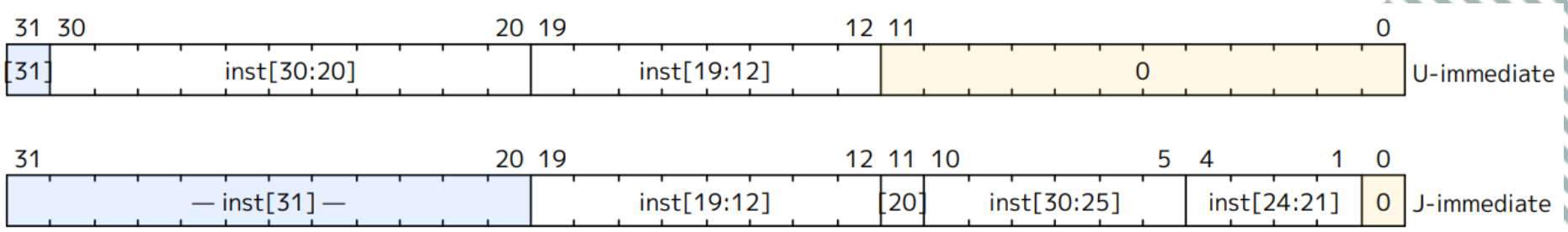


動作	控制訊號
pc_next_ ← pc_rr + imm_r rd_value_ ← pc_rr + 4	sel_pc_r = 1 flush_IFID_r = 1 flush_IDEX_r = 1 sel_alu_a_r = 1 sel_alu_b_r = 2 sel_jump_r = 1 op_r = `ALUOP_ADD write_regf_en_r = 1





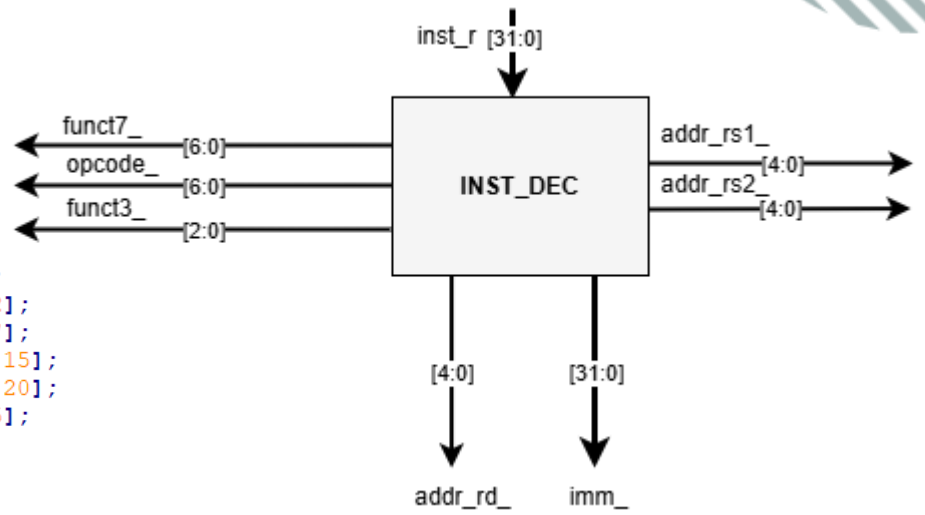
立即值擴充



- **JAL 指令**
  - 將指令 (inst_r) 重組為立即值後做符號擴展 (sign extension) 至 32 位元。

- **JALR 指令**
  - 同立即數定址指令之立即值

- **AUIPC & LUI 指令**
  - 取指令 (inst_r) 的第 12 ~ 31 個位元 (共 20 個位元) 作為立即值較高的 20 個位元。
  - 立即值較低的 12 位元補零。




```
assign opcode_ = inst_r[6:0];
assign funct3_ = inst_r[14:12];
assign addr_rd_ = inst_r[11:7];
assign addr_rs1_ = inst_r[19:15];
assign addr_rs2_ = inst_r[24:20];
assign funct7_ = inst_r[31:25];

logic[31:0] IMM_I;
logic[31:0] IMM_B;
logic[31:0] IMM_JAL;
logic[31:0] IMM_LUI_AUIPC;
assign IMM_I         = {{20{inst_r[31]}}, inst_r[31:20]};
assign IMM_B         = {{20{inst_r[31]}}, inst_r[7], inst_r[30:25], inst_r[11:8], 1'b0};
assign IMM_JAL       = {{12{inst_r[31]}}, inst_r[19:12], inst_r[20], inst_r[30:21], 1'b0};
assign IMM_LUI_AUIPC = {inst_r[31:12], 12'b0};

always_comb begin
    unique case (opcode_)
        `Opcode_I:      imm_ = IMM_I;
        `Opcode_B:      imm_ = IMM_B;
        `Opcode_JAL:    imm_ = IMM_JAL;
        `Opcode_JALR:   imm_ = IMM_I;
        `Opcode_LUI:    imm_ = IMM_LUI_AUIPC;
        `Opcode_AUIPC:  imm_ = IMM_LUI_AUIPC;
    endcase
end
```

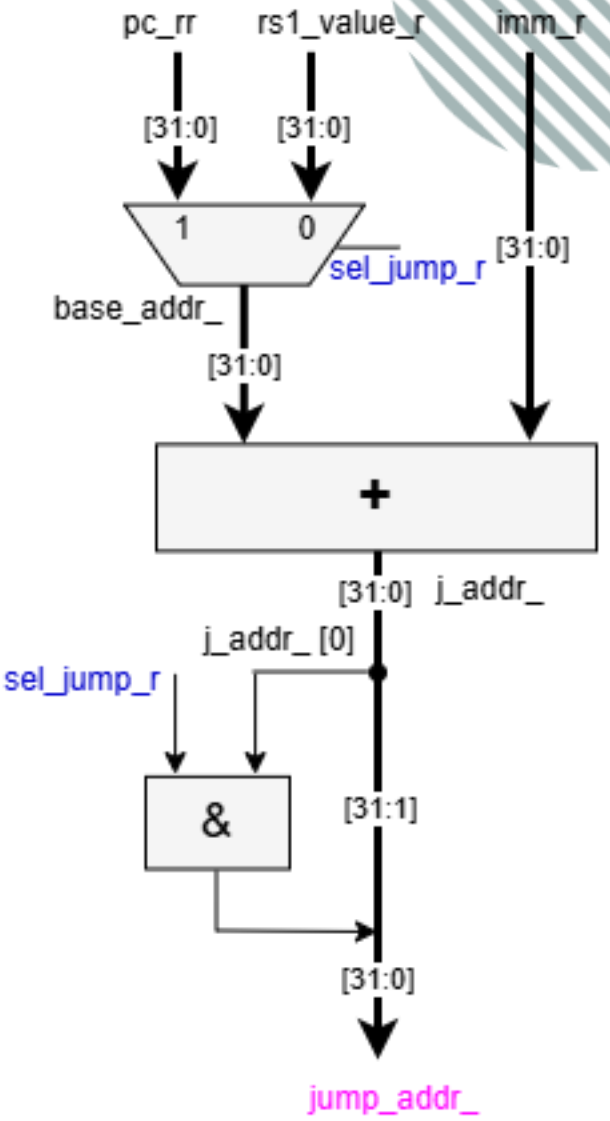
跳躍位址計算

- **JAL & Branch 指令**
  - JAL 和 Branch 指令相同都是以 Program Counter 作為基準位址。

- **JALR 指令**
  - JALR 指令則是以 x[rs1] 作為基準位址。
  - 因為 JALR 指令需要確保其運算出的結果 (j_addr_) 為偶數，因此有了 & 的運算。

※RV32IM 指令集架構中的程式記憶體 (PROM) 每一筆指令的位址皆為 4 的倍數，因此在使用 JALR 需要特別注意 x[rs1] 是否為 4 的倍數。

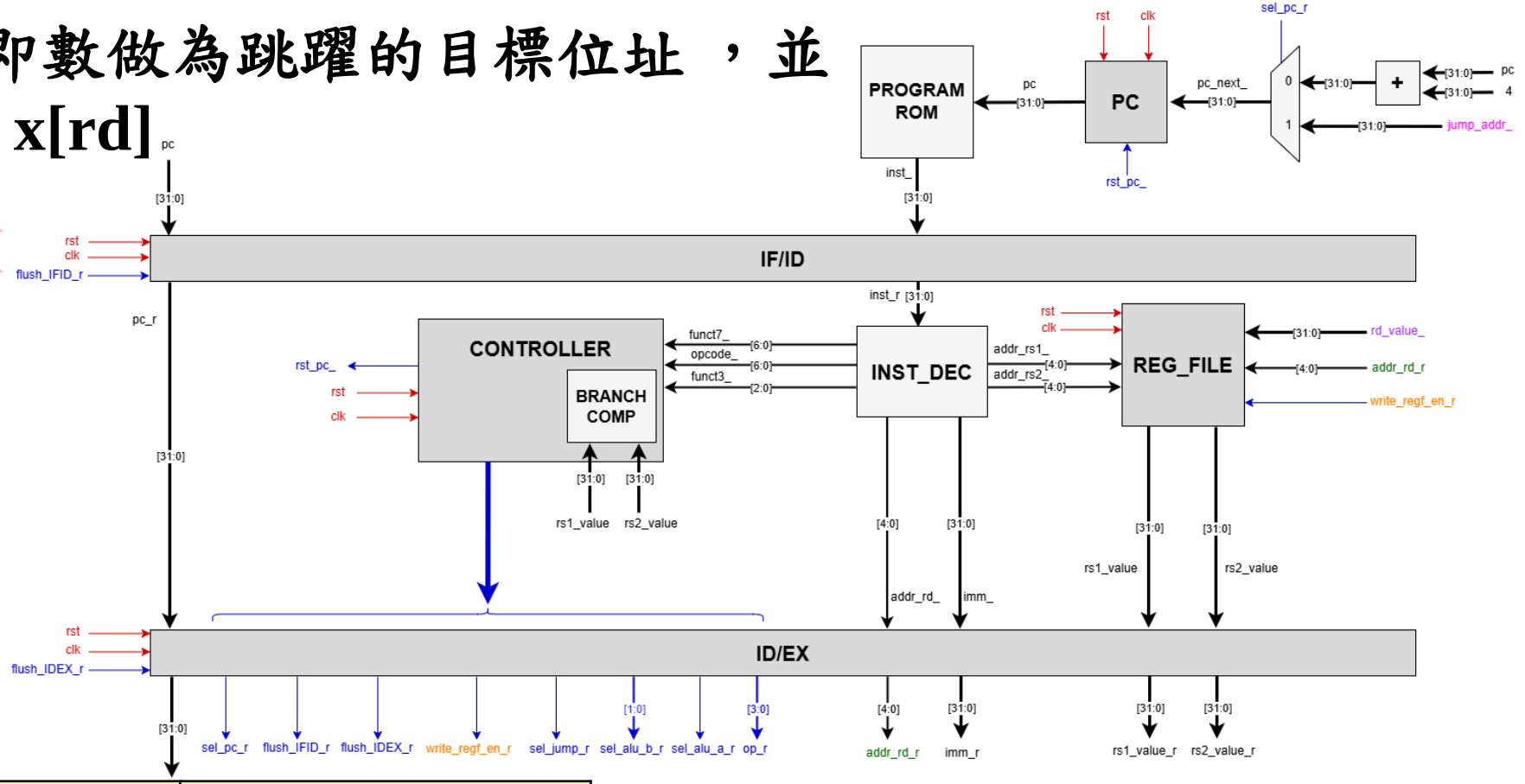
```
//Jump address calculation
//JAL & Branch 以 pc 作為基準位址，JALR 則以 x[rs1] 作為基準
assign base_addr_ = sel_jump_r ? pc_rr : rs1_value_r;
assign j_addr_ = base_addr_ + imm_r;
assign jump_addr_ = {j_addr_[31:1], (j_addr_[0] & sel_jump_r)};
```



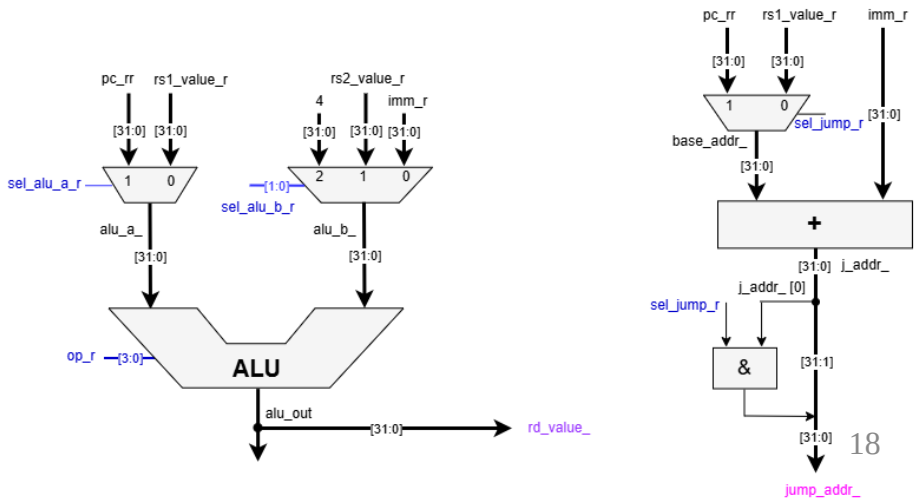
JAL：將 pc_rr 加上立即數做為跳躍的目標位址 ，並將返回位址 pc_rr 存入 x[rd]

```
`define Opcode_JAL      7'b1101111
`define Opcode_JALR     7'b1100111
```

動作	控制訊號
if (opcode_ == `Opcode_JAL) 發出 JAL 的控制訊號	sel_pc_ = 1 flush_IFID_ = 1 flush_IDEX_ = 1 sel_alu_a_ = 1 sel_alu_b_ = 2 sel_jump_ = 1 op_ = `ALUOP_ADD write_regf_en_ = 1

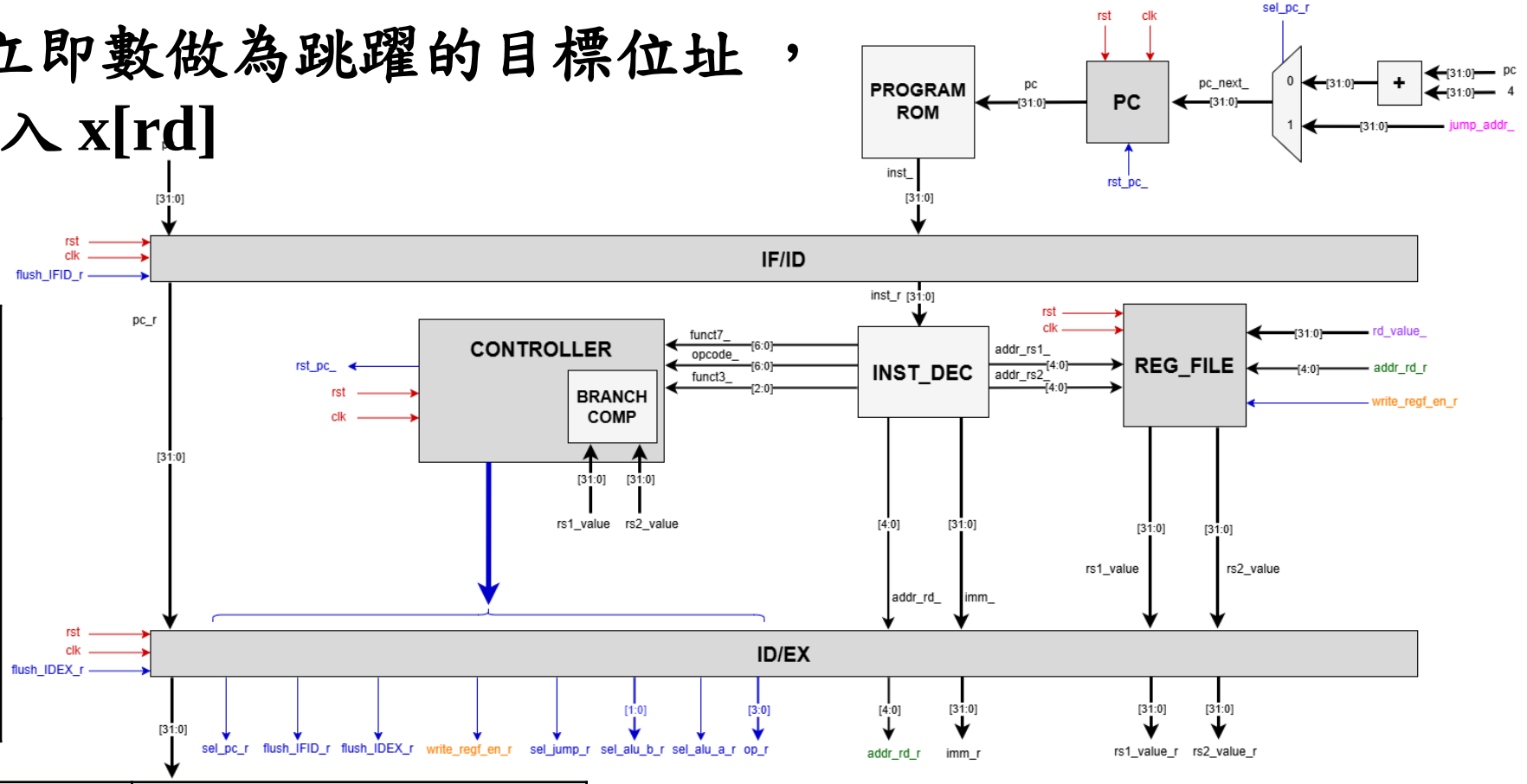


動作	控制訊號
pc_next_ ← pc_rr + imm_r rd_value_ ← pc_rr + 4	sel_pc_r = 1 flush_IFID_r = 1 flush_IDEX_r = 1 sel_alu_a_r = 1 sel_alu_b_r = 2 sel_jump_r = 1 op_r = `ALUOP_ADD write_regf_en_r = 1

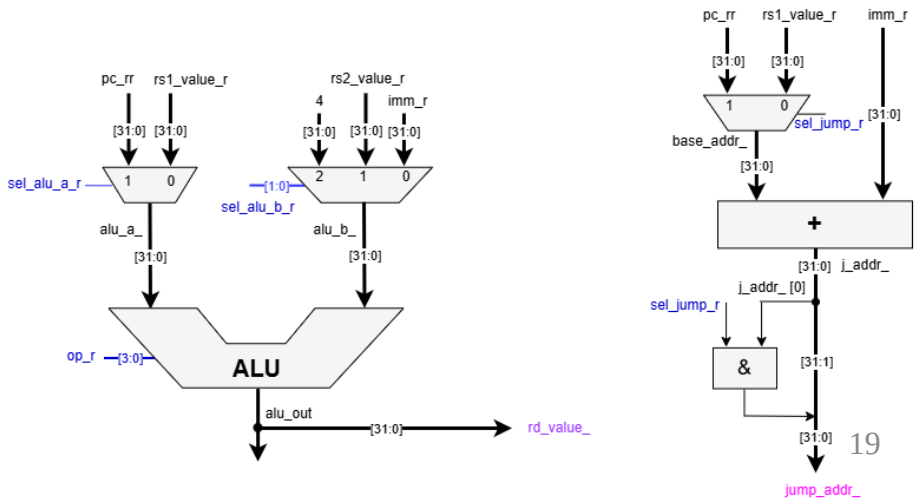


JALR：將 x[rs1] 加上立即數做為跳躍的目標位址，並將返回位址 pc_rr 存入 x[rd]

動作	控制訊號
if (opcode_ == `Opcode_JALR) 發出 JALR 的控制訊號	sel_pc_ = 1 flush_IFID_ = 1 flush_IDEX_ = 1 sel_alu_a_ = 1 sel_alu_b_ = 2 sel_jump_ = 0 op_ = `ALUOP_ADD write_regf_en_ = 1



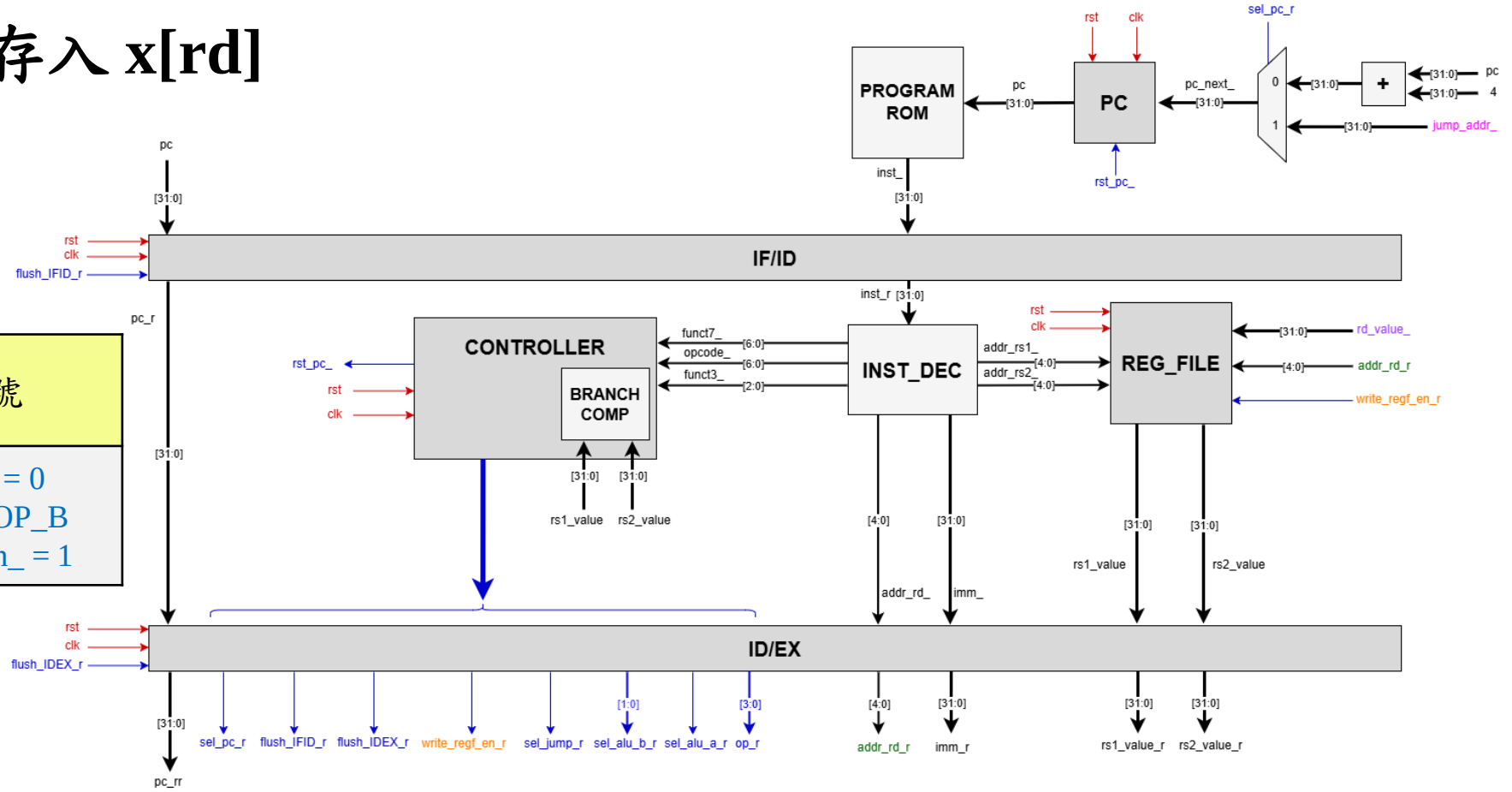
動作	控制訊號
pc_next_ ← rs1_value_r + imm_r rd_value_ ← pc_rr + 4	sel_pc_r = 1 flush_IFID_r = 1 flush_IDEX_r = 1 sel_alu_a_r = 1 sel_alu_b_r = 2 sel_jump_r = 0 op_r = `ALUOP_ADD write_regf_en_r = 1



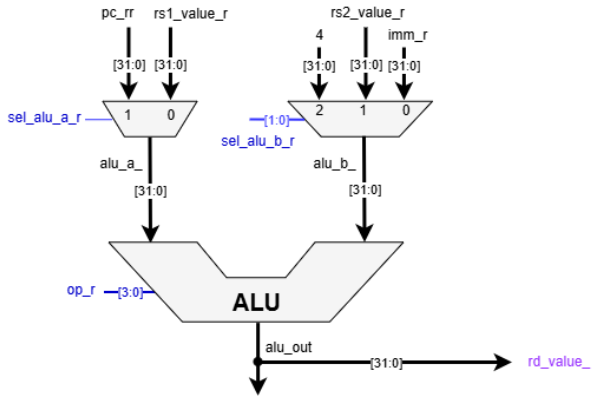
LUI：將高位的立即數存入 x[rd]

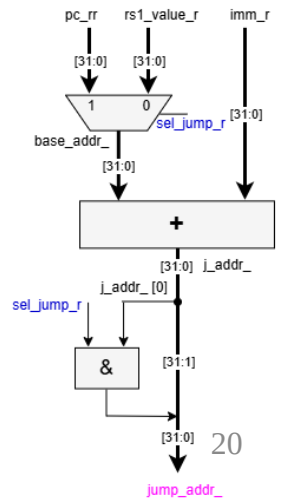
```
`define Opcode_LUI      7'b0110111
`define Opcode_AUIPC    7'b0010111
```

動作	控制訊號
if (opcode_ == `Opcode_LUI) 發出 LUI 的控制訊號	sel_alu_b_ = 0 op_ = `ALUOP_B write_regf_en_ = 1



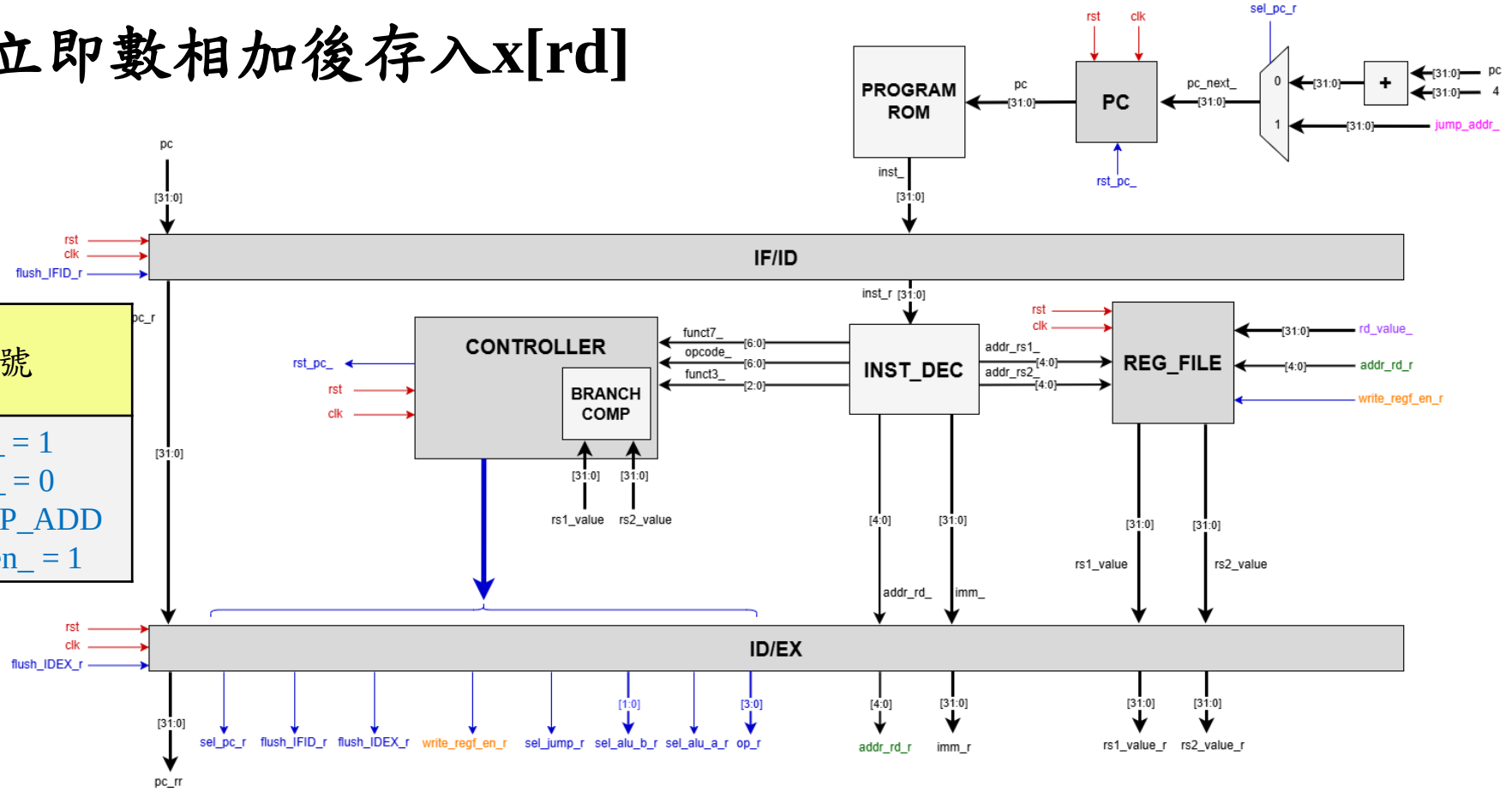
動作	控制訊號
rd_value_ ← imm_r	sel_alu_b_r = 0 op_r = `ALUOP_B write_regf_en_r = 1



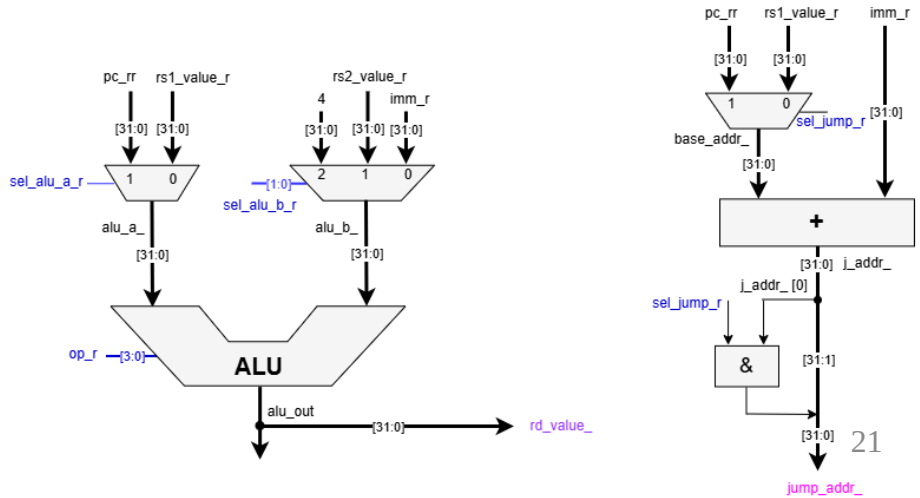


AUIPC：PC 和高位的立即數相加後存入x[rd]

動作	控制訊號
if (opcode_ ==`Opcode_AUIPC) 發出 AUIPC 的控制訊號	sel_alu_a_ = 1 sel_alu_b_ = 0 op_ = `ALUOP_ADD write_regf_en_ = 1



動作	控制訊號
rd_value_ ← pc_rr + imm_r	sel_alu_a_r = 1 sel_alu_b_r = 0 op_r = `ALUOP_ADD write_regf_en_r = 1

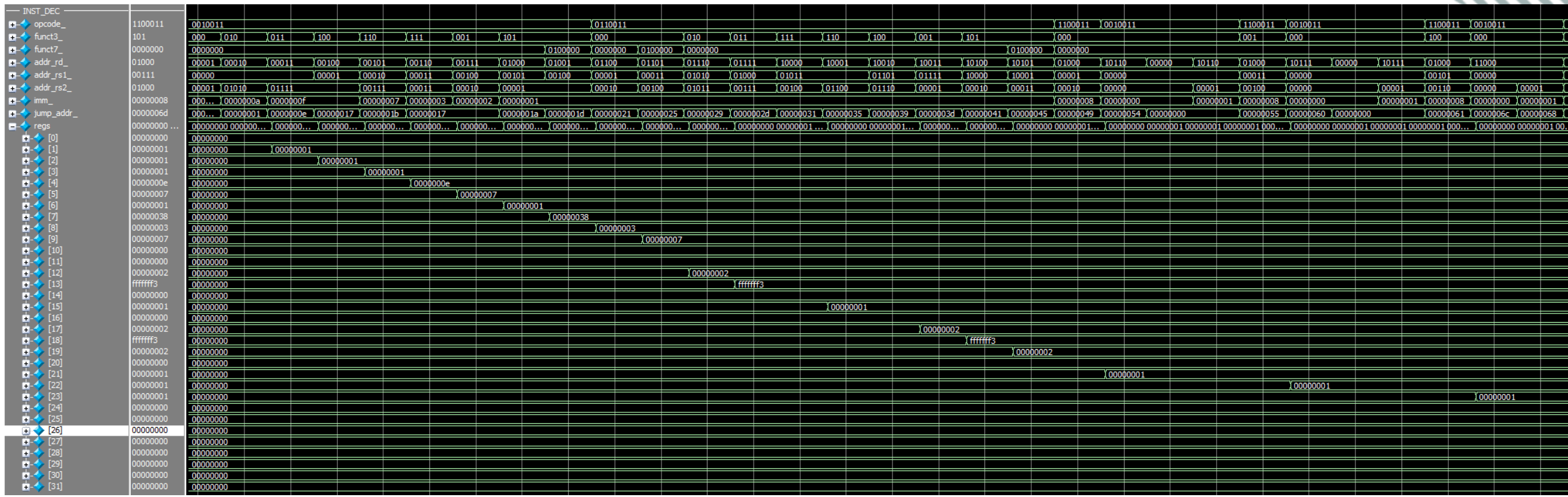


上課實作：ModelSim 模擬

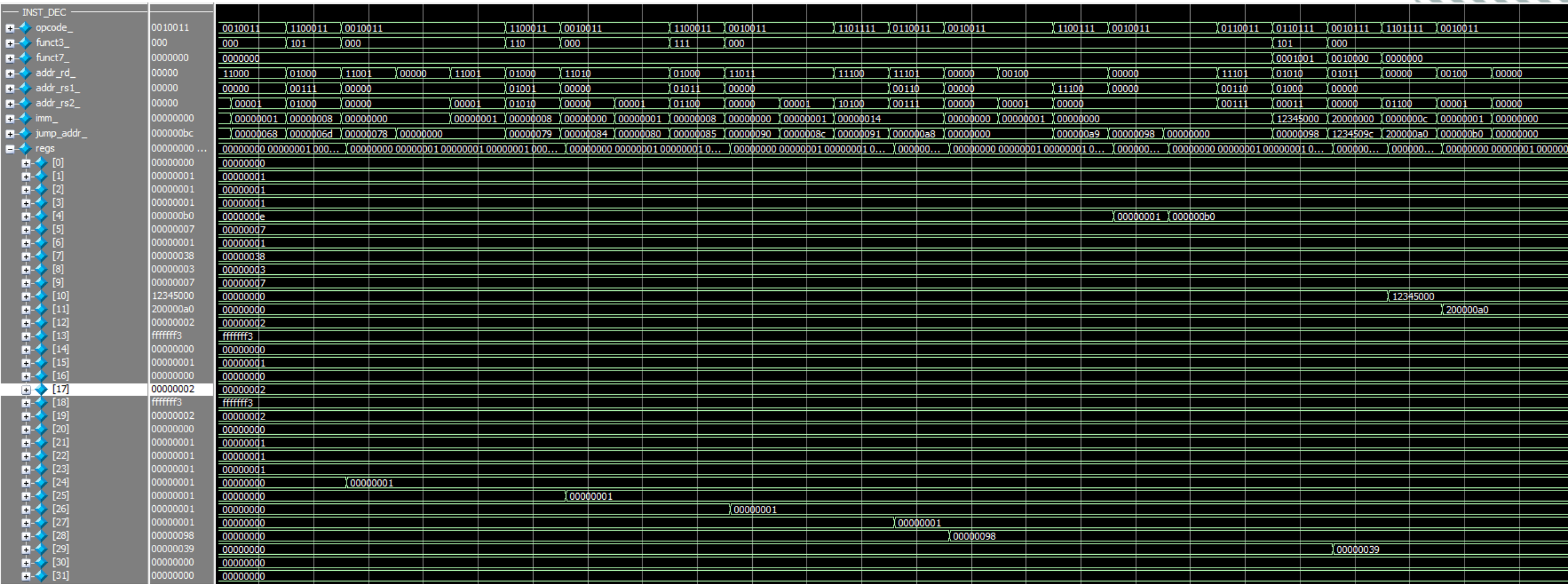
```
#立即值定址
    addi x1, x0, 1
    slti x2, x0, 10
    sltiu x3, x0, 15
    xori x4, x1, 0xF
    ori x5, x2, 0x7
    andi x6, x3, 0x3
    slli x7, x4, 2
    srli x8, x5, 1
    srai x9, x4, 1

#暫存器定址
    add x12, x1, x2
    sub x13, x3, x4
    slt x14, x10, x11
    sltu x15, x8, x7
    and x16, x11, x4
    or  x17, x11, x12
    xor x18, x13, x14
    sll x19, x15, x1
    srl x20, x16, x2
    sra x21, x17, x3

#分支預測
        beq x1, x2, test_beq
        addi x22, x0, 0
    test_beq:
        addi x22, x0, 1
        bne x3, x4, test_bne
        addi x23, x0, 0
    test_bne:
        addi x23, x0, 1
        blt x5, x6, test_blt
        addi x24, x0, 0
    test_blt:
        addi x24, x0, 1
        bge x7, x8, test_bge
        addi x25, x0, 0
    test_bge:
        addi x25, x0, 1
        bltu x9, x10, test_bltu
        addi x26, x0, 0
    test_bltu:
        addi x26, x0, 1
        bgeu x11, x12, test_bgeu
        addi x27, x0, 0
    test_bgeu:
        addi x27, x0, 1

#無條件跳躍
    jal x28, jump
    add x29, x6, x7
    lui x10, 0x12345
    auipc x11, 0x20000
    j end
jump:
    addi x4, x0, 1
    jalr x4, x28, 0
end:
    nop
```

```
32'h00000000 : Rom_data = 32'h00100093; // addi x1 x0 1
32'h00000004 : Rom_data = 32'h00A02113; // slti x2 x0 10
32'h00000008 : Rom_data = 32'h00F03193; // sltiu x3 x0 15
32'h0000000C : Rom_data = 32'h00F0C213; // xori x4 x1 15
32'h00000010 : Rom_data = 32'h00716293; // ori x5 x2 7
32'h00000014 : Rom_data = 32'h0031F313; // andi x6 x3 3
32'h00000018 : Rom_data = 32'h00221393; // slli x7 x4 2
32'h0000001C : Rom_data = 32'h0012D413; // srli x8 x5 1
32'h00000020 : Rom_data = 32'h40125493; // srai x9 x4 1
32'h00000024 : Rom_data = 32'h00208633; // add x12 x1 x2
32'h00000028 : Rom_data = 32'h404186B3; // sub x13 x3 x4
32'h0000002C : Rom_data = 32'h00B52733; // slt x14 x10 x11
32'h00000030 : Rom_data = 32'h007437B3; // sltu x15 x8 x7
32'h00000034 : Rom_data = 32'h0045F833; // and x16 x11 x4
32'h00000038 : Rom_data = 32'h00C5E8B3; // or x17 x11 x12
32'h0000003C : Rom_data = 32'h00E6C933; // xor x18 x13 x14
32'h00000040 : Rom_data = 32'h001799B3; // sll x19 x15 x1
32'h00000044 : Rom_data = 32'h00285A33; // srl x20 x16 x2
32'h00000048 : Rom_data = 32'h4038DAB3; // sra x21 x17 x3
32'h0000004C : Rom_data = 32'h00208463; // beq x1 x2 8
32'h00000050 : Rom_data = 32'h00000B13; // addi x22 x0 0
32'h00000054 : Rom_data = 32'h00100B13; // addi x22 x0 1
32'h00000058 : Rom_data = 32'h00419463; // bne x3 x4 8
32'h0000005C : Rom_data = 32'h00000B93; // addi x23 x0 0
32'h00000060 : Rom_data = 32'h00100B93; // addi x23 x0 1
32'h00000064 : Rom_data = 32'h0062C463; // blt x5 x6 8
32'h00000068 : Rom_data = 32'h00000C13; // addi x24 x0 0
32'h0000006C : Rom_data = 32'h00100C13; // addi x24 x0 1
32'h00000070 : Rom_data = 32'h0083D463; // bge x7 x8 8
32'h00000074 : Rom_data = 32'h00000C93; // addi x25 x0 0
32'h00000078 : Rom_data = 32'h00100C93; // addi x25 x0 1
32'h0000007C : Rom_data = 32'h00A4E463; // bltu x9 x10 8
32'h00000080 : Rom_data = 32'h00000D13; // addi x26 x0 0
32'h00000084 : Rom_data = 32'h00100D13; // addi x26 x0 1
32'h00000088 : Rom_data = 32'h00C5F463; // bgeu x11 x12 8
32'h0000008C : Rom_data = 32'h00000D93; // addi x27 x0 0
32'h00000090 : Rom_data = 32'h00100D93; // addi x27 x0 1
32'h00000094 : Rom_data = 32'h01400E6F; // jal x28 20
32'h00000098 : Rom_data = 32'h00730EB3; // add x29 x6 x7
32'h0000009C : Rom_data = 32'h12345537; // lui x10 74565
32'h000000A0 : Rom_data = 32'h20000597; // auipc x11 131072
32'h000000A4 : Rom_data = 32'h00C0006F; // jal x0 12
32'h000000A8 : Rom_data = 32'h00100213; // addi x4 x0 1
32'h000000AC : Rom_data = 32'h000E0267; // jalr x4 x28 0
32'h000000B0 : Rom_data = 32'h00000013; // addi x0 x0 0
default : Rom_data = 32'h00000013;      // NOP
```

上課實作：ModelSim 模擬



上課實作：ModelSim 模擬



上課實作：FPGA 燒錄

- 將暫存器 x31 的內容 mapping 到 FPGA 的七段顯示器中，透過下面的組合語言完成上數計數器的功能。
- 本次燒錄無須再使用除頻器，而是透過 Delay 副程式來達成除頻的效果。

```
1  init:
2      # 初始化計數器
3      li x31, 0
4
5  loop:
6
7      # 計數器加 1
8      addi x31, x31, 1
9
10     call delay
11
12     # 無窮迴圈
13     j loop
14
15 # 延遲副程式
16 delay:
17     # 外層迴圈次數
18     li x10, 5000       # 外層迴圈次數 (5000)
19 outer_loop:
20     # 內層迴圈次數
21     li x11, 10000      # 內層迴圈次數 (10000)
22     nop
23     nop
24 inner_loop:
25     addi x11, x11, -1 # 內層迴圈遞減
26     nop
27     nop
28     bnez x11, inner_loop # 如果 x11 不為零，繼續內層迴圈
29     addi x10, x10, -1 # 外層迴圈遞減
30     nop
31     nop
32     bnez x10, outer_loop # 如果 x10 不為零，繼續外層迴圈
33
34     ret                # 返回主程式
```

可以看到假指令被組譯成兩種以上的指令

```
1   0x00000000    0x00000F93    addi x31 x0 0
2   0x00000004    0x001F8F93    addi x31 x31 1
3   0x00000008    0x00000317    auipc x6 0
4   0x0000000C    0x00C300E7    jalr x1 x6 12
5   0x00000010    0xFF5FF06F    jal x0 -12
6   0x00000014    0x00001537    lui x10 1
7   0x00000018    0x38850513    addi x10 x10 904
8   0x0000001C    0x000025B7    lui x11 2
9   0x00000020    0x71058593    addi x11 x11 1808
10  0x00000024    0x00000013    addi x0 x0 0
11  0x00000028    0x00000013    addi x0 x0 0
12  0x0000002C    0xFFF58593    addi x11 x11 -1
13  0x00000030    0x00000013    addi x0 x0 0
14  0x00000034    0x00000013    addi x0 x0 0
15  0x00000038    0xFE059AE3    bne x11 x0 -12
16  0x0000003C    0xFFF50513    addi x10 x10 -1
17  0x00000040    0x00000013    addi x0 x0 0
18  0x00000044    0x00000013    addi x0 x0 0
19  0x00000048    0xFC051AE3    bne x10 x0 -44
20  0x0000004C    0x00008067    jalr x0 x1 0
```

上課實作：FPGA 燒錄

- 本次燒錄無須再使用除頻器，而是透過 Delay 副程式來達成除頻的效果。

DE0_CV.sv (作為最上層的模組)

```
=============================================================
REG/WIRE declarations
=============================================================

//如果`define USE_CLOCK_DIVIDER 被註解掉的話
//clk 會被設為 CLOCK_50，否則會被設為 clk_div
//`define USE_CLOCK_DIVIDER
logic [31:0] regs_31;
logic clk;
logic clk_div;

=============================================================
Structural coding
=============================================================

`ifdef USE_CLOCK_DIVIDER
    clock_divider u_clock_divider(
        .clk        (CLOCK_50       ),
        .rst        (~RESET_N       ),
        .DIVISOR    (1_000_000      ),
        //
        .clk_out    (clk_div        )
    );

    assign clk = clk_div;
`else
    assign clk = CLOCK_50;
`endif

RISCV_Core u_Core(
    .clk        (clk        ),
    .rst        (~RESET_N   ),
    //
    .regs_31    (regs_31    )
);

assign LEDR = regs_31[9:0];
```

```
seven_segment_display hex0(
    .digit(regs_31[3:0]),
    //
    .seg(HEX0)
);

seven_segment_display hex1(
    .digit(regs_31[7:4]),
    //
    .seg(HEX1)
);

seven_segment_display hex2(
    .digit(regs_31[11:8]),
    //
    .seg(HEX2)
);

seven_segment_display hex3(
    .digit(regs_31[15:12]),
    //
    .seg(HEX3)
);

seven_segment_display hex4(
    .digit(regs_31[19:16]),
    //
    .seg(HEX4)
);

seven_segment_display hex5(
    .digit(regs_31[23:20]),
    //
    .seg(HEX5)
);
```

七段顯示器

```
module seven_segment_display(
    input  logic [3:0] digit,  // 4位元二進制數字輸入
    output logic [6:0] seg     // 7段顯示器輸出 (g, f, e, d, c, b, a)
);
    //共陽
    always_comb begin
        case (digit)
            4'h0: seg = 7'b1000000;     // 顯示 "0"
            4'h1: seg = 7'b1111001;     // 顯示 "1"
            4'h2: seg = 7'b0100100;     // 顯示 "2"
            4'h3: seg = 7'b0110000;     // 顯示 "3"
            4'h4: seg = 7'b0011001;     // 顯示 "4"
            4'h5: seg = 7'b0010010;     // 顯示 "5"
            4'h6: seg = 7'b0000010;     // 顯示 "6"
            4'h7: seg = 7'b1111000;     // 顯示 "7"
            4'h8: seg = 7'b0000000;     // 顯示 "8"
            4'h9: seg = 7'b0010000;     // 顯示 "9"
            4'ha: seg = 7'b0001000;     // 顯示 "A"
            4'hb: seg = 7'b0000011;     // 顯示 "B"
            4'hc: seg = 7'b1000110;     // 顯示 "C"
            4'hd: seg = 7'b0100001;     // 顯示 "D"
            4'he: seg = 7'b0000110;     // 顯示 "E"
            4'hf: seg = 7'b0001110;     // 顯示 "F"
            default: seg = 7'b1111111;  // 不顯示
        endcase
    end

endmodule
```

THANK YOU