計算機系統設計

暫存器定址

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

RISC-V RV32IM INSTRUCTION SET

Integer Register-Register Instructions

Mnemonic, Operands	Description	Implementation
ADD rd, rs1, rs2	Adds the registers *rs1* and *rs2* and stores the result in *rd*. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result.	x[rd] = x[rs1] + x[rs2]
SUB rd, rs1, rs2	Subs the register *rs2* from *rs1* and stores the result in *rd*. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result.	x[rd] = x[rs1] - x[rs2]
SLT rd, rs1, rs2	Place the value 1 in register *rd* if register *rs1* is less than register *rs2* when both are treated as signed numbers, else 0 is written to *rd*.	x[rd] = x[rs1] $<_s$ x[rs2]
SLTU rd, rs1, rs2	Place the value 1 in register *rd* if register *rs1* is less than register *rs2* when both are treated as unsigned numbers, else 0 is written to *rd*.	x[rd] = x[rs1] $<_u$ x[rs2]
XOR rd, rs1, rs2	Performs bitwise XOR on registers *rs1* and *rs2* and place the result in *rd*.	x[rd] = x[rs1] ^ x[rs2]

RISC-V RV32IM INSTRUCTION SET

Integer Register-Register Instructions

Mnemonic, Operands	Description	Implementation
OR rd, rs1, rs2	Performs bitwise OR on registers *rs1* and *rs2* and place the result in *rd*.	x[rd] = x[rs1] | x[rs2]
AND rd, rs1, rs2	Performs bitwise AND on registers *rs1* and *rs2* and place the result in *rd*.	x[rd] = x[rs1] & x[rs2]
SLL rd, rs1, rs2	Performs logical left shift on the value in register *rs1* by the shift amount held in the lower 5 bits of register *rs2*.	x[rd] = x[rs1] << x[rs2]
SRL rd, rs1, rs2	Performs logical right shift on the value in register *rs1* by the shift amount held in the lower 5 bits of register *rs2*.	x[rd] = x[rs1] $>>_u$ x[rs2]
SRA rd, rs1, rs2	Performs arithmetic right shift on the value in register *rs1* by the shift amount held in the lower 5 bits of register *rs2*.	x[rd] = x[rs1] $>>_s$ x[rs2]

Pseudo-Instruction (假指令)

Mnemonic, Operands	Actual Instruction	Description
NEG rd, rs2	SUB rd, x0, rs2	Used to negate the value of register *rs2* and place the result in *rd*.
SNEZ rd, rs2	SLTU rd, x0, rs2	If the value of register *rs2* is not zero, set the register *rd* to 1; otherwise, set it to 0.

RISC-V RV32IM INSTRUCTION SET

Integer Register-Register Instructions

Funct7	Source registers 2	Source registers 1	Funct3	Destination registers	Opcode	Instruction
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
31 25	24 20	19 15	14 12	11 7	6 0	

RISC-V RV32IM INSTRUCTION SET

- **Assembly to Machine Code**
  - add x1, x2, x3 => 0x003100b3

Funct7	Source registers 2	Source registers 1	Funct3	Destination registers	Opcode	Instruction
0000_000	0_0011	0001_0	000	0000_1	011_0011	ADD
31 25	24 20	19 15	14 12	11 7	6 0	

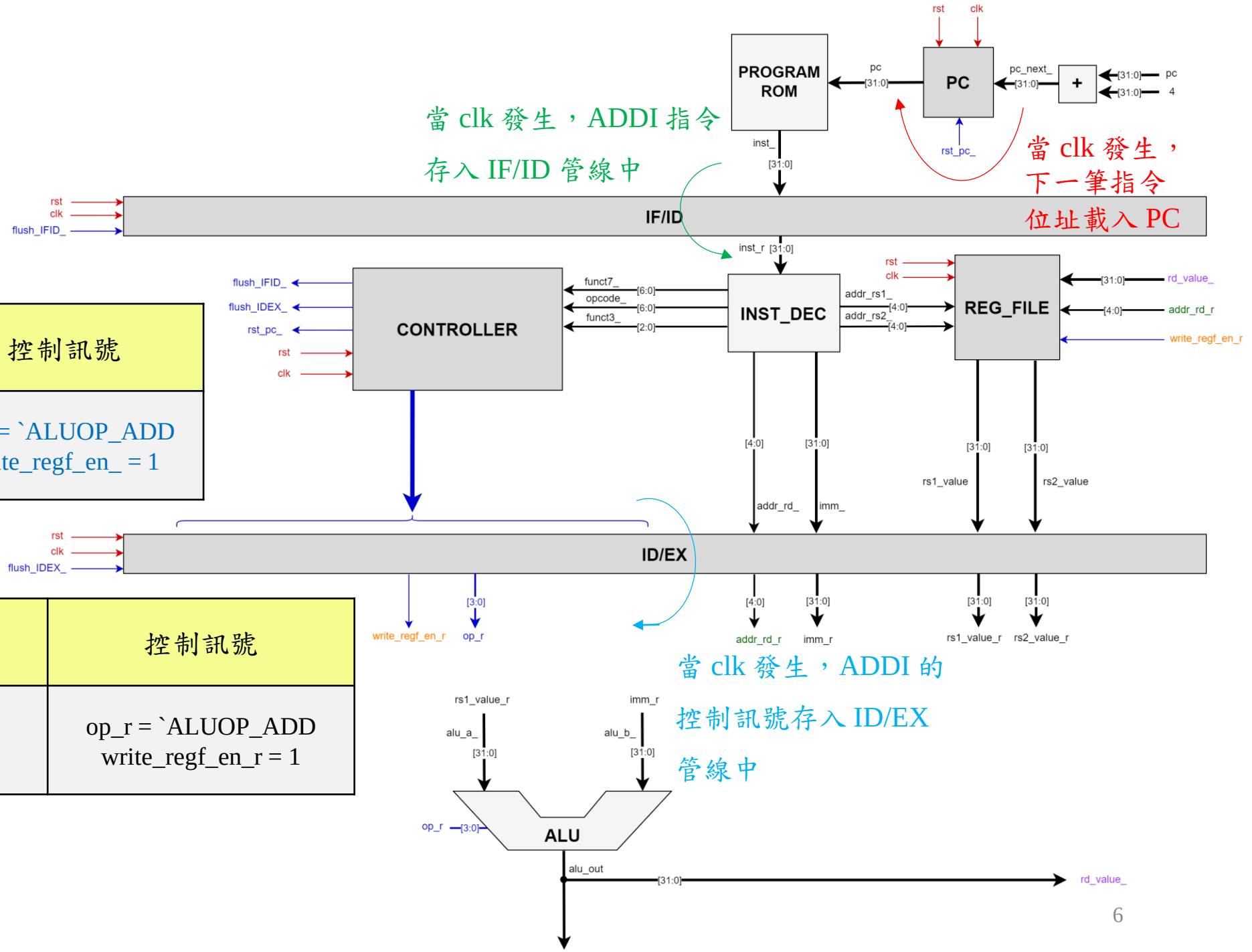
  - sub x1, x2, x3 => 0x403100b3

Funct7	Source registers 2	Source registers 1	Funct3	Destination registers	Opcode	Instruction
0100_000	0_0011	0001_0	000	0000_1	011_0011	SUB
31 25	24 20	19 15	14 12	11 7	6 0	

  - NEG x1, x2 => sub x1, x0, x2 => 0x402000b3

Funct7	Source registers 2	Source registers 1	Funct3	Destination registers	Opcode	Instruction
0100_000	0_0011	0000_0	000	0000_1	011_0011	SUB
31 25	24 20	19 15	14 12	11 7	6 0	

上周完成之架構



動作	控制訊號
if ((opcode_ == `Opcode_I) && (funct3_ == `F_ADDI)) 發出 ADDI 的控制訊號	op_ = `ALUOP_ADD write_regf_en_ = 1

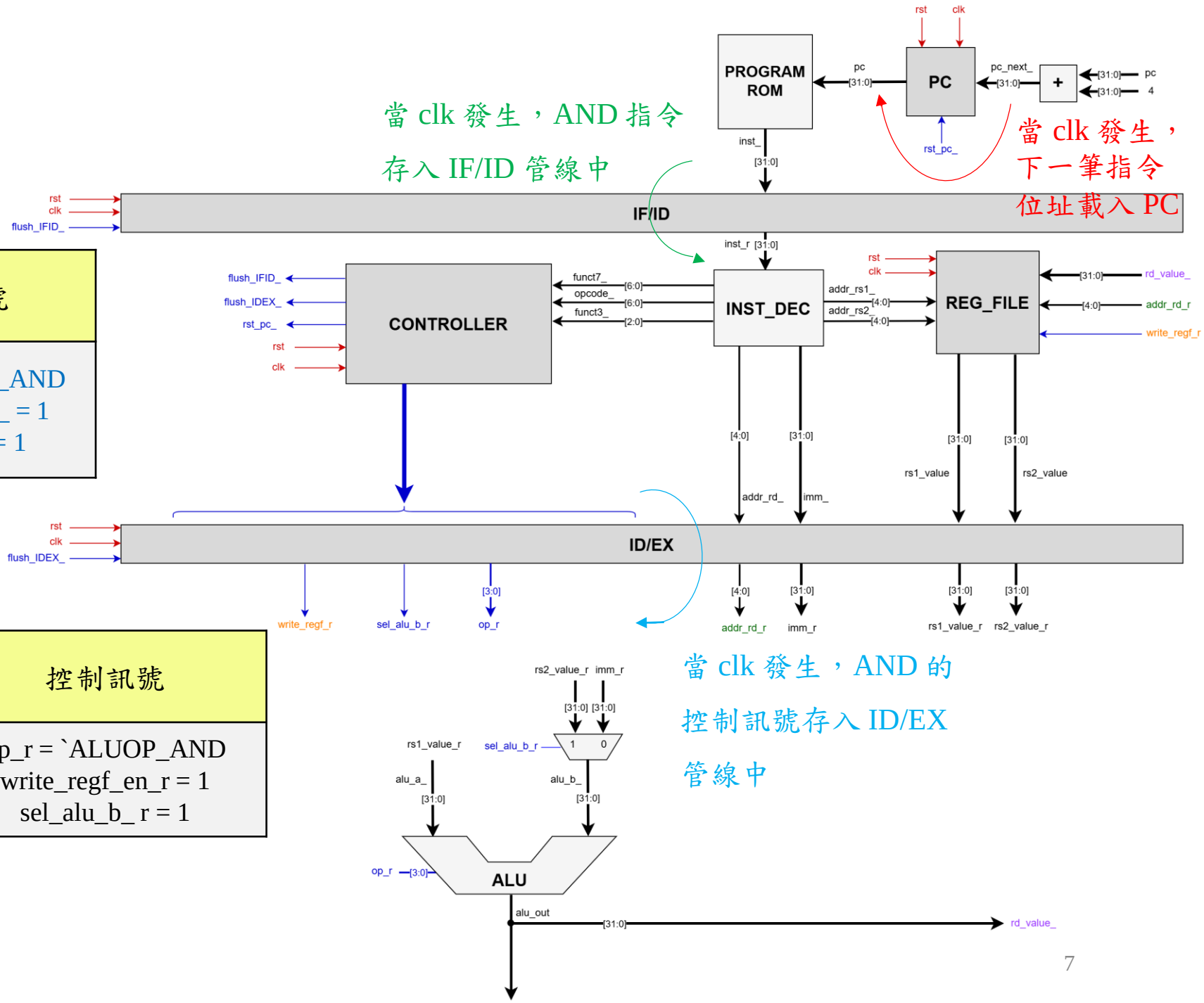
動作	控制訊號
x[addr_rd_r] ← rs1_value_r + imm_r	op_r = `ALUOP_ADD write_regf_en_r = 1

本周架構

```
`define Opcode_R_M      7'b0110011
```

動作	控制訊號
if ((opcode_ == `Opcode_R_M) && (funct7_ == 7'b000_0000) && (funct3_ == `F_AND)) 發出 AND 的控制訊號	op_ = `ALUOP_AND write_regf_en_ = 1 sel_alu_b_ = 1

動作	控制訊號
x[addr_rd_r] ← rs1_value_r & rs2_value_r	op_r = `ALUOP_AND write_regf_en_r = 1 sel_alu_b_ r = 1



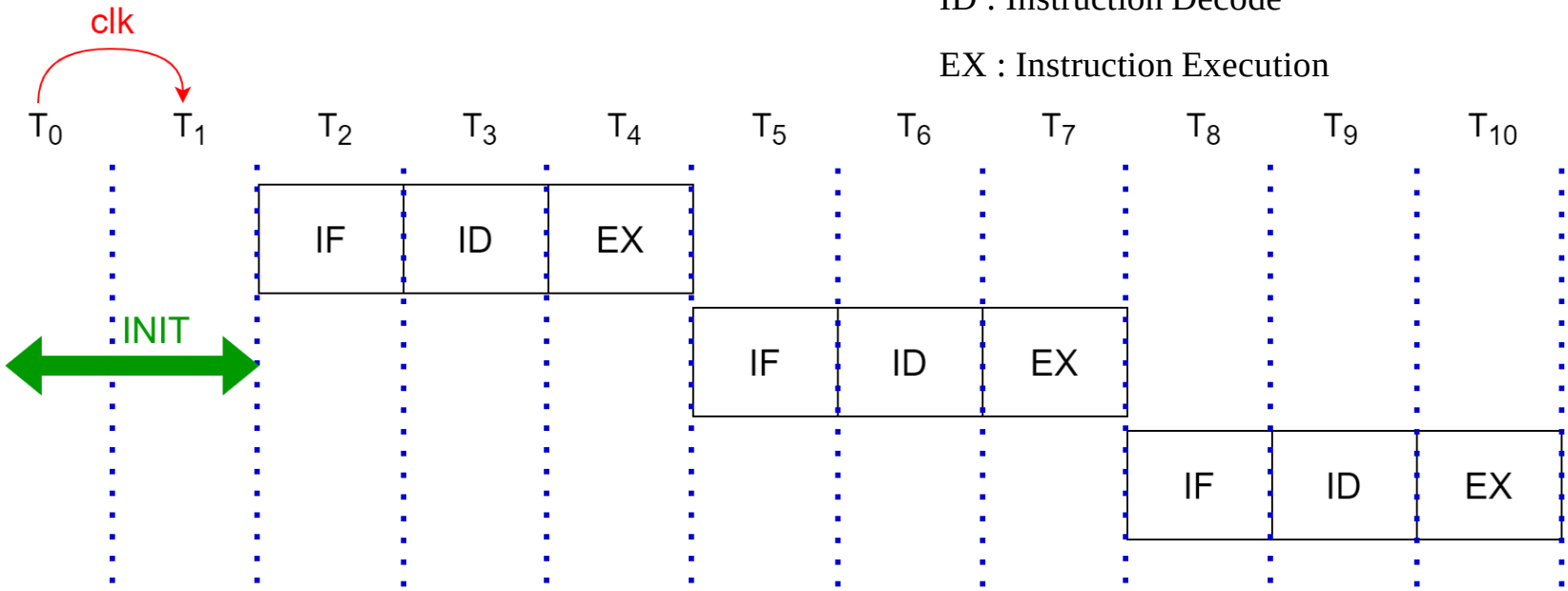
管線化 (Pipeline)(1/2)

IF : Instruction Fetch

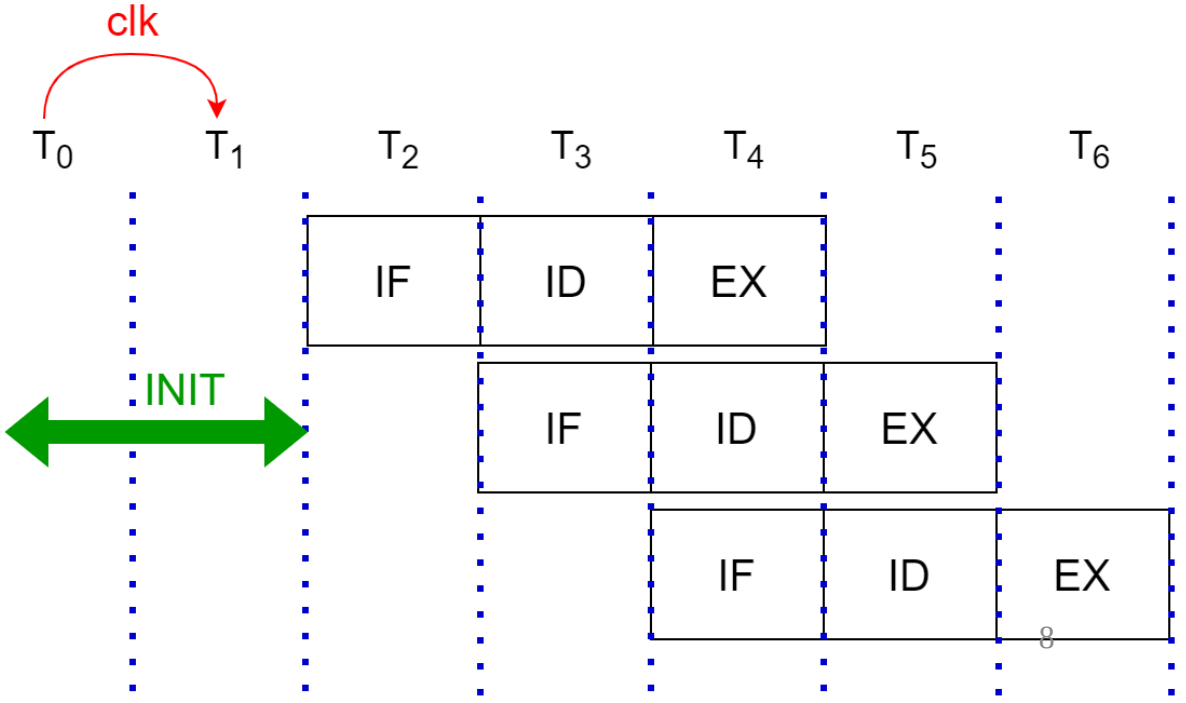
ID : Instruction Decode

EX : Instruction Execution

- **無管線化**
  - 每一個指令完整執行完畢後才會讀取下一個位址的指令。

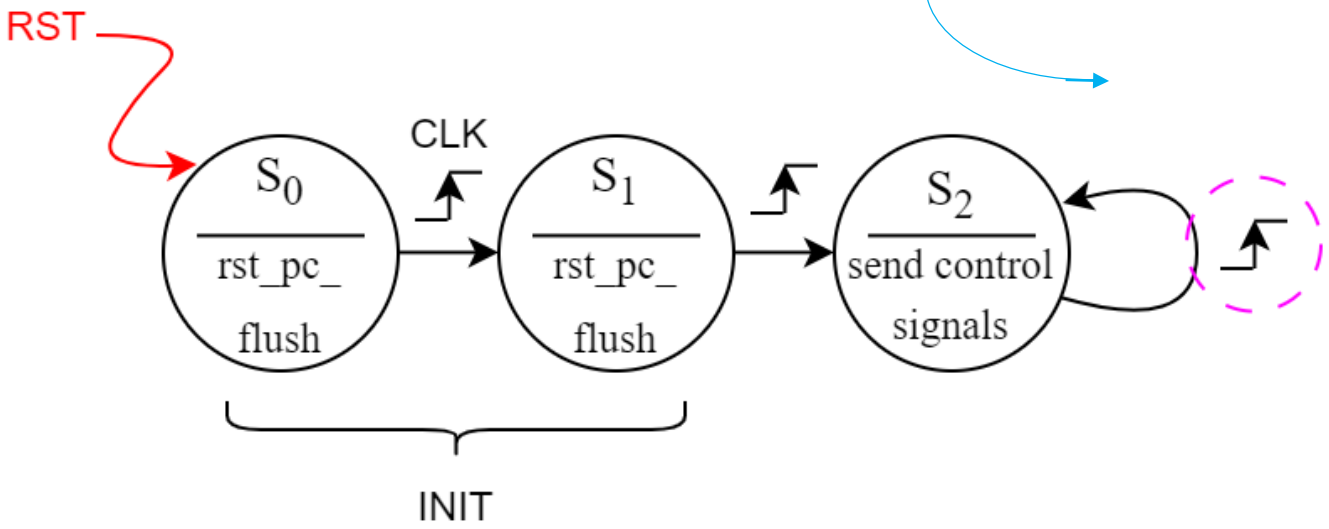


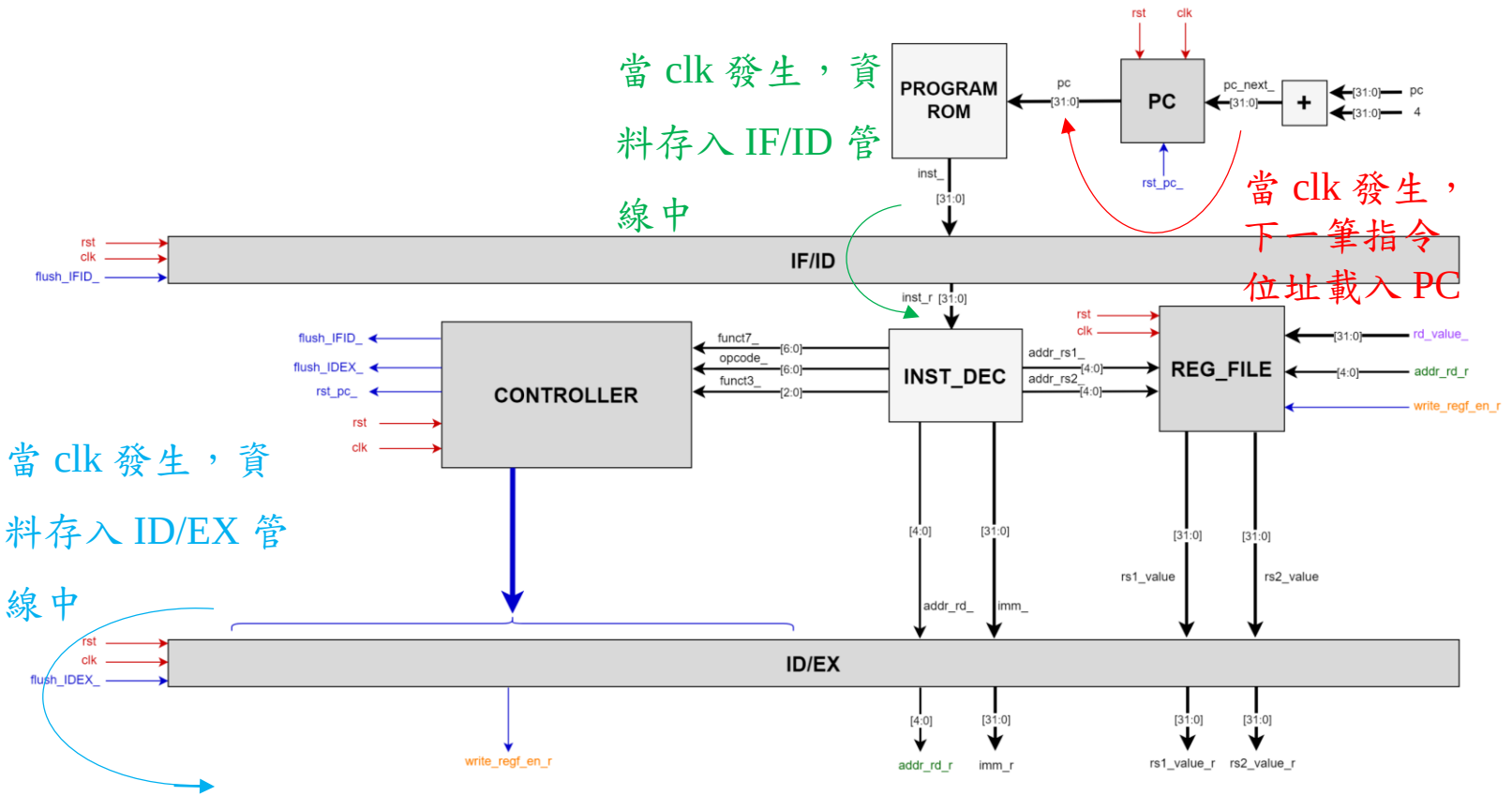
- **管線化(Pipeline)**
  - 程式脫離 INIT (Initialization) 狀態後，每一個時脈都會讀取一筆指令來執行。



管線化 (Pipeline)(2/2)

- 課程中使用三層管線化架構，狀態圖如下:
  - 在 INIT 的兩個狀態 S_0, S_1 和皆發出 flush 和 rst_pc_ 訊號。
  - 進入 S_2 後，每當 clk 發生便會執行右圖中的動作。



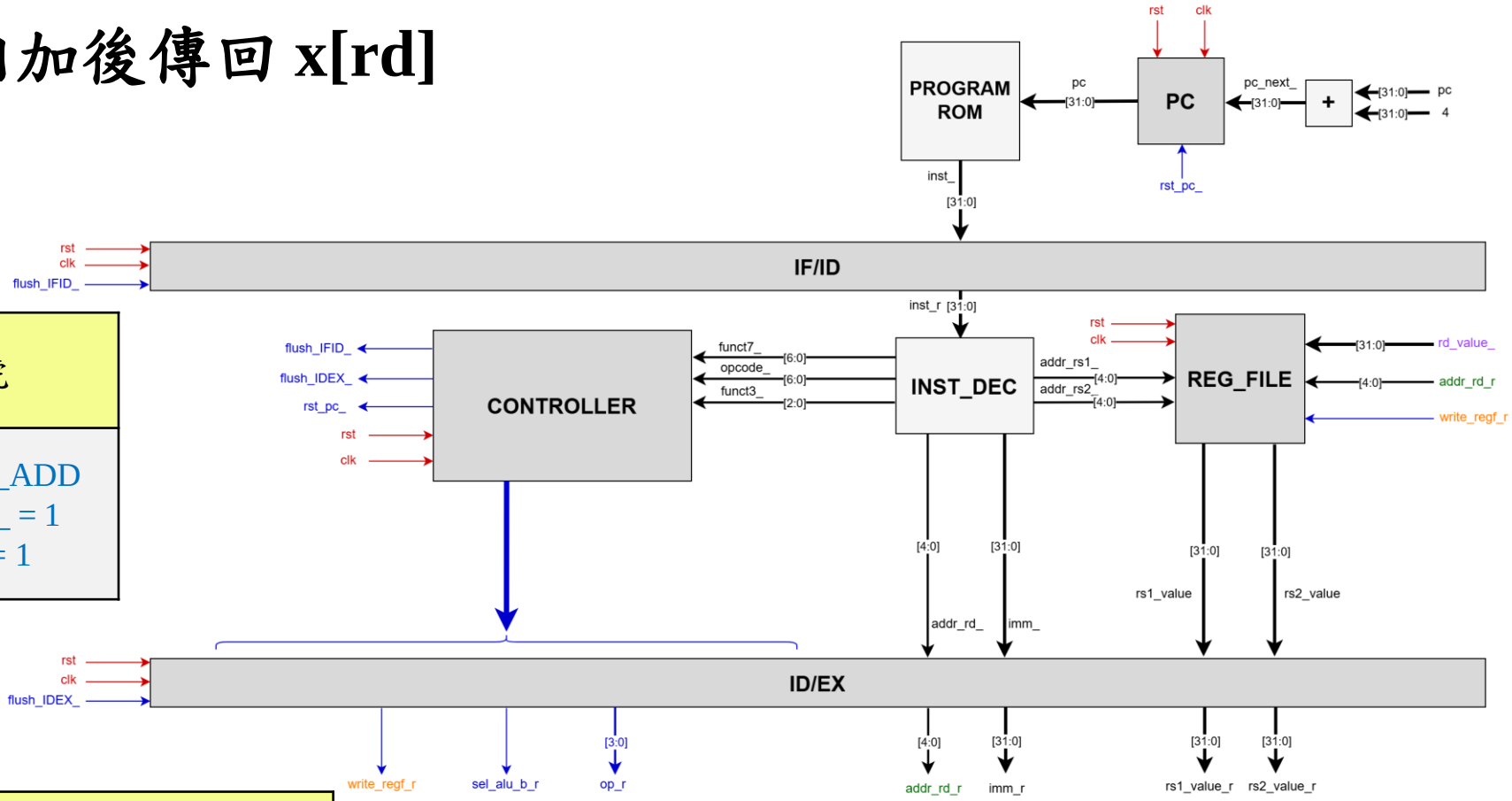


狀態	動作
S_2	pc ← pc_next_ inst_r ← inst_

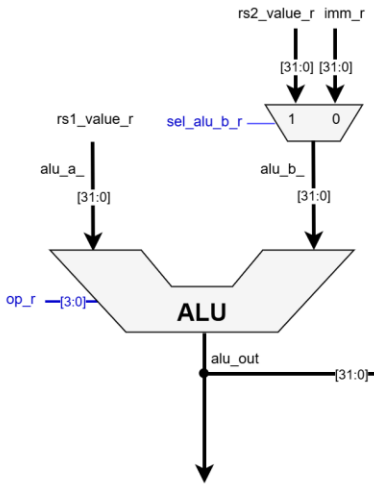
ADD : x[rs1] 和 x[rs2] 相加後傳回 x[rd]

```
`define F7_ADD          7'b0000000
`define F7_SUB          7'b0100000
```

動作	控制訊號
if ((opcode_ == `Opcode_R_M) && (funct7_ == `F7_ADD) && (funct3_ == `F_ADD_SUB)) 發出 ADD 的控制訊號	op_ = `ALUOP_ADD write_regf_en_ = 1 sel_alu_b_ = 1



動作	控制訊號
x[addr_rd_r] ← rs1_value_r + rs2_value_r	op_r = `ALUOP_ADD write_regf_en_r = 1 sel_alu_b_ r = 1

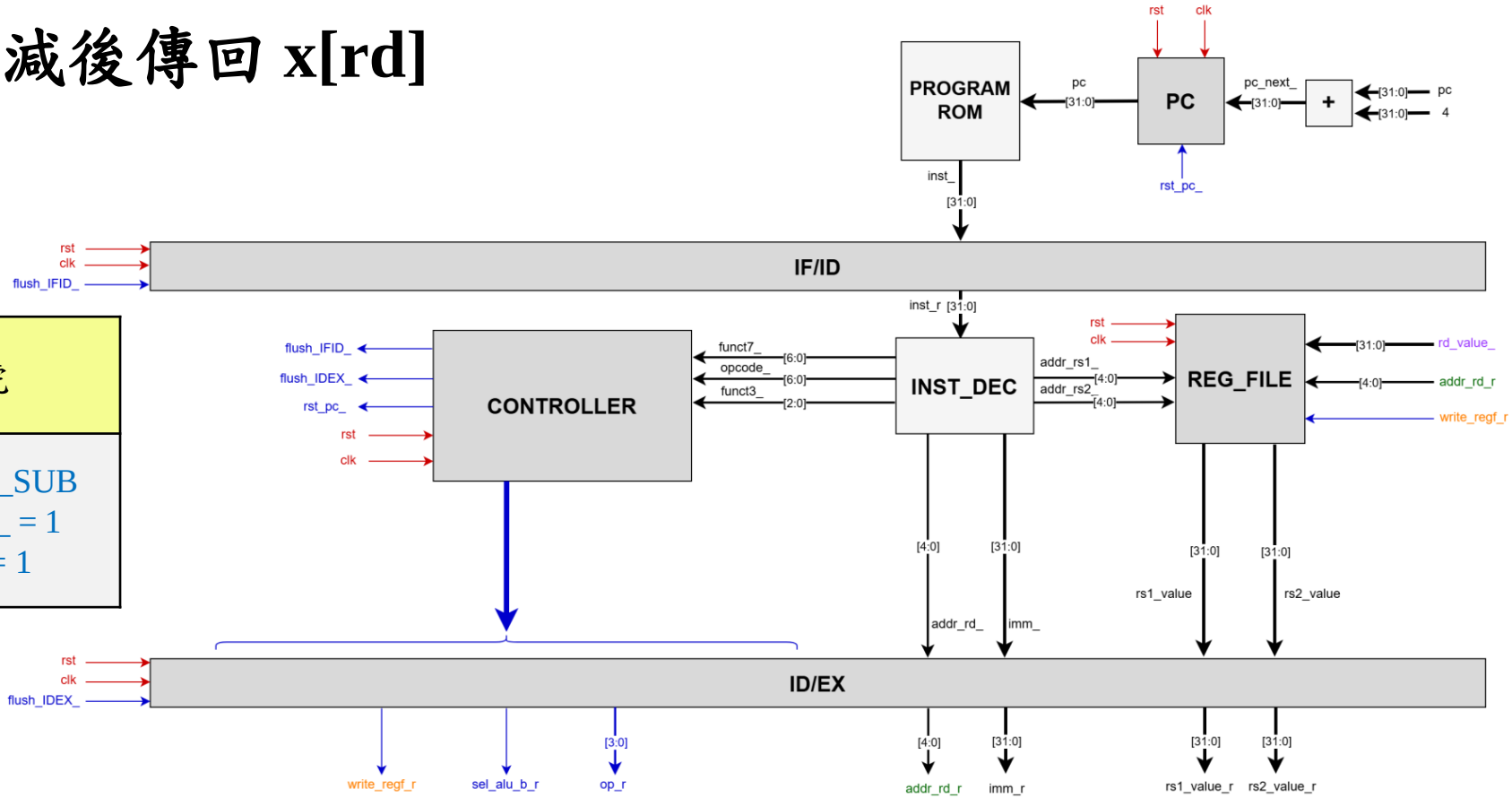



```
`define F_ADD_SUB       3'b000
`define F_SLL           3'b001
`define F_SLT           3'b010
`define F_SLTU          3'b011
`define F_XOR           3'b100
`define F_SRL_SRA       3'b101
`define F_OR            3'b110
`define F_AND           3'b111
```

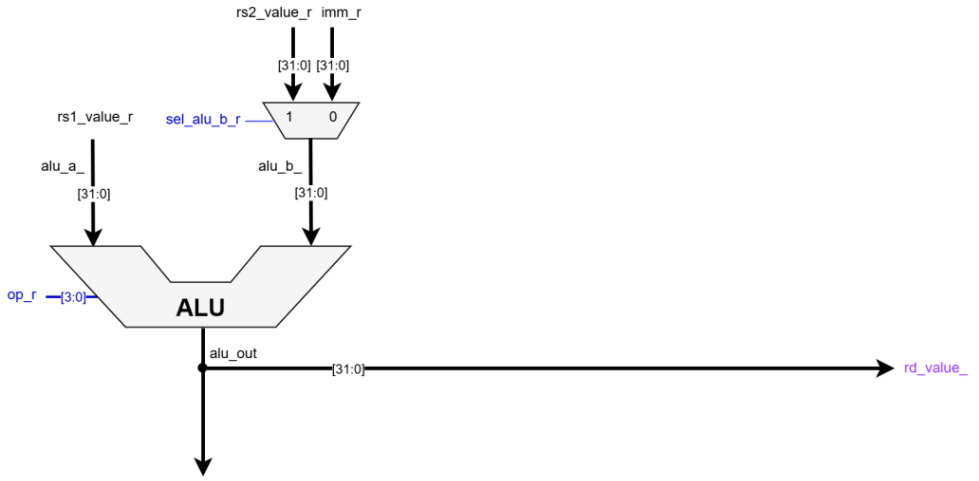
SUB : x[rs1] 和 x[rs2] 相減後傳回 x[rd]

```
`define F7_ADD          7'b0000000
`define F7_SUB          7'b0100000
```

動作	控制訊號
if ((opcode_ == `Opcode_R_M) && (funct7_ == `F7_SUB) && (funct3_ == `F_ADD_SUB)) 發出 SUB 的控制訊號	op_ = `ALUOP_SUB write_regf_en_ = 1 sel_alu_b_ = 1



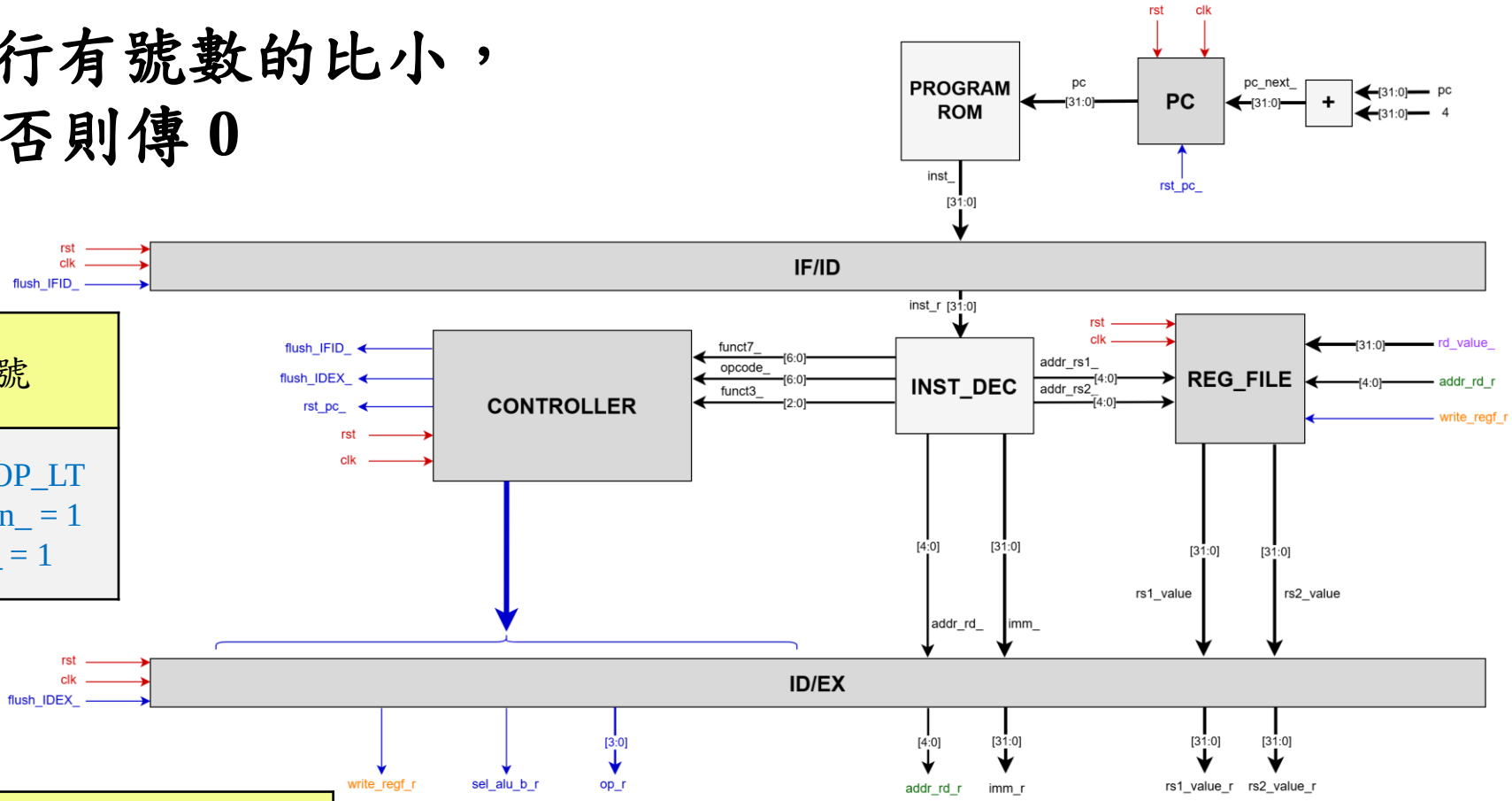
動作	控制訊號
x[addr_rd_r] ← rs1_value_r - rs2_value_r	op_r = `ALUOP_SUB write_regf_en_r = 1 sel_alu_b_ r = 1



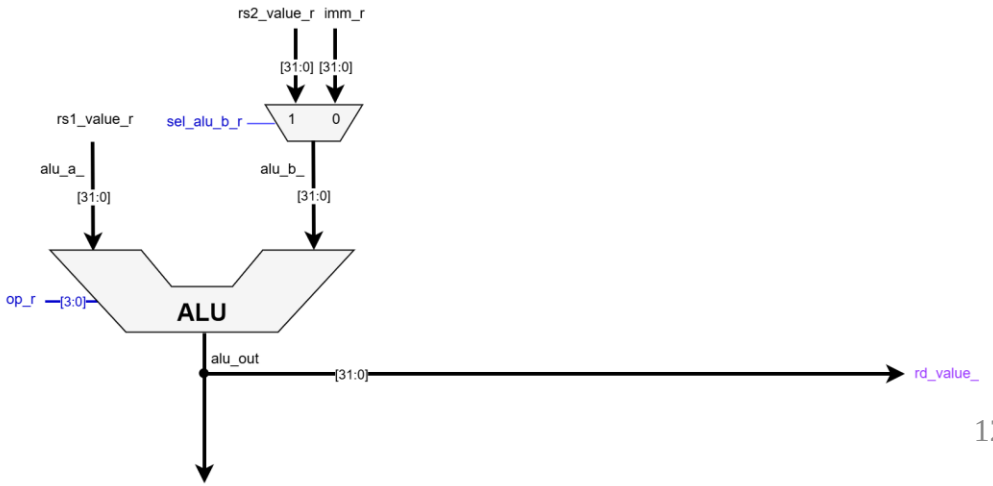
SLT : x[rs1] 和 x[rs2] 進行有號數的比小，若真則傳回 1 至 x[rd]，否則傳 0

```
`define F7_OPCODE_R      7'b0000000
```

動作	控制訊號
if ((opcode_ == `Opcode_R_M) && (funct7_ == `F7_OPCODE_R) && (funct3_ == `F_SLT)) 發出 SLT 的控制訊號	op_ = `ALUOP_LT write_regf_en_ = 1 sel_alu_b_ = 1

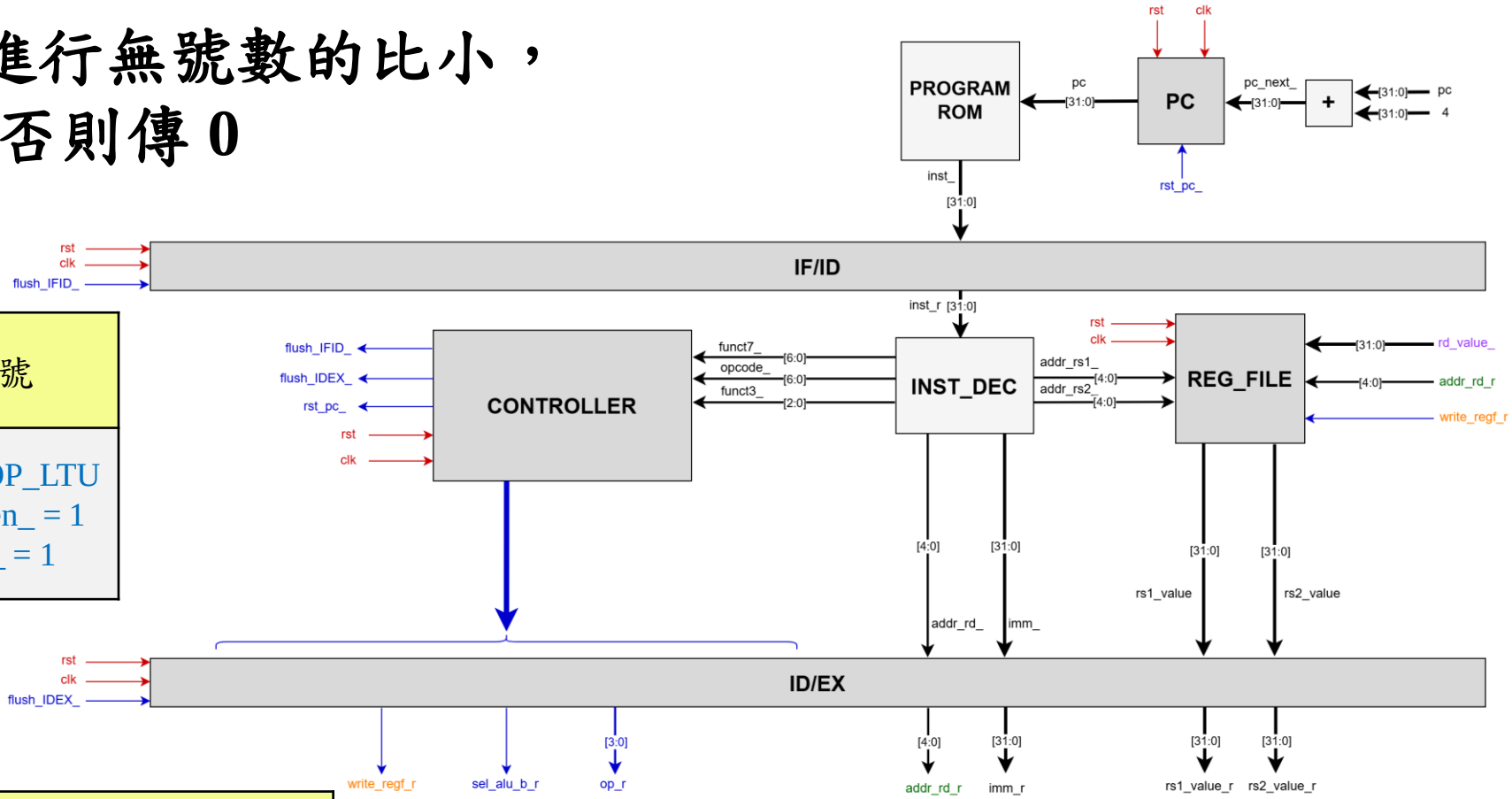


動作	控制訊號
x[addr_rd_r] ← rs1_value_r $<_s$ rs2_value_r	op_r = `ALUOP_LT write_regf_en_r = 1 sel_alu_b_ r = 1

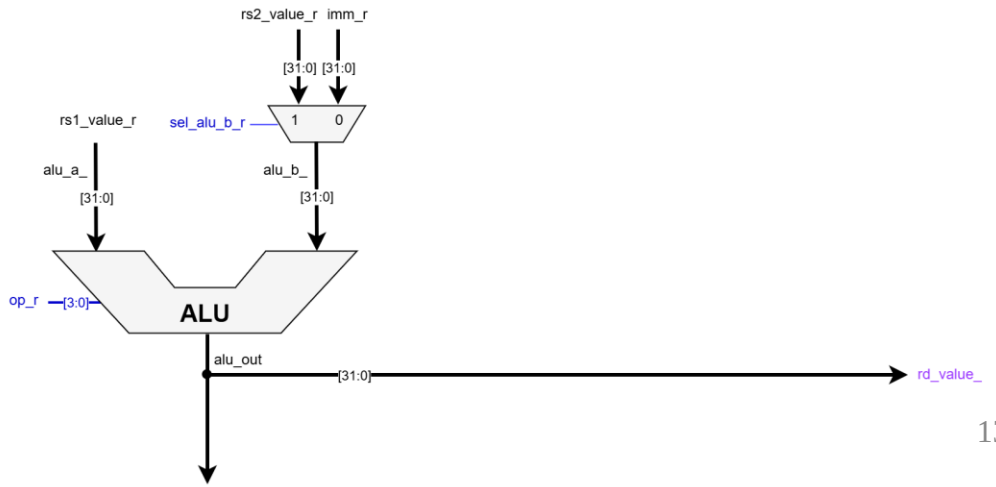


SLTU : x[rs1] 和 x[rs2] 進行無號數的比小，若真則傳回 1 至 x[rd]，否則傳 0

動作	控制訊號
if ((opcode_ == `Opcode_R_M) && (funct7_ == `F7_OPCODE_R) && (funct3_ == `F_SLTU)) 發出 SLTU 的控制訊號	op_ = `ALUOP_LTU write_regf_en_ = 1 sel_alu_b_ = 1

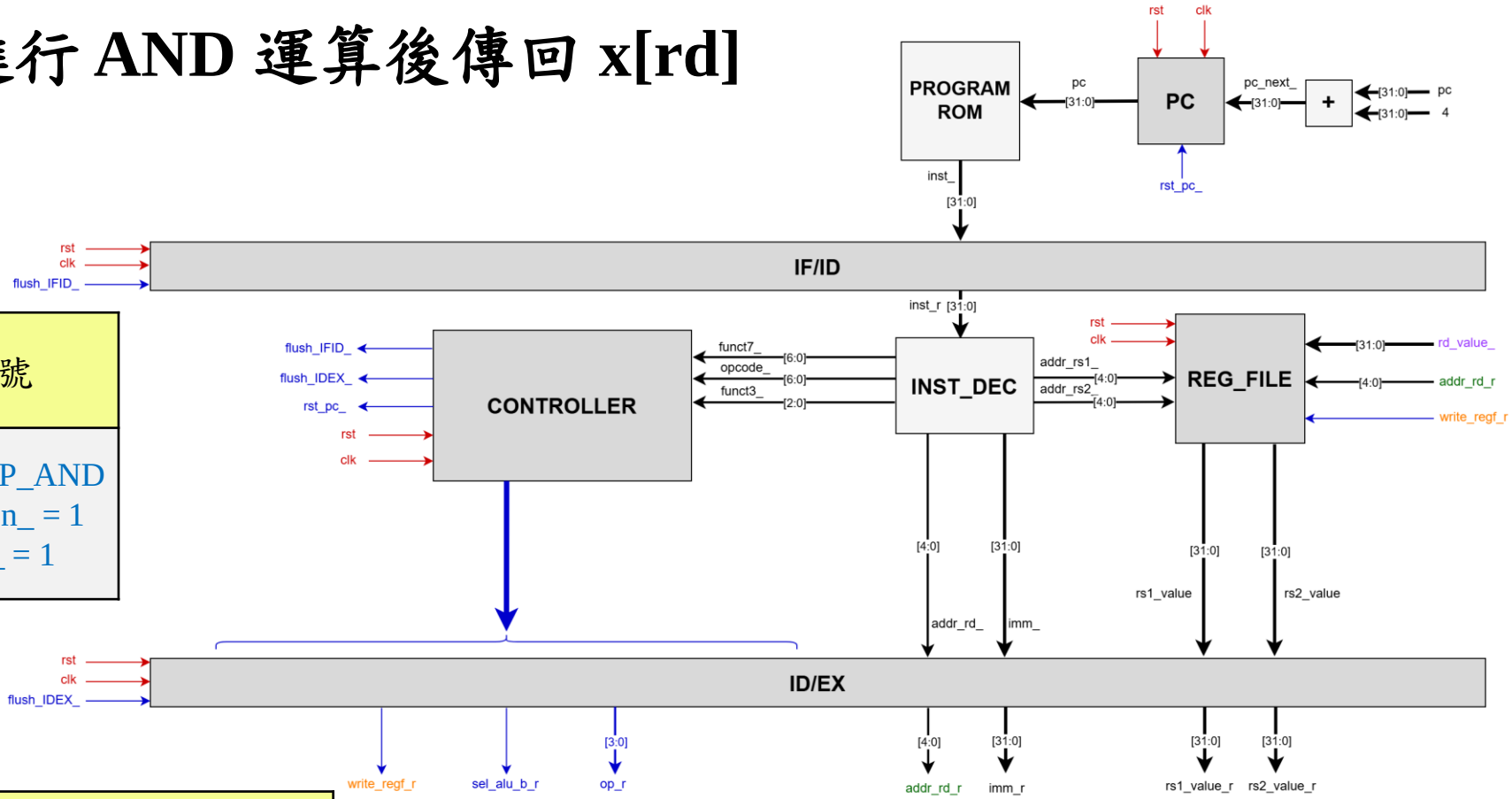


動作	控制訊號
x[addr_rd_r] ← rs1_value_r $<_u$ rs2_value_r	op_r = `ALUOP_LTU write_regf_en_r = 1 sel_alu_b_ r = 1

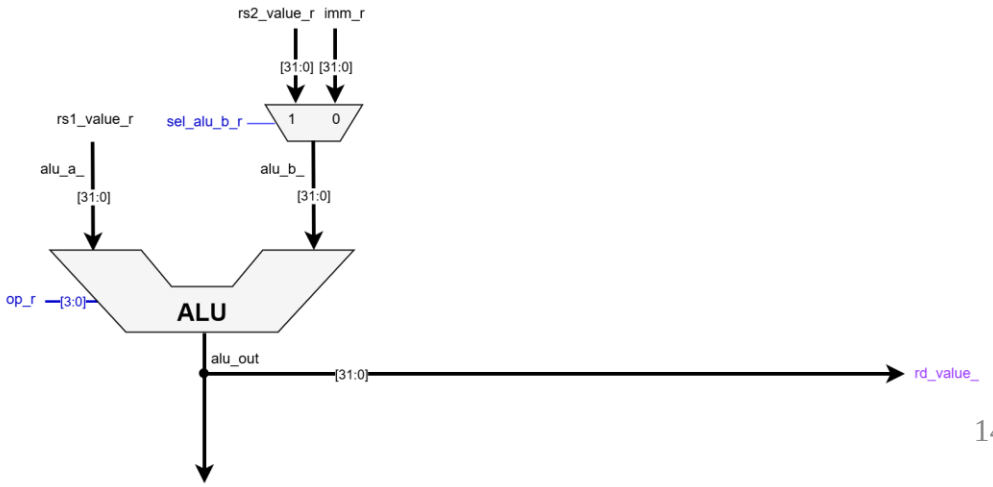


AND : x[rs1] 和 x[rs2] 進行 AND 運算後傳回 x[rd]

動作	控制訊號
if ((opcode_ == `Opcode_R_M) && (funct7_ == `F7_OPCODE_R) && (funct3_ == `F_AND)) 發出 AND 的控制訊號	op_ = `ALUOP_AND write_regf_en_ = 1 sel_alu_b_ = 1

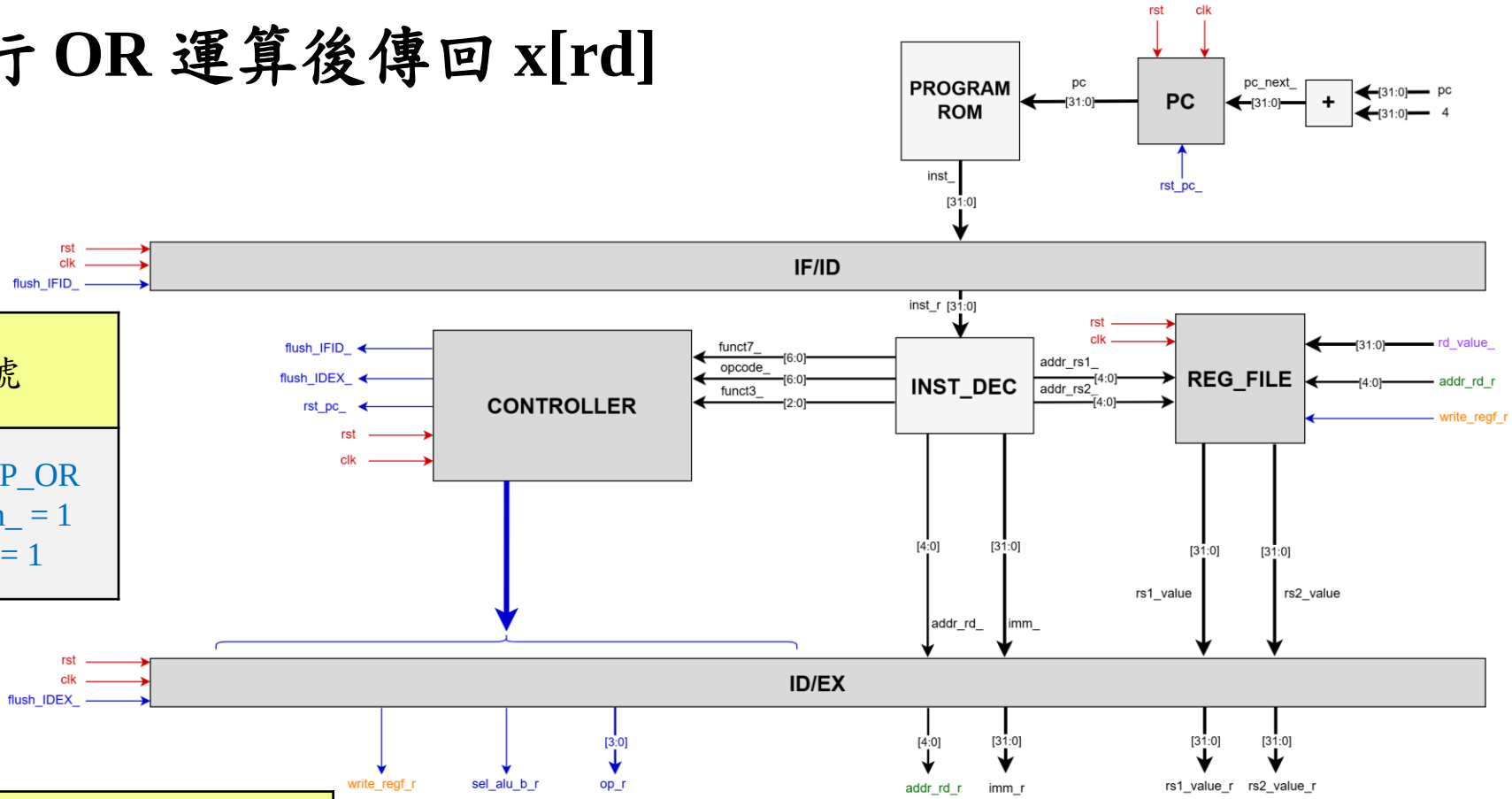


動作	控制訊號
x[addr_rd_r] ← rs1_value_r & rs2_value_r	op_r = `ALUOP_AND write_regf_en_r = 1 sel_alu_b_ r = 1

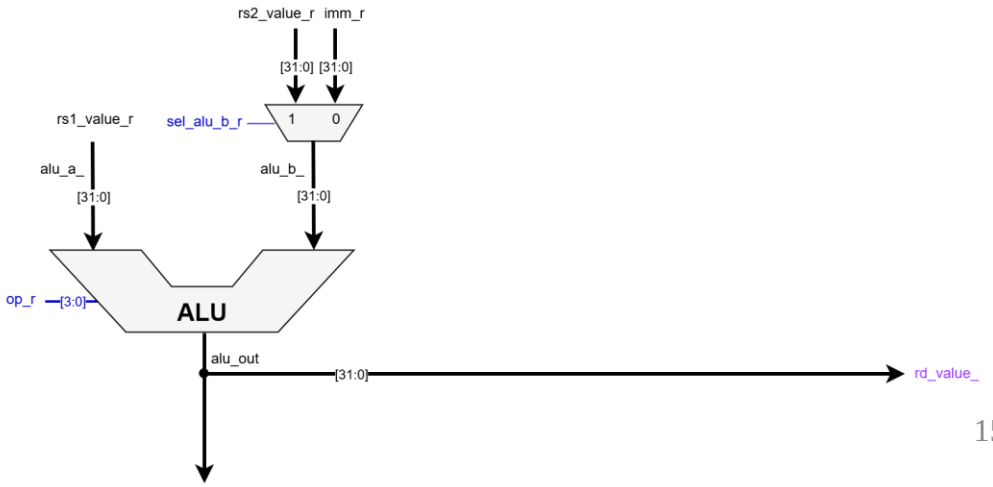


OR : x[rs1] 和 x[rs2] 進行 OR 運算後傳回 x[rd]

動作	控制訊號
if ((opcode_ == `Opcode_R_M) && (funct7_ == `F7_OPCODE_R) && (funct3_ == `F_OR)) 發出 OR 的控制訊號	op_ = `ALUOP_OR write_regf_en_ = 1 sel_alu_b_ = 1

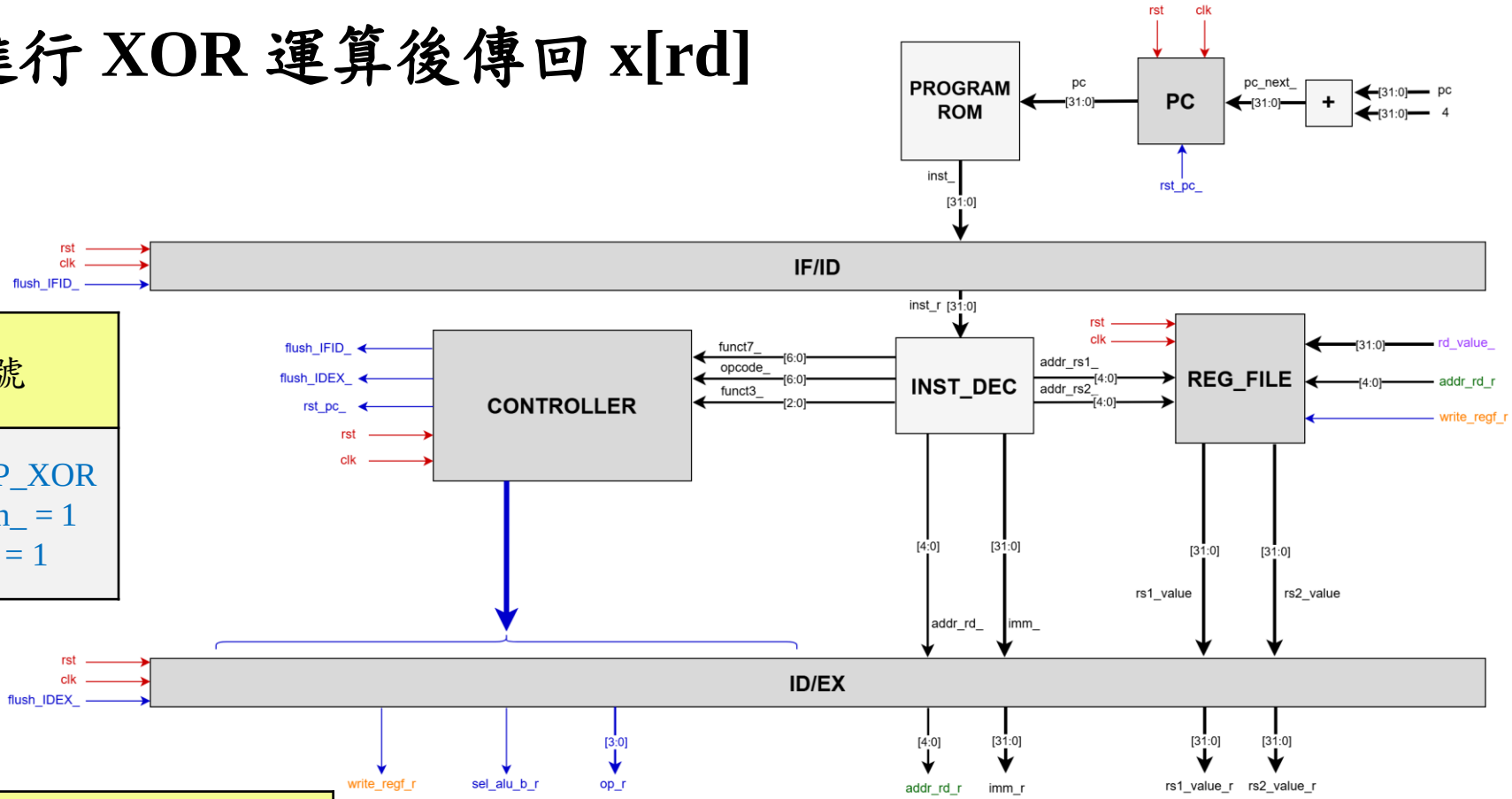


動作	控制訊號
x[addr_rd_r] ← rs1_value_r | rs2_value_r	op_r = `ALUOP_OR write_regf_en_r = 1 sel_alu_b_ r = 1

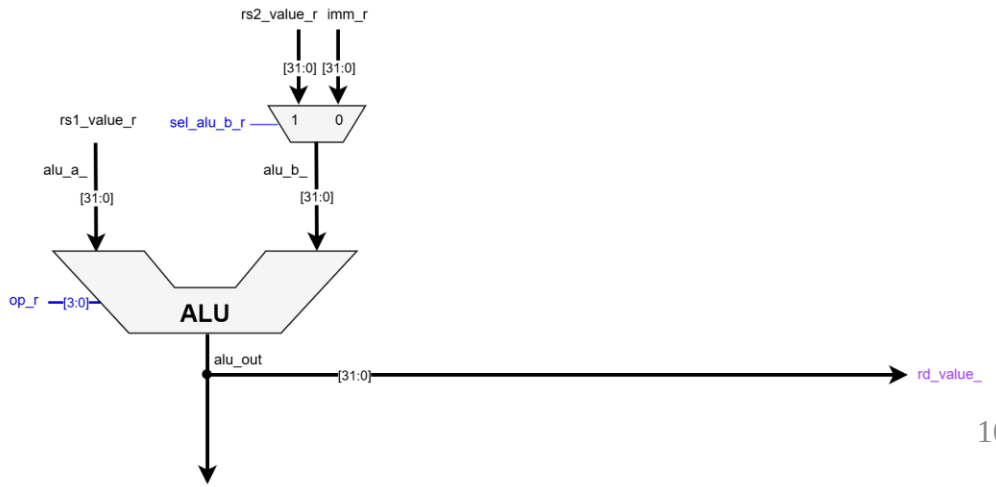


XOR : x[rs1] 和 x[rs2] 進行 XOR 運算後傳回 x[rd]

動作	控制訊號
if ((opcode_ == `Opcode_R_M) && (funct7_ == `F7_OPCODE_R) && (funct3_ == `F_XOR)) 發出 XOR 的控制訊號	op_ = `ALUOP_XOR write_regf_en_ = 1 sel_alu_b_ = 1

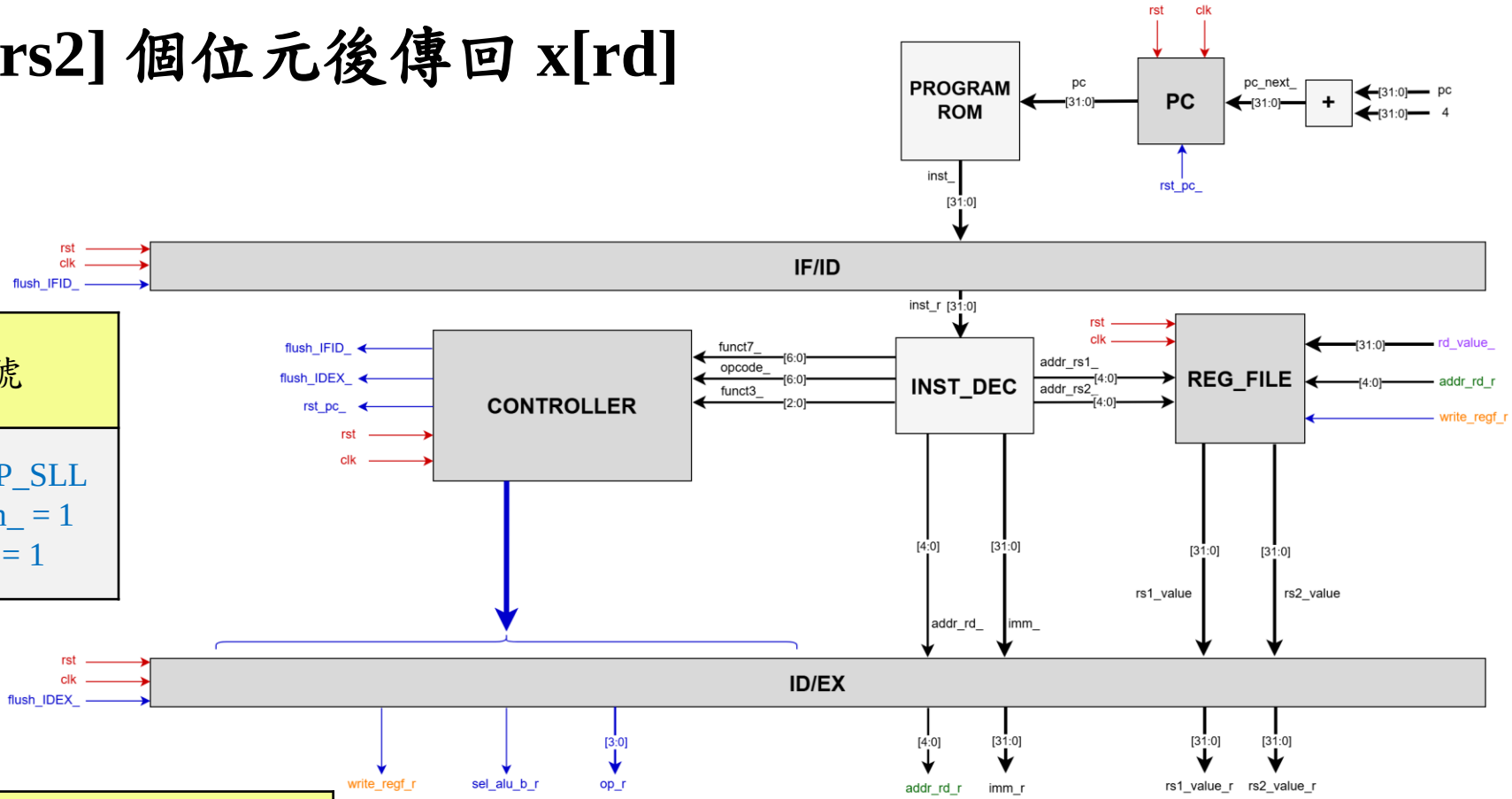


動作	控制訊號
x[addr_rd_r] ← rs1_value_r ^ rs2_value_r	op_r = `ALUOP_XOR write_regf_en_r = 1 sel_alu_b_ r = 1

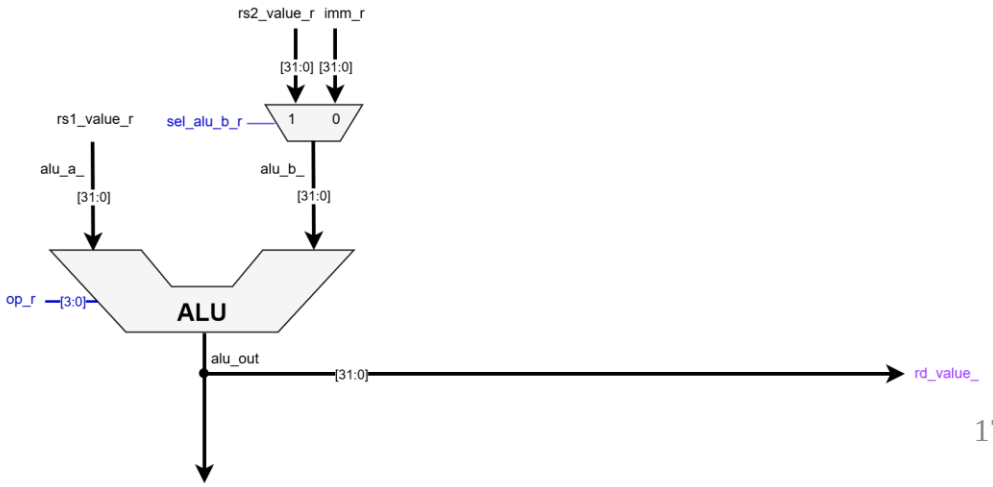


SLL : x[rs1] 邏輯左移 x[rs2] 個位元後傳回 x[rd]

動作	控制訊號
if ((opcode_ == `Opcode_R_M) && (funct7_ == `F7_OPCODE_R) && (funct3_ == `F_SLL)) 發出 SLL 的控制訊號	op_ = `ALUOP_SLL write_regf_en_ = 1 sel_alu_b_ = 1



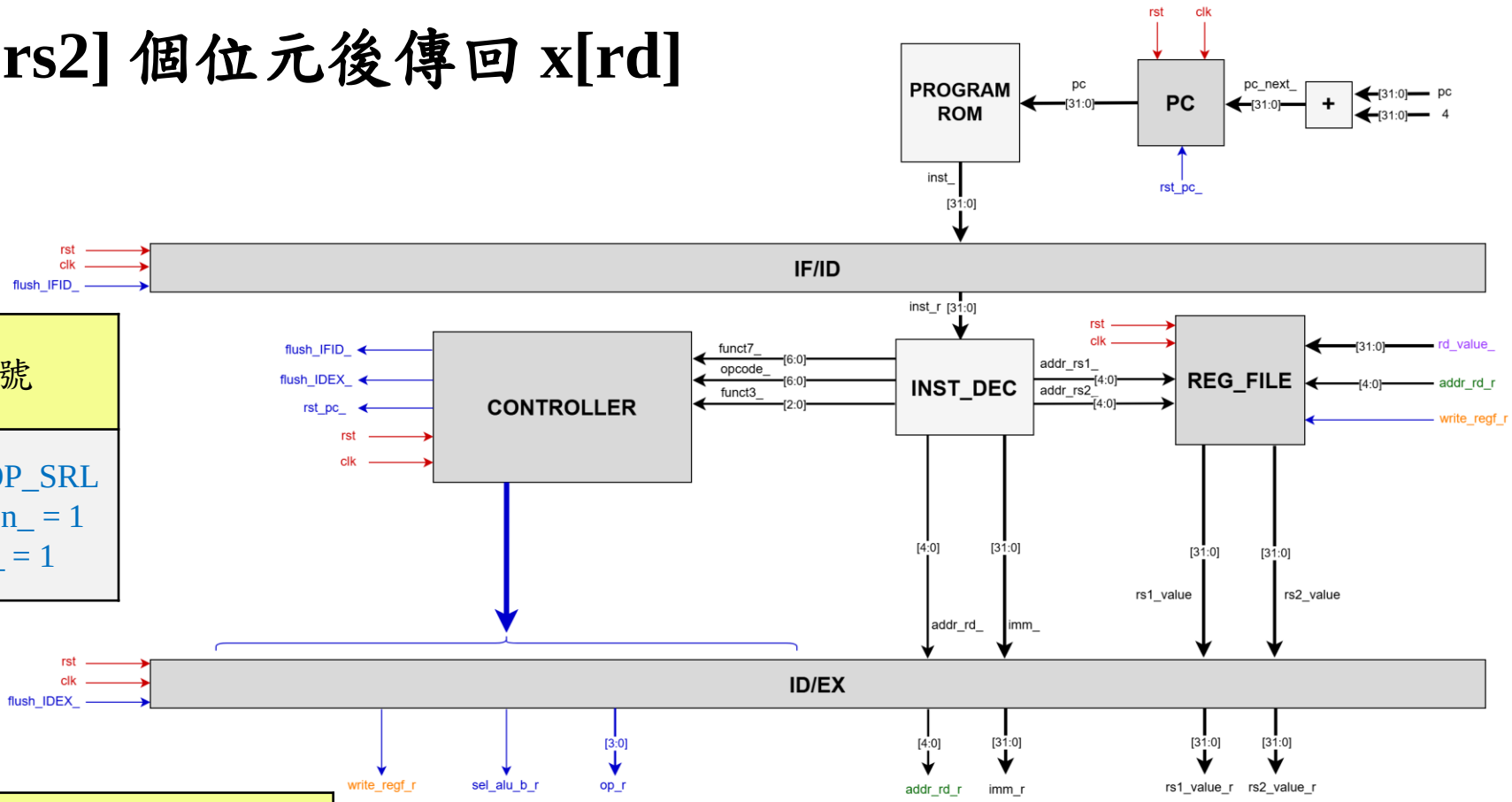
動作	控制訊號
x[addr_rd_r] ← rs1_value_r << rs2_value_r	op_r = `ALUOP_SLL write_regf_en_r = 1 sel_alu_b_ r = 1



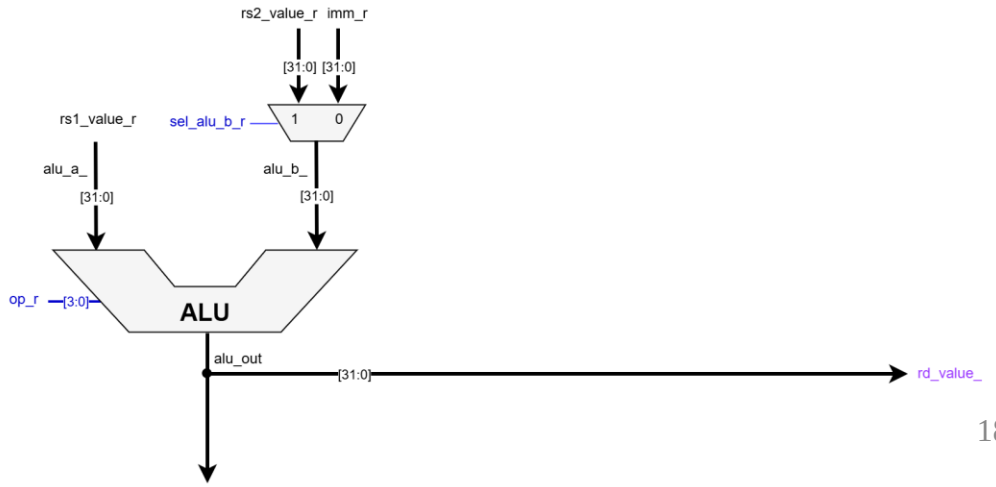
SRL : x[rs1] 邏輯右移 x[rs2] 個位元後傳回 x[rd]

```
`define F7_SRL          7'b0000000
`define F7_SRA          7'b0100000
```

動作	控制訊號
if ((opcode_ == `Opcode_R_M) && (funct7_ == `F7_OPCODE_R) && (funct3_ == `F_SRL_SRA)) 發出 SRL 的控制訊號	op_ = `ALUOP_SRL write_regf_en_ = 1 sel_alu_b_ = 1



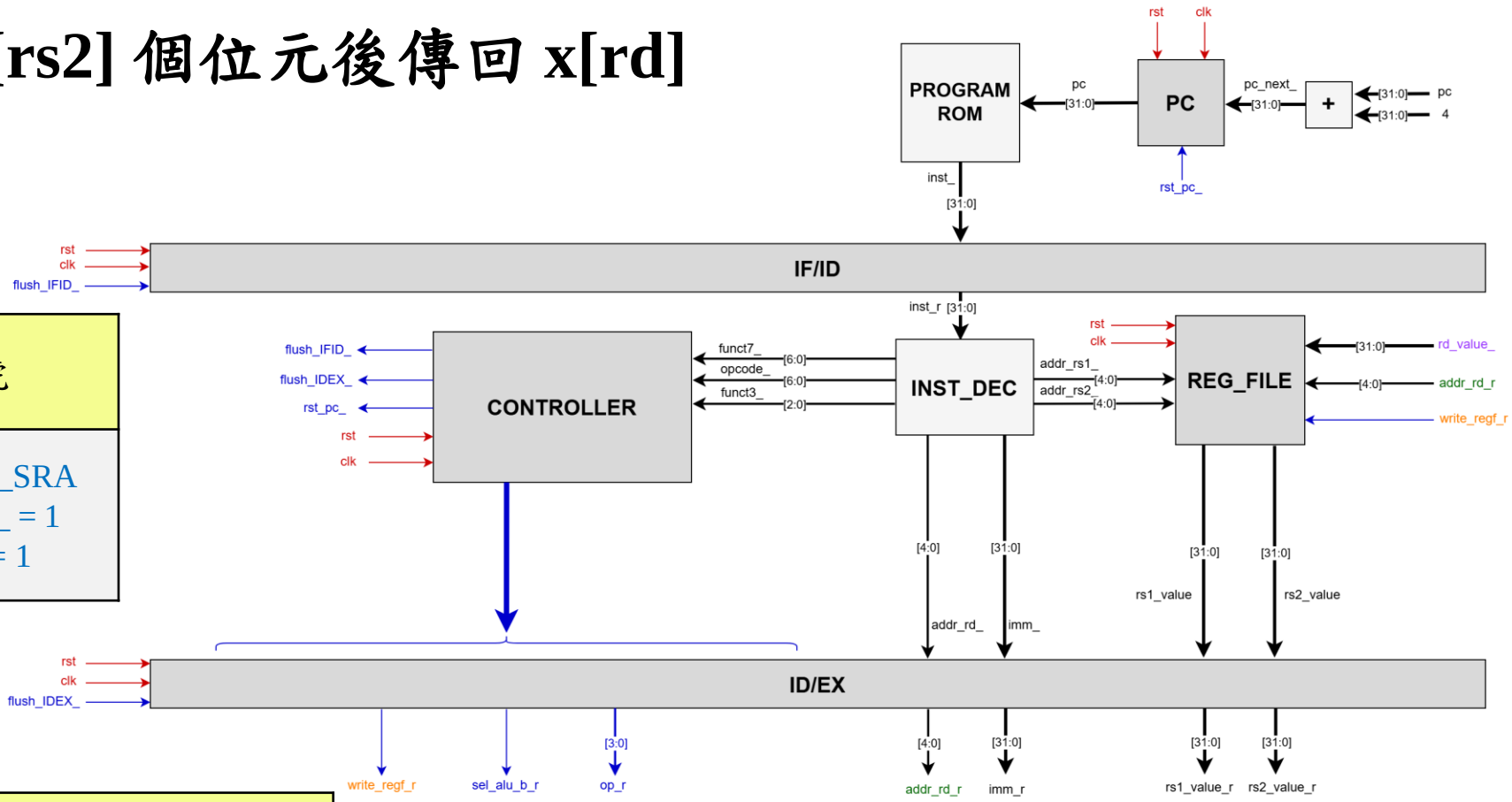
動作	控制訊號
x[addr_rd_r] ← rs1_value_r >>$_u$ rs2_value_r	op_r = `ALUOP_SRL write_regf_en_r = 1 sel_alu_b_ r = 1



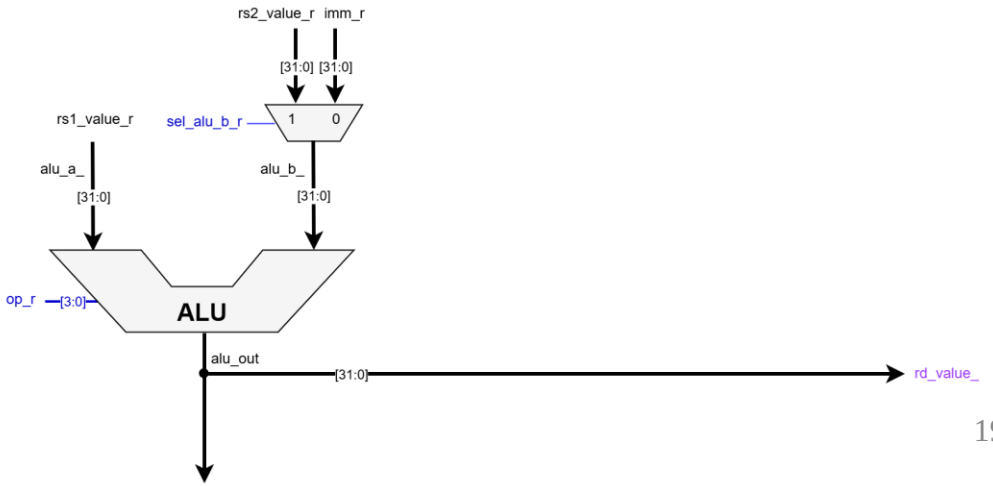
SRA : x[rs1] 算術右移 x[rs2] 個位元後傳回 x[rd]

```
`define F7_SRL          7'b0000000
`define F7_SRA          7'b0100000
```

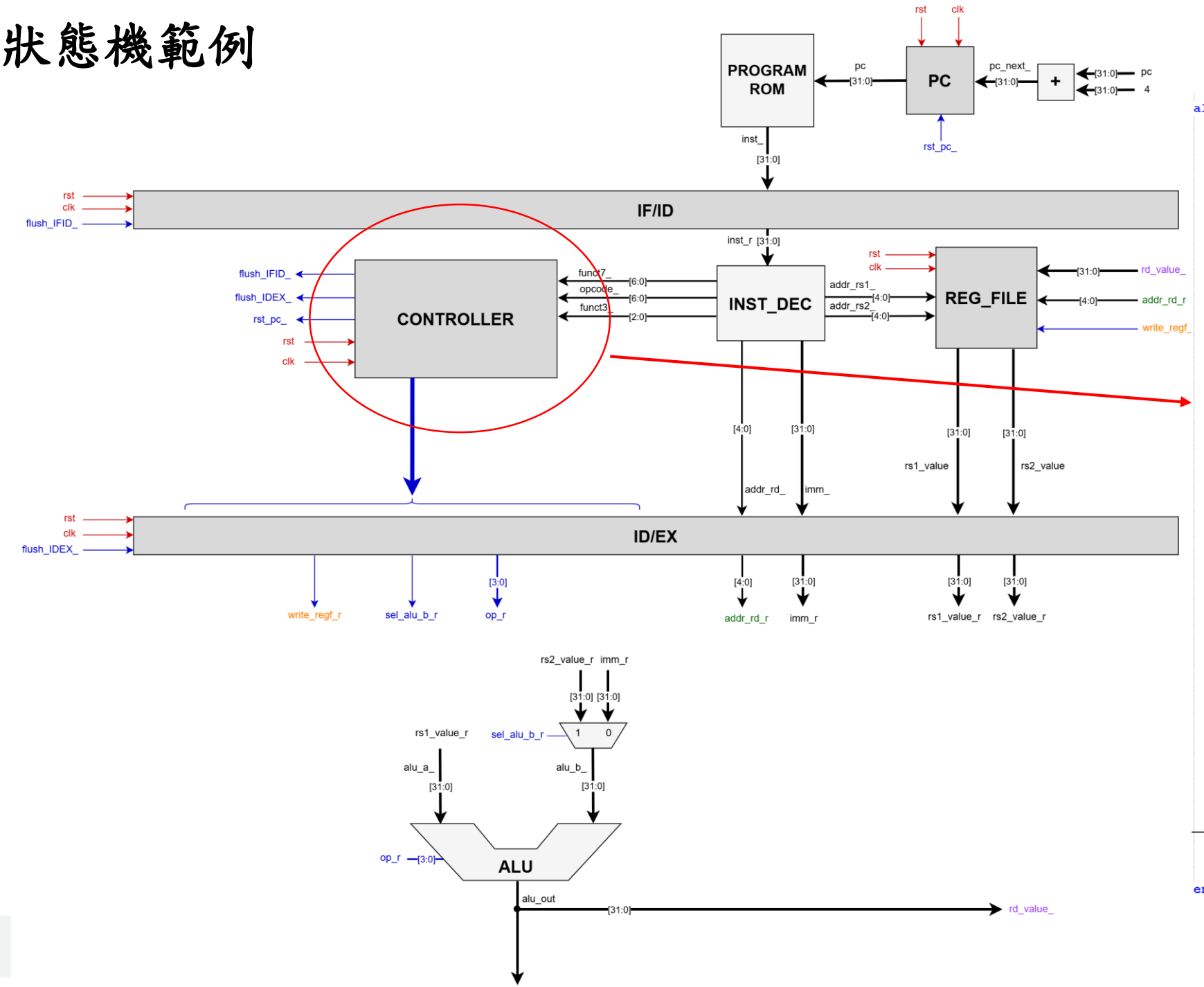
動作	控制訊號
if ((opcode_ == `Opcode_R_M) && (funct7_ == `F7_SRA) && (funct3_ == `F_SRL_SRA)) 發出 SRA 的控制訊號	op_ = `ALUOP_SRA write_regf_en_ = 1 sel_alu_b_ = 1



動作	控制訊號
x[addr_rd_r] ← rs1_value_r >>$_s$ rs2_value_r	op_r = `ALUOP_SRA write_regf_en_r = 1 sel_alu_b_ r = 1



狀態機範例



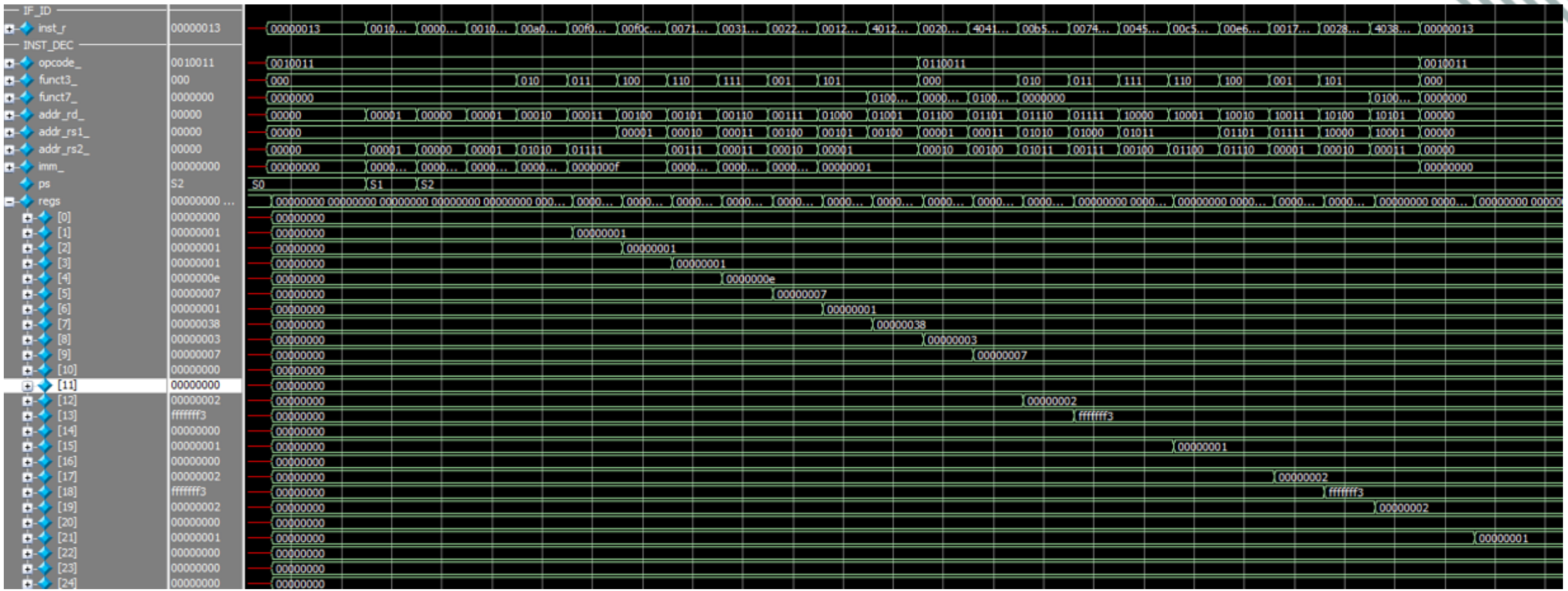

```
always_comb begin
    flush_IFID_   = 0;
    flush_IDEX_   = 0;
    rst_pc_       = 0;
    write_regf_en = 0;
    op_           = 0;
    sel_alu_b_    = 0;
    ns            = ps;
    unique case(ps)
        S0:
            begin
                flush_IFID_ = 1;
                flush_IDEX_ = 1;
                rst_pc_     = 1;
                ns          = S1;
            end
        S1:
            begin
                flush_IFID_ = 1;
                flush_IDEX_ = 1;
                rst_pc_     = 1;
                ns          = S2;
            end
        S2:
            begin
                unique case(opcode_)
                    `Opcode_I: begin
                        write_regf_en = 1;
                        unique case(funct3_)
                            `F_ADDI  : op_ = `ALUOP_ADD;
                            `F_SLTI  : op_ = `ALUOP_LT;
                            `F_SLTIU : op_ = `ALUOP_LTU;
                            `F_ANDI  : op_ = `ALUOP_AND;
                            `F_ORI   : op_ = `ALUOP_OR;
                            `F_XORI  : op_ = `ALUOP_XOR;
                            `F_SLLI  : op_ = `ALUOP_SLL;
                            `F_SRLI_SRAI : begin
                                unique case(funct7_)
                                    `F7_SRLI : op_ = `ALUOP_SRL;
                                    `F7_SRAI : op_ = `ALUOP_SRA;
                                endcase
                            end
                        endcase
                    end
                    `Opcode_R: begin
                    endcase
                end
    endcase
end
```

上課實作：ModelSim 模擬

請至 Tronclass 下載 Program_Rom.sv 檔案

```
always_comb begin
    case (Rom_addr)
        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
        default : Rom_data = 32'h00000013;      // NOP
    endcase
end
```

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 LED 中，透過下面的組合語言完成 LED 左右來回閃爍的功能

```
addi x31, x0, 0x00000001
addi x31, x0, 0x00000010
addi x31, x0, 0x00000100
addi x31, x0, 0x00000010
addi x31, x0, 0x00000001
addi x31, x0, 0x00000010
addi x31, x0, 0x00000100
addi x31, x0, 0x00000010
addi x31, x0, 0x00000001
addi x31, x0, 0x00000010
addi x31, x0, 0x00000100
addi x31, x0, 0x00000010
addi x31, x0, 0x00000001
```

組合語言

```
always_comb begin
   case (Rom_addr)
      32'h00000000 : Rom_data = 32'h00100F93; // addi x31 x0 1
      32'h00000004 : Rom_data = 32'h01000F93; // addi x31 x0 16
      32'h00000008 : Rom_data = 32'h10000F93; // addi x31 x0 256
      32'h0000000C : Rom_data = 32'h01000F93; // addi x31 x0 16
      32'h00000010 : Rom_data = 32'h00100F93; // addi x31 x0 1
      32'h00000014 : Rom_data = 32'h01000F93; // addi x31 x0 16
      32'h00000018 : Rom_data = 32'h10000F93; // addi x31 x0 256
      32'h0000001C : Rom_data = 32'h01000F93; // addi x31 x0 16
      32'h00000020 : Rom_data = 32'h00100F93; // addi x31 x0 1
      32'h00000024 : Rom_data = 32'h01000F93; // addi x31 x0 16
      32'h00000028 : Rom_data = 32'h10000F93; // addi x31 x0 256
      32'h0000002C : Rom_data = 32'h01000F93; // addi x31 x0 16
      32'h00000030 : Rom_data = 32'h00100F93; // addi x31 x0 1
         default: Rom_data = 32'h00000013;    //NOP
   endcase
end
```

Program_Rom.sv

上課實作：FPGA 燒錄

此 FPGA 上只提供 50MHz 的時脈，因此若要觀察到 LED 的變化，需要除頻器將時脈降至人眼可辨別的頻率內。

`define USE_CLOCK_DIVIER => clk = clk_div (如果 USE_CLOCK_DIVIER 有被定義的話，clk 會被設為 clk_div)

//`define USE_CLOCK_DIVIER => clk = CLOCK_50 (否則 clk 會被設為 CLOCK_50)

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    (10_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    )
    );
        將暫存器 x31 mapping 到 LED 上
    assign LEDR = regs_31[9:0];
```

RISCV_Core.sv

```
module RISCV_Core(
    input   logic clk,
    input   logic rst,
    //
    output logic [31:0] regs_31
);

//Register file
Reg_file Reg_file_1     (
    .clk                (clk                ),
    .rst                (rst                ),
    .write_regf_en      (write_regf_en_r    ),
    .addr_rd            (addr_rd_r          ),
    .addr_rs1           (addr_rs1_          ),
    .addr_rs2           (addr_rs2_          ),
    .rd_value           (rd_value_          ),
    //
    .rs1_value          (rs1_value          ),
    .rs2_value          (rs2_value          ),
    .regs_31            (regs_31            )
);
```

Reg_file.sv

```
module Reg_file(
    input   logic clk,
    input   logic rst,
    input   logic write_regf_en,
    input   logic [4:0] addr_rd,
    input   logic [4:0] addr_rs1,
    input   logic [4:0] addr_rs2,
    input   logic [31:0] rd_value,

    output  logic [31:0] rs1_value,
    output  logic [31:0] rs2_value,
    output  logic [31:0] regs_31
);

    logic [31:0] regs[0:31];
    logic addr_rd_not_0;
    integer i;

    assign regs_31 = regs[31];
```

THANK YOU