計算機系統設計

立即數定址

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

RV32IM 指令集中的定址方式

- 立即數定址 (Immediate Addressing)
  - 暫存器與立即數的運算。例如：addi x1, x2, 10，將暫存器 x2 的值與立即數 10 相加，結果存到暫存器 x1 中。
- 暫存器定址 (Register Addressing)
  - 暫存器與暫存器間的運算。例如：add x1, x2, x3，將暫存器 x2 和 x3 的值相加，結果存到暫存器 x1 中。
- 基址加偏移定址 (Base + Offset Addressing)
  - 透過基址暫存器和立即數的偏移量相加來計算記憶體位址 (Byte Addressing)。例如：lw x1, 8(x2)，從暫存器 x2 加 8 的記憶體位置讀取資料並存到暫存器 x1。
- PC 相對定址 (PC-Relative Addressing)
  - 用於跳躍和分支指令。偏移量是相對於程式計數器 (PC) 的值，用來計算目標地址。例如：jal x1, 8，跳轉到當前 PC 加 8 的地址，並將返回地址 (PC + 4) 存入 x1。
- 間接定址(Indirect Addressing)
  - 用於跳躍指令。偏移量是相對於指定暫存器的值，用來計算目標地址。例如：jalr x1, x2, 8，跳轉到當前暫存器 x2 加 8 的地址，並將返回地址 (PC + 4) 存入 x1。

RISC-V RV32IM INSTRUCTION SET

Integer Register-Immediate Instructions

Mnemonic, Operands	Description	Implementation
ADDI rd, rs1, imm	Adds the sign-extended 12-bit immediate to register rs1. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result.	x[rd] = x[rs1] + sext(immediate)
SLTI rd, rs1, imm	Place the value 1 in register *rd* if register *rs1* is less than the sign-extended immediate when both are treated as signed numbers, else 0 is written to *rd*.	x[rd] = x[rs1] $<_s$ sext(immediate)
SLTIU rd, rs1, imm	Place the value 1 in register *rd* if register *rs1* is less than the immediate when both are treated as unsigned numbers, else 0 is written to *rd*.	x[rd] = x[rs1] $<_u$ sext(immediate)
ANDI rd, rs1, imm	Performs bitwise AND on register *rs1* and the sign-extended 12-bit immediate and place the result in *rd*.	x[rd] = x[rs1] & sext(immediate)
ORI rd, rs1, imm	Performs bitwise OR on register *rs1* and the sign-extended 12-bit immediate and place the result in *rd*.	x[rd] = x[rs1] | sext(immediate)
XORI rd, rs1, imm	Performs bitwise XOR on register *rs1* and the sign-extended 12-bit immediate and place the result in *rd*.	x[rd] = x[rs1] ^ sext(immediate)

※ sign-extended => sext

RISC-V RV32IM INSTRUCTION SET

Integer Register-Immediate Instructions

Mnemonic, Operands	Description	Implementation
SLLI rd, rs1, imm	Performs logical left shift on the value in register *rs1* by the shift amount held in the lower 5 bits of the immediate.	x[rd] = x[rs1] << shamt
SRLI rd, rs1, imm	Performs logical right shift on the value in register *rs1* by the shift amount held in the lower 5 bits of the immediate.	x[rd] = x[rs1] $>>_u$ shamt
SRAI rd, rs1, imm	Performs arithmetic right shift on the value in register *rs1* by the shift amount held in the lower 5 bits of the immediate.	x[rd] = x[rs1] $>>_s$ shamt

Pseudo-Instruction (假指令)

Mnemonic, Operands	Actual Instruction	Description
MV rd, rs1	ADDI rd, rs1, 0	Used to copy the value from register *rs1* to register *rd*.
NOP	ADDI x0, x0, 0	No operation.
NOT rd, rs1	XORI rd, rs1, -1	Used to bitwise negate the value of register *rs1* and place the result in *rd*.
SEQZ rd, rs1	SLTIU rd, rs1, 1	If the value of register *rs1* is zero, set the register *rd* to 1; otherwise, set it to 0.

RISC-V RV32IM INSTRUCTION SET

Integer Register-Immediate Instructions

Immediate value	Source registers 1	Funct3	Destination registers	Opcode	Instruction
imm[11:0]	rs1	000	rd	0010011	ADDI
imm[11:0]	rs1	010	rd	0010011	SLTI
imm[11:0]	rs1	011	rd	0010011	SLTIU
imm[11:0]	rs1	111	rd	0010011	ANDI
imm[11:0]	rs1	110	rd	0010011	ORI
imm[11:0]	rs1	100	rd	0010011	XORI
31 20	19 15	14 12	11 7	6 0	

Funct7	Immediate value	Source registers 1	Funct3	Destination registers	Opcode	Instruction
0000000	imm[4:0]	rs1	001	rd	0010011	SLLI
0000000	imm[4:0]	rs1	101	rd	0010011	SRLI
0100000	imm[4:0]	rs1	101	rd	0010011	SRAI
31 25	24 20	19 15	14 12	11 7	6 0	

RISC-V RV32IM INSTRUCTION SET

- **Assembly to Machine Code**
  - addi x1, x2, 321 => 0x14110093

Immediate value	Source registers 1	Funct3	Destination registers	Opcode	Instruction
0001_0100_0001	0001_0	000	0000_1	001_0011	ADDI
31 20	19 15	14 12	11 7	6 0	

  - slli x31, x5, 20 => 0x01429f93

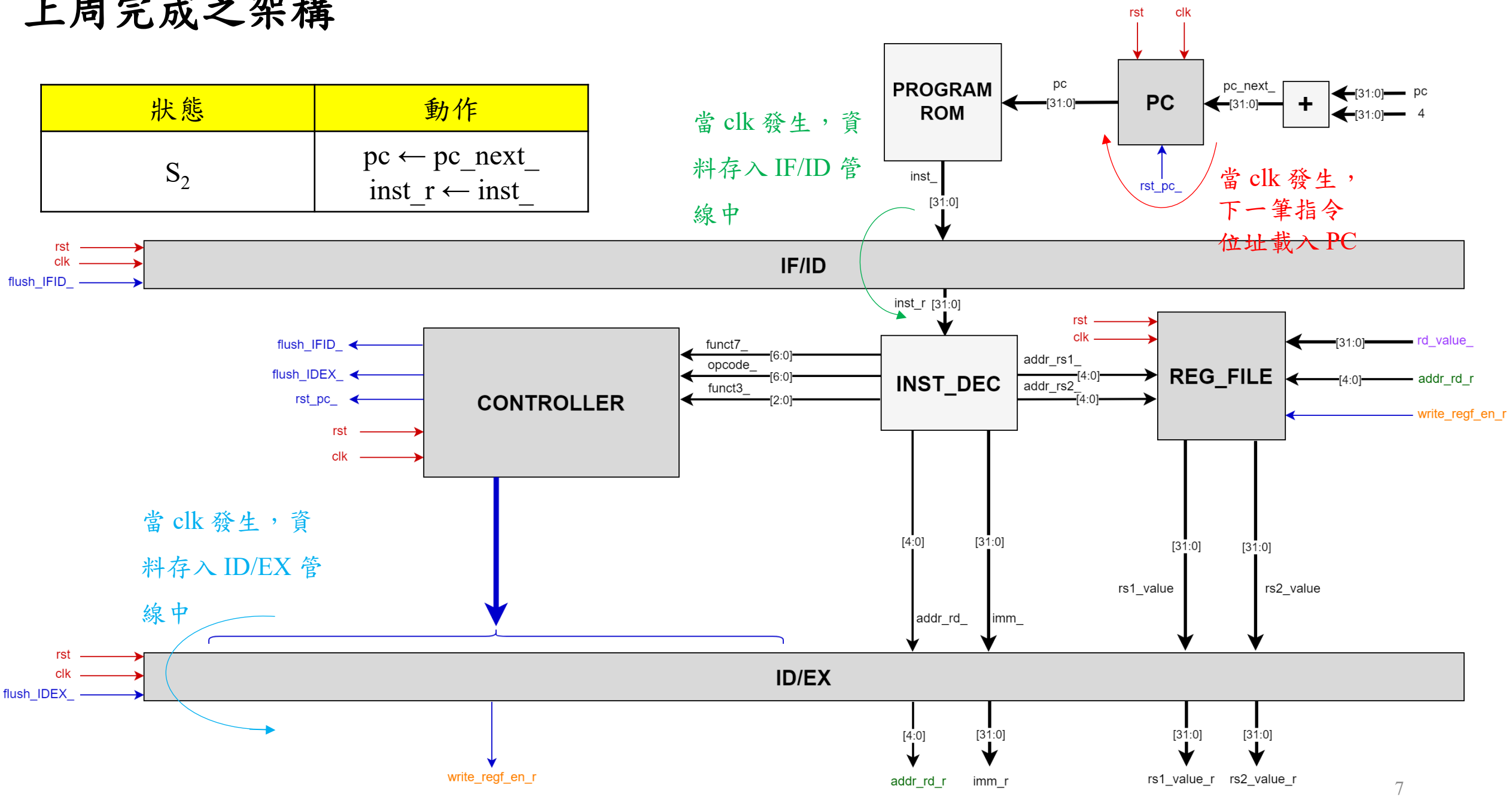
Funct7	Immediate value	Source registers 1	Funct3	Destination registers	Opcode	Instruction
0000_000	1_0100	0010_1	001	1111_1	001_0011	SLLI
31 25	24 20	19 15	14 12	11 7	6 0	

  - NOP => addi x0, x0, 0 => 0x00000013

Immediate value	Source registers 1	Funct3	Destination registers	Opcode	Instruction
0000_0000_0000	0000_0	000	0000_0	001_0011	ADDI
31 20	19 15	14 12	11 7	6 0	

上周完成之架構

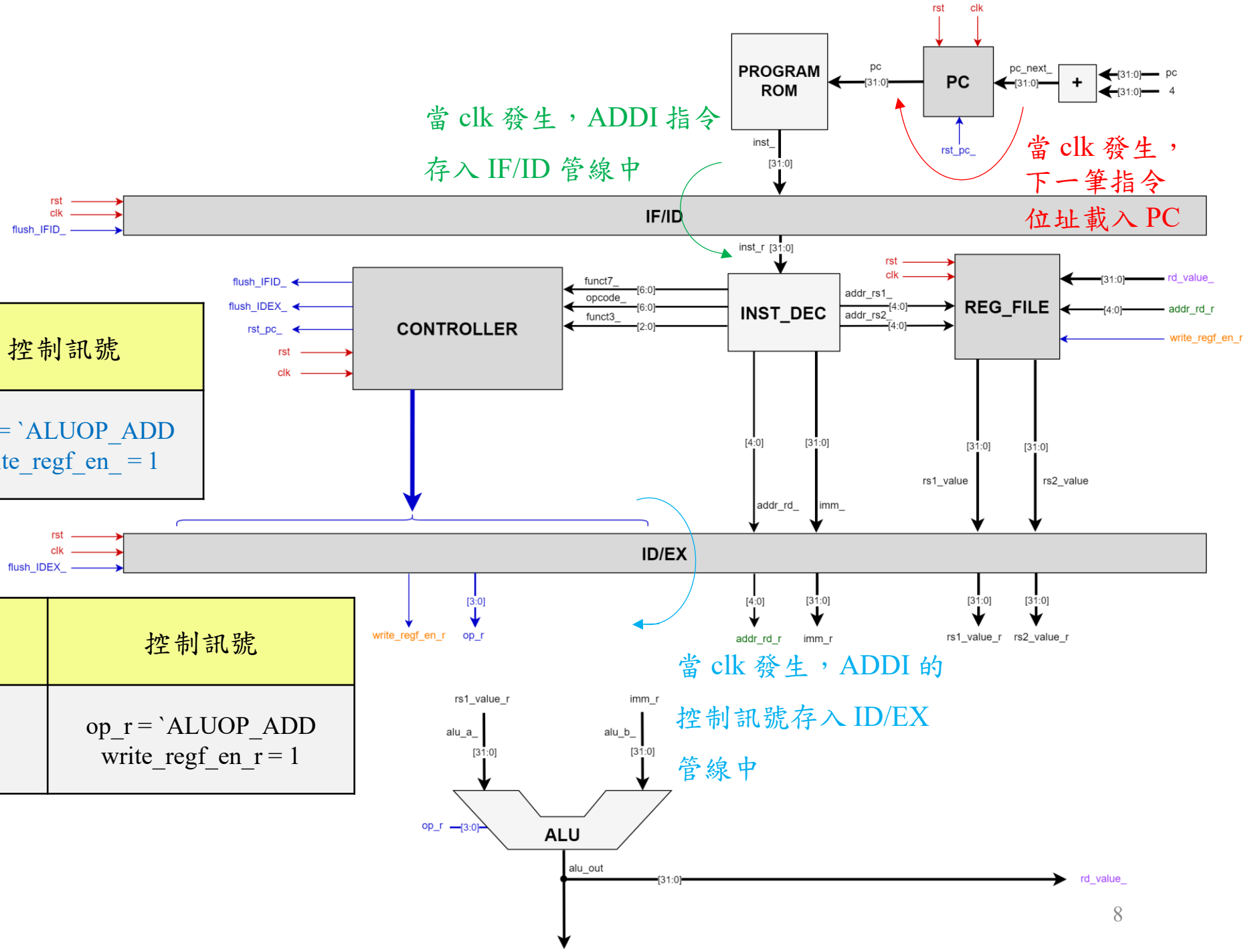
狀態	動作
S_2	pc ← pc_next_ inst_r ← inst_



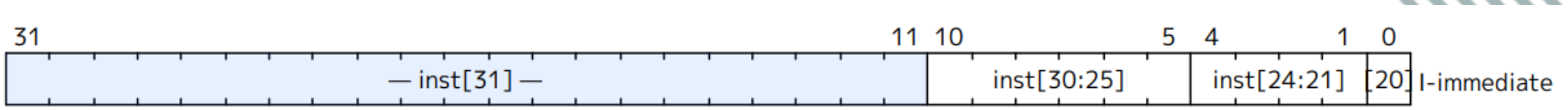
本周架構

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

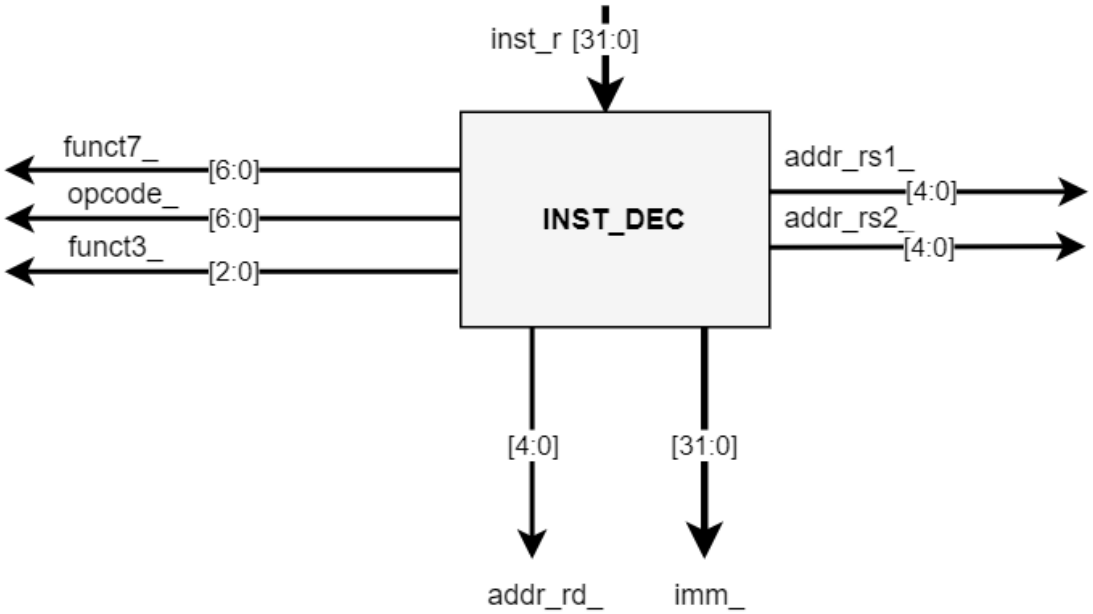
動作	控制訊號
x[addr_rd_r] ← rs1_value_r + imm_r	op_r = `ALUOP_ADD write_regf_en_r = 1



立即值擴充



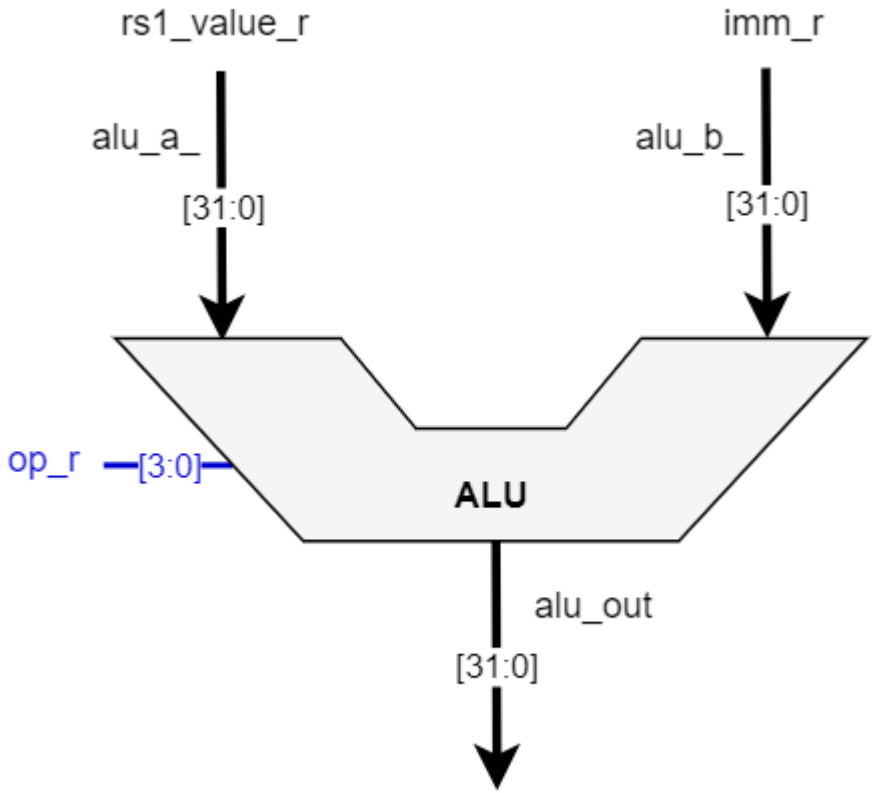
- 立即值定址指令
  - 取指令 (inst_r) 的第 20 ~ 31 個位元 (共 12 個位元) 作為立即值。
  - 做符號擴展 (sign extension) 至 32 位元。




```
`define Opcode_I        7'b0010011

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;
assign IMM_I        = {{20{inst_r[31]}}, inst_r[31:20]};

always_comb begin
    unique case (opcode_)
        `Opcode_I: imm = IMM_I;
    endcase
end
```

ALU

- 在 SystemVerilog 中，logic 預設是無號數，當要進行有號數運算時，如有號數的減法或比較，就需要用到 $signed() 函數強制轉換數值型態。




```
//有//為本周立即數定址會使用到之ALU OP
always_comb begin
    unique case(op_r)
        `ALUOP_ADD      : alu_out = alu_a_ + alu_b_;//
        `ALUOP_SUB      : alu_out = $signed(alu_a_) - $signed(alu_b_);
        `ALUOP_AND      : alu_out = alu_a_ & alu_b_;//
        `ALUOP_OR       : alu_out = alu_a_ | alu_b_;//
        `ALUOP_XOR      : alu_out = alu_a_ ^ alu_b_;//
        `ALUOP_A        : alu_out = alu_a_;
        `ALUOP_A_ADD_4  : alu_out = alu_a_ + 4;
        `ALUOP_LTU      : alu_out = alu_a_ < alu_b_;//
        `ALUOP_LT       : alu_out = $signed(alu_a_) < $signed(alu_b_);//
        `ALUOP_SLL      : alu_out = alu_a_ << alu_b_[4:0];//
        `ALUOP_SRL      : alu_out = alu_a_ >> alu_b_[4:0];//
        `ALUOP_SRA      : alu_out = $signed(alu_a_) >>> alu_b_[4:0];//
        `ALUOP_B        : alu_out = alu_b_;
        default : alu_out = alu_a_;
    endcase
end
```

```
module ALU(
    input   logic [3:0] op,
    input   logic [31:0] alu_a,
    input   logic [31:0] alu_b,

    output  logic [31:0] alu_out
);

ALU ALU_1       (
    .op         (op_r       ),
    .alu_a      (alu_a_     ),
    .alu_b      (alu_b_     ),
    //
    .alu_out    (alu_out    )
);
```

```
//------------------------------------------
//              ALU operation
//------------------------------------------

`define ALUOP_ADD       4'h0
`define ALUOP_SUB       4'h1
`define ALUOP_AND       4'h2
`define ALUOP_OR        4'h3
`define ALUOP_XOR       4'h4
`define ALUOP_A         4'h5
`define ALUOP_A_ADD_4   4'h6
`define ALUOP_LTU       4'h7
`define ALUOP_LT        4'h8
`define ALUOP_SLL       4'h9
`define ALUOP_SRL       4'hA
`define ALUOP_SRA       4'hB
`define ALUOP_B         4'hC
```

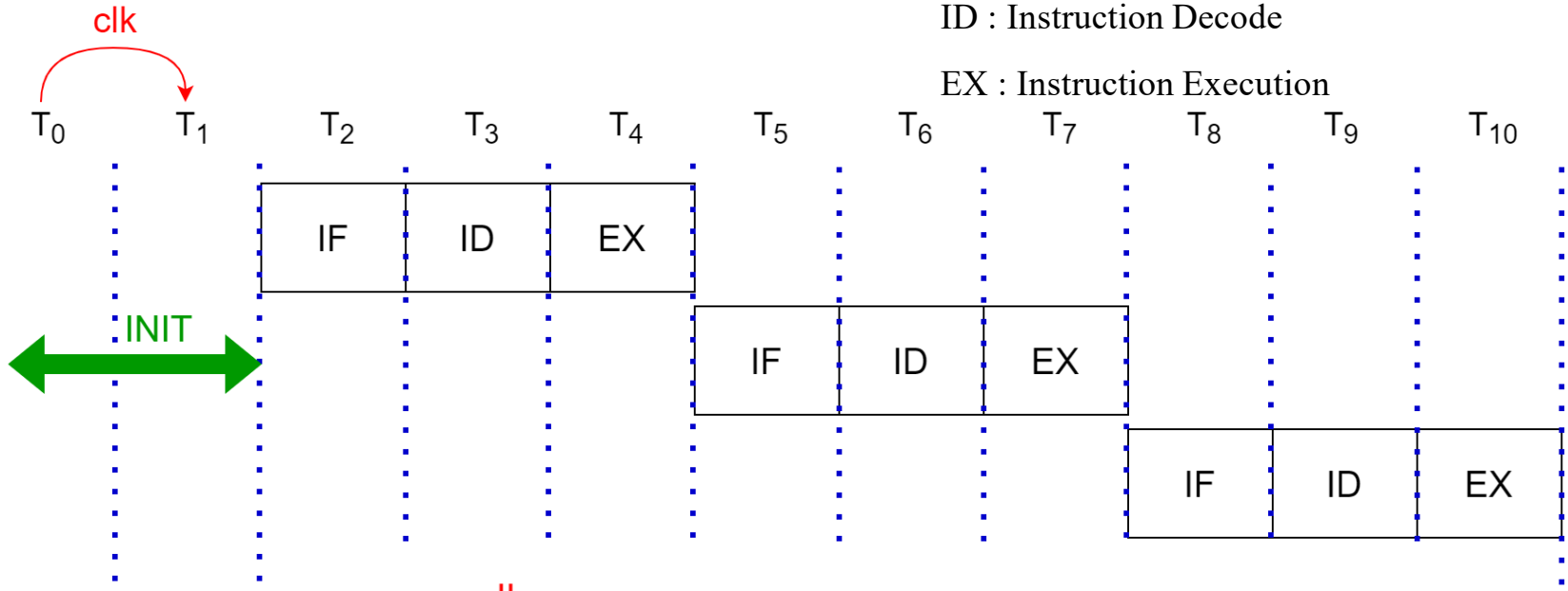
管線化 (Pipeline)(1/2)

IF : Instruction Fetch

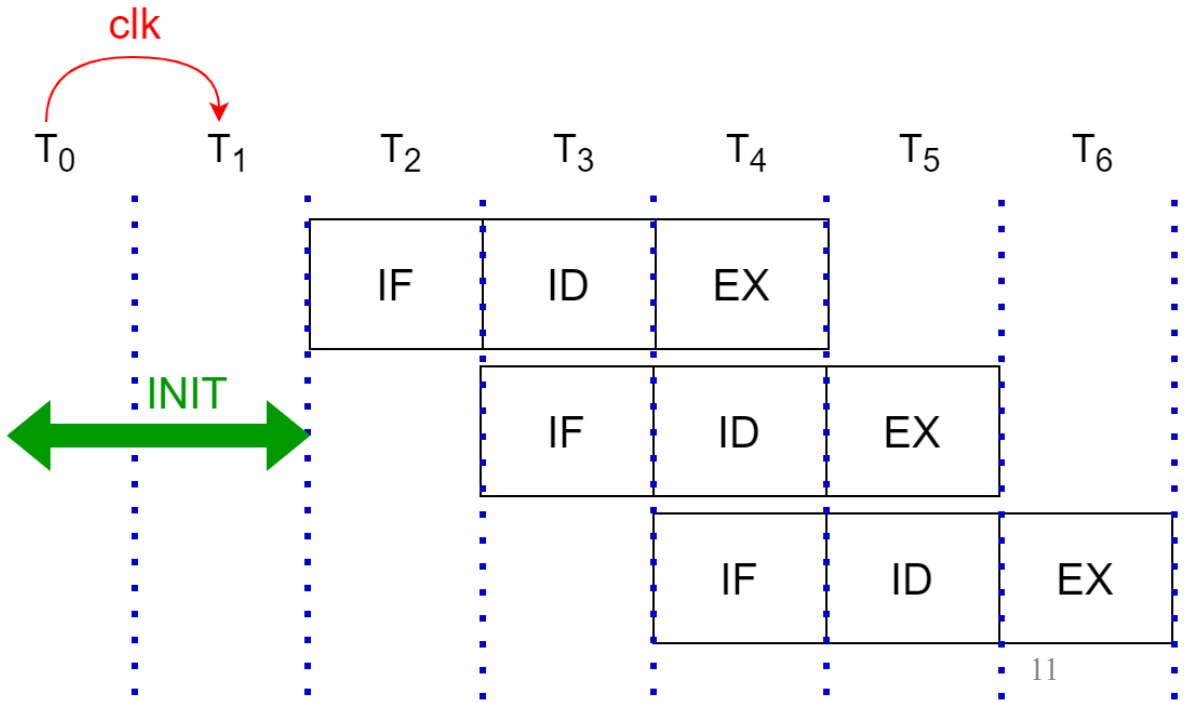
ID : Instruction Decode

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。

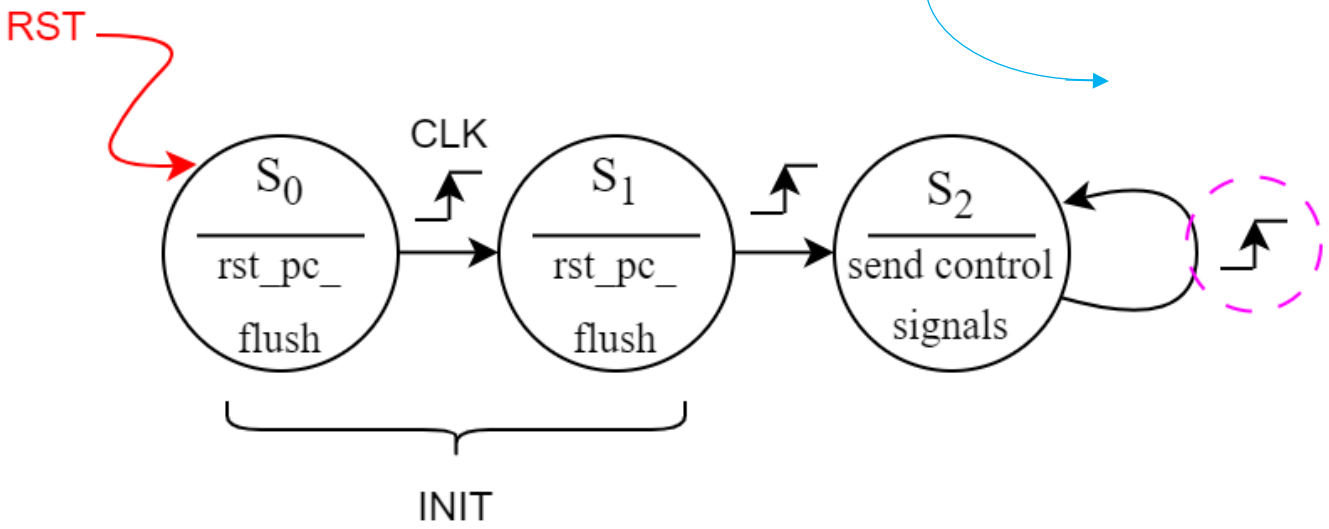


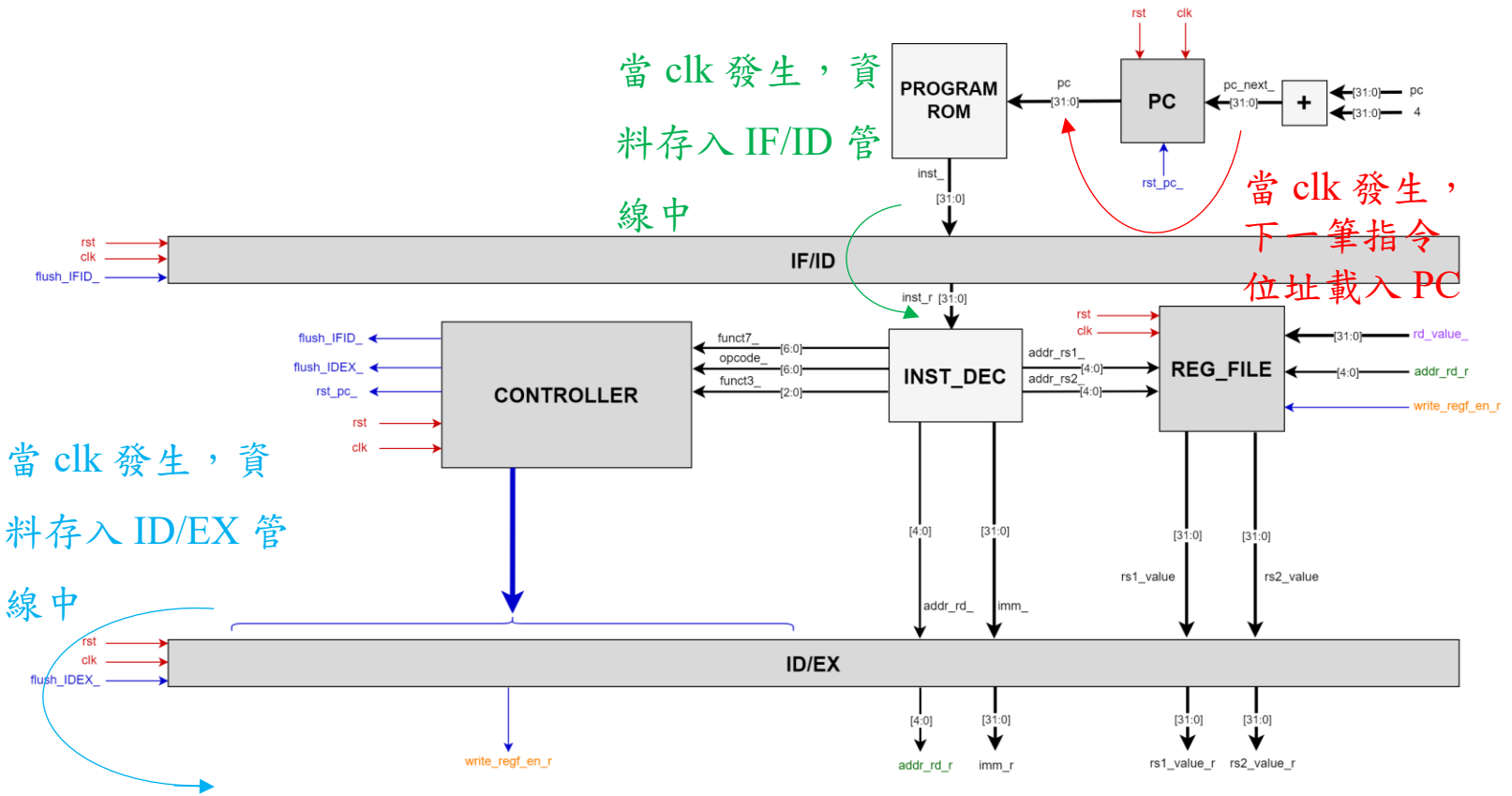
- **管線化(Pipeline)**
  - 程式脫離 INIT (Initialization) 狀態後，每一個時脈都會讀取一筆指令來執行。



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。





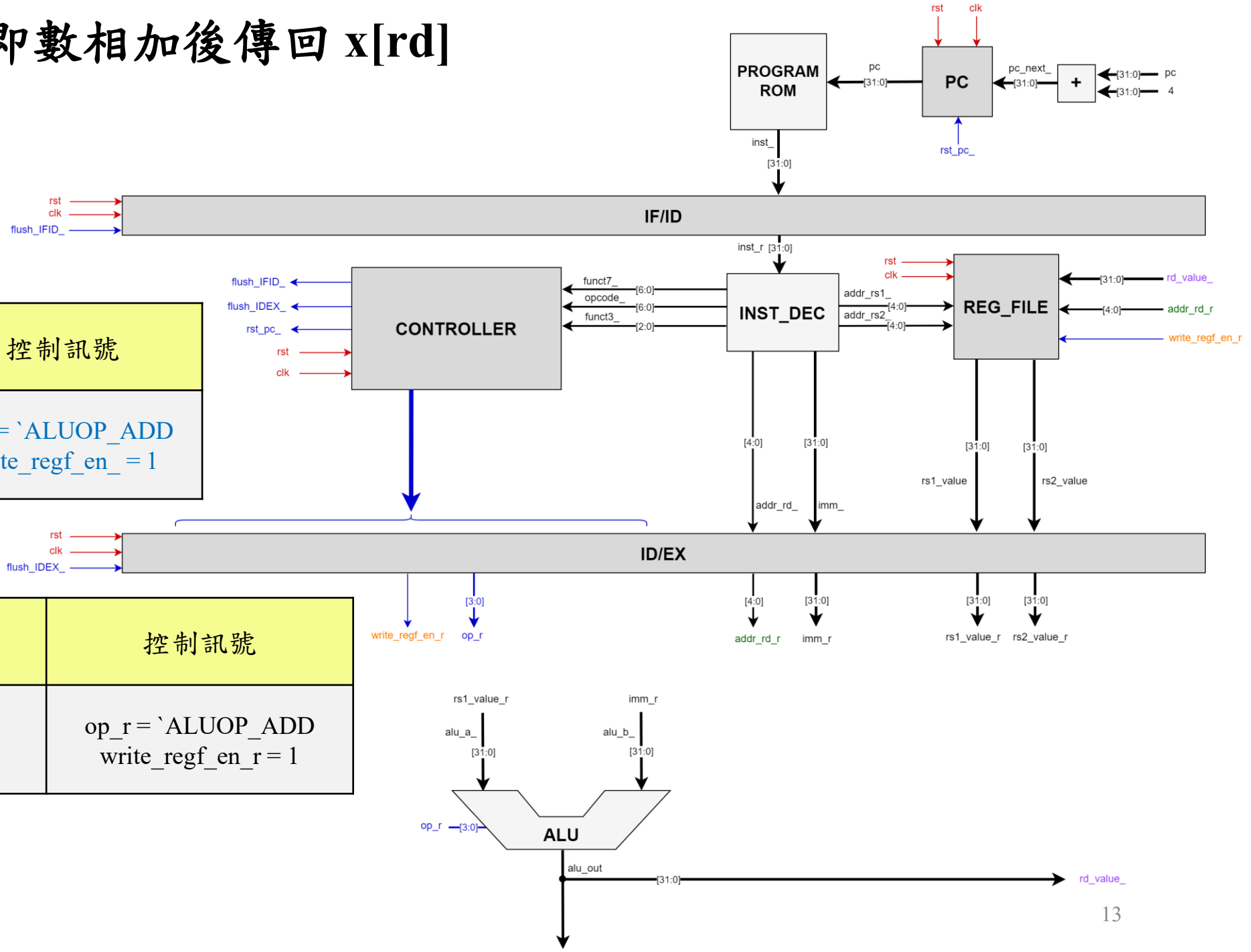
狀態	動作
S_2	pc ← pc_next_ inst_r ← inst_

ADDI : x[rs1] 和立即數相加後傳回 x[rd]

```
`define F_ADDI          3'b000
`define F_SLTI          3'b010
`define F_SLTIU         3'b011
`define F_XORI          3'b100
`define F_ORI           3'b110
`define F_ANDI          3'b111
`define F_SLLI          3'b001
`define F_SRLI_SRAI     3'b101
```

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

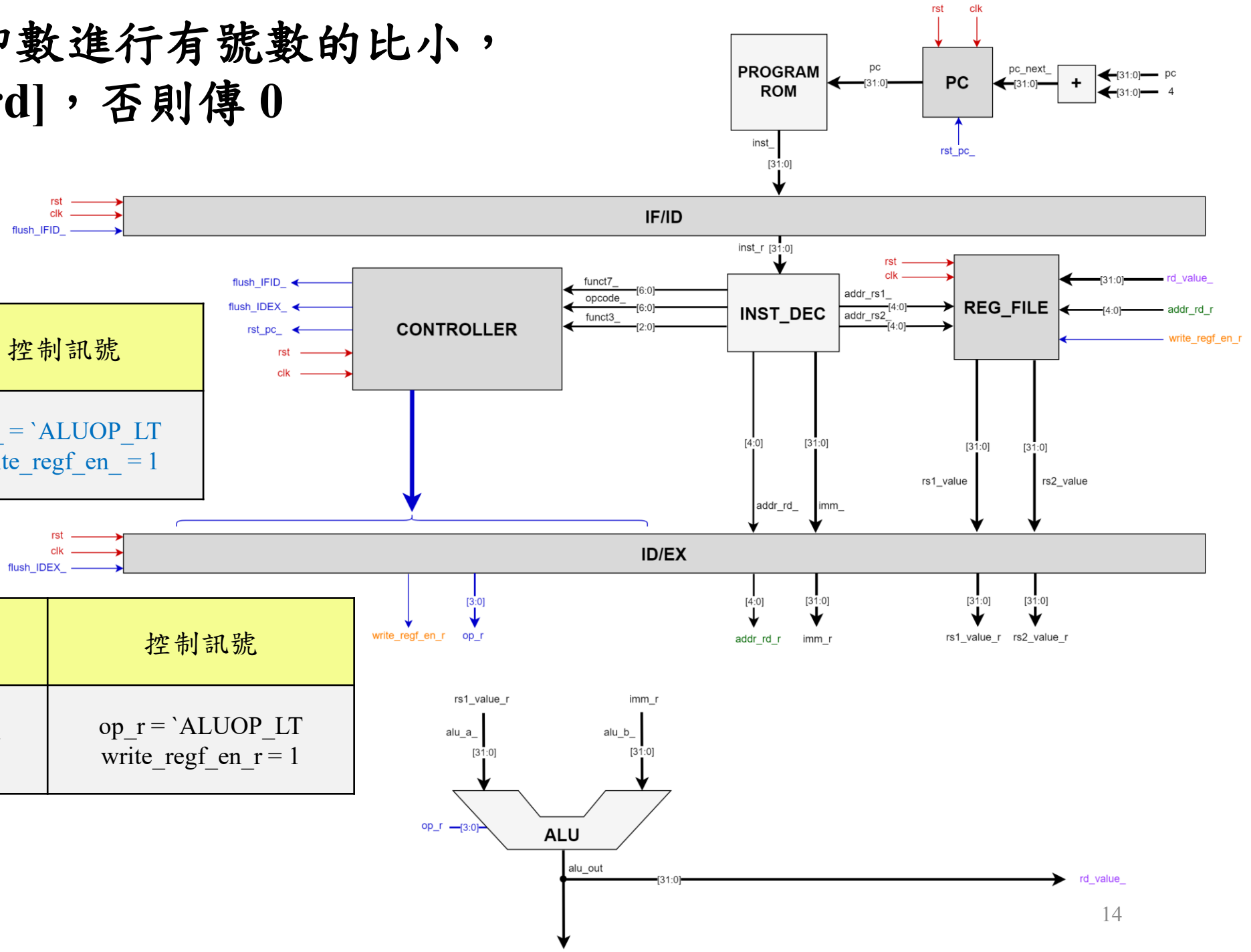
動作	控制訊號
x[addr_rd_r] ← rs1_value_r + imm_r	op_r = `ALUOP_ADD write_regf_en_r = 1



SLTI : x[rs1] 和立即數進行有號數的比小，若真則傳回 1 至 x[rd]，否則傳 0

動作	控制訊號
if ((opcode_ == `Opcode_I) && (funct3_ == `F_SLTI)) 發出 SLTI 的控制訊號	op_ = `ALUOP_LT write_regf_en_ = 1

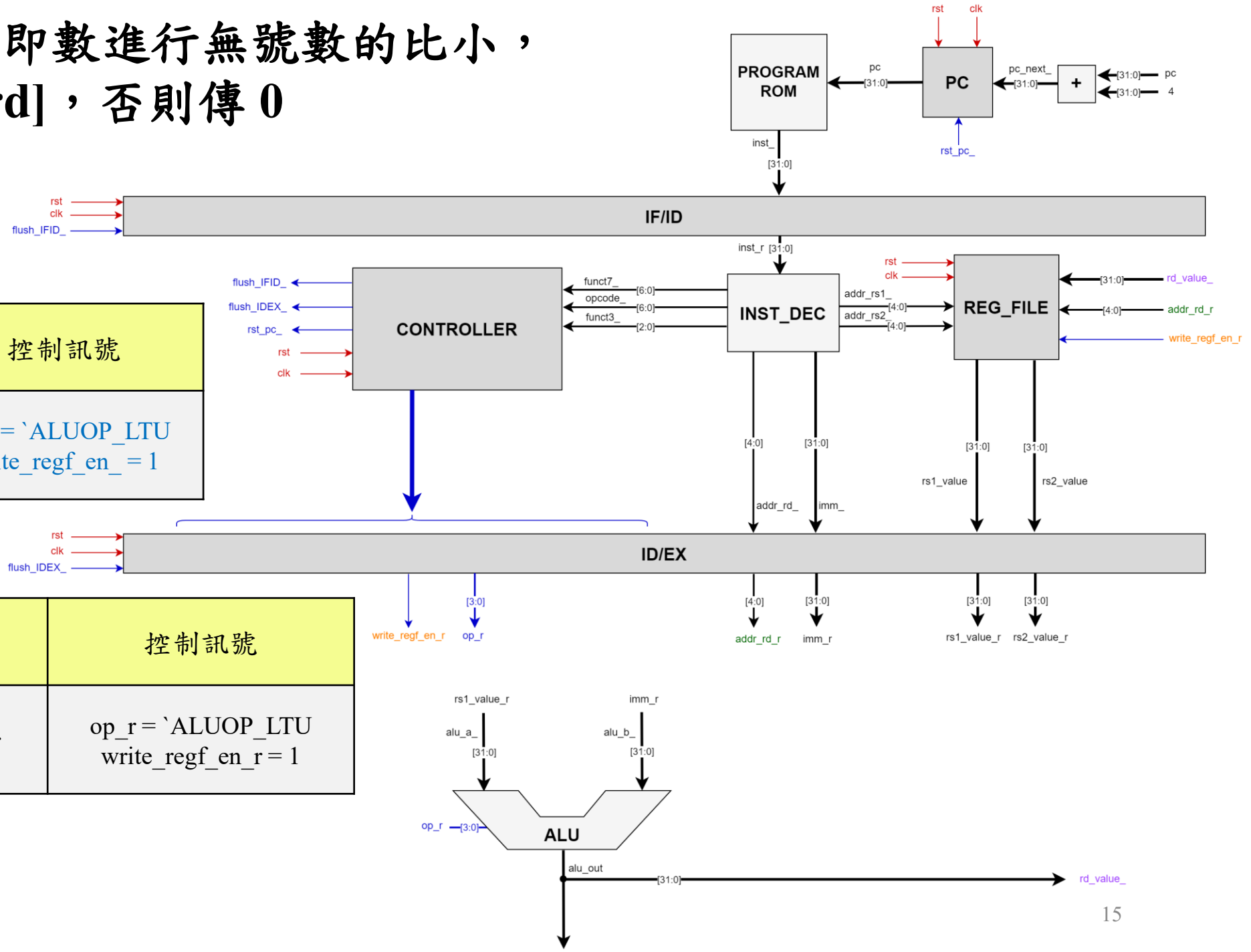
動作	控制訊號
x[addr_rd_r] ← rs1_value_r <$_s$ imm_r	op_r = `ALUOP_LT write_regf_en_r = 1



SLTIU：x[rs1] 和立即數進行無號數的比小，若真則傳回 1 至 x[rd]，否則傳 0

動作	控制訊號
if ((opcode_ == `Opcode_I) && (funct3_ == `F_SLTIU)) 發出 SLTIU 的控制訊號	op_ = `ALUOP_LTU write_regf_en_ = 1

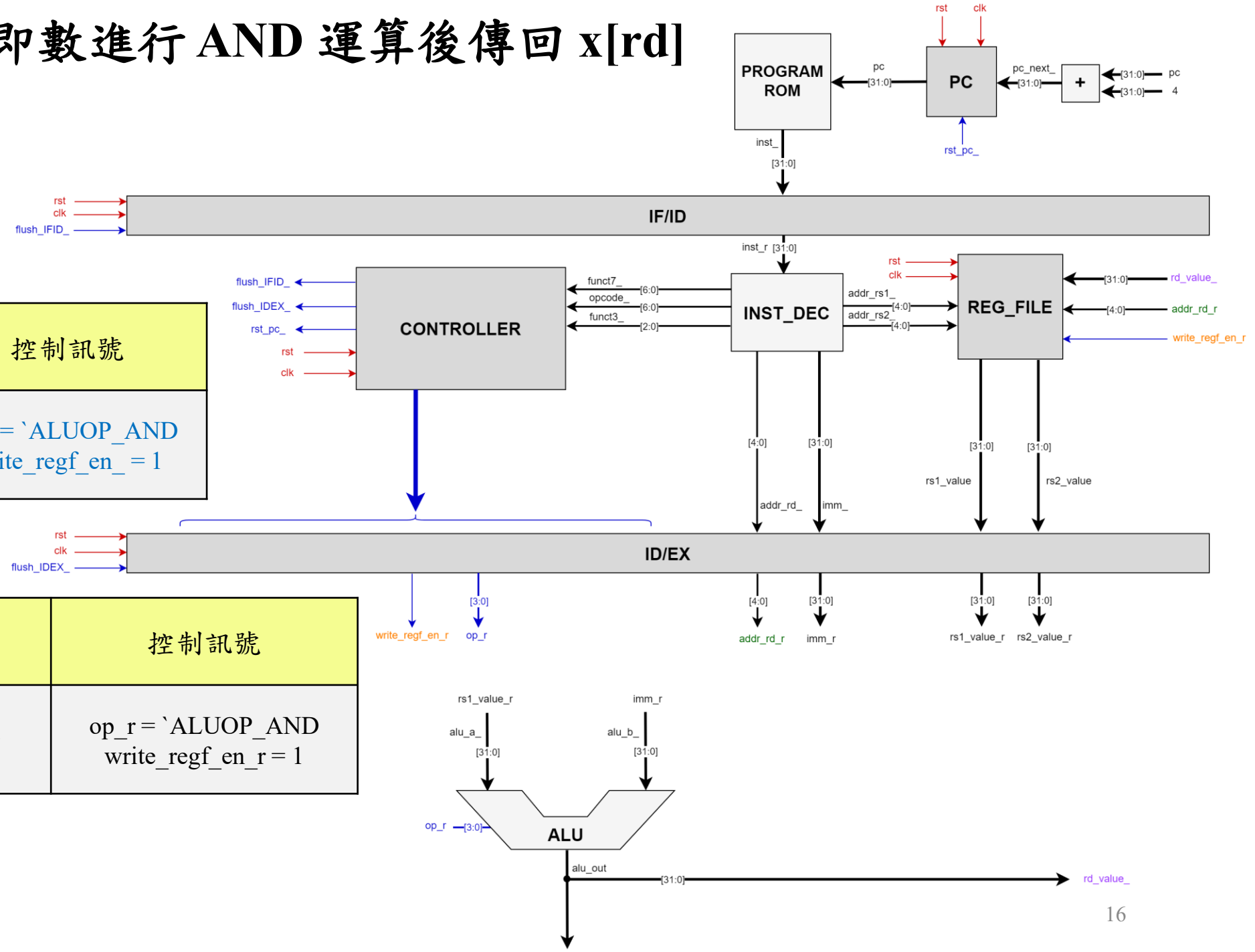
動作	控制訊號
x[addr_rd_r] ← rs1_value_r $<_u$ imm_r	op_r = `ALUOP_LTU write_regf_en_r = 1



ANDI : x[rs1] 和立即數進行 AND 運算後傳回 x[rd]

動作	控制訊號
if ((opcode_ == `Opcode_I) && (funct3_ == `F_ANDI)) 發出 ANDI 的控制訊號	op_ = `ALUOP_AND write_regf_en_ = 1

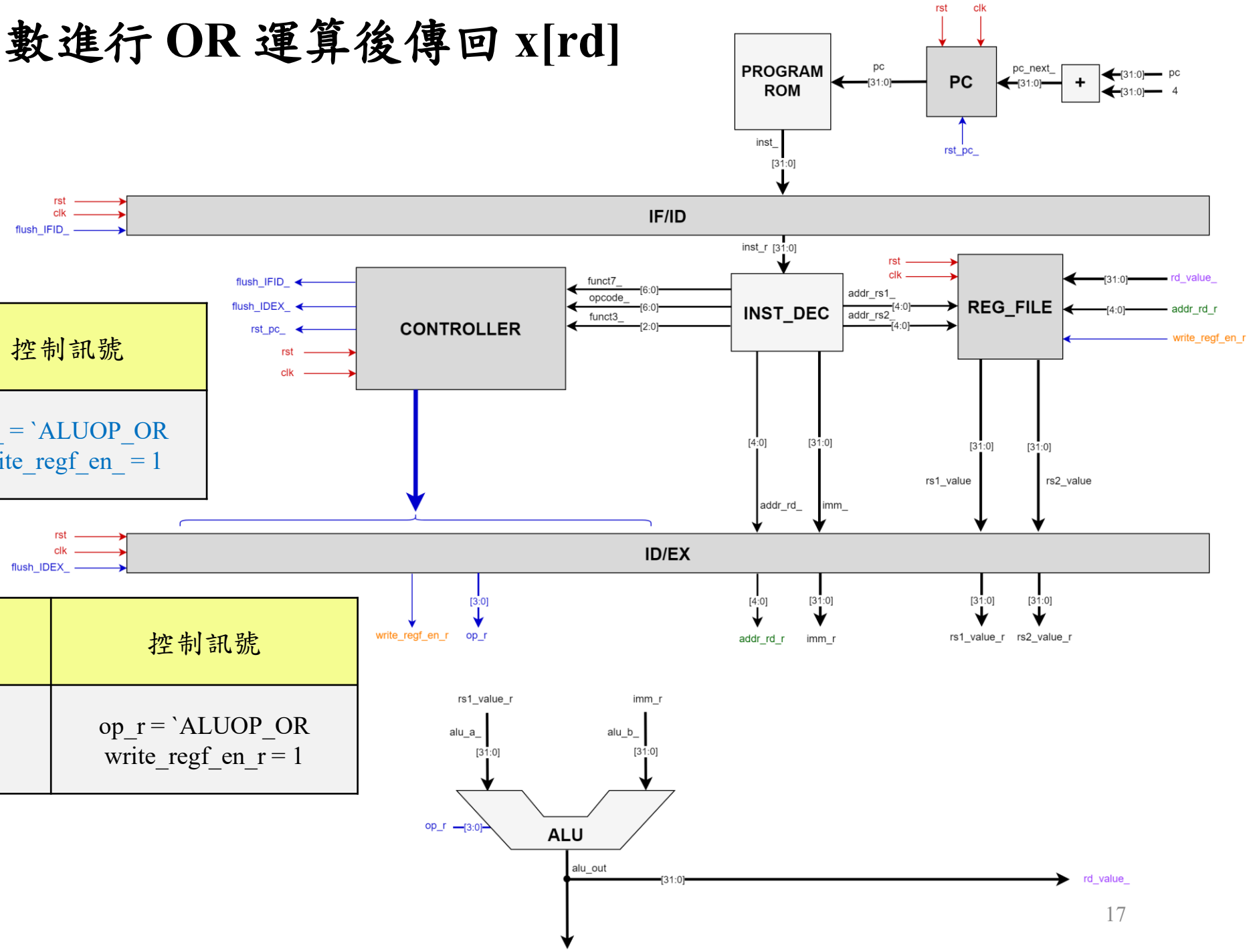
動作	控制訊號
x[addr_rd_r] ← rs1_value_r & imm_r	op_r = `ALUOP_AND write_regf_en_r = 1



ORI : x[rs1] 和立即數進行 OR 運算後傳回 x[rd]

動作	控制訊號
if ((opcode_ == `Opcode_I) && (funct3_ == `F_ORI)) 發出 ORI 的控制訊號	op_ = `ALUOP_OR write_regf_en_ = 1

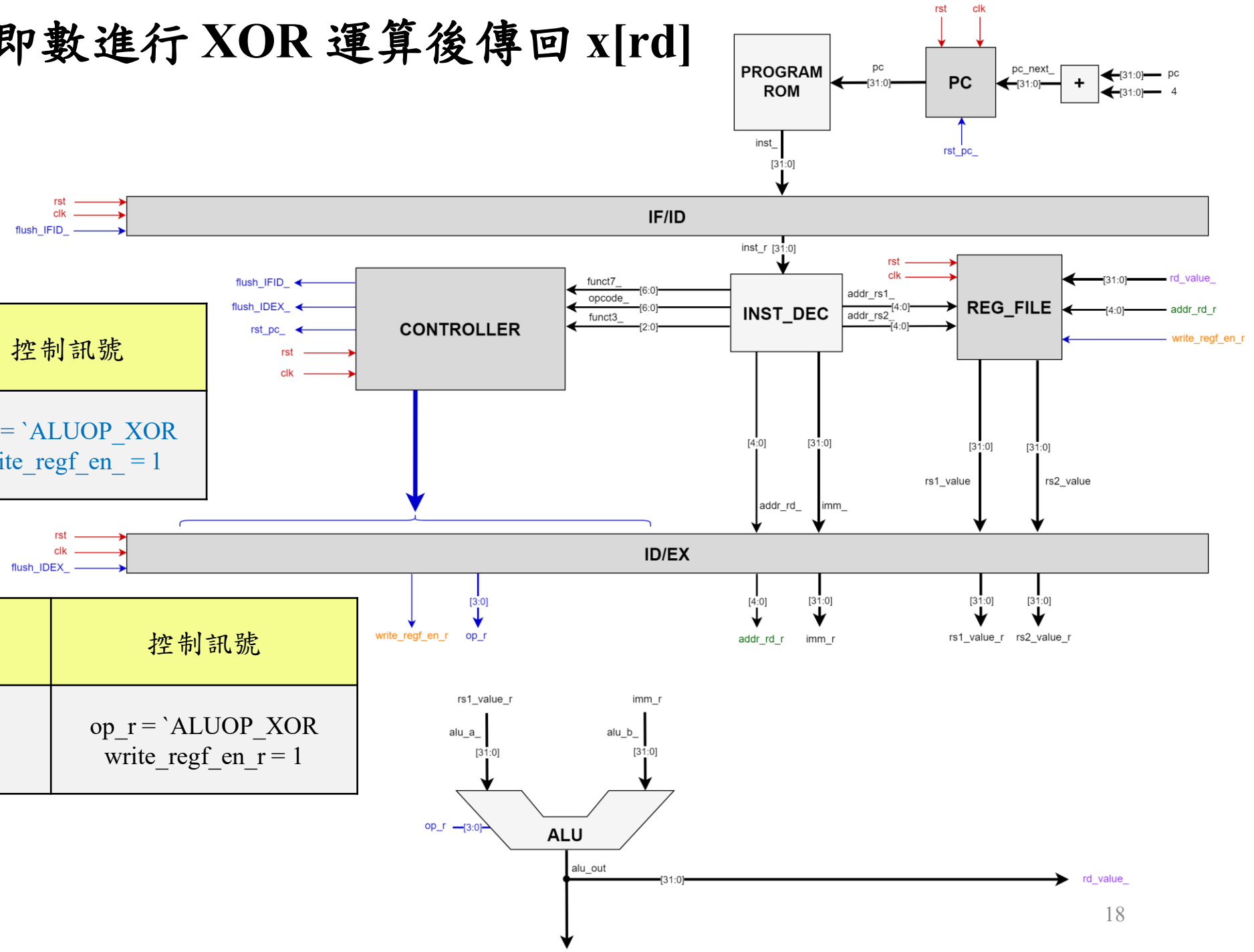
動作	控制訊號
x[addr_rd_r] ← rs1_value_r | imm_r	op_r = `ALUOP_OR write_regf_en_r = 1



XORI：x[rs1] 和立即數進行 XOR 運算後傳回 x[rd]

動作	控制訊號
if ((opcode_ == `Opcode_I) && (funct3_ == `F_XORI)) 發出 XORI 的控制訊號	op_ = `ALUOP_XOR write_regf_en_ = 1

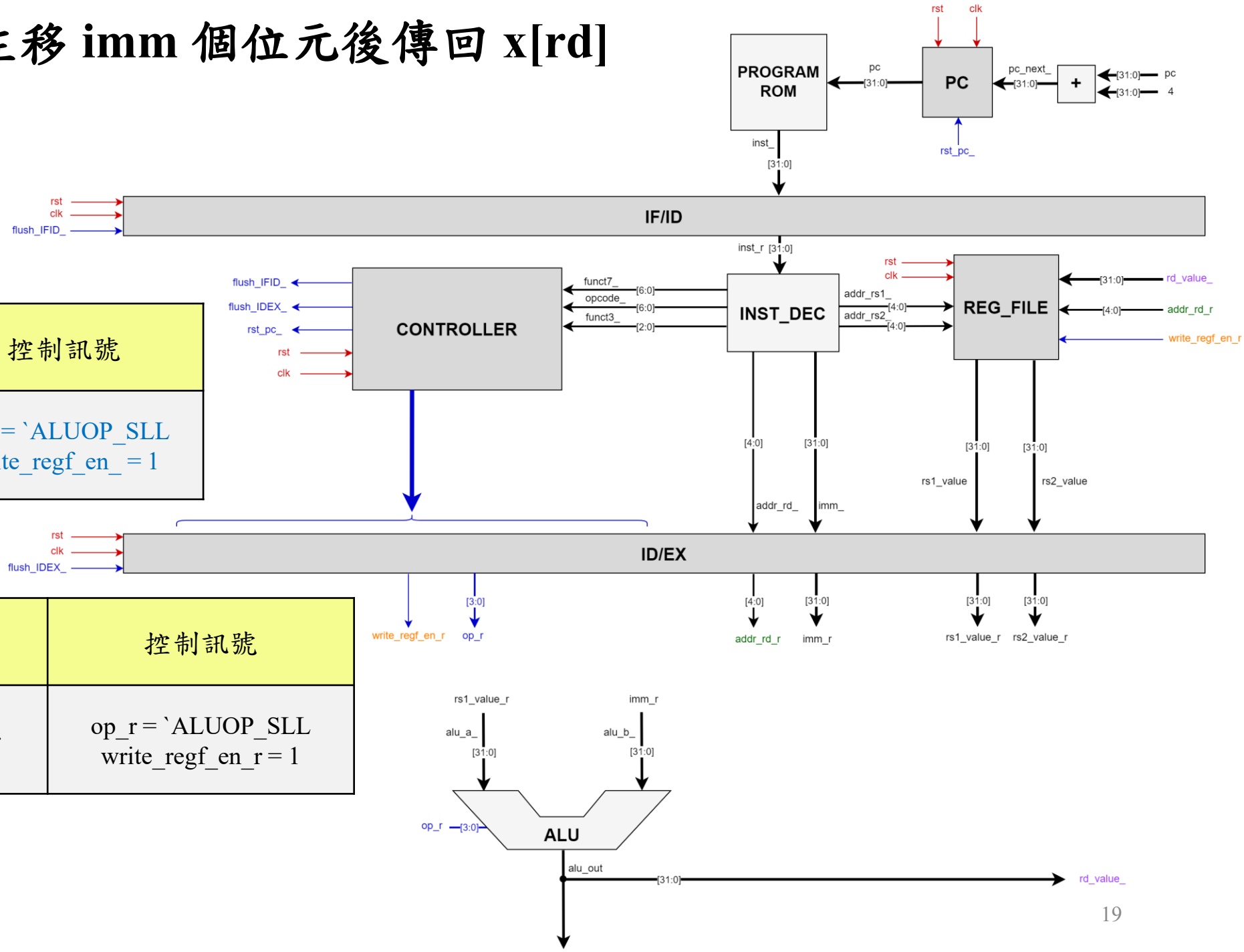
動作	控制訊號
x[addr_rd_r] ← rs1_value_r ^ imm_r	op_r = `ALUOP_XOR write_regf_en_r = 1



SLLI : x[rs1] 邏輯左移 imm 個位元後傳回 x[rd]

動作	控制訊號
if ((opcode_ == `Opcode_I) && (funct3_ == `F_SLLI)) 發出 SLLI 的控制訊號	op_ = `ALUOP_SLL write_regf_en_ = 1

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

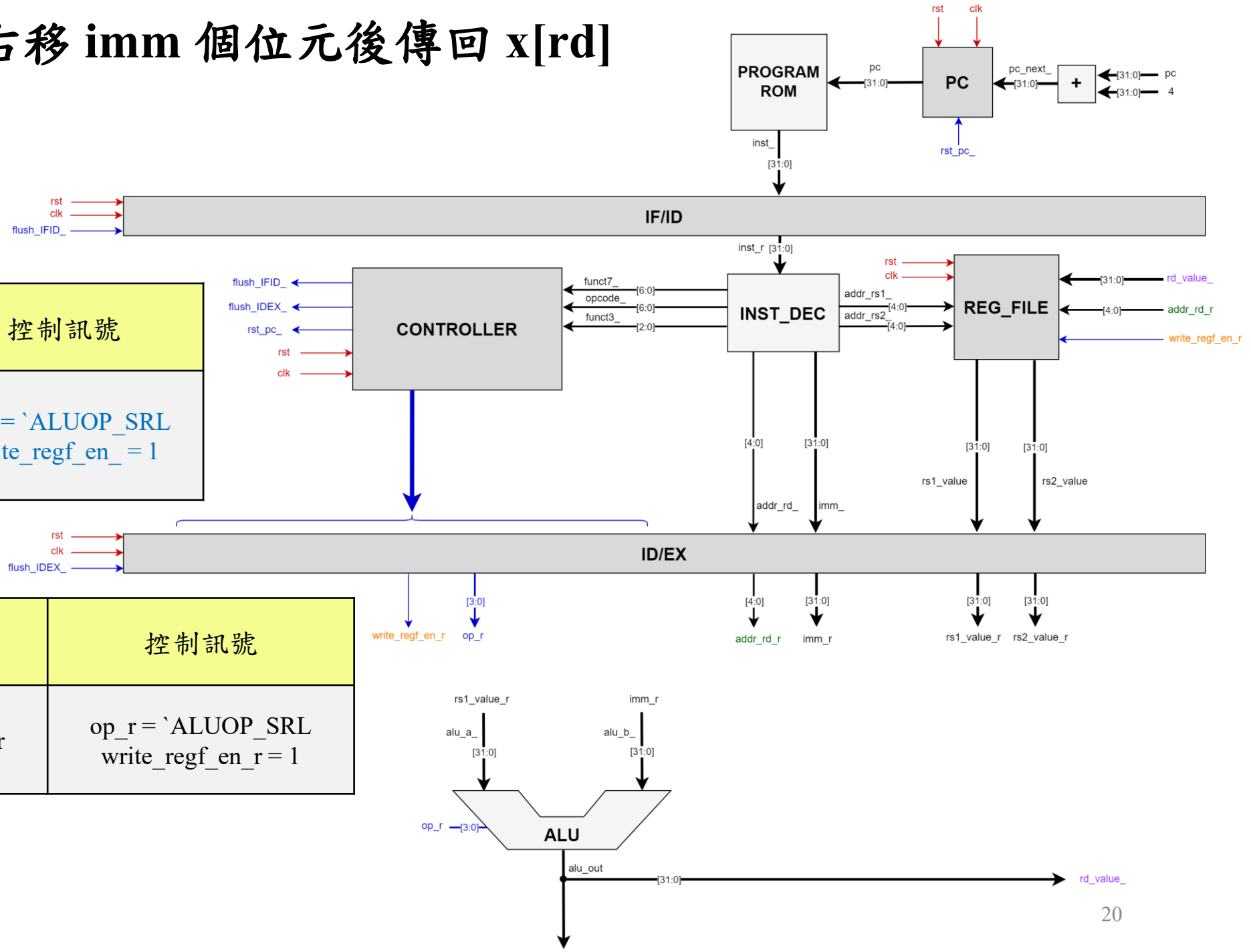


SRLI : x[rs1] 邏輯右移 imm 個位元後傳回 x[rd]

```
`define F7_SRLI      7'b0000000
`define F7_SRAI      7'b0100000
```

動作	控制訊號
if ((opcode_ == `Opcode_I) && (funct3_ == `F_SRLI_SRAI) && (funct7_==`F7_SRLI)) 發出 SRLI 的控制訊號	op_ = `ALUOP_SRL write_regf_en_ = 1

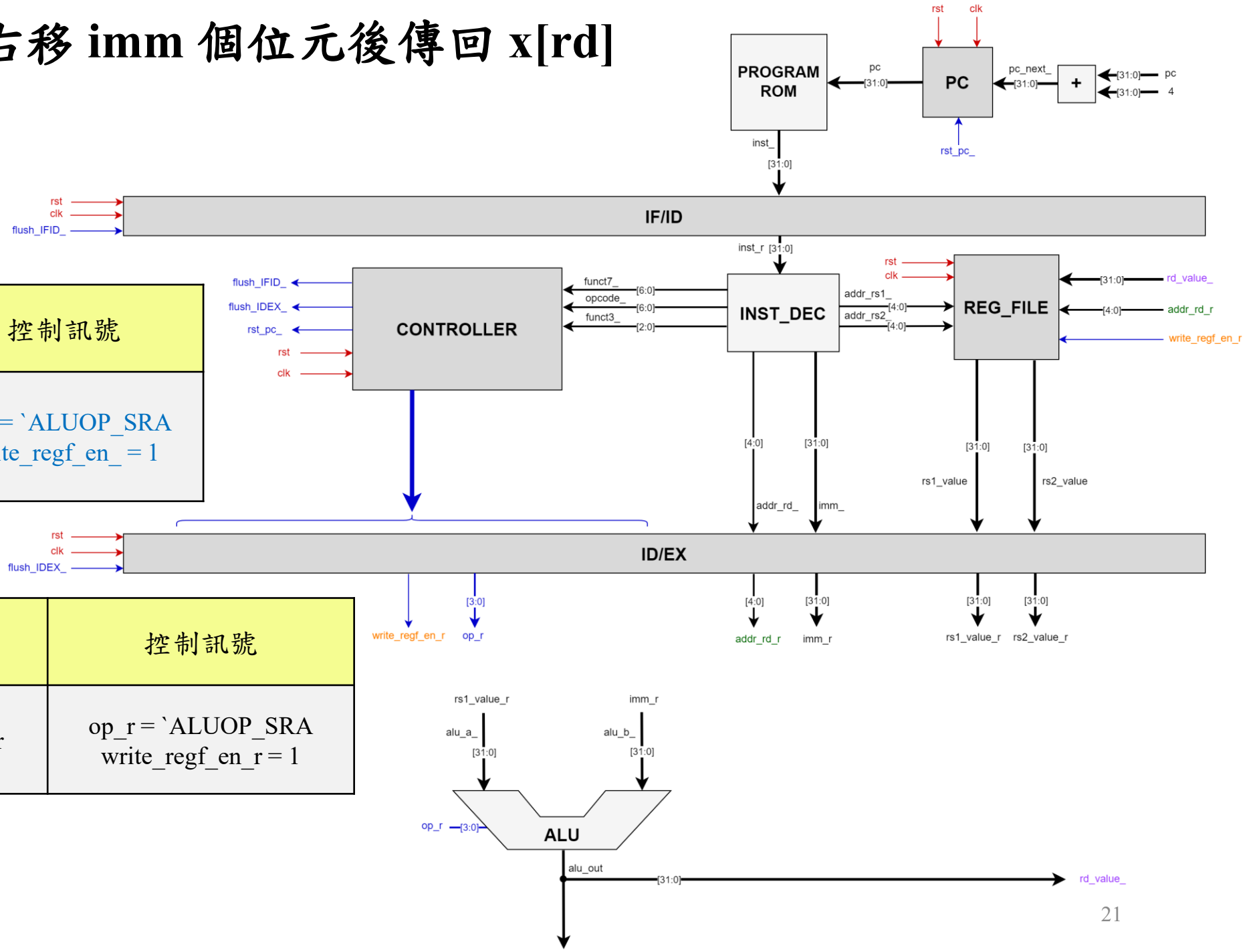
動作	控制訊號
x[addr_rd_r] ← rs1_value_r >>$_u$ imm_r	op_r = `ALUOP_SRL write_regf_en_r = 1



SRAI : x[rs1] 算術右移 imm 個位元後傳回 x[rd]

動作	控制訊號
if ((opcode_ == `Opcode_I) && (funct3_ == `F_SRLI_SRAI) && (funct7_==`F7_SRAI)) 發出 SRAI 的控制訊號	op_ = `ALUOP_SRA write_regf_en_ = 1

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



上課實作

- 因為 RAW 的危障問題，三個相連的指令間不能對同一暫存器進行先寫後讀。
- 在管線化的運作下，第一筆指令需要三個時脈才能計算出數值存入 x1，因此第二第三筆指令所讀取的 x1 值為非使用者預期的。(危障問題會在後續課程解決)

  如 addi x1, x0, 1 (寫入 x1)

  addi x2, x1, 2 (讀取 x1)

  addi x3, x1, 3 (讀取 x1)

- 若非要讀取 x1 則可以在寫入的指令後加上兩個 NOP 指令

  如 addi x1, x0, 1 (寫入 x1)

  NOP

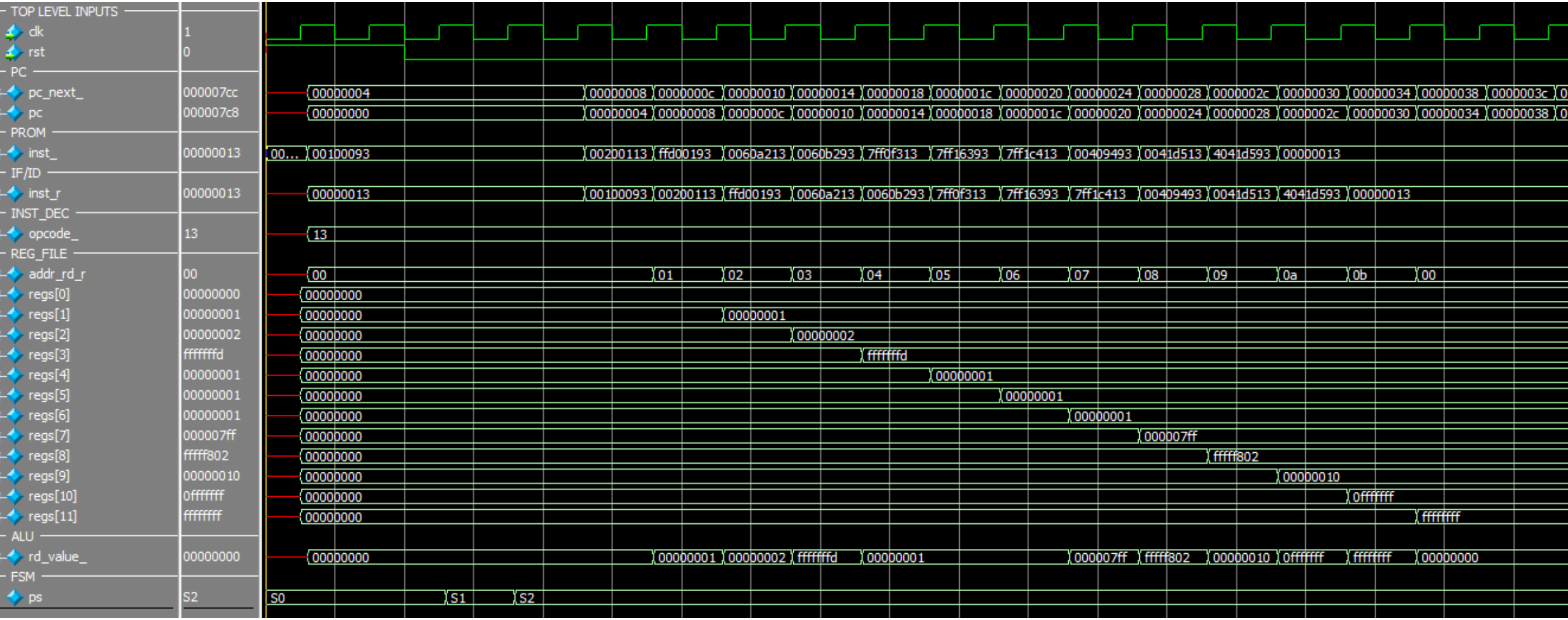
  NOP

  addi x2, x1, 2 (讀取 x1)

  addi x3, x1, 3 (讀取 x1)

```
`timescale 1ns/100ps
module Program_Rom(
    input   logic [31:0] Rom_addr,
    output  logic [31:0] Rom_data
);

    always_comb begin
        case (Rom_addr)
            32'h0  : Rom_data = 32'h00100093;   //addi  x1, x0, 1
            32'h4  : Rom_data = 32'h00200113;   //addi  x2, x0, 2
            32'h8  : Rom_data = 32'hffd00193;   //addi  x3, x0, -3
            32'hc  : Rom_data = 32'h0060a213;   //slti  x4, x1, 6
            32'h10 : Rom_data = 32'h0060b293;   //sltiu x5, x1, 6
            32'h14 : Rom_data = 32'h7ff0f313;   //andi  x6, x1, 2047
            32'h18 : Rom_data = 32'h7ff16393;   //ori   x7, x2, 2047
            32'h1c : Rom_data = 32'h7ff1c413;   //xori  x8, x3, 2047
            32'h20 : Rom_data = 32'h00409493;   //slli  x9, x1, 4
            32'h24 : Rom_data = 32'h0041d513;   //srli  x10, x3, 4
            32'h28 : Rom_data = 32'h4041d593;   //srai  x11, x3, 4
            default: Rom_data = 32'h00000013;   //NOP
        endcase
    end
endmodule
```

上課實作


TOP LEVEL INPUTS
clk 1
rst 0
PC
pc_next_ 000007cc
pc 000007c8
PROM
inst_ 00000013
IF/ID
inst_r 00000013
INST_DEC
opcode_ 13
REG_FILE
addr_rd_r 00
regs[0] 00000000
regs[1] 00000001
regs[2] 00000002
regs[3] fffffffd
regs[4] 00000001
regs[5] 00000001
regs[6] 00000001
regs[7] 000007ff
regs[8] fffff802
regs[9] 00000010
regs[10] 0fffffff
regs[11] ffffffff
ALU
rd_value_ 00000000
FSM
ps S2
00000004 00000008 0000000c 00000010 00000014 00000018 0000001c 00000020 00000024 00000028 0000002c 00000030 00000034 00000038
00000000 00000004 00000008 0000000c 00000010 00000014 00000018 0000001c 00000020 00000024 00000028 0000002c 00000030 00000034 00000038
00... 00100093 00200113 ffd00193 0060a213 0060b293 7ff0f313 7ff16393 7ff1c413 00409493 0041d513 4041d593 00000013
00000013 00100093 00200113 ffd00193 0060a213 0060b293 7ff0f313 7ff16393 7ff1c413 00409493 0041d513 4041d593 00000013
13
00 01 02 03 04 05 06 07 08 09 0a 0b 00
00000000
00000001
00000002
fffffffd
00000001
00000001
00000001
000007ff
fffff802
00000010
0fffffff
ffffffff
00000000 00000001 00000002 fffffffd 00000001 000007ff fffff802 00000010 0fffffff ffffffff 00000000
S0 S1 S2

THANK YOU