計算機系統設計

條件分支指令

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

RISC-V RV32IM INSTRUCTION SET

Conditional Branch Instructions

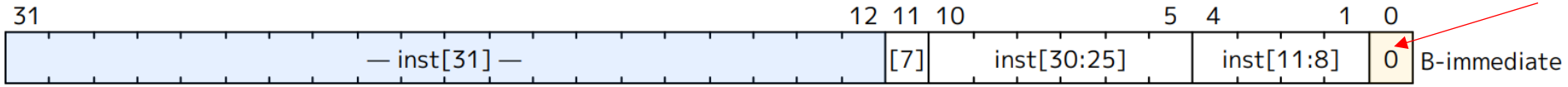
Mnemonic, Operands	Description	Implementation
BEQ rs1, rs2, offset	Take the branch if registers *rs1* and *rs2* are equal.	if (x[rs1] == x[rs2]) pc += sext(offset)
BNE rs1, rs2, offset	Take the branch if registers *rs1* and *rs2* are not equal.	if (x[rs1] != x[rs2]) pc += sext(offset)
BLT rs1, rs2, offset	Take the branch if registers *rs1* is less than *rs2*, using signed comparison.	if (x[rs1] $<_s$ x[rs2]) pc += sext(offset)
BGE rs1, rs2, offset	Take the branch if registers *rs1* is greater than or equal to *rs2*, using signed comparison.	if (x[rs1] $>=_s$ x[rs2]) pc += sext(offset)
BLTU rs1, rs2, offset	Take the branch if registers *rs1* is less than *rs2,* using unsigned comparison.	if (x[rs1] $<_u$ x[rs2]) pc += sext(offset)
BGEU rs1, rs2, offset	Take the branch if registers *rs1* is greater than or equal to *rs2*, using unsigned comparison.	if (x[rs1] $>=_u$ x[rs2]) pc += sext(offset)

RISC-V RV32IM INSTRUCTION SET

Conditional Branch Instructions

Immediate value	Source registers 2	Source registers 1	Funct3	Immediate value	Opcode	Instruction
{ imm[12], imm[10:5] }	rs2	rs1	000	{ imm[4:1], imm[11] }	1100011	BEQ
{ imm[12], imm[10:5] }	rs2	rs1	001	{ imm[4:1], imm[11] }	1100011	BNE
{ imm[12], imm[10:5] }	rs2	rs1	100	{ imm[4:1], imm[11] }	1100011	BLT
{ imm[12], imm[10:5] }	rs2	rs1	101	{ imm[4:1], imm[11] }	1100011	BGE
{ imm[12], imm[10:5] }	rs2	rs1	110	{ imm[4:1], imm[11] }	1100011	BLTU
{ imm[12], imm[10:5] }	rs2	rs1	111	{ imm[4:1], imm[11] }	1100011	BGEU
31 25	24 20	19 15	14 12	11 7	6 0	

Branch 指令之立即值 (offset)



RISC-V 的所有指令中長度最短為兩個位元組 (Byte)，因此 offset 的第零個位元永遠設為 0，以此確保分支跳躍的位置一定是 2 的倍數。

Branch 指令的跳轉範圍：$-2^{10} \sim 2^{10}-1$ 個 RV32I 指令，意即 ± 4096 Bytes

RISC-V RV32IM INSTRUCTION SET

- **Assembly to Machine Code**
  - beq x2, x1, label => beq x2, x1, 0x10 => 0x00110863

Immediate value	Source registers 2	Source registers 1	Funct3	Immediate value	Opcode	Instruction
0000_000	0_0001	0001_0	000	1000_0	110_0011	BEQ
31 25	24 20	19 15	14 12	11 7	6 0	

指令位址	指令
0x10	beq x2, x1, label
0x14	and x6, x4, x5
0x18	xor x7, x4, x5
0x1C	andi x8, x4, 1
0x20	label: andi x3, x0, 2

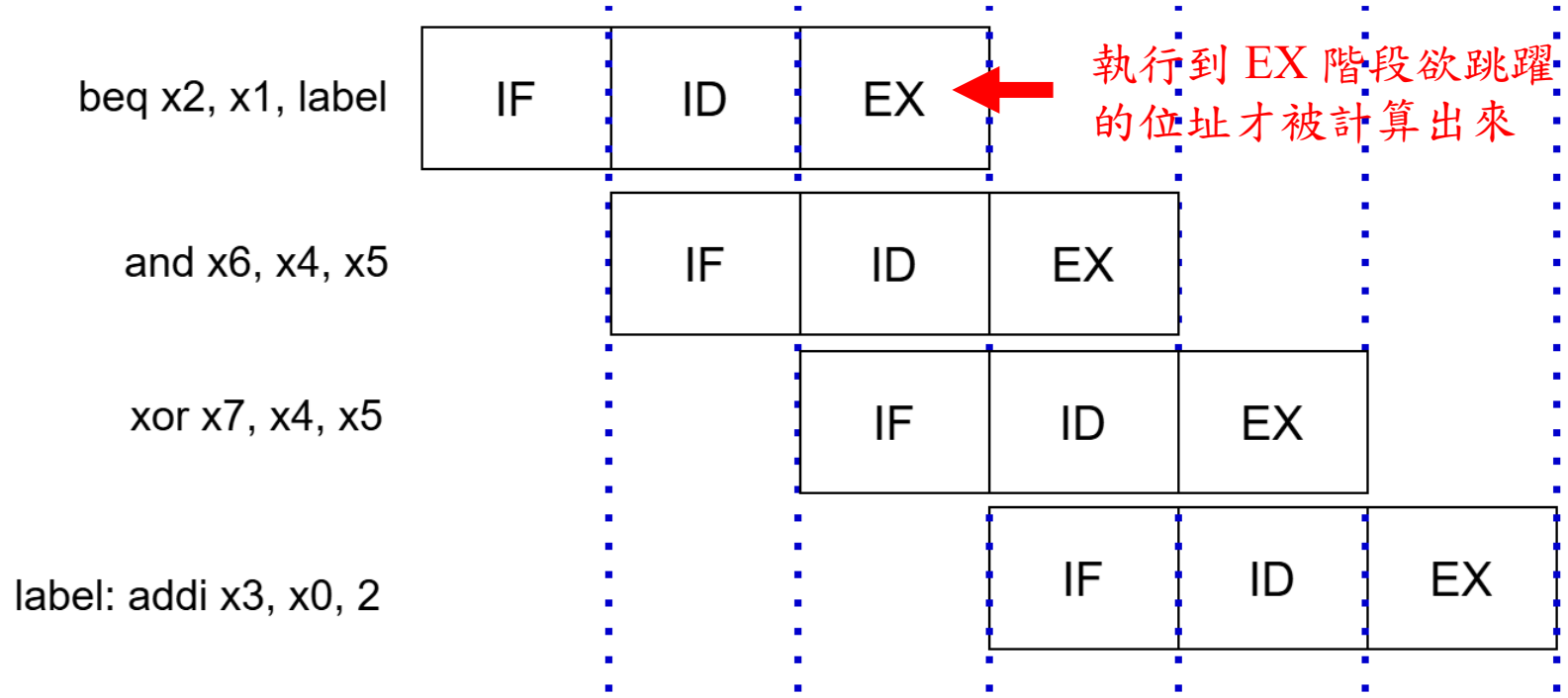
指令位址相差 0x10

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，假設第一條指令 (beq x2, x1, label) 比較完的結果是需要跳躍的，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。

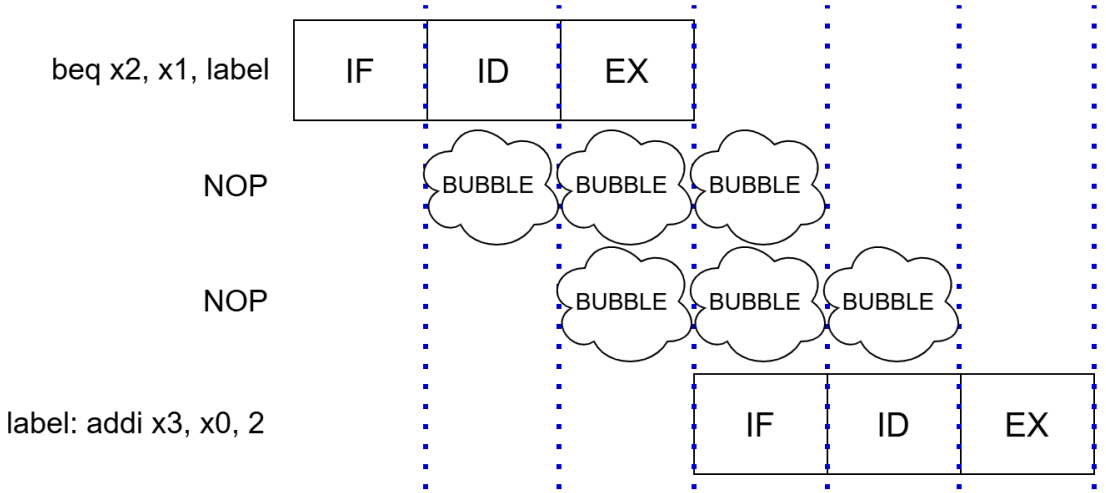
指令位址	指令
0x10	beq x2, x1, label
0x14	and x6, x4, x5
0x18	xor x7, x4, x5
0x1C	andi x8, x4, 1
0x20	label: andi x3, x0, 2

正常的程式執行順序

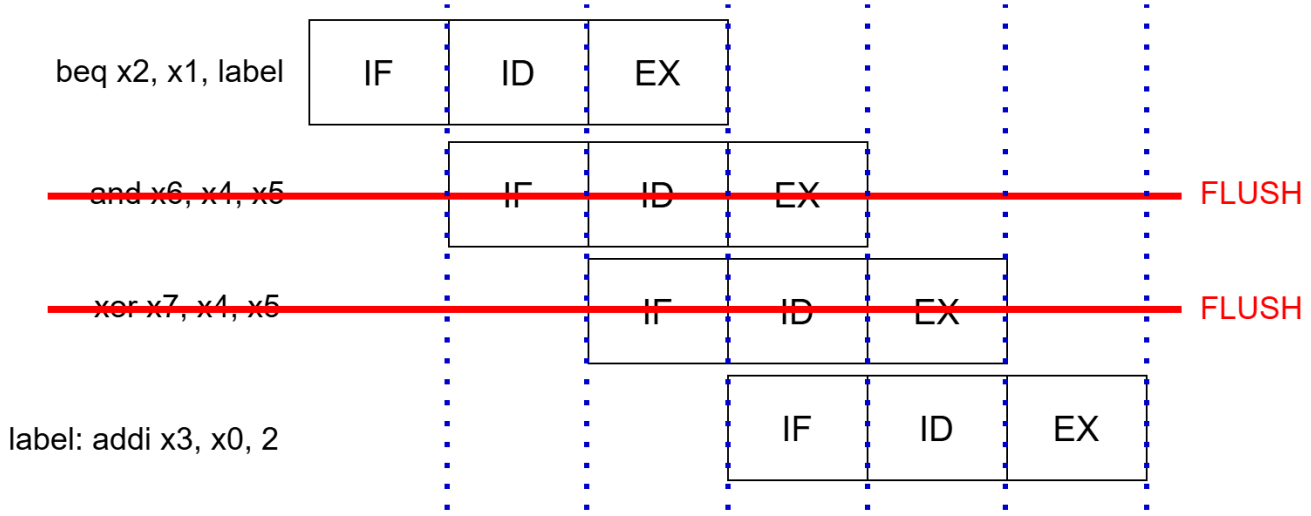


控制危障解決方法

- **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 (flush) 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。

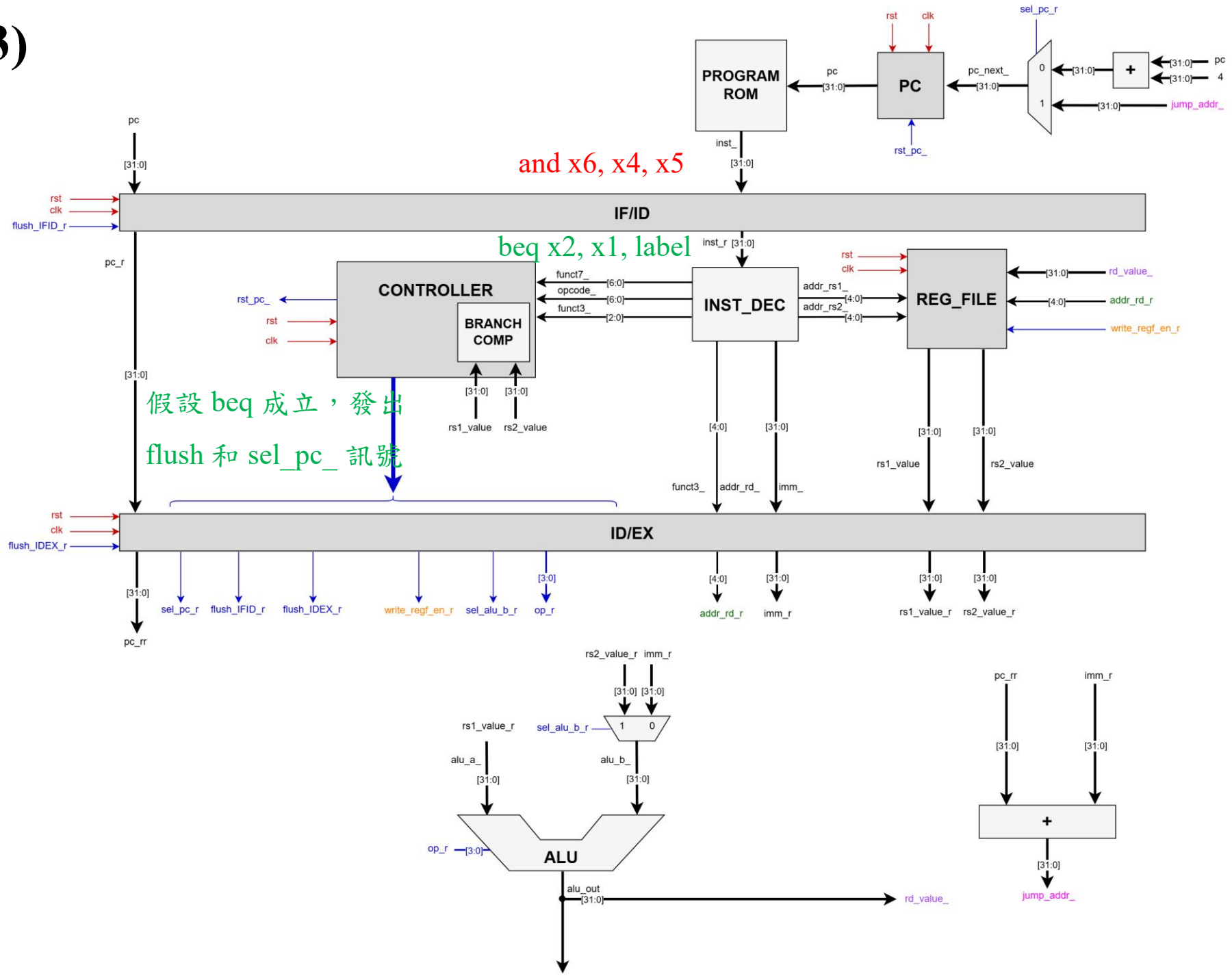


軟體方面

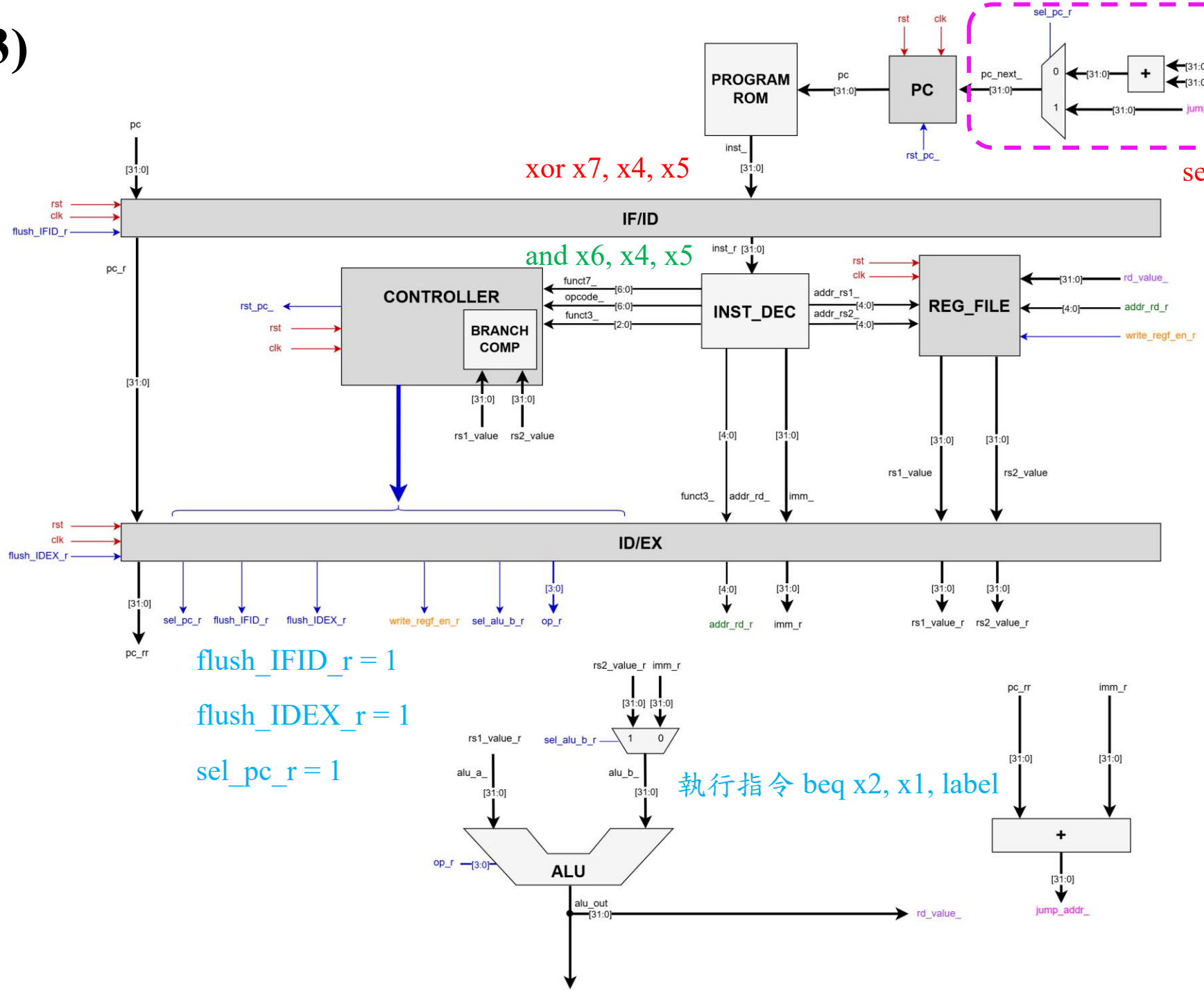


硬體方面

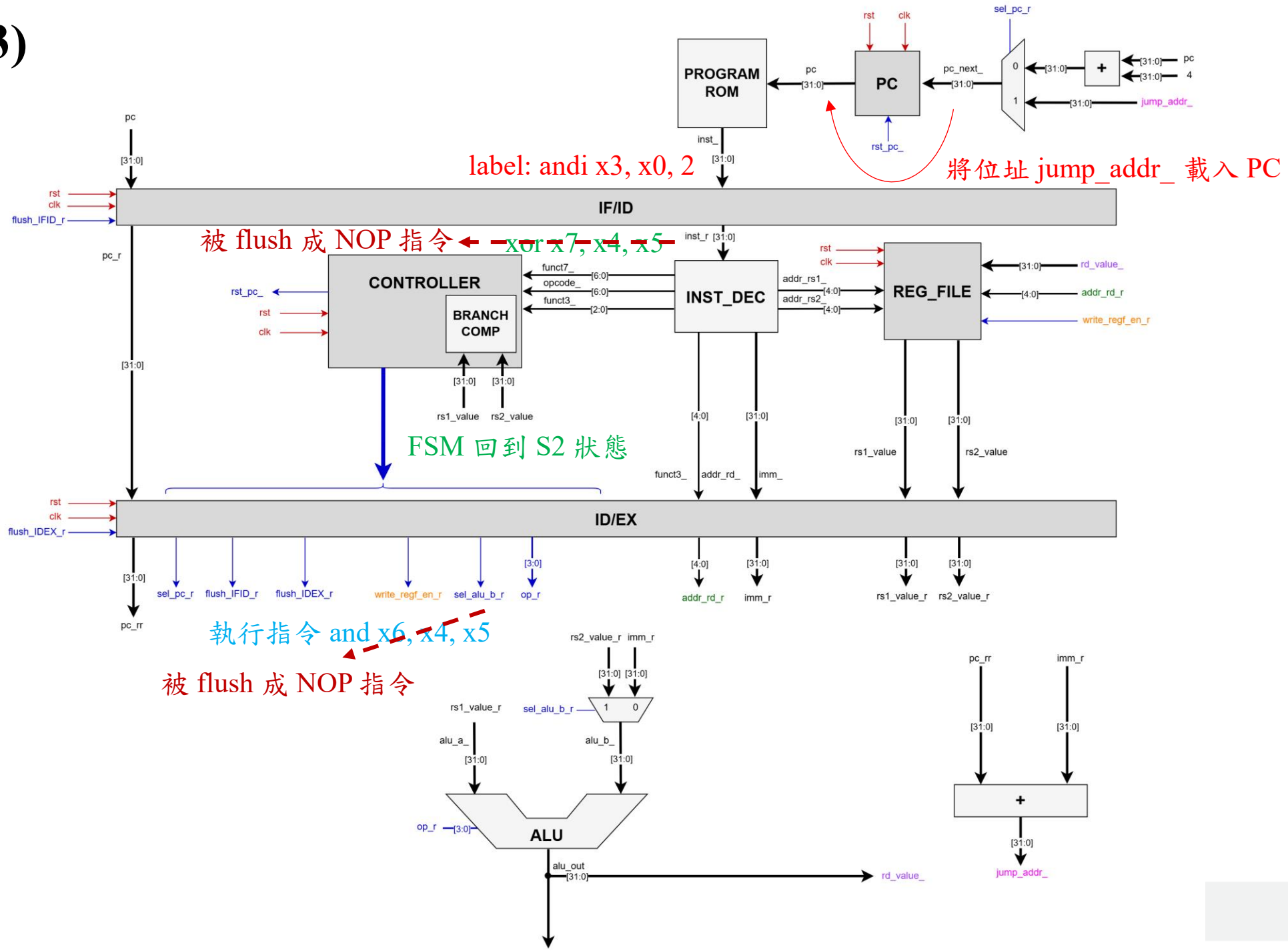
範例(1/3)


and x6, x4, x5
beq x2, x1, label
假設 beq 成立，發出
flush 和 sel_pc_ 訊號
PROGRAM ROM
PC
IF/ID
CONTROLLER
BRANCH COMP
INST_DEC
REG_FILE
ID/EX
ALU
pc
pc_next_
pc_r
pc_rr
inst_
inst_r
rst
clk
rst_pc_
flush_IFID_r
flush_IDEX_r
sel_pc_r
sel_alu_b_r
write_regf_en_r
funct7_
opcode_
funct3_
addr_rs1_
addr_rs2_
addr_rd_
imm_
rd_value_
jump_addr_
rs1_value
rs2_value
rs1_value_r
rs2_value_r
addr_rd_r
imm_r
op_r
alu_a_
alu_b_
alu_out

範例(2/3)


PROGRAM ROM
PC
sel_pc_r
pc_next_
jump_addr_
xor x7, x4, x5
sel_pc_r 選擇 1
IF/ID
and x6, x4, x5
CONTROLLER
BRANCH COMP
INST_DEC
REG_FILE
rd_value_
addr_rd_r
write_regf_en_r
rs1_value
rs2_value
ID/EX
flush_IFID_r = 1
flush_IDEX_r = 1
sel_pc_r = 1
執行指令 beq x2, x1, label
ALU
alu_out
jump_addr_

範例(3/3)


label: andi x3, x0, 2
將位址 jump_addr_ 載入 PC
被 flush 成 NOP 指令
xor x7, x4, x5
FSM 回到 S2 狀態
執行指令 and x6, x4, x5
被 flush 成 NOP 指令
PROGRAM ROM
PC
IF/ID
CONTROLLER
BRANCH COMP
INST_DEC
REG_FILE
ID/EX
ALU
pc
pc_next_
pc_r
inst_
inst_r
rst_pc_
funct7_
opcode_
funct3_
addr_rs1_
addr_rs2_
rd_value_
addr_rd_r
write_regf_en_r
rs1_value
rs2_value
addr_rd_
imm_
sel_pc_r
flush_IFID_r
flush_IDEX_r
write_regf_en_r
sel_alu_b_r
op_r
pc_rr
rs1_value_r
rs2_value_r
imm_r
alu_a_
alu_b_
alu_out
jump_addr_

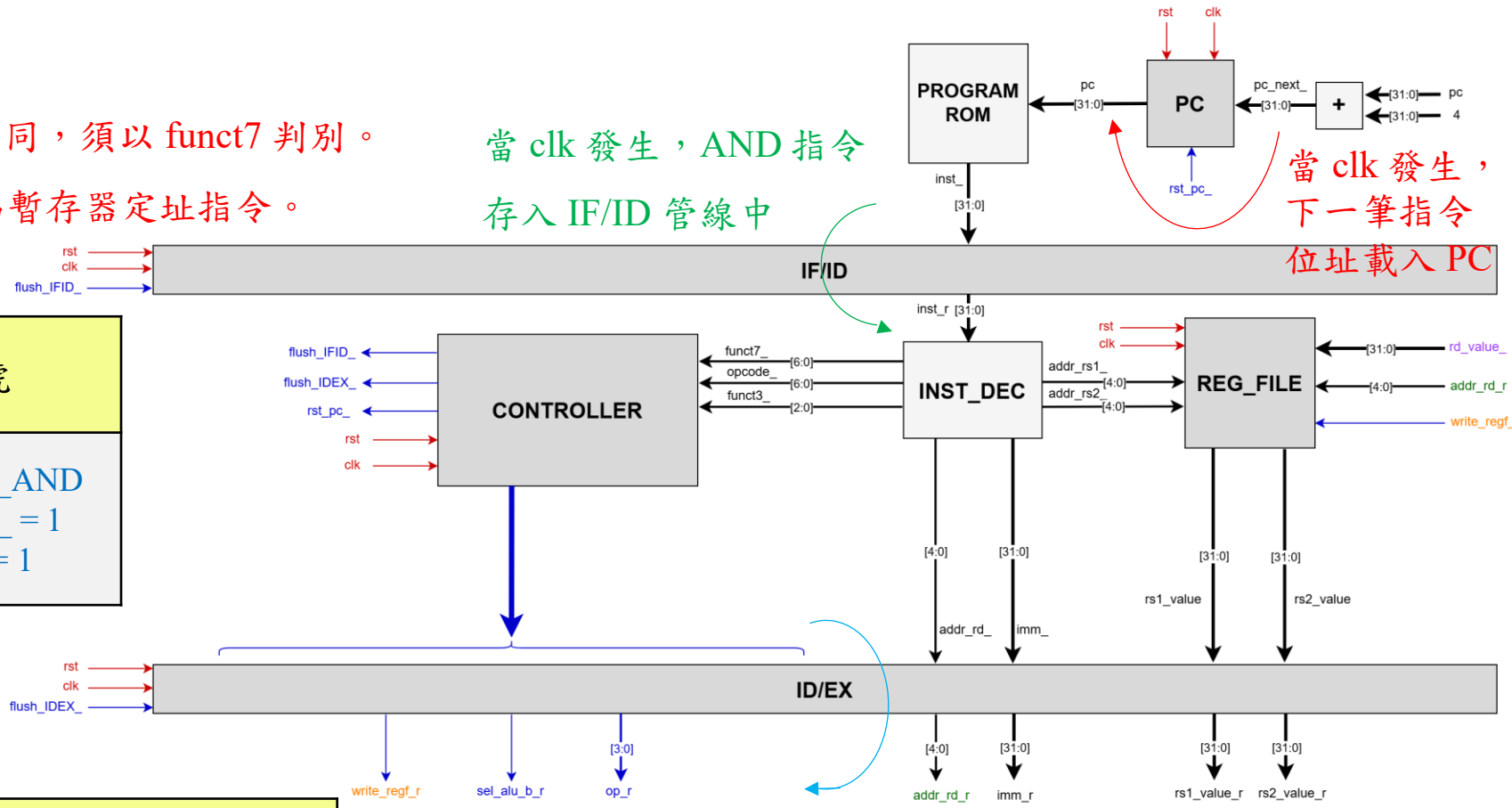
上周完成之架構

暫存器定址指令和乘除法指令的 opcode 相同，須以 funct7 判別。

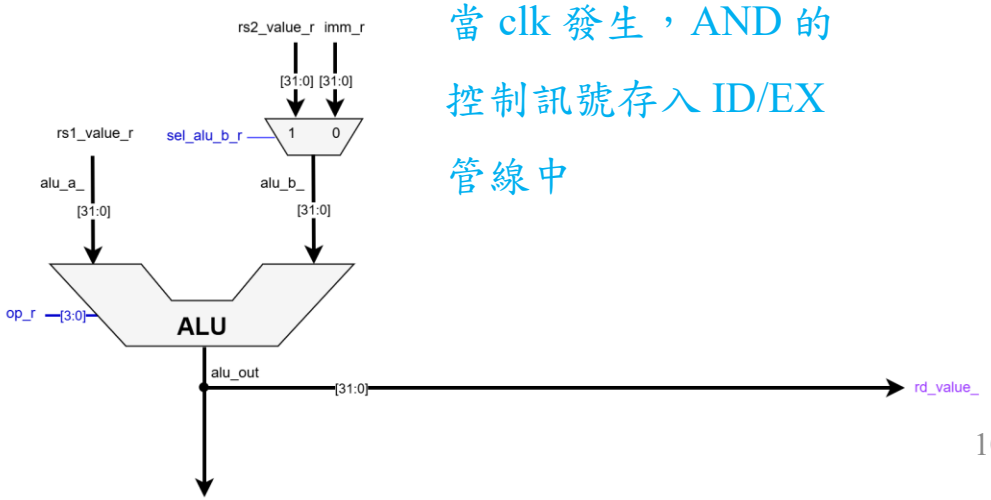
funct7 為 7'b000_0000 或 7'b010_0000 皆為暫存器定址指令。

```
`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

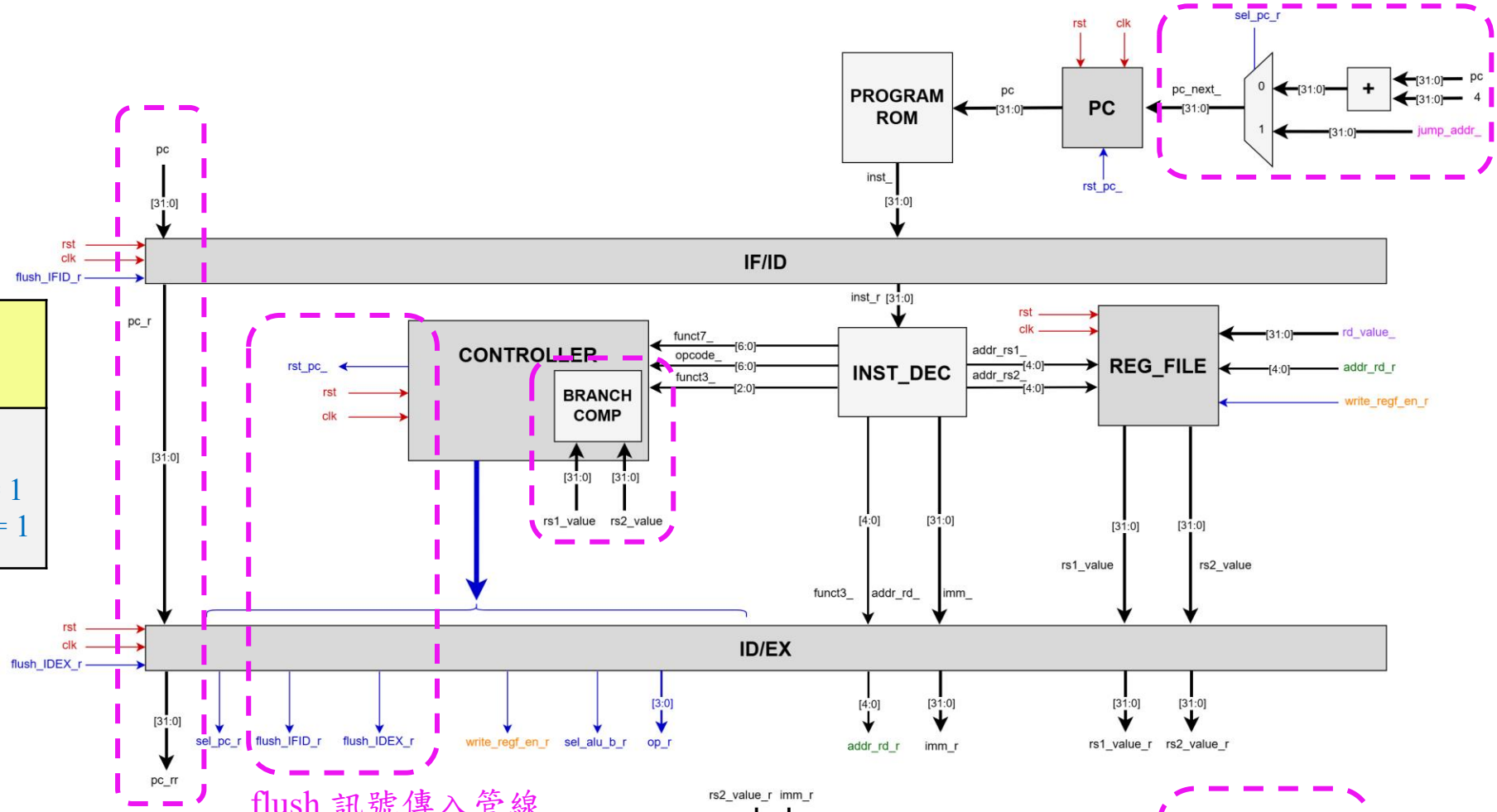


動作	控制訊號
x[addr_rd_r] ← rs1_value_r & rs2_value_r	op_r = `ALUOP_AND write_regf_en_r = 1 sel_alu_b_r = 1

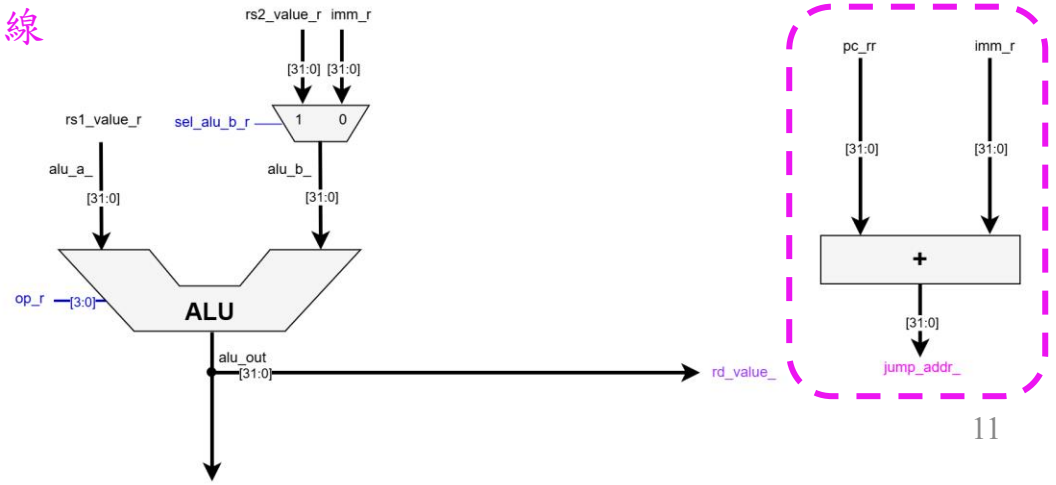


本周架構

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

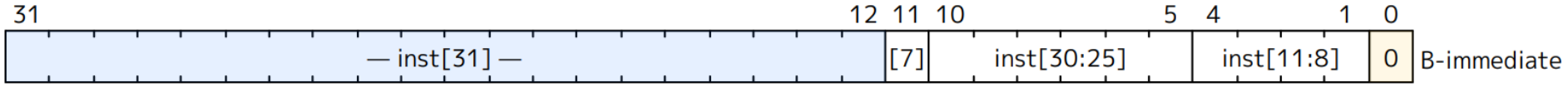


動作	控制訊號
pc_next_ ← pc_rr + imm_r	sel_pc_ r = 1 flush_IFID_r = 1 flush_IDEX_r = 1

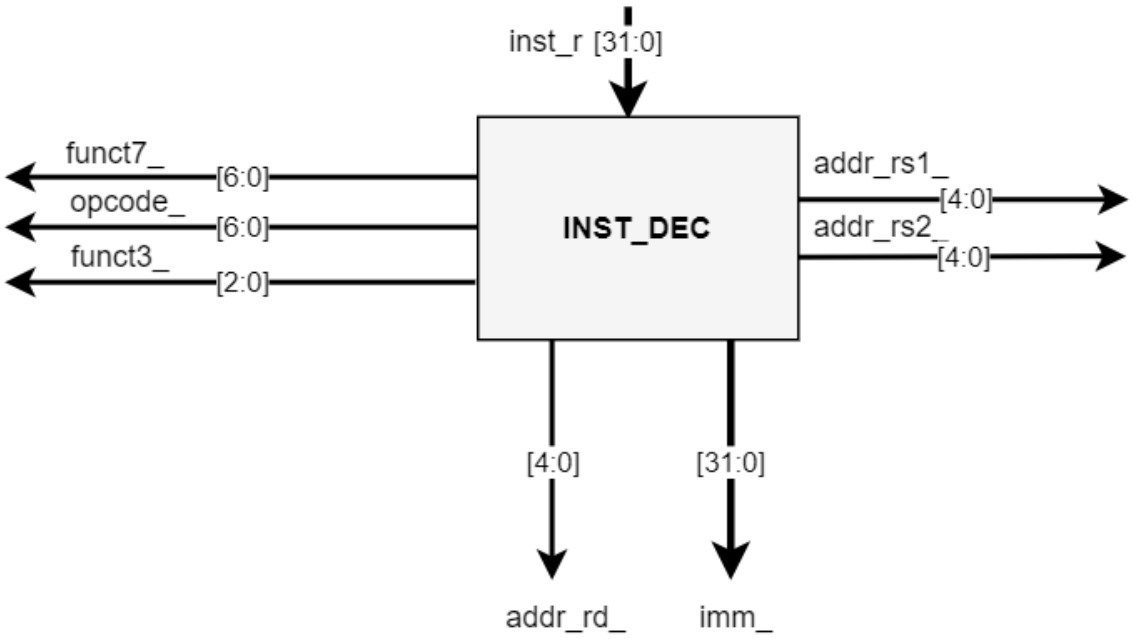


立即值擴充

- **Branch 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。




```
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;
assign IMM_I = {{20{inst_r[31]}}, inst_r[31:20]};
assign IMM_B = {{20{inst[31]}}, inst[7], inst[30:25], inst[11:8], 1'b0};

always_comb begin
    unique case (opcode_)
        `Opcode_I: imm_ = IMM_I;
        `Opcode_B: imm_ = IMM_B;
    endcase
end
```

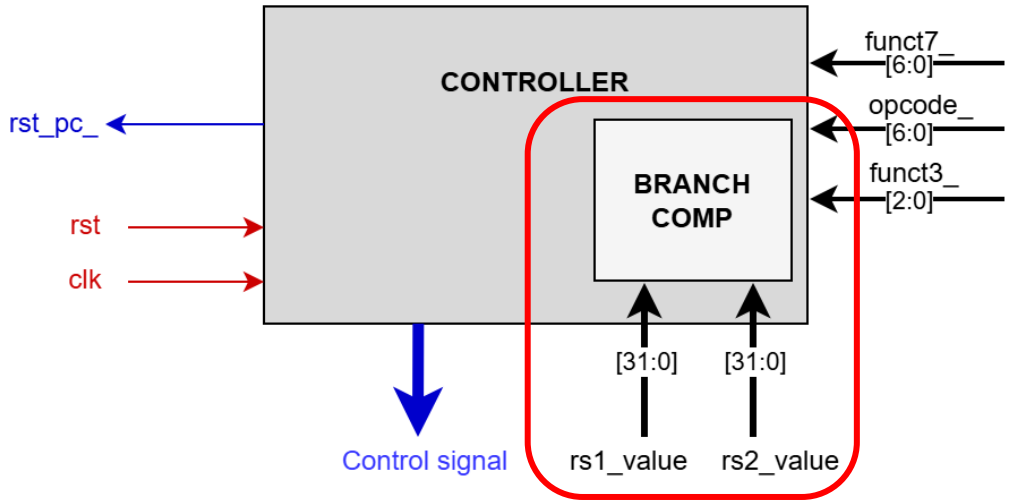


Branch Compare

- **Branch Flag**
  - 比較 x[rs1] 和 x[rs2] 的值。
  - 使用 1 個位元的旗標紀錄比較結果。
  - 在 FSM 中使用旗標來判斷是否要跳躍。

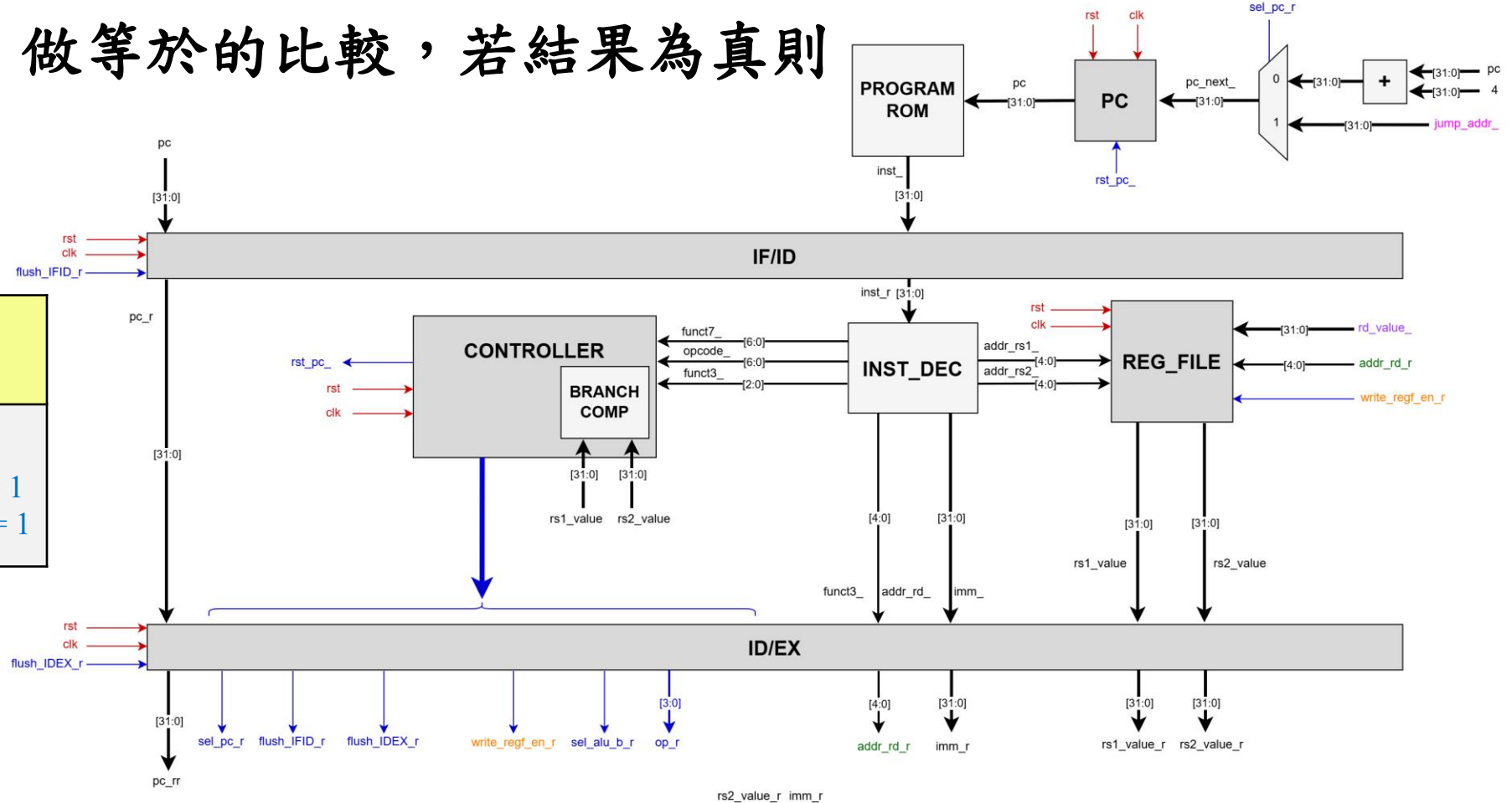
```
`Opcode_B: begin
    unique case (funct3_)
        //等於
        `F_BEQ: begin
            if(BEQ_FLAG) begin //x[rs1] == x[rs2] 跳躍
                sel_pc_ = 1;
                flush_IFID_ = 1;
                flush_IDEX_ = 1;
            end
        end
        //不等於
        `F_BNE: begin
```

```
logic BEQ_FLAG;
logic BNE_FLAG;
logic BLT_FLAG;
logic BGE_FLAG;
logic BLTU_FLAG;
logic BGEU_FLAG;

assign BEQ_FLAG  = (rs1_value == rs2_value);
assign BNE_FLAG  = (rs1_value != rs2_value);
assign BLT_FLAG  = ($signed(rs1_value) <  $signed(rs2_value));
assign BGE_FLAG  = ($signed(rs1_value) >=  $signed(rs2_value));
assign BLTU_FLAG = (rs1_value <  rs2_value);
assign BGEU_FLAG = (rs1_value >=  rs2_value);
```

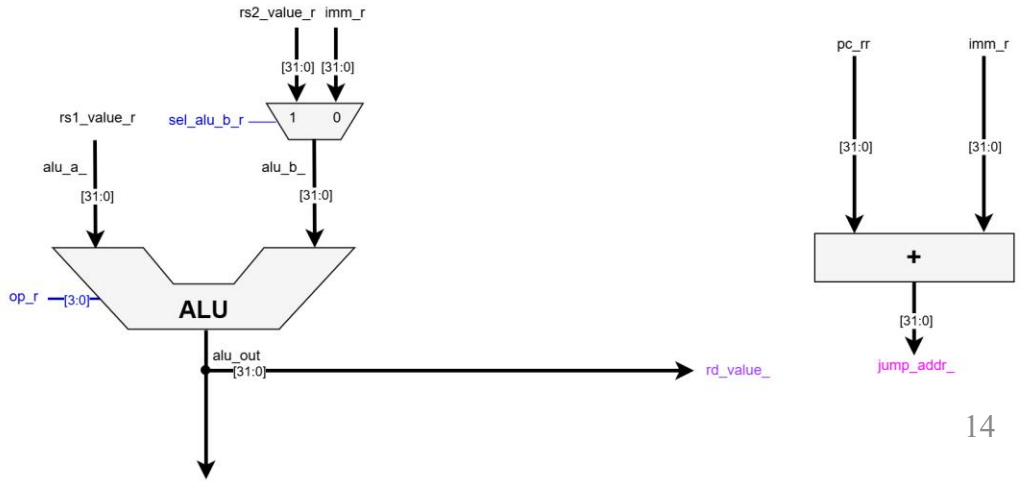


BEQ : x[rs1] 和 x[rs2] 做等於的比較，若結果為真則跳躍，否則不動作

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

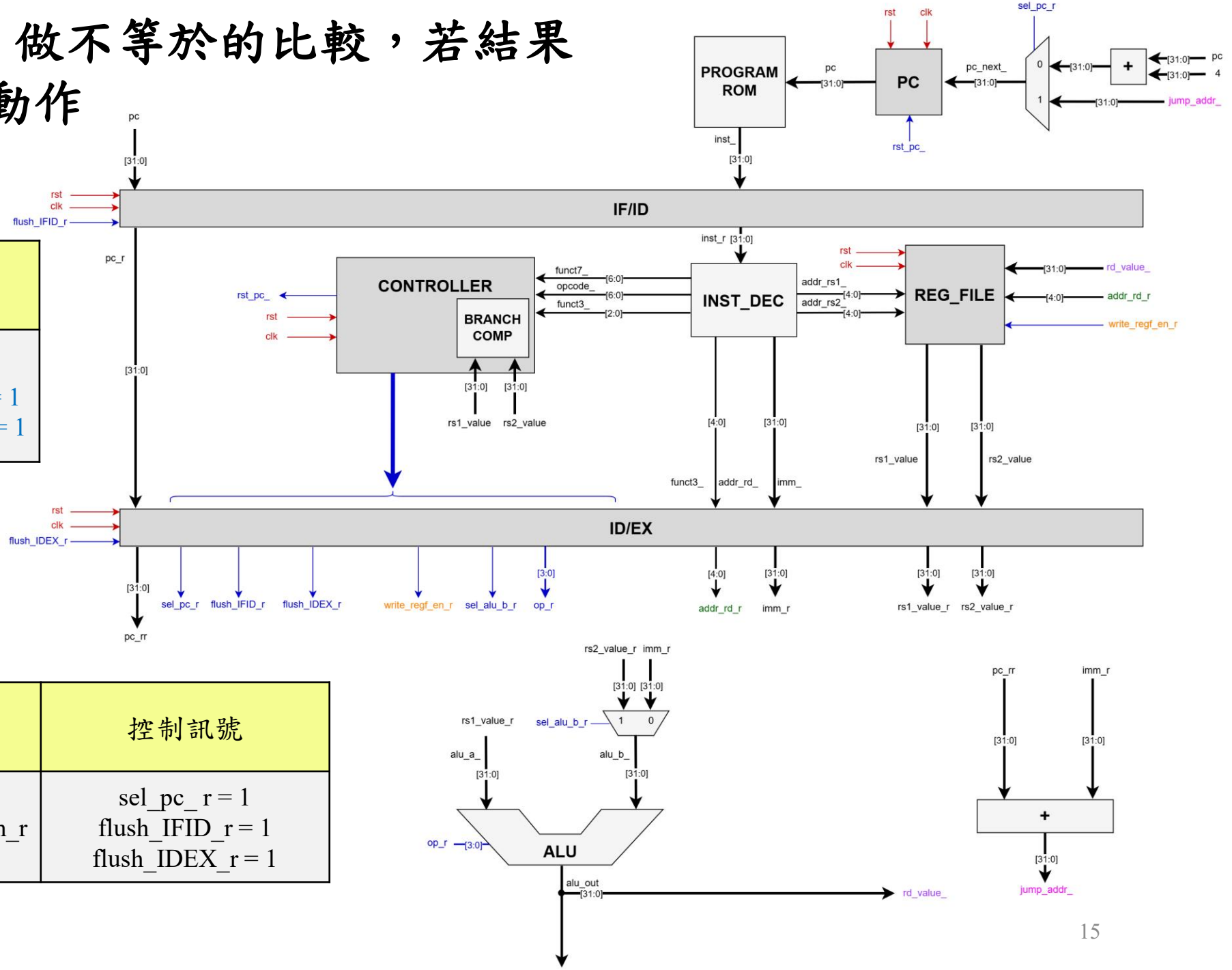


動作	控制訊號
pc_next_ ← pc_rr + imm_r	sel_pc_r = 1 flush_IFID_r = 1 flush_IDEX_r = 1



BNE : x[rs1] 和 x[rs2] 做不等於的比較，若結果為真則跳躍，否則不動作

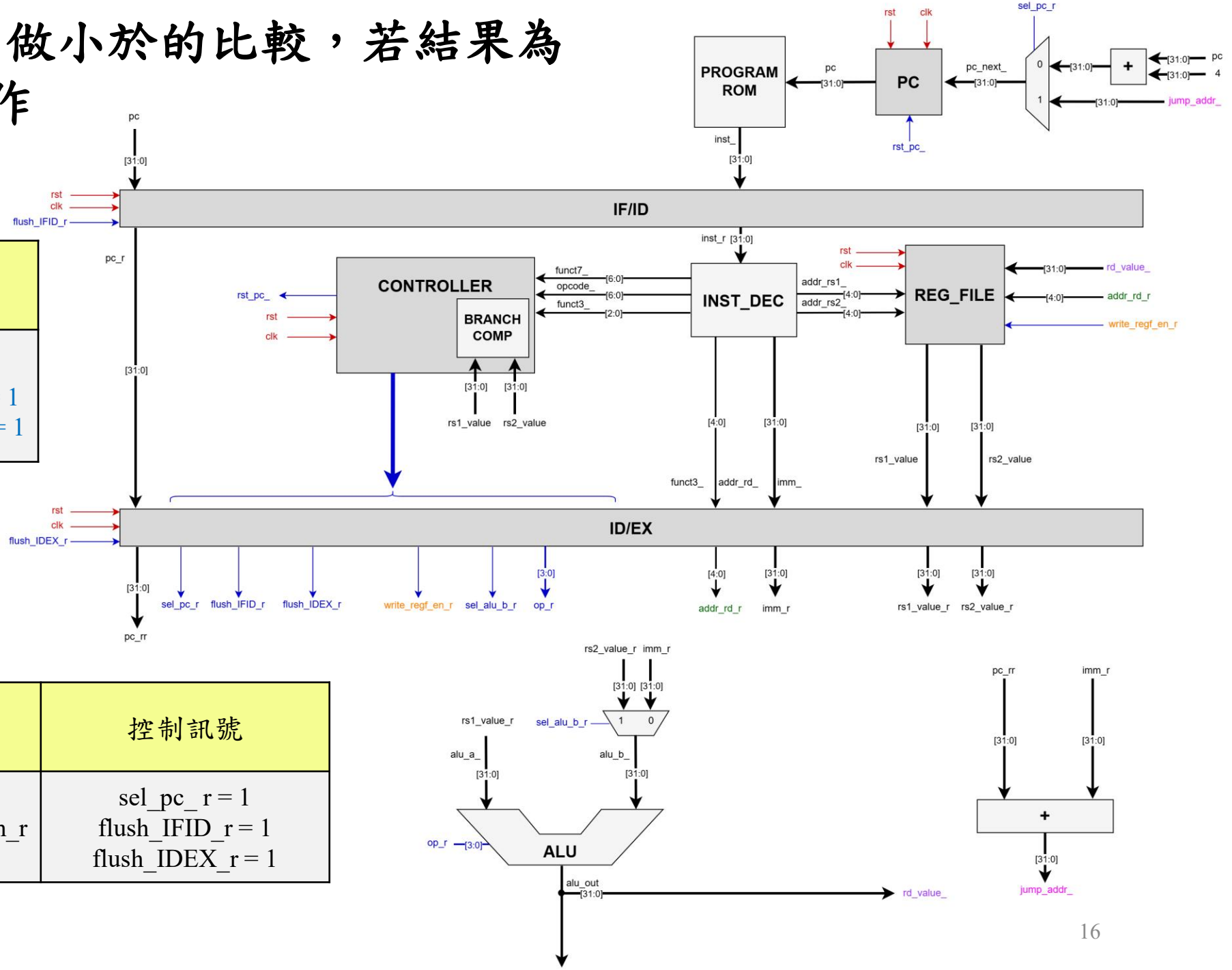
動作	控制訊號
if ((opcode_ == `Opcode_B) && (funct3_ == `F_BNE) && BNE_FLAG) 發出 BNE 的控制訊號	sel_pc_ = 1 flush_IFID_ = 1 flush_IDEX_ = 1



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

BLT : x[rs1] 和 x[rs2] 做小於的比較，若結果為真則跳躍，否則不動作

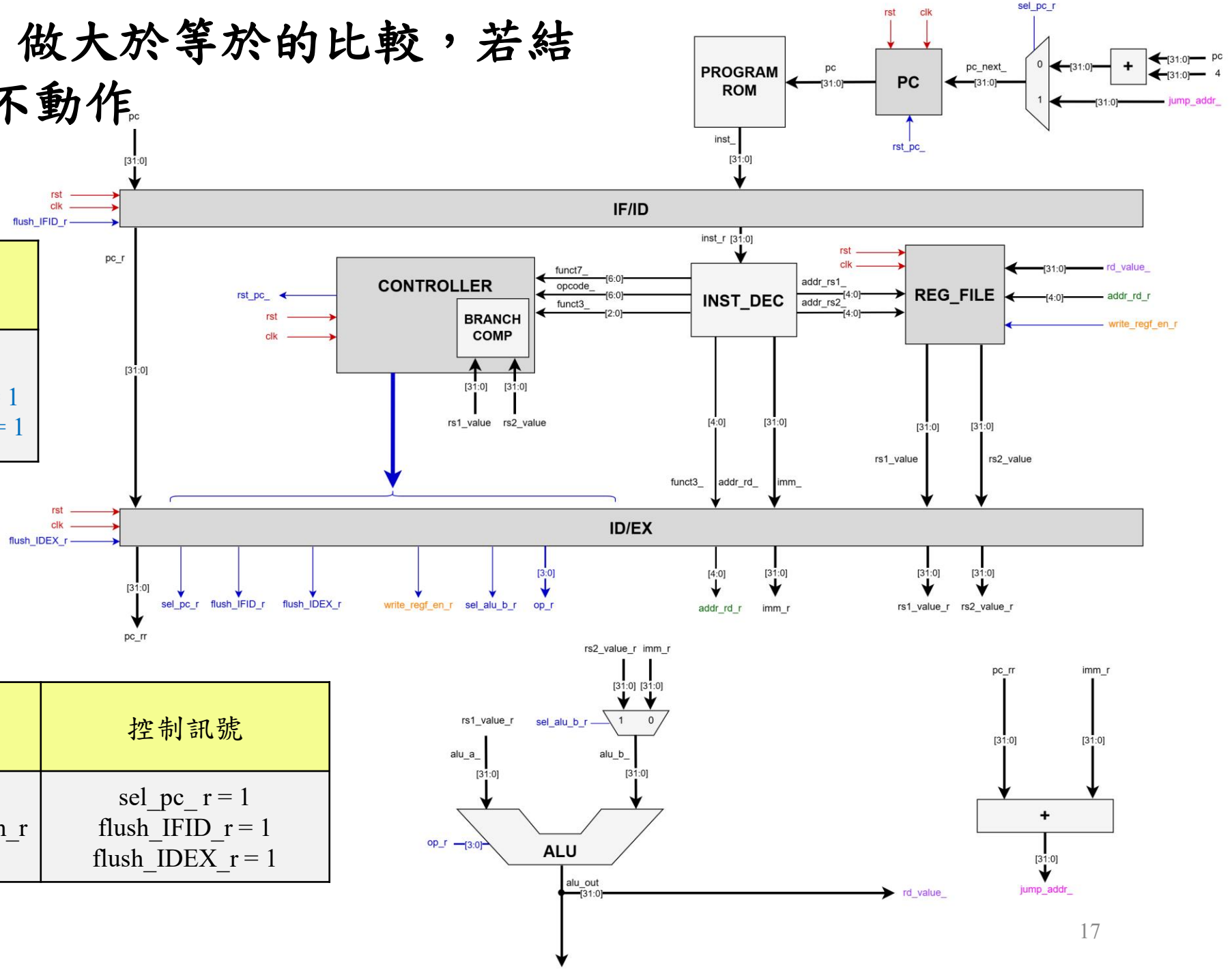
動作	控制訊號
if ((opcode_ == `Opcode_B) && (funct3_ == `F_BLT) && BLT_FLAG) 發出 BLT 的控制訊號	sel_pc_ = 1 flush_IFID_ = 1 flush_IDEX_ = 1



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

BGE：x[rs1] 和 x[rs2] 做大於等於的比較，若結果為真則跳躍，否則不動作

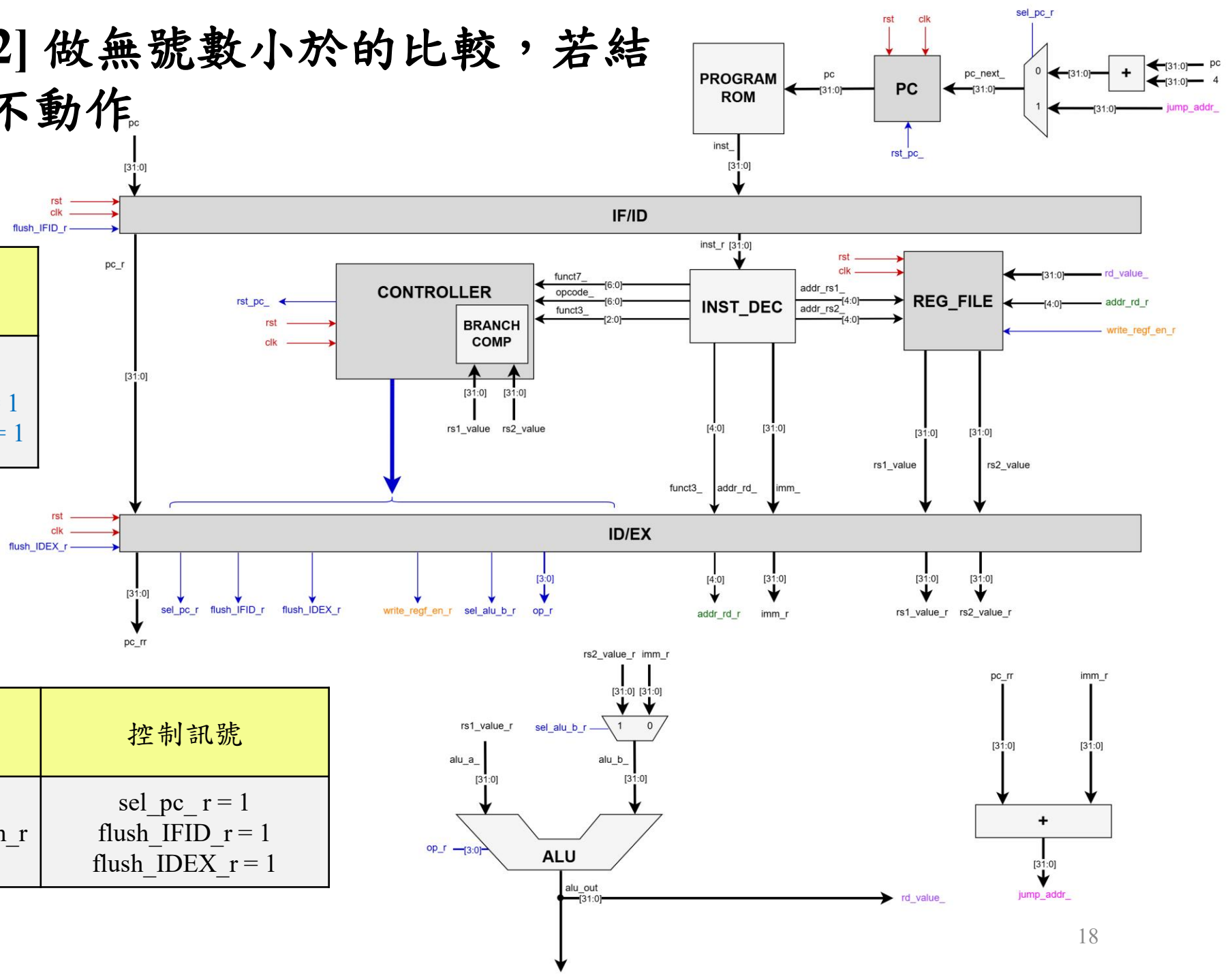
動作	控制訊號
if ((opcode_ == `Opcode_B) && (funct3_ == `F_BGE) && BGE_FLAG) 發出 BGE 的控制訊號	sel_pc_ = 1 flush_IFID_ = 1 flush_IDEX_ = 1



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

BLTU：x[rs1] 和 x[rs2] 做無號數小於的比較，若結果為真則跳躍，否則不動作

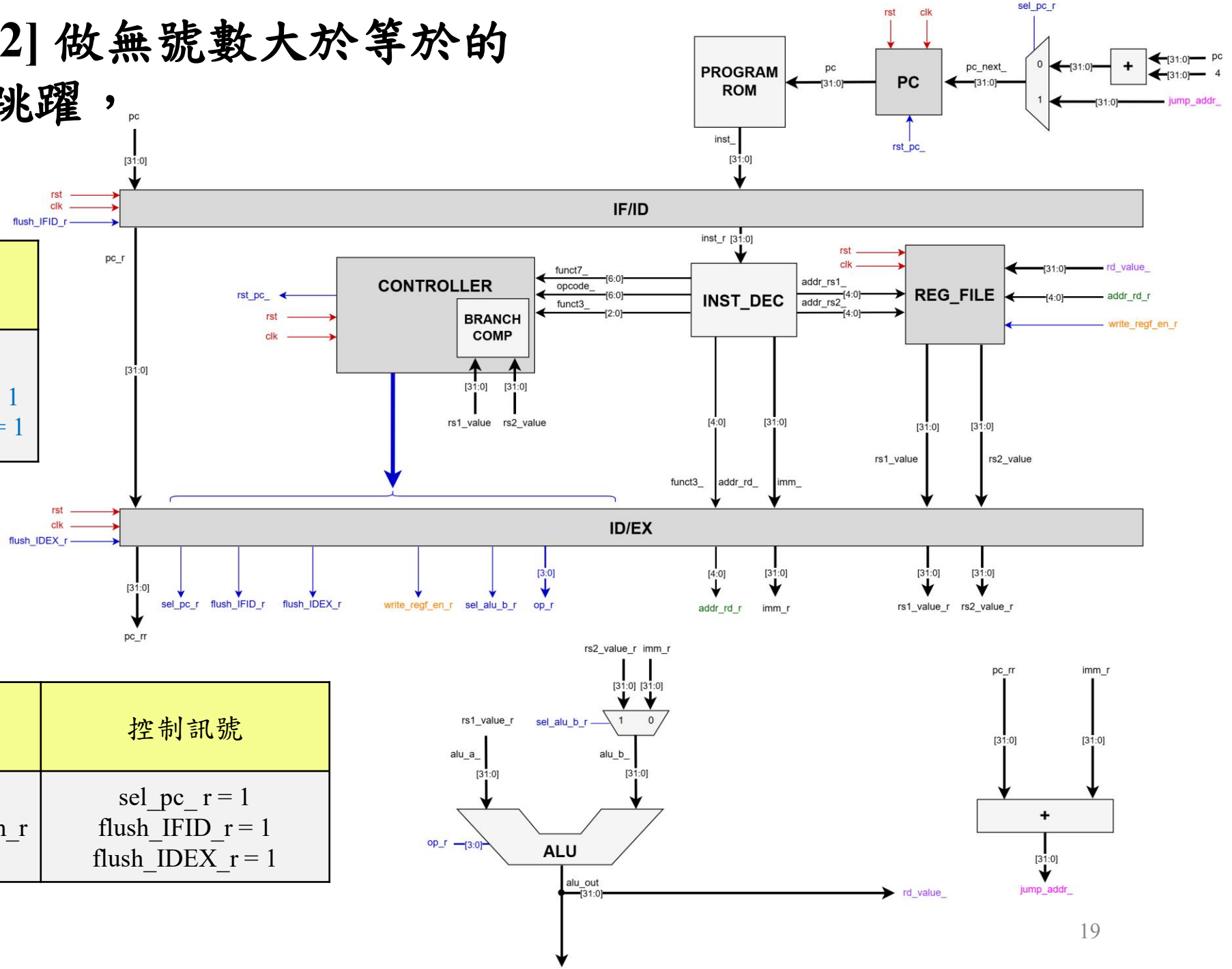
動作	控制訊號
if ((opcode_ == `Opcode_B) && (funct3_ == `F_BLTU) && BLTU_FLAG) 發出 BLTU 的控制訊號	sel_pc_ = 1 flush_IFID_ = 1 flush_IDEX_ = 1



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

BGEU：x[rs1] 和 x[rs2] 做無號數大於等於的比較，若結果為真則跳躍，否則不動作

動作	控制訊號
if ((opcode_ == `Opcode_B) && (funct3_ == `F_BGEU) && BGEU_FLAG) 發出 BGEU 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

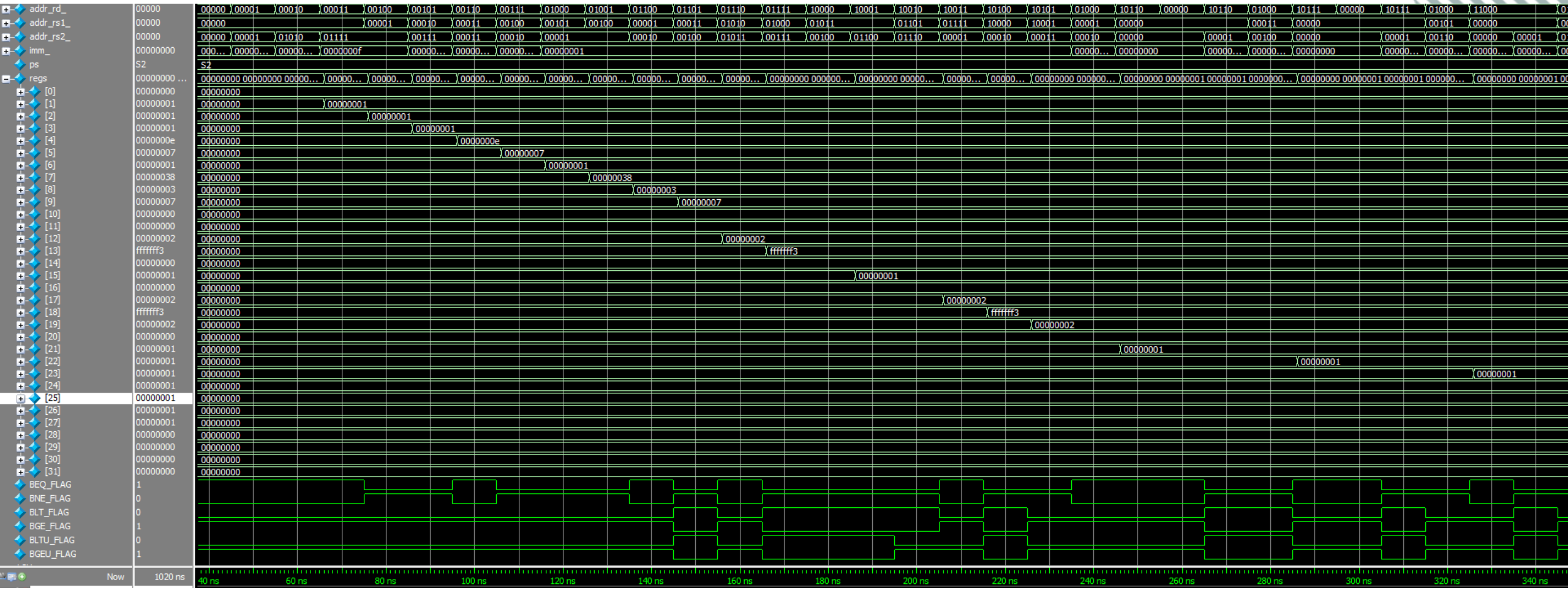
上課實作：ModelSim 模擬

```
#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
```

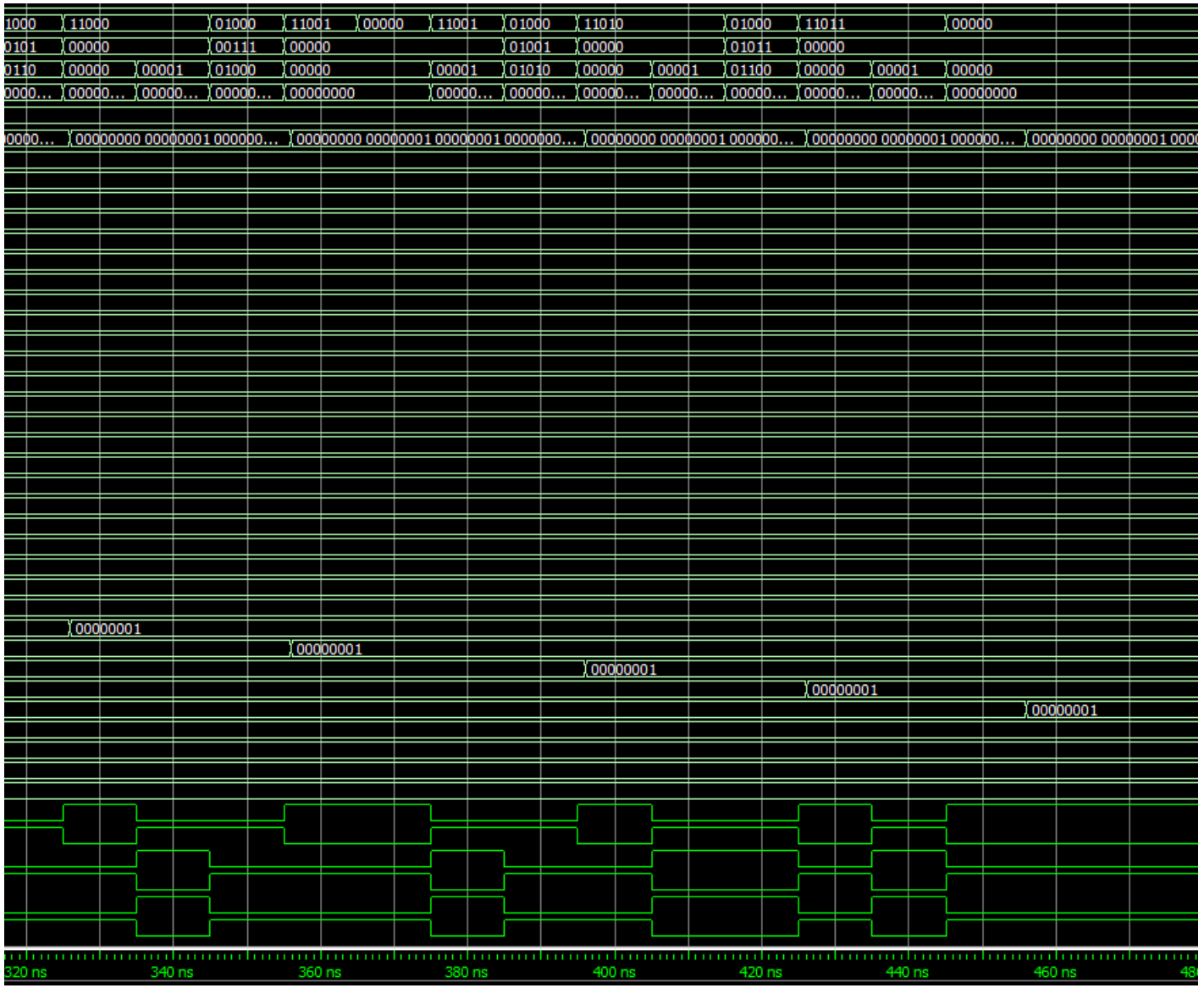
```
#分支預測
        beq x1, x2, test_beq
        addi x22, x0, 0
    test_beq:
        addi x22, x0, 1
        bne x3, x4, test_bne
        addi x23, x0, 0
    test_bne:
        addi x23, x0, 1
        blt x5, x6, test_blt
        addi x24, x0, 0
    test_blt:
        addi x24, x0, 1
        bge x7, x8, 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
```

```
module Program_Rom(
    output logic [31:0] Rom_data,
    input [31:0] Rom_addr
);

    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
            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
            default : Rom_data = 32'h00000013;      // NOP
        endcase
    end
endmodule
```

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 LED 中，透過下面的組合語言完成 LED 左右來回閃爍的功能

```
# Initialize registers and constants
init:
    addi x31, x0, 1      # Set x31 to 1 (used to control LED states)
    addi x3, x0, 1       # Set x3 to 1 (shift amount for each iteration)
    addi x2, x0, 0       # Initialize counter x2 to 0
    addi x1, x0, 9       # Set loop count x1 to 9 (number of shifts)

# Shift left loop: shift LED pattern to the left
shift left:
    sll x31, x31, x3     # Shift x31 left by 1 (move the LED pattern left)
    nop                  # Avoid data dependency issues
    addi x2, x2, 1       # Increment counter x2
    nop                  # Avoid data dependency issues
    nop                  # Avoid data dependency issues
    blt x2, x1, shift left  # Repeat until x2 reaches x1 (9 shifts)
    addi x2, x0, 0       # Reset counter x2 to 0 for the next loop

# Shift right loop: shift LED pattern to the right
shift right:
    srl x31, x31, x3     # Shift x31 right by 1 (move the LED pattern right)
    nop                  # Avoid data dependency issues
    addi x2, x2, 1       # Increment counter x2
    nop                  # Avoid data dependency issues
    nop                  # Avoid data dependency issues
    blt x2, x1, shift right  # Repeat until x2 reaches x1 (9 shifts)
    addi x2, x0, 0       # Reset counter x2 to 0 for the next loop
    blt x0, x1, shift left   # Repeat the left-right shift cycle indefinitely
```

組合語言中的 nop 指令是為了避免資料危障，跟控制危障無關

```
`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'h00100f93;   //init:          addi x31, x0, 1
            32'h4  : Rom_data = 32'h00100193;   //                addi x3, x0, 1
            32'h8  : Rom_data = 32'h00000113;   //                addi x2, x0, 0
            32'hc  : Rom_data = 32'h00900093;   //                addi x1, x0, 9
            32'h10 : Rom_data = 32'h003f9fb3;   //shift_left:    sll x31, x31, x3
            32'h14 : Rom_data = 32'h00000013;   //                nop
            32'h18 : Rom_data = 32'h00110113;   //                addi x2, x2, 1
            32'h1c : Rom_data = 32'h00000013;   //                nop
            32'h20 : Rom_data = 32'h00000013;   //                nop
            32'h24 : Rom_data = 32'hfe1146e3;   //                blt x2, x1, shift_left
            32'h28 : Rom_data = 32'h00000113;   //                addi x2, x0, 0
            32'h2c : Rom_data = 32'h003fdfb3;   //shift_right:   srl x31, x31, x3
            32'h30 : Rom_data = 32'h00000013;   //                nop
            32'h34 : Rom_data = 32'h00110113;   //                addi x2, x2, 1
            32'h38 : Rom_data = 32'h00000013;   //                nop
            32'h3c : Rom_data = 32'h00000013;   //                nop
            32'h40 : Rom_data = 32'hfe1146e3;   //                blt x2, x1, shift_right
            32'h44 : Rom_data = 32'h00000113;   //                addi x2, x0, 0
            32'h48 : Rom_data = 32'hfc1044e3;   //                blt x0, x1, shift_left
            default: Rom_data = 32'h00000013;   //NOP
        endcase
    end
endmodule
```

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

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

//=======================================================
//  Structural coding
//=======================================================

`ifdef Burn
    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];將暫存器 x31 mapping 到 LED 上
```

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