計算機系統設計

記憶體存取

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

RISC-V RV32IM INSTRUCTION SET

● 記憶體介紹

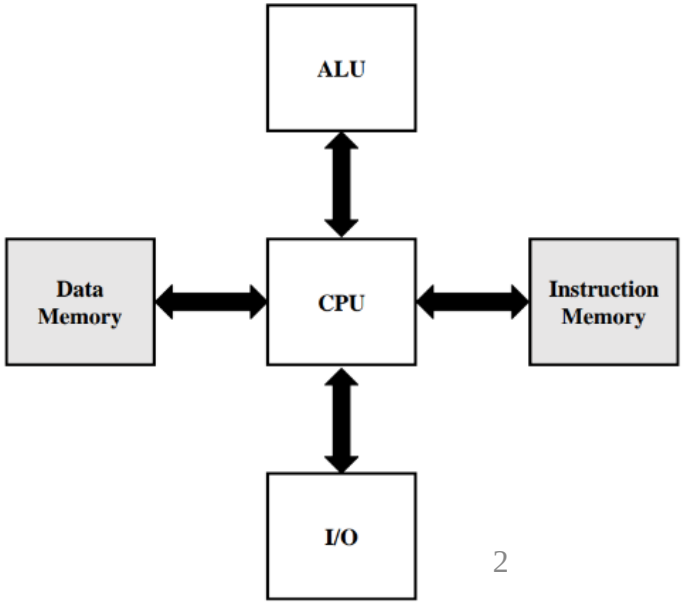
➢ 哈佛架構 (Harvard architecture)

一種將程式指令儲存和資料儲存分開的記憶體結構 (Split Cache)。這一詞起源於 Harvard Mark I 型繼電器式電腦，它儲存指令 (24位元) 在紙帶上和資料於機電計數器上。 中央處理器首先到程式指令儲存器中讀取程式指令內容，解碼後得到資料位址，再到相應的資料儲存器中讀取資料，並進行下一步的操作 (通常是執行)。程式指令儲存和資料儲存分開，資料和指令的儲存可以同時進行，可以使指令和資料有不同的資料寬度。

https://zh.wikipedia.org/wiki/%E5%93%88%E4%BD%9B%E7%BB%93%E6%9E%84

➢ Data Memory

- RISC-V 的資料記憶體定址空間高達 2^{32} 個位元組，但本實作中只使用 2^{7} x 4 個位元組。
- 在 RISC-V 中 Word 表示 32 個位元，而 Half word 表示 16 個位元。
- RISC-V 的記憶體為 Byte Addressing，意即記憶體中每個位址代表一個 Byte。



RISC-V RV32IM INSTRUCTION SET

Load and Store Instructions

※ **addr = x[rs1] + sext(offset)**

Mnemonic, Operands	Description	Implementation
LB rd, offset(rs1)	Loads a 8-bit value from memory and sign-extends this to XLEN bits before storing it in register *rd*.	x[rd] = sext(M[addr])
LH rd, offset(rs1)	Loads a 16-bit value from memory and sign-extends this to XLEN bits before storing it in register *rd*.	x[rd] = sext(M[addr + 1 : addr])
LW rd, offset(rs1)	Loads a 32-bit value from memory and store it in register *rd*.	x[rd] = M[addr + 3 : addr]
LBU rd, offset(rs1)	Loads a 8-bit value from memory and zero-extends this to XLEN bits before storing it in register *rd*.	x[rd] = zext(M[addr])
LHU rd, offset(rs1)	Loads a 16-bit value from memory and zero-extends this to XLEN bits before storing it in register *rd*.	x[rd] = zext(M[addr + 1 : addr])
SB rs2, offset(rs1)	Store 8-bit, values from the low bits of register *rs2* to memory.	M[addr] = x[rs2][7:0]
SH rs2, offset(rs1)	Store 16-bit, values from the low bits of register *rs2* to memory.	M[addr + 1 : addr] = x[rs2][15:0]
SW rs2, offset(rs1)	Store 32-bit, values from the low bits of register *rs2* to memory.	M[addr + 3 : addr] = x[rs2][31:0]

※ **Word** (W) : 32 bits、**Half Word** (H) : 16 bits 、**Byte** (B) : 8 bits

※ zext : zero extension

※ sext : sign extension

RISC-V RV32IM INSTRUCTION SET

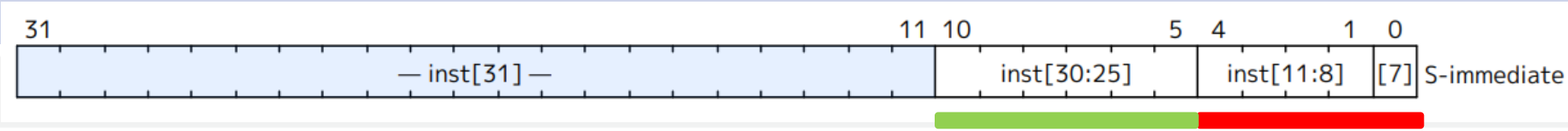
Load and Store Instructions

Immediate value	Source registers 1	Funct3	Destination registers	Opcode	Instruction
imm[11:0]	rs1	000	rd	0000011	LB
imm[11:0]	rs1	001	rd	0000011	LH
imm[11:0]	rs1	010	rd	0000011	LW
imm[11:0]	rs1	100	rd	0000011	LBU
imm[11:0]	rs1	101	rd	0000011	LHU

31 20 19 15 14 12 11 7 6 0

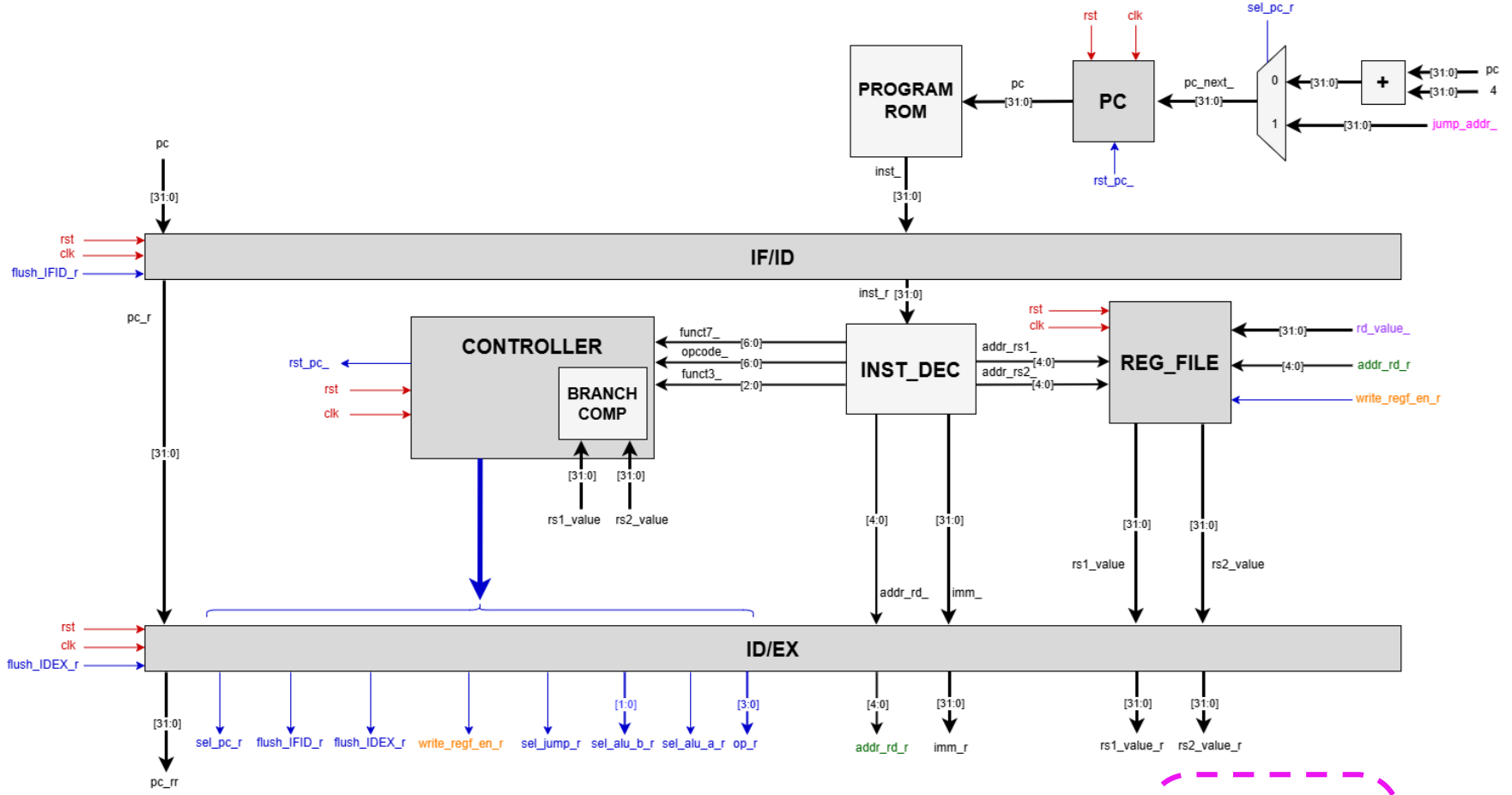
Immediate value	Source registers 2	Source registers 1	Funct3	Immediate value	Opcode	Instruction
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

31 25 24 20 19 15 14 12 11 7 6 0

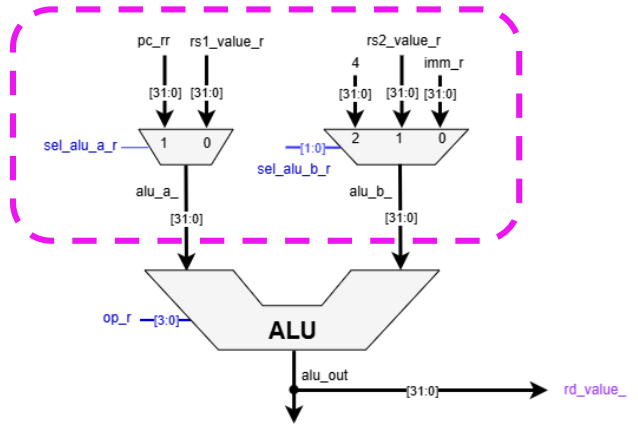


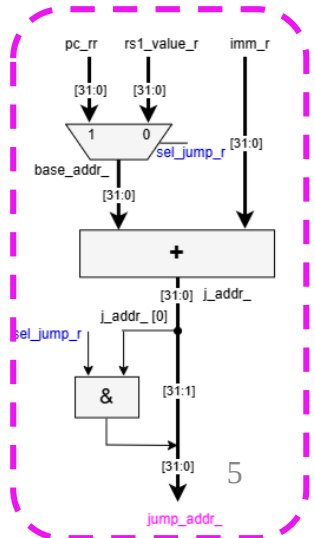
上周完成之架構

動作	控制訊號
if (opcode_ == `Opcode_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



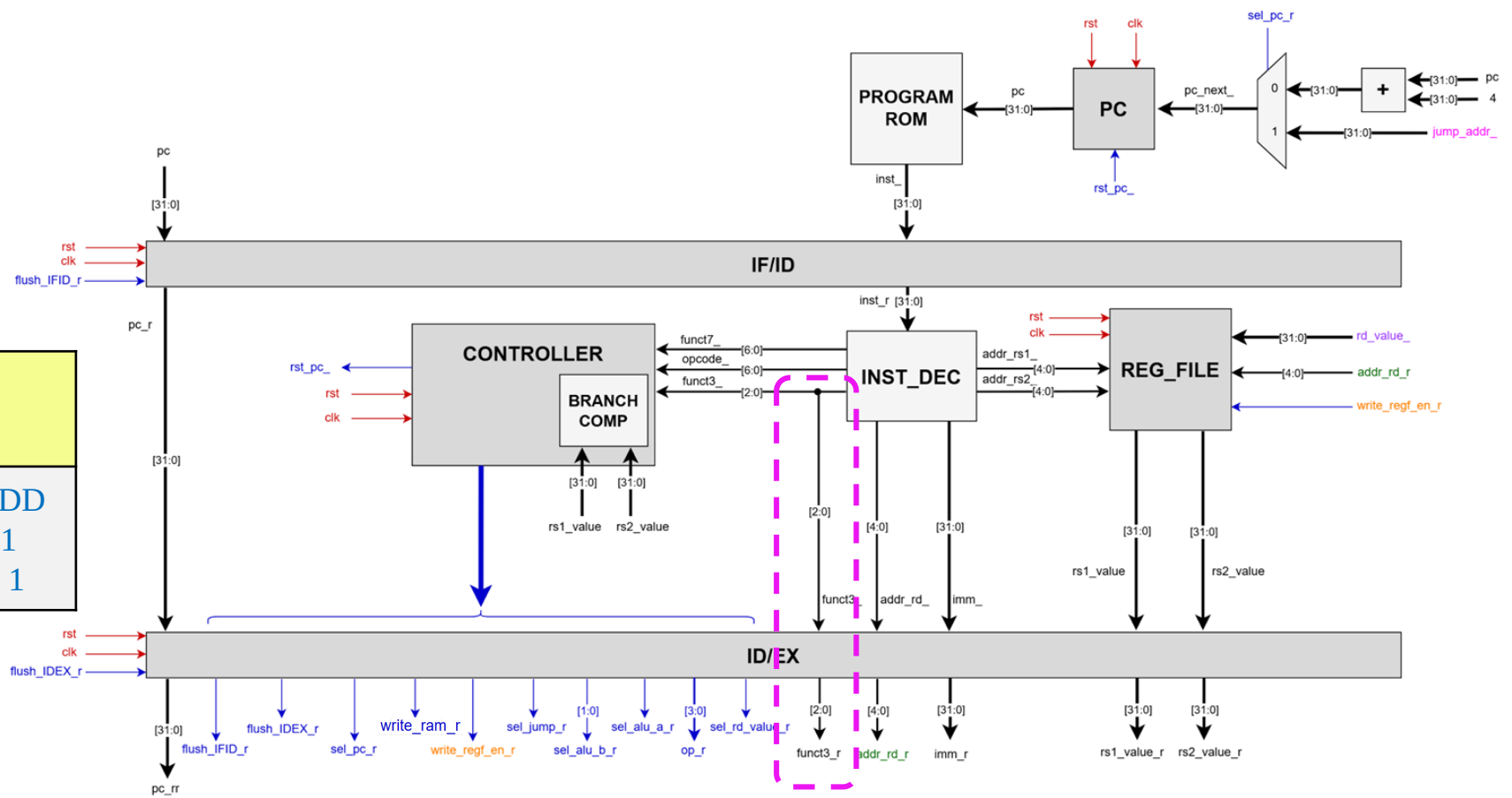
動作	控制訊號
pc_next_ ← pc_rr + imm_r rd_value_ ← pc_rr + 4	sel_pc_r = 1 flush_IFID_r = 1 flush_IDEX_r = 1 sel_alu_a_r = 1 sel_alu_b_r = 2 sel_jump_r = 1 op_r = `ALUOP_ADD write_regf_en_r = 1



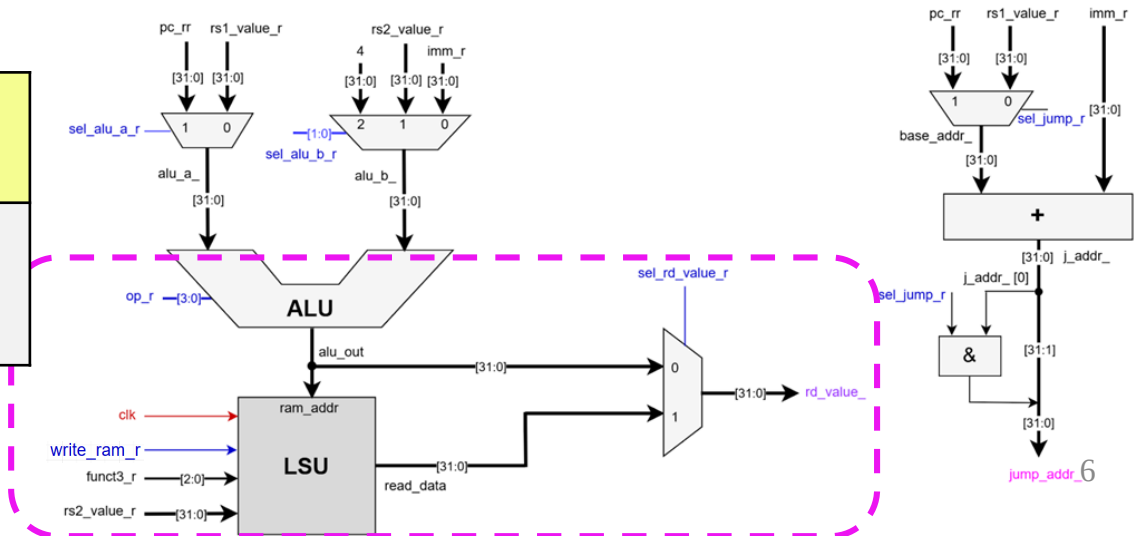


新架構

動作	控制訊號
if ((opcode_ == `Opcode_L) && (funct3_ == `F_LB)) 發出 LB 的控制訊號	op_ = `ALUOP_ADD sel_rd_value_ = 1 write_regf_en_ = 1



動作	控制訊號
※ addr = rs1_value_r + imm_r rd_value_ ← sext(M[addr])	op_r = `ALUOP_ADD sel_rd_value_r = 1 write_regf_en_r = 1



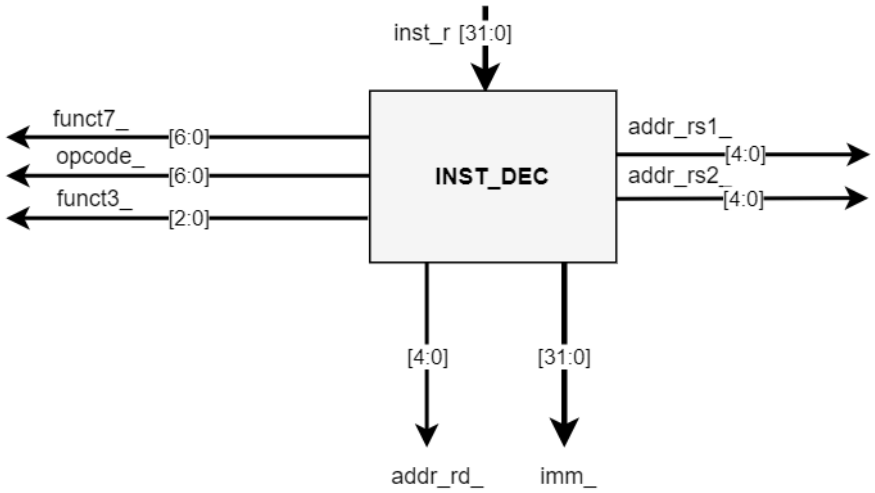
立即值擴充

● Load 指令

- 擴展方式和立即值定址指令相同。

● Store 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。



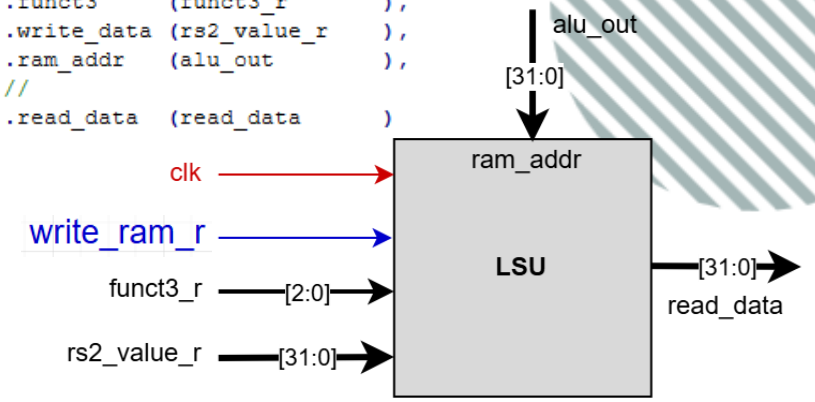

```
assign opcode_ = inst_r[6:0];
assign funct3_ = inst_r[14:12];
assign addr_rd_ = inst_r[11:7];
assign addr_rs1_ = inst_r[19:15];
assign addr_rs2_ = inst_r[24:20];
assign funct7_ = inst_r[31:25];

logic[31:0] IMM_I;
logic[31:0] IMM_B;
logic[31:0] IMM_JAL;
logic[31:0] IMM_LUI_AUIPC;
logic[31:0] IMM_S;
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};
assign IMM_S            = {{20{inst_r[31]}}, inst_r[31:25], inst_r[11:7]};

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;
        `Opcode_L:      imm_ = IMM_I;
        `Opcode_S:      imm_ = IMM_S;
    endcase
end
```

LSU

●介紹

- 載入儲存單元 (Load Store Unit, LSU) 負責執行記憶體存取，包括讀取資料(載入)和寫入資料(儲存)。
- RISC-V 中對記憶體存取的最小單位為 Byte (Byte Addressing)。

```
//Load and Store Unit
LSU LSU_1       (
    .clk        (clk            ),
    .write_ram  (write_ram_r    ),
    .funct3     (funct3_r       ),
    .write_data (rs2_value_r    ),
    .ram_addr   (alu_out        ),
    //
    .read_data  (read_data      )
);
```

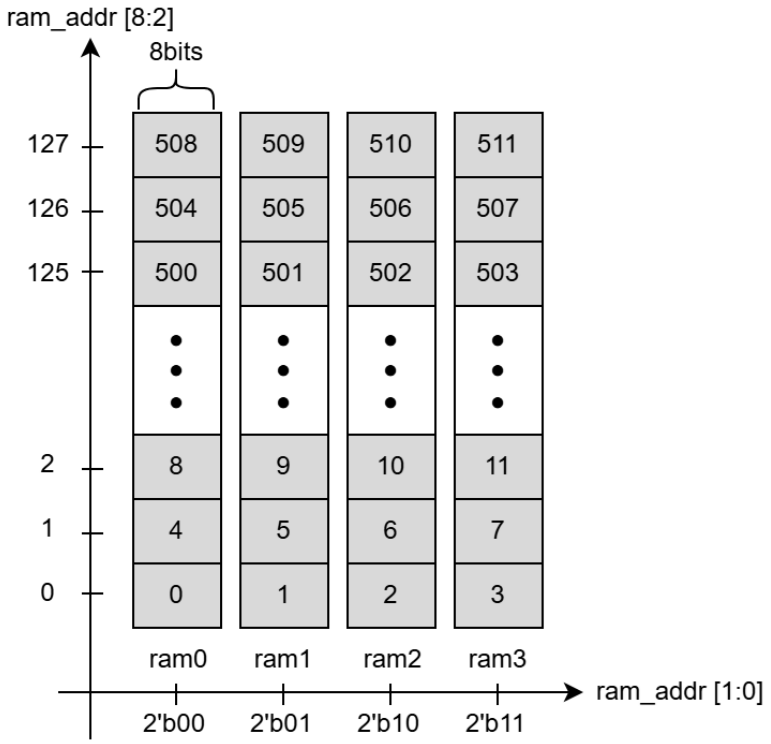


指令	funct3_r	write_ram_en_r	說明
LB	000	0	讀取一個 Byte，並做符號擴充至四個 Byte
LH	001	0	讀取兩個 Byte，並做符號擴充至四個 Byte
LW	010	0	讀取四個 Byte
LBU	100	0	讀取一個 Byte，並做零擴充至四個 Byte
LHU	101	0	讀取兩個 Byte，並做零擴充至四個 Byte
SB	000	1	寫入一個 Byte
SH	001	1	寫入兩個 Byte
SW	010	1	寫入四個 Byte

LSU

● RAM

- RISC-V 的定址空間高達 2^{32} 個位元組，但本實作中只使用 2^7 x 4 個位元組。
- 一個隨機存取記憶體 (Random-Access Memory, RAM) 的寬度為 1 Byte，深度為 2^7。
- 本實作使用四塊相同大小的 RAM 實現記憶體結構。
- ram_addr [1:0] 作為水平方向的定址，ram_addr [8:2] 作為垂直方向的定址。



memory addr = ram_addr [8:2] * 4 + ram_addr [1:0]

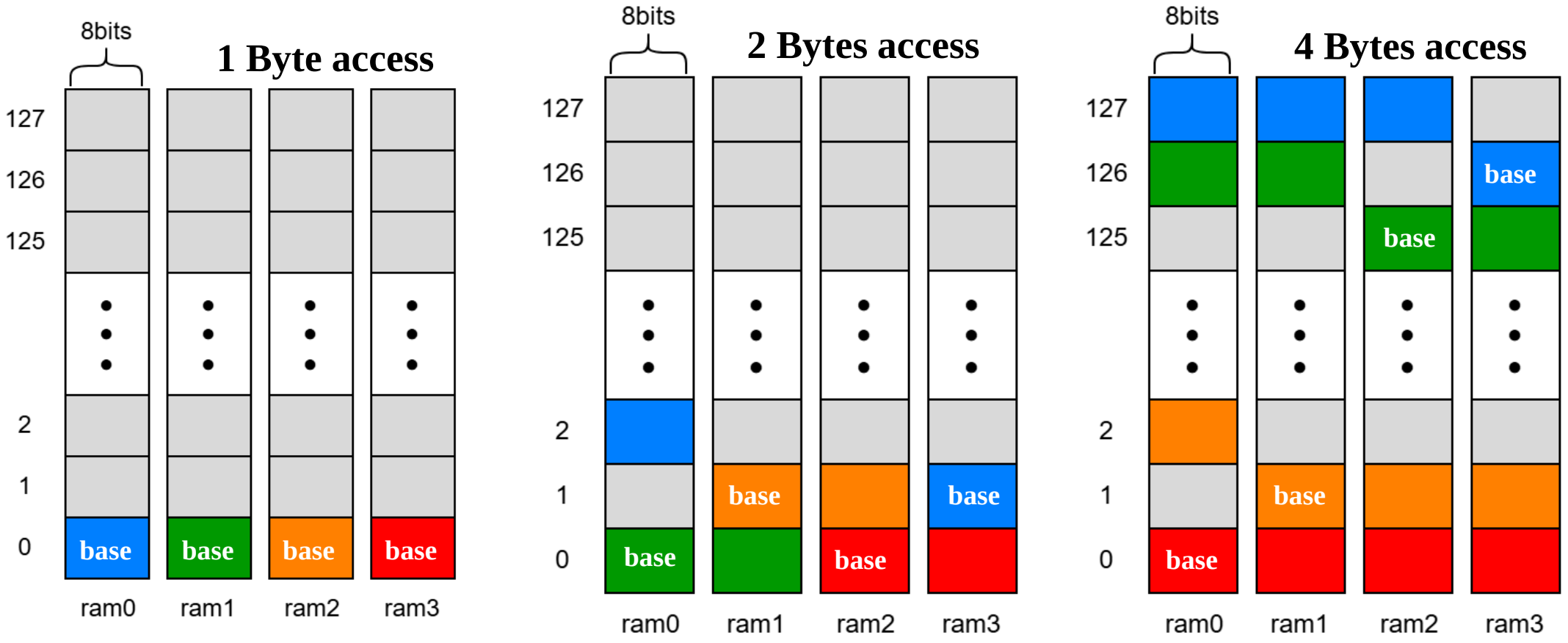
```
module RAM(
    input   logic clk,
    input   logic write,
    input   logic [7:0] write_data,
    input   logic [29:0] ram_addr,

    output  logic [7:0] read_data
);

    logic [7:0] ram[0:127];

    assign read_data = ram[ram_addr];

    always_ff @(posedge clk) begin
        if (write) begin
            ram[ram_addr] <= #1 write_data;
        end
    end

endmodule
```

```
RAM ram_0   (
    .clk            (clk            ),
    .write          (write_ram_0    ),
    .write_data     (write_data_0   ),
    .ram_addr       (ram_addr_0     ),

    .read_data      (read_data_0    )
);

RAM ram_1   (
    .clk            (clk            ),
    .write          (write_ram_1    ),
    .write_data     (write_data_1   ),
    .ram_addr       (ram_addr_1     ),

    .read_data      (read_data_1    )
);

RAM ram_2   (
    .clk            (clk            ),
    .write          (write_ram_2    ),
    .write_data     (write_data_2   ),
    .ram_addr       (ram_addr_2     ),

    .read_data      (read_data_2    )
);

RAM ram_3   (
    .clk            (clk            ),
    .write          (write_ram_3    ),
    .write_data     (write_data_3   ),
    .ram_addr       (ram_addr_3     ),

    .read_data      (read_data_3    )
);
```

LSU

●記憶體存取

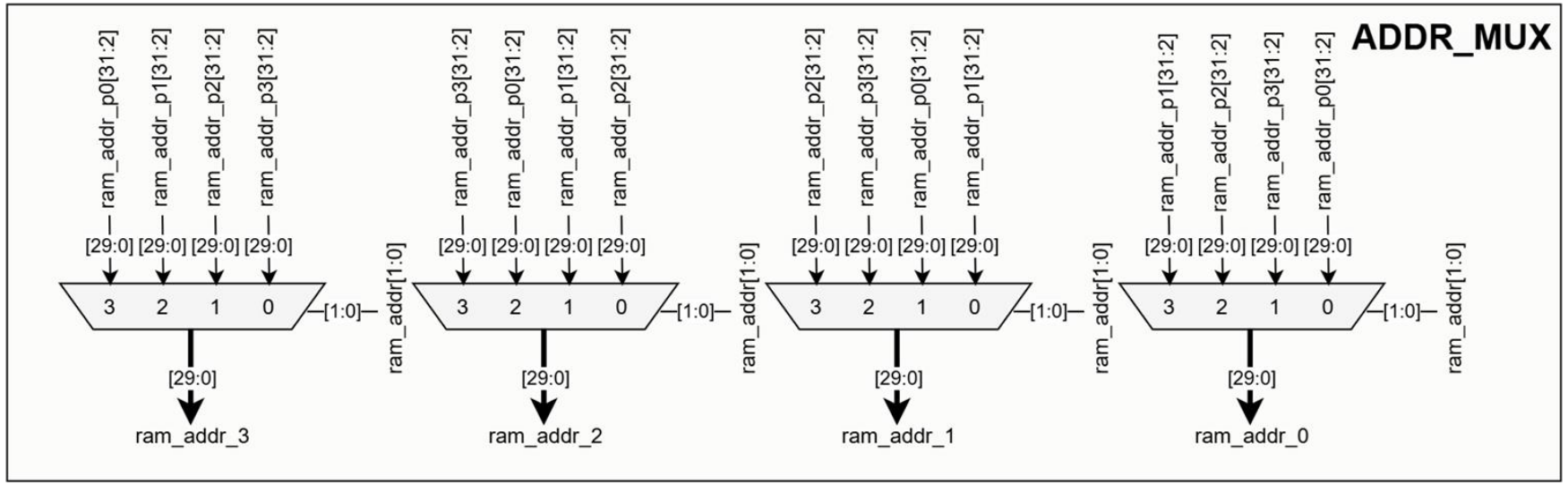
➢ 三種大小的記憶體存取 (1/2/4 Byte) 依據 funct3_r 決定，ALU 只會給予基準位址 (base)。



LSU

● 硬體架構 - ADDR_MUX

- 依據 base addr 是位於哪個 RAM 來分配每個 RAM 的位址
  -




```
logic [31:0] ram_addr_p0;
logic [31:0] ram_addr_p1;
logic [31:0] ram_addr_p2;
logic [31:0] ram_addr_p3;
logic [29:0] ram_addr_0;
logic [29:0] ram_addr_1;
logic [29:0] ram_addr_2;
logic [29:0] ram_addr_3;

assign ram_addr_p0 = ram_addr;
assign ram_addr_p1 = ram_addr + 1;
assign ram_addr_p2 = ram_addr + 2;
assign ram_addr_p3 = ram_addr + 3;

always_comb begin
    unique case (ram_addr_p0[1:0])
        2'b00: begin
            ram_addr_0 = ram_addr_p0[31:2];
            ram_addr_1 = ram_addr_p1[31:2];
            ram_addr_2 = ram_addr_p2[31:2];
            ram_addr_3 = ram_addr_p3[31:2];
        end
        2'b01: begin
            ram_addr_0 = ram_addr_p3[31:2];
            ram_addr_1 = ram_addr_p0[31:2];
            ram_addr_2 = ram_addr_p1[31:2];
            ram_addr_3 = ram_addr_p2[31:2];
        end
        2'b10: begin
            ram_addr_0 = ram_addr_p2[31:2];
            ram_addr_1 = ram_addr_p3[31:2];
            ram_addr_2 = ram_addr_p0[31:2];
            ram_addr_3 = ram_addr_p1[31:2];
        end
        2'b11: begin
            ram_addr_0 = ram_addr_p1[31:2];
            ram_addr_1 = ram_addr_p2[31:2];
            ram_addr_2 = ram_addr_p3[31:2];
            ram_addr_3 = ram_addr_p0[31:2];
        end
    endcase
end
```

LSU

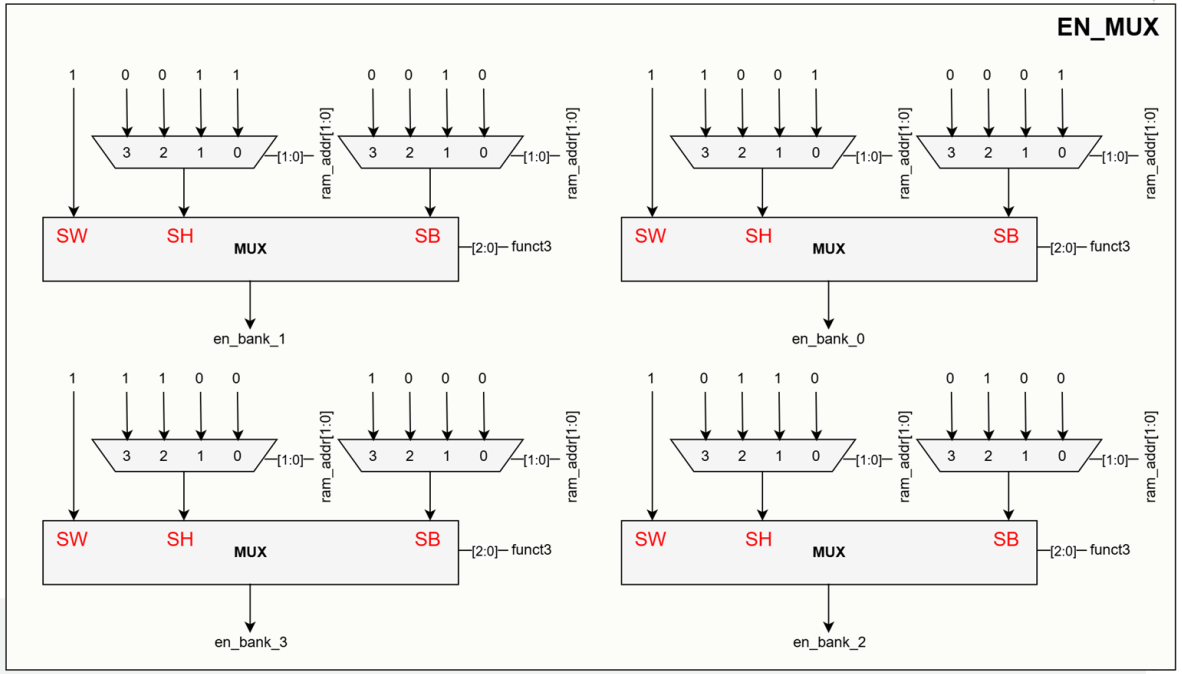
● 硬體架構 - EN_MUX

- 依據 funct3 和 base addr 是位於哪個 RAM 來決定哪些 RAM 可以被寫入。
- Case 1 : SB (Store Byte)

  擁有 base addr 的 RAM_n 可以被寫入。
- Case 2 : SH (Store Half word)

  擁有 base addr 的 RAM_n 和 RAM_{n+1} 可以被寫入。
- Case 3 : SW (Store Word)

  四個 RAM 都要被寫入。



EN_MUX 的四個輸出還須與 FSM 發出的 write_ram 做 AND 運算。

```
logic write_ram_0;
logic write_ram_1;
logic write_ram_2;
logic write_ram_3;
assign write_ram_0 = en_bank_0 & write_ram;
assign write_ram_1 = en_bank_1 & write_ram;
assign write_ram_2 = en_bank_2 & write_ram;
assign write_ram_3 = en_bank_3 & write_ram;
```

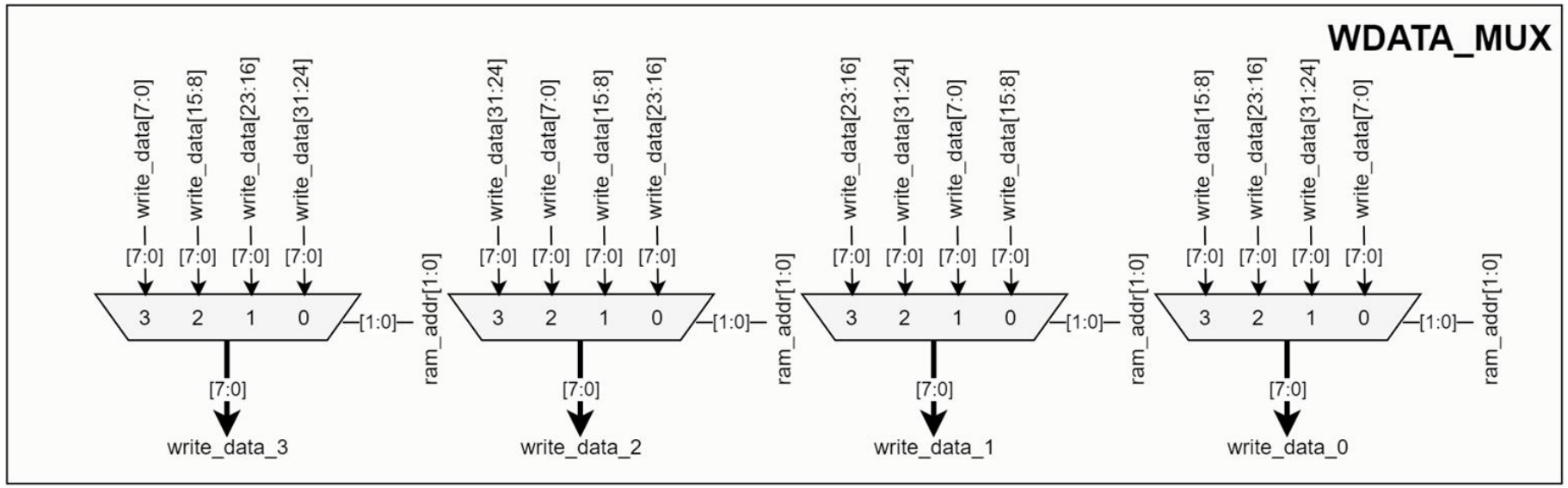
```
logic en_bank_0;
logic en_bank_1;
logic en_bank_2;
logic en_bank_3;

always_comb begin
    en_bank_0 = 0;
    en_bank_1 = 0;
    en_bank_2 = 0;
    en_bank_3 = 0;
    unique case (funct3)
        `F_SB: begin
            unique case (ram_addr[1:0])
                2'b00: en_bank_0 = 1;
                2'b01: en_bank_1 = 1;
                2'b10: en_bank_2 = 1;
                2'b11: en_bank_3 = 1;
            endcase
        end
        `F_SH: begin
            unique case (ram_addr[1:0])
                2'b00: begin
                    en_bank_0 = 1;
                    en_bank_1 = 1;
                end
                2'b01: begin
                    en_bank_1 = 1;
                    en_bank_2 = 1;
                end
                2'b10: begin
                    en_bank_2 = 1;
                    en_bank_3 = 1;
                end
                2'b11: begin
                    en_bank_3 = 1;
                    en_bank_0 = 1;
                end
            endcase
        end
        `F_SW: begin
            en_bank_0 = 1;
            en_bank_1 = 1;
            en_bank_2 = 1;
            en_bank_3 = 1;
        end
    endcase
end
```

LSU

● 硬體架構 - WDATA_MUX

- 依據 base addr 是位於哪個 RAM 來分配寫入資料。
- 若 base addr 位於 RAM_n，資料由 RAM_n 開始由位元小到大依序分配。

```
logic [7:0] write_data_0;
logic [7:0] write_data_1;
logic [7:0] write_data_2;
logic [7:0] write_data_3;

always_comb begin
    unique case (ram_addr[1:0])
        2'b00: begin
            write_data_0 = write_data[7:0];
            write_data_1 = write_data[15:8];
            write_data_2 = write_data[23:16];
            write_data_3 = write_data[31:24];
        end
        2'b01: begin
            write_data_0 = write_data[31:24];
            write_data_1 = write_data[7:0];
            write_data_2 = write_data[15:8];
            write_data_3 = write_data[23:16];
        end
        2'b10: begin
            write_data_0 = write_data[23:16];
            write_data_1 = write_data[31:24];
            write_data_2 = write_data[7:0];
            write_data_3 = write_data[15:8];
        end
        2'b11: begin
            write_data_0 = write_data[15:8];
            write_data_1 = write_data[23:16];
            write_data_2 = write_data[31:24];
            write_data_3 = write_data[7:0];
        end
    endcase
end
```



LSU

● 硬體架構 - RDATA_MUX (1/2)

- 依據 funct3 和 base addr 是位於哪個 RAM 來重組讀取資料。
- Case 1 : LB (Load Byte)

  只需擁有 base addr 的 RAM_n 讀取出資料做符號擴充。
- Case 2 : LH (Load Half word)

  擁有 base addr 的 RAM_n 和 RAM_{n+1} 讀取出資料做符號擴充
- Case 3 : LW (Load Word)

  四塊 RAM 讀取出的資料做重組。
- Case 4 : LBU (Load Byte Unsigned)

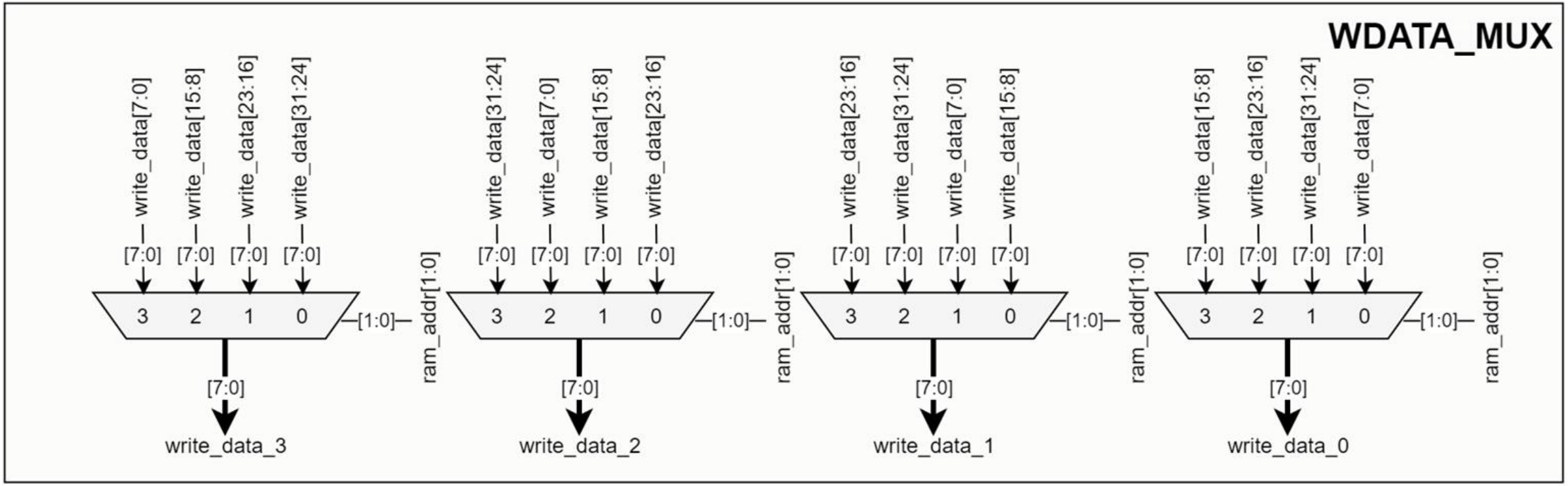
  只需擁有 base addr 的 RAM_n 讀取出資料做零擴充。
- Case 5 : LHU (Load Half word Unsigned)

  擁有 base addr 的 RAM_n 和 RAM_{n+1} 讀取出資料做零擴充。

```
always_comb begin
    unique case (funct3)
        `F_LB: begin
            unique case (ram_addr[1:0])
                2'b00: read_data = {{24{read_data_0[7]}}, read_data_0};
                2'b01: read_data = {{24{read_data_1[7]}}, read_data_1};
                2'b10: read_data = {{24{read_data_2[7]}}, read_data_2};
                2'b11: read_data = {{24{read_data_3[7]}}, read_data_3};
            endcase
        end
        `F_LH: begin
            unique case (ram_addr[1:0])
                2'b00: read_data = {{16{read_data_1[7]}}, read_data_1, read_data_0};
                2'b01: read_data = {{16{read_data_2[7]}}, read_data_2, read_data_1};
                2'b10: read_data = {{16{read_data_3[7]}}, read_data_3, read_data_2};
                2'b11: read_data = {{16{read_data_0[7]}}, read_data_0, read_data_3};
            endcase
        end
        `F_LW: begin
            unique case (ram_addr[1:0])
                2'b00: read_data = {read_data_3, read_data_2, read_data_1, read_data_0};
                2'b01: read_data = {read_data_0, read_data_3, read_data_2, read_data_1};
                2'b10: read_data = {read_data_1, read_data_0, read_data_3, read_data_2};
                2'b11: read_data = {read_data_2, read_data_1, read_data_0, read_data_3};
            endcase
        end
        `F_LBU: begin
            unique case (ram_addr[1:0])
                2'b00: read_data = {24'h0, read_data_0};
                2'b01: read_data = {24'h0, read_data_1};
                2'b10: read_data = {24'h0, read_data_2};
                2'b11: read_data = {24'h0, read_data_3};
            endcase
        end
        `F_LHU: begin
            unique case (ram_addr[1:0])
                2'b00: read_data = {16'h0, read_data_1, read_data_0};
                2'b01: read_data = {16'h0, read_data_2, read_data_1};
                2'b10: read_data = {16'h0, read_data_3, read_data_2};
                2'b11: read_data = {16'h0, read_data_0, read_data_3};
            endcase
        end
    endcase
end
```

LSU

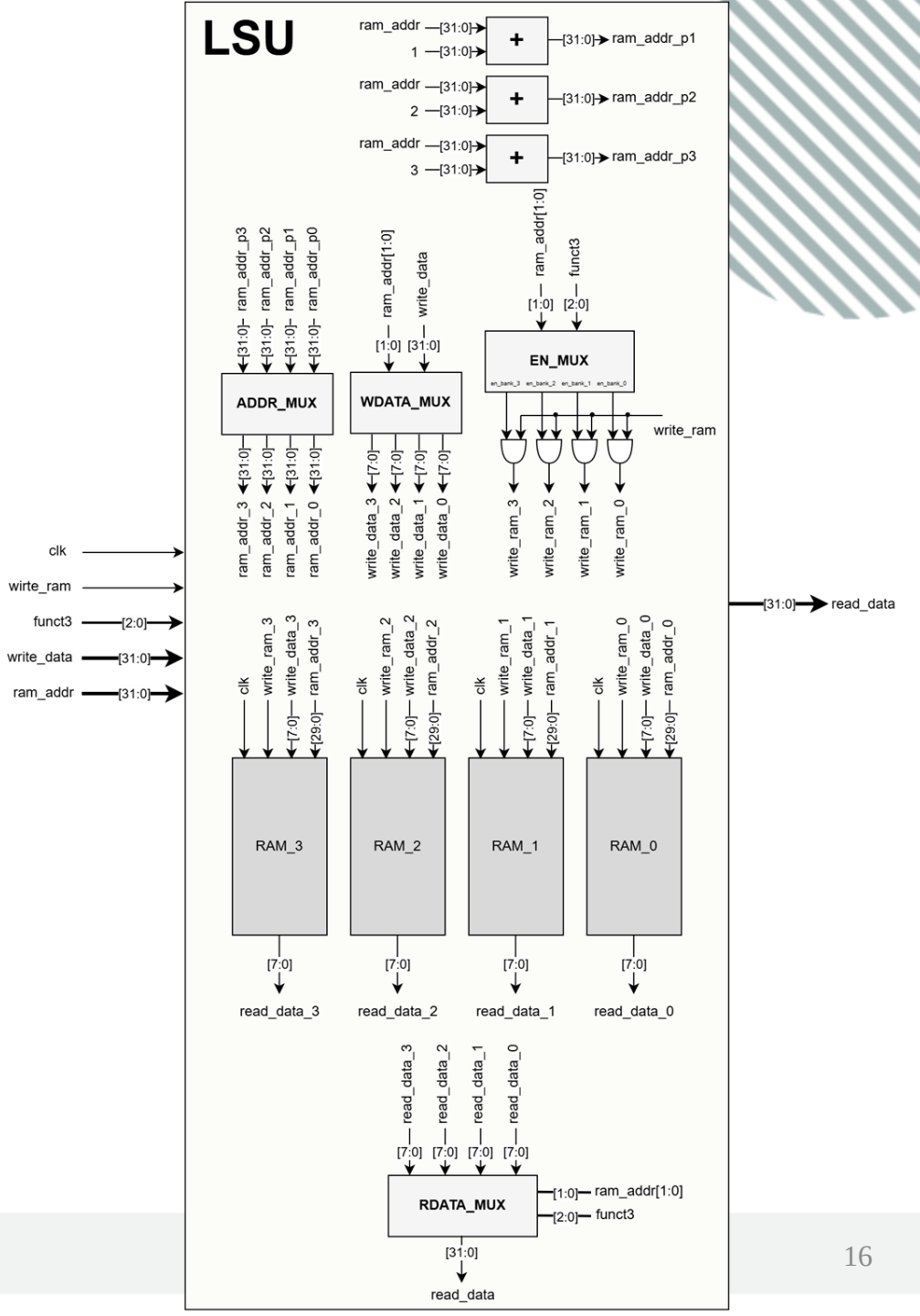
●硬體架構 - RDATA_MUX (2/2)



LSU

●硬體架構 - LSU

- ADDER
  - ◆ 計算除了 base addr 以外的記憶體位址。
- ADDR_MUX
  - ◆ 指派每塊記憶體的位址。
- WDATA_MUX
  - ◆ 將 32 位元的寫入資料分拆成 4 等份，並依據 ram_addr[1:0] 去派遣。
- EN_MUX
  - ◆ 根據 ram_addr[1:0] 和 funct3 決定哪些 RAM 可以被寫入，最後再和 write_ram 訊號做 AND 運算。
- RAM
  - ◆ 執行 8 位元讀取和寫入的動作。
- RDATA_MUX
  - ◆ 將 4 塊 RAM 讀取出來的 8 位元資料進行重組。

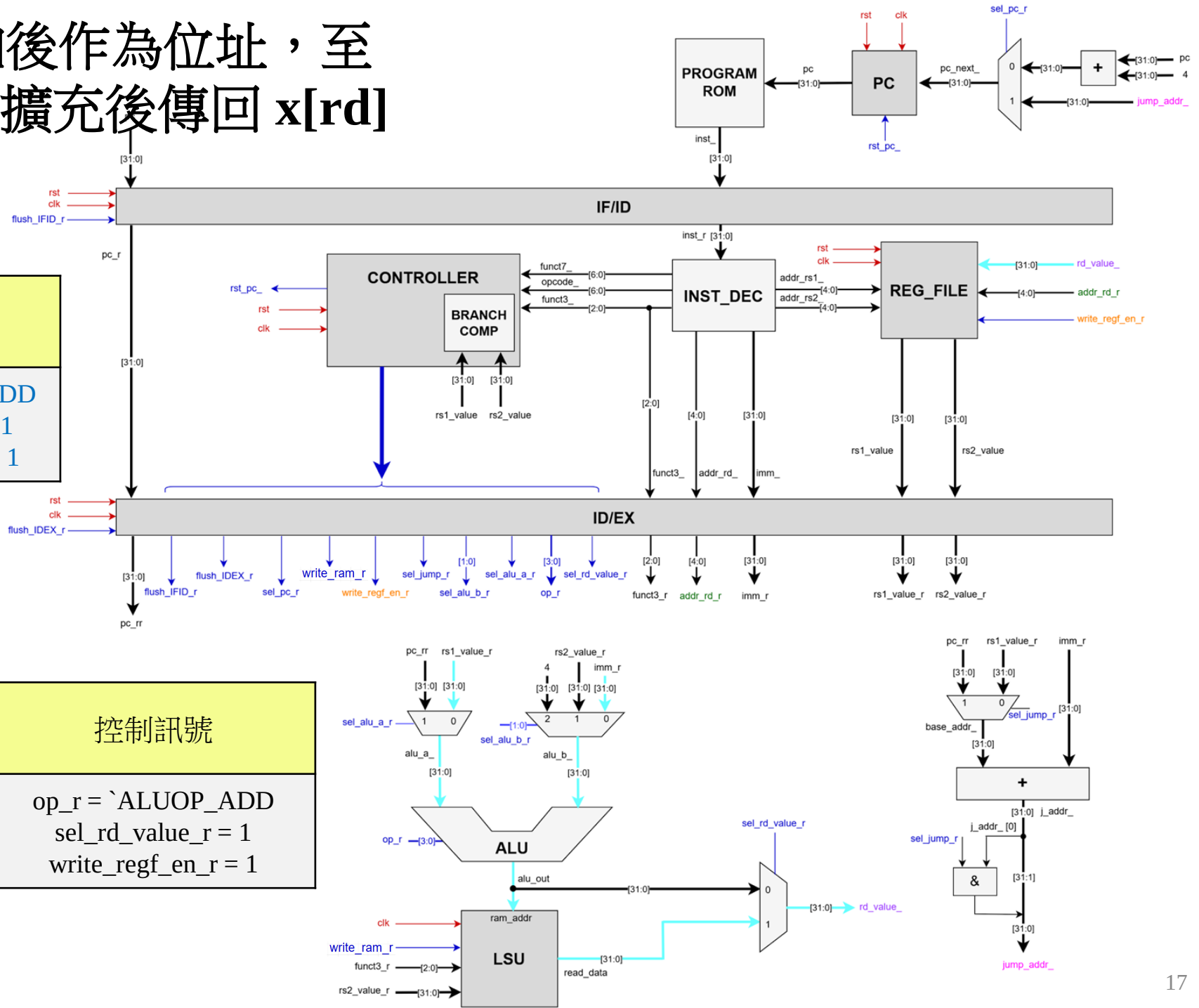


LB : x[rs1] 和 offset 相加後作為位址，至 RAM 中取 8 位元做符號擴充後傳回 x[rd]

動作	控制訊號
if ((opcode_ == `Opcode_L) && (funct3_ == `F_LB)) 發出 LB 的控制訊號	op_ = `ALUOP_ADD sel_rd_value_ = 1 write_regf_en_ = 1

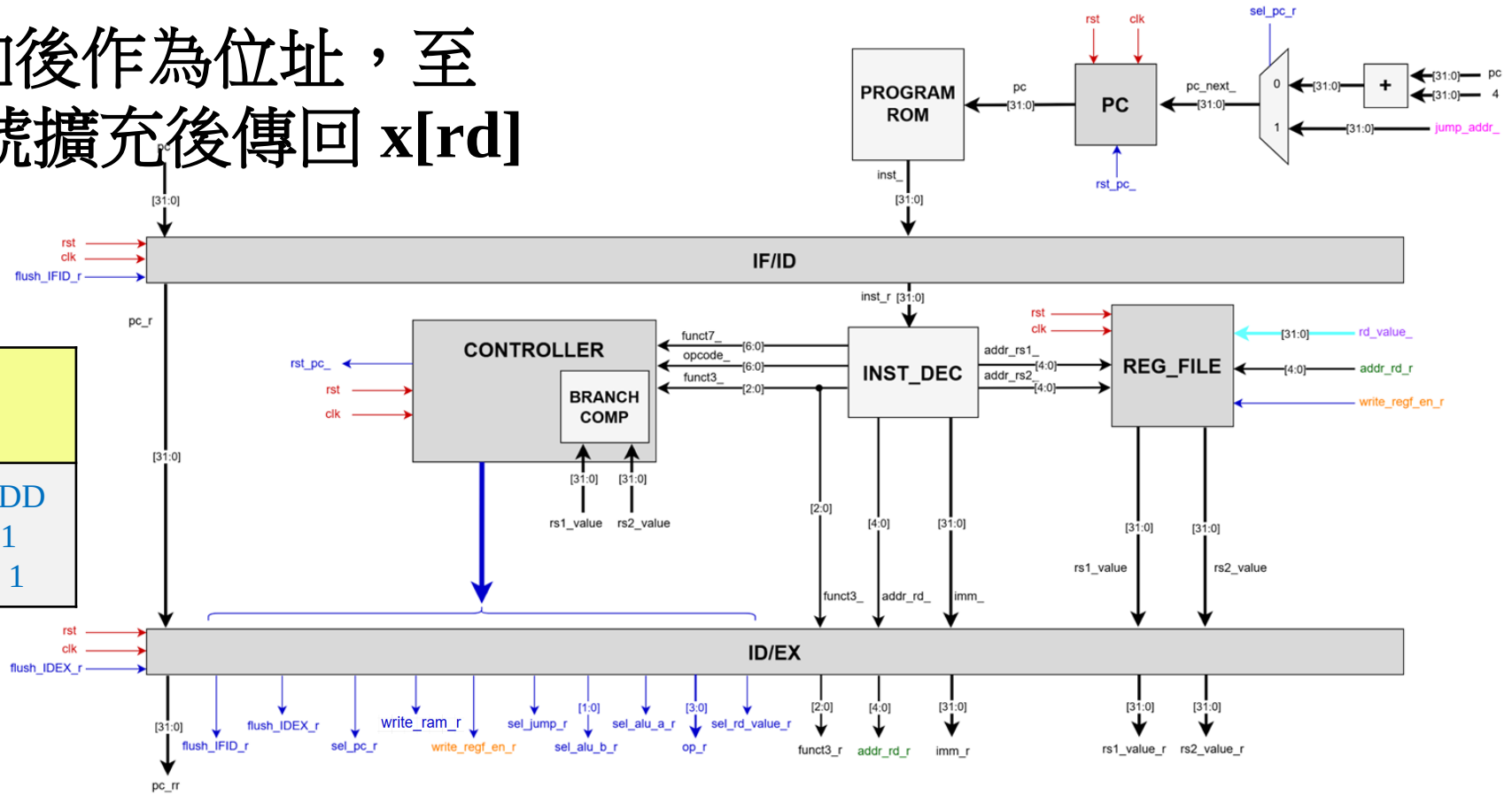
```
`define Opcode_L        7'b0000011
`define F_LB            3'b000
`define F_LH            3'b001
`define F_LW            3'b010
`define F_LBU           3'b100
`define F_LHU           3'b101
```

動作	控制訊號
※ addr = rs1_value_r + imm_r rd_value_ ← sext(M[addr])	op_r = `ALUOP_ADD sel_rd_value_r = 1 write_regf_en_r = 1

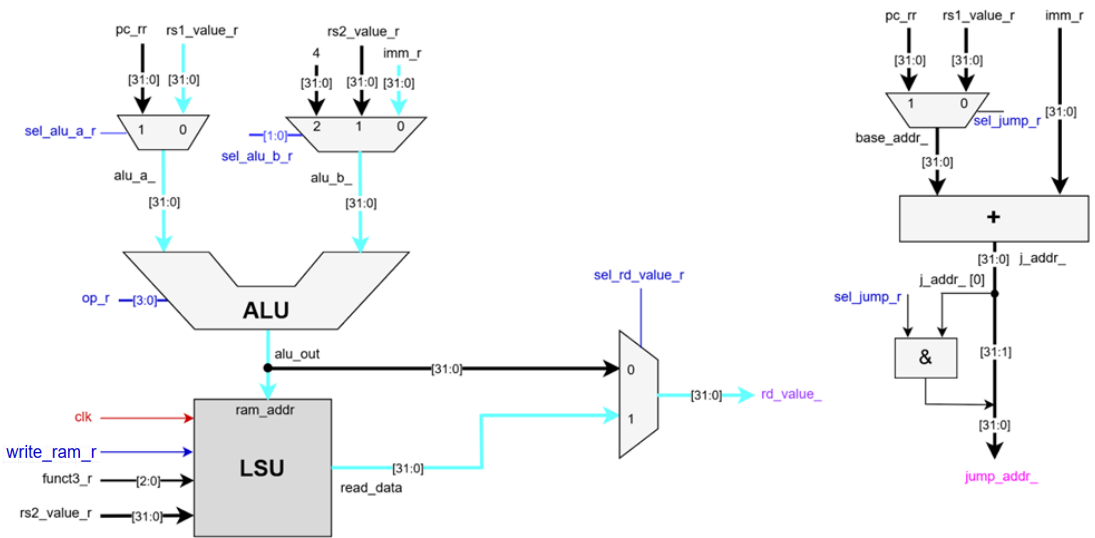


LH : x[rs1] 和 offset 相加後作為位址，至 RAM 中取 16 位元做符號擴充後傳回 x[rd]

動作	控制訊號
if ((opcode_ == `Opcode_L) && (funct3_ == `F_LH)) 發出 LH 的控制訊號	op_ = `ALUOP_ADD sel_rd_value_ = 1 write_regf_en_ = 1

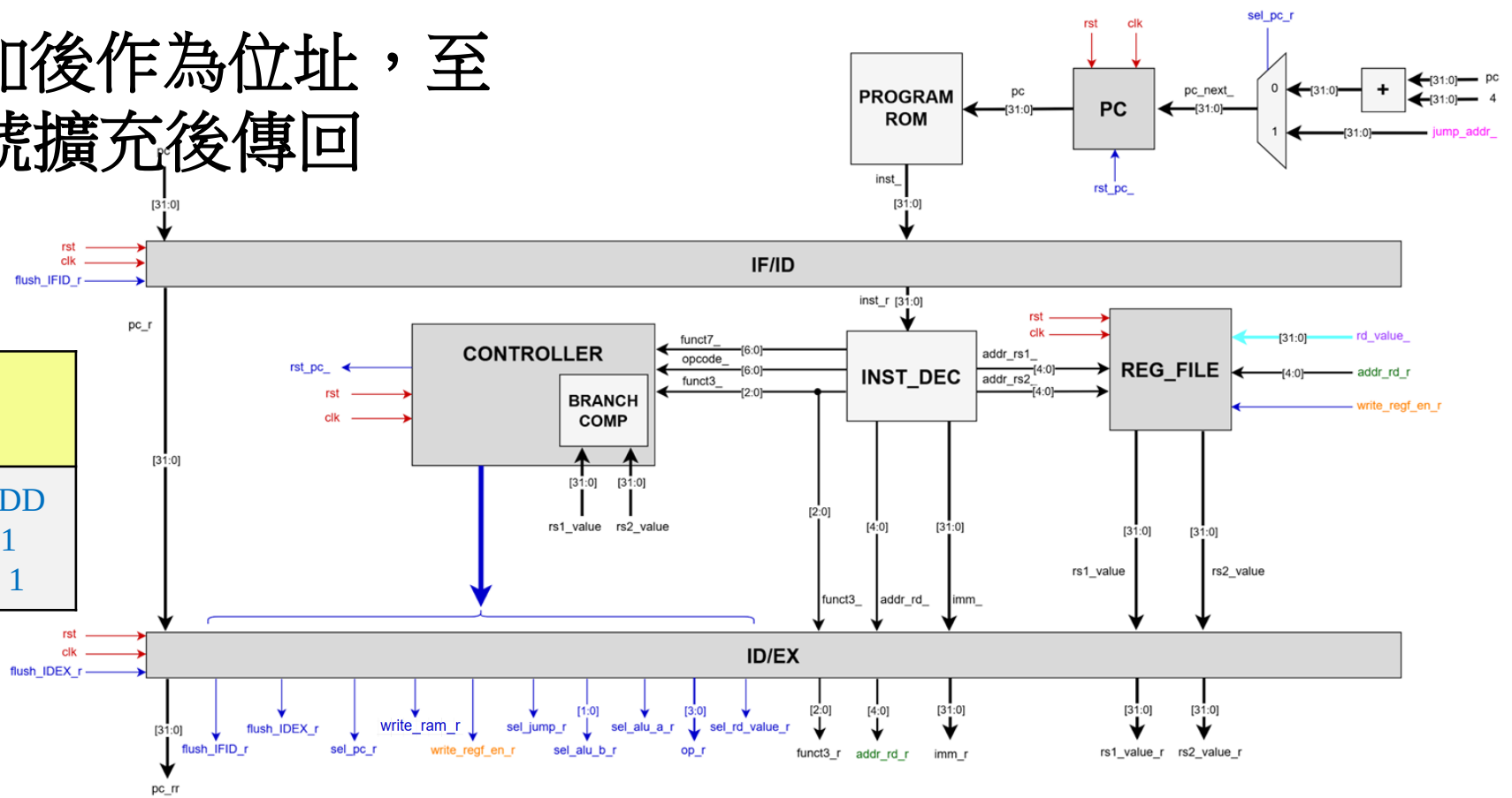


動作	控制訊號
※ addr = rs1_value_r + imm_r rd_value_ ← sext(M[addr + 1 : addr])	op_r = `ALUOP_ADD sel_rd_value_r = 1 write_regf_en_r = 1

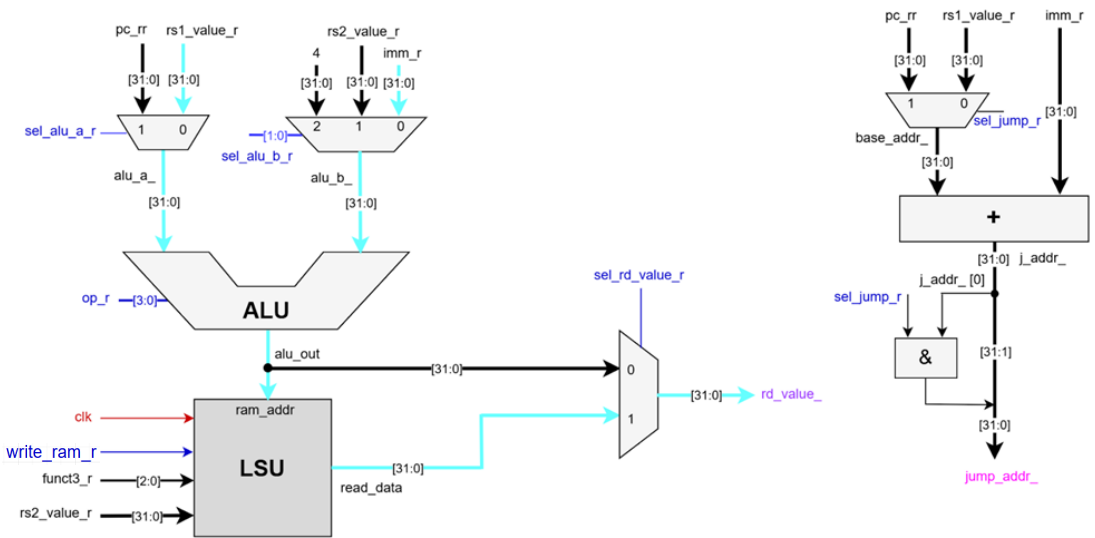


LW : x[rs1] 和 offset 相加後作為位址，至 RAM 中取 32 位元做符號擴充後傳回 x[rd]

動作	控制訊號
if ((opcode_ == `Opcode_L) && (funct3_ == `F_LW)) 發出 LW 的控制訊號	op_ = `ALUOP_ADD sel_rd_value_ = 1 write_regf_en_ = 1

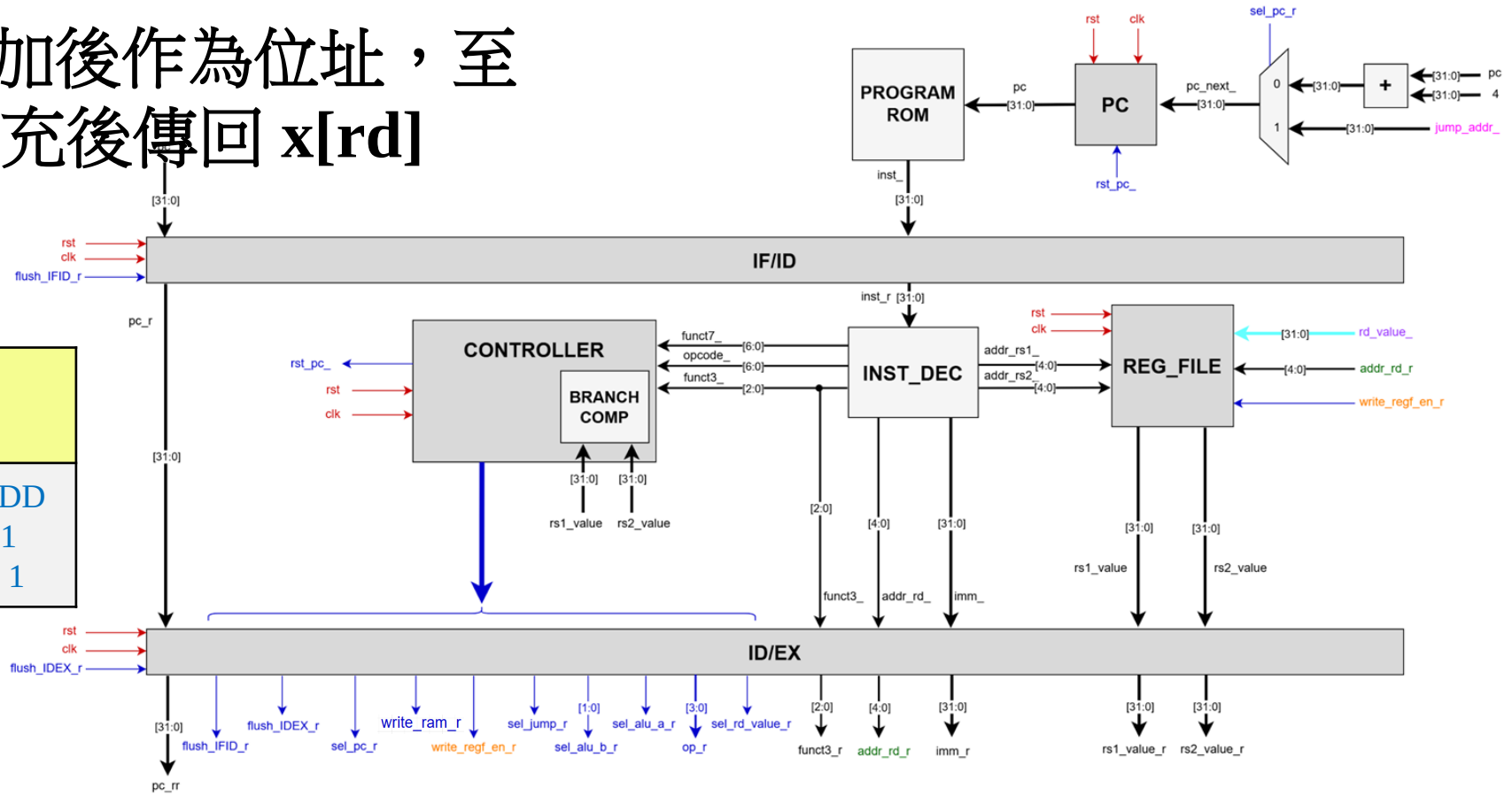


動作	控制訊號
※ addr = rs1_value_r + imm_r rd_value_ ← sext(M[addr + 3 : addr])	op_r = `ALUOP_ADD sel_rd_value_r = 1 write_regf_en_r = 1



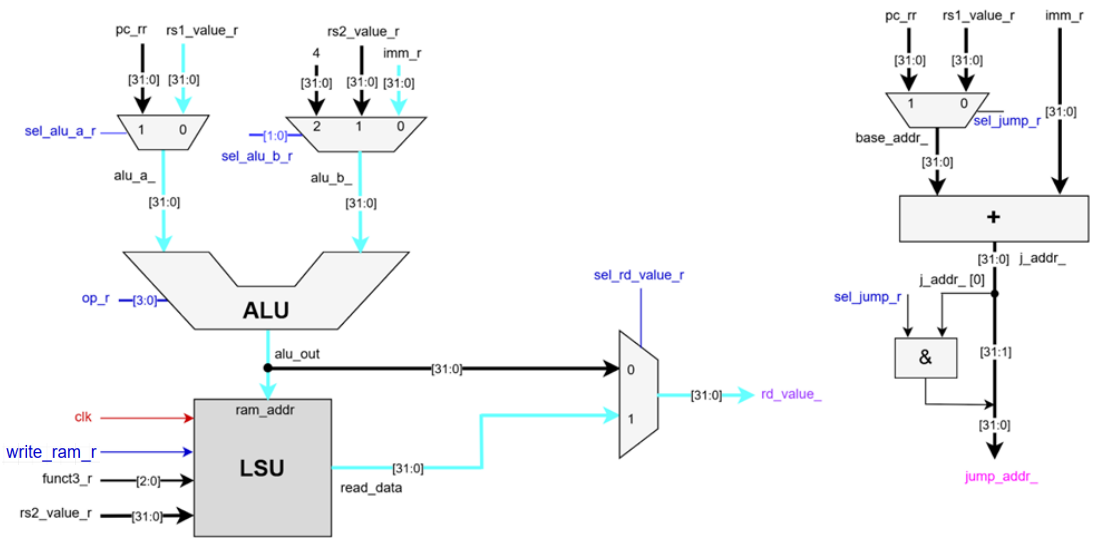
LBU : x[rs1] 和 offset 相加後作為位址，至 RAM 中取 8 位元做零擴充後傳回 x[rd]

動作	控制訊號
if ((opcode_ == `Opcode_L) && (funct3_ == `F_LBU)) 發出 LBU 的控制訊號	op_ = `ALUOP_ADD sel_rd_value_ = 1 write_regf_en_ = 1



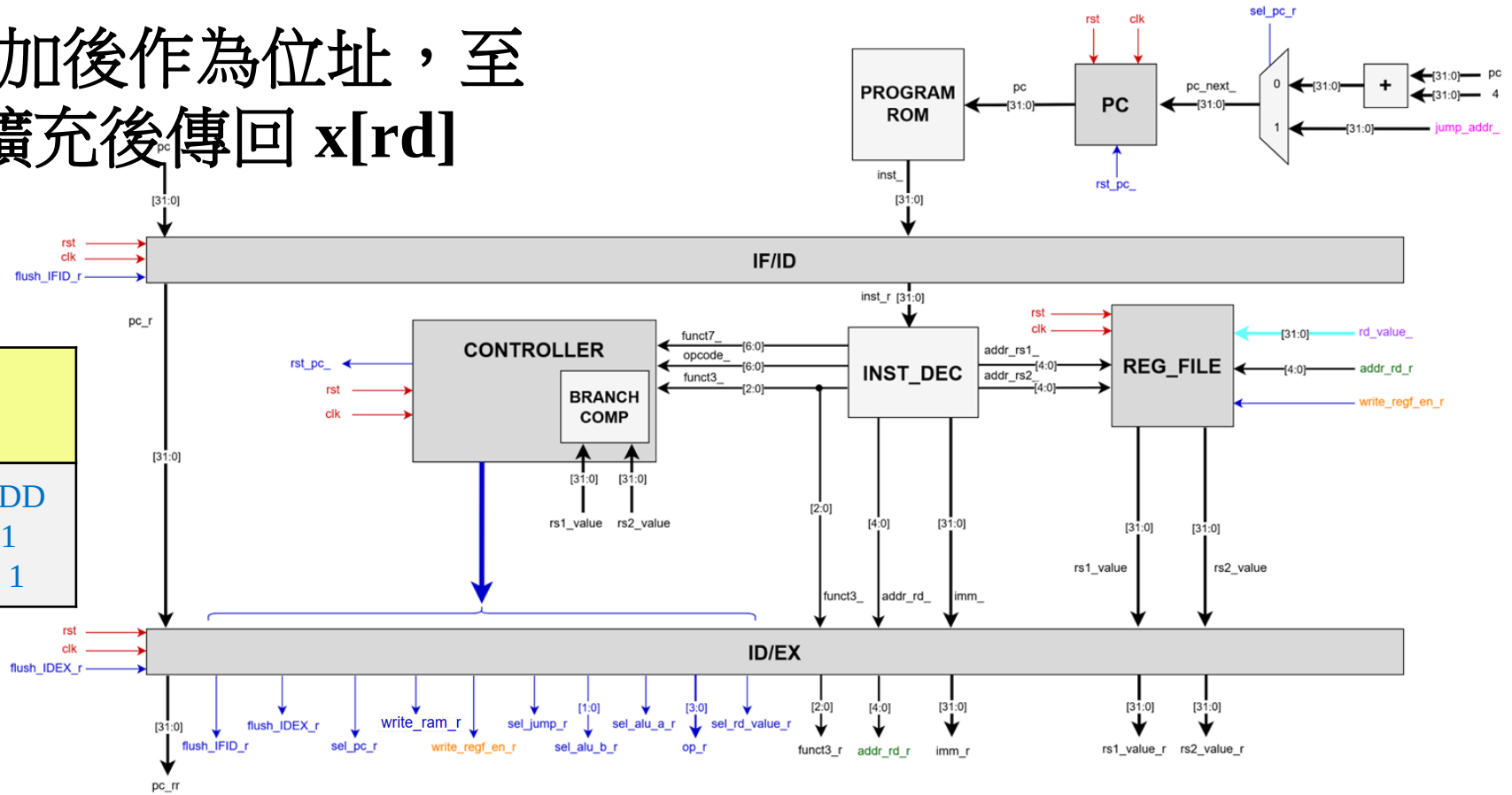
動作	控制訊號
※ addr = rs1_value_r + imm_r rd_value_ ← zext(M[addr])	op_r = `ALUOP_ADD sel_rd_value_r = 1 write_regf_en_r = 1

※zext : zero extension

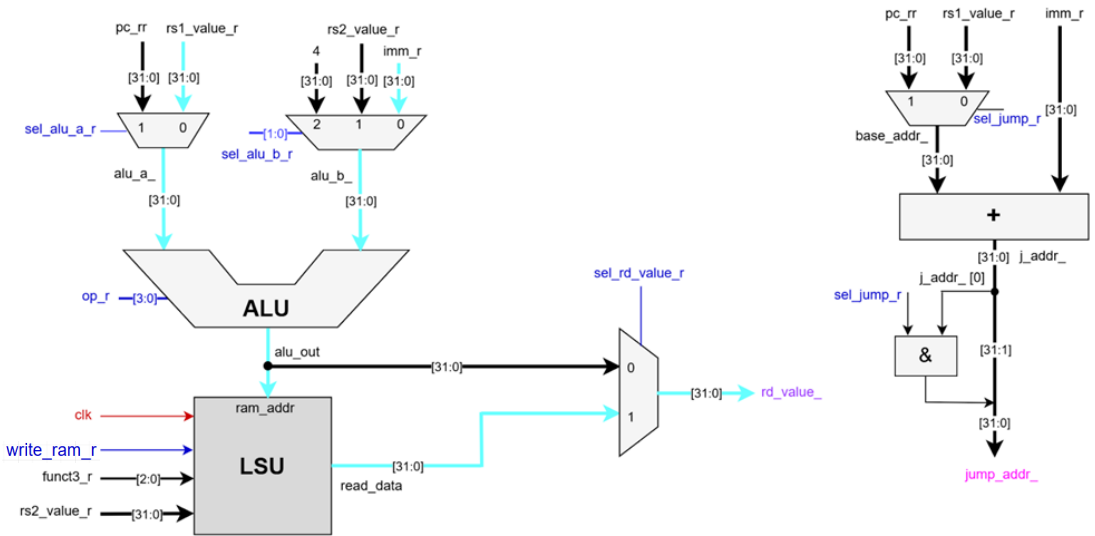


LHU : x[rs1] 和 offset 相加後作為位址，至 RAM 中取 16 位元做零擴充後傳回 x[rd]

動作	控制訊號
if ((opcode_ == `Opcode_L) && (funct3_ == `F_LHU)) 發出 LHU 的控制訊號	op_ = `ALUOP_ADD sel_rd_value_ = 1 write_regf_en_ = 1



動作	控制訊號
※ addr = rs1_value_r + imm_r rd_value_ ← zext(M[addr + 1 : addr])	op_r = `ALUOP_ADD sel_rd_value_r = 1 write_regf_en_r = 1

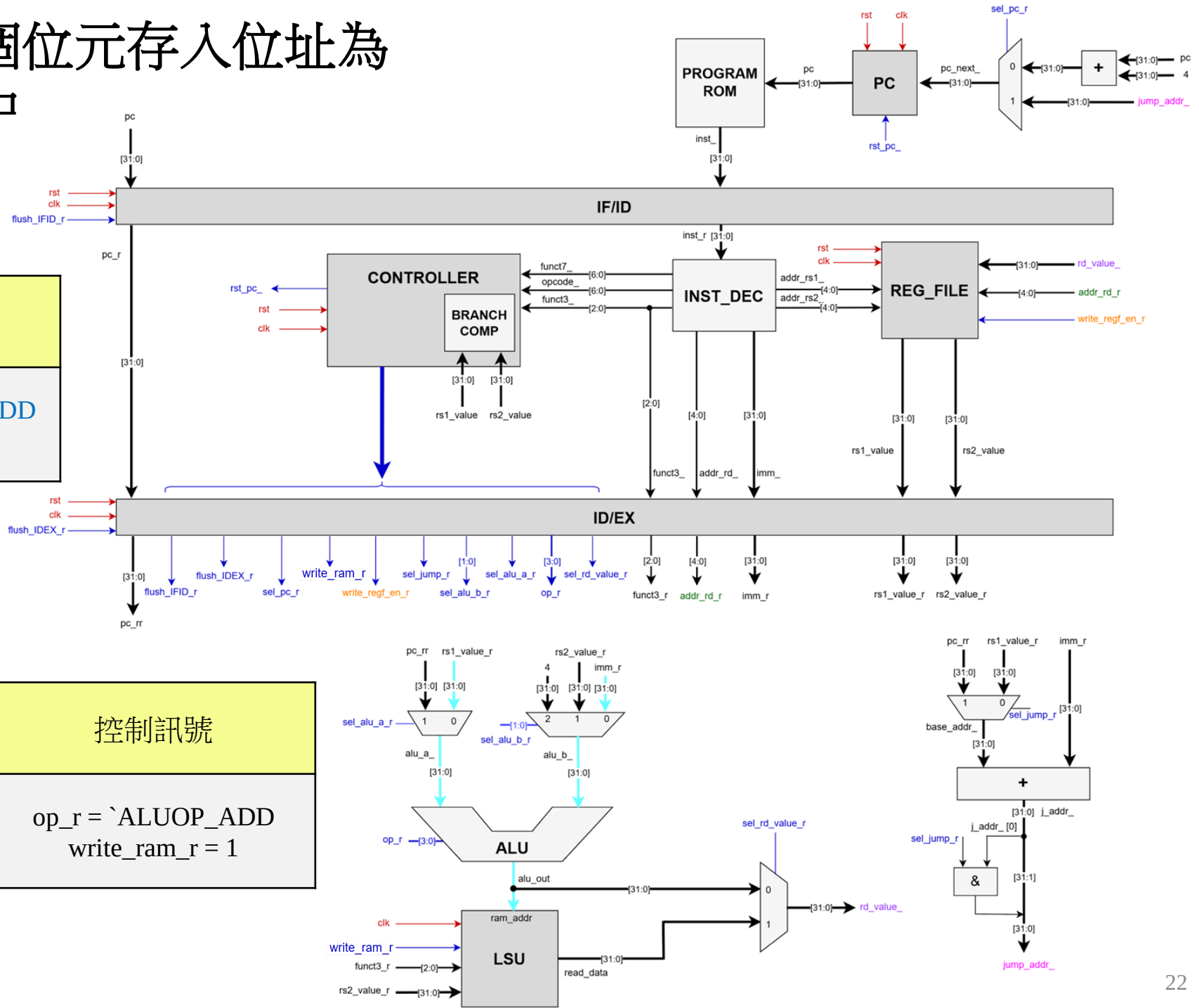


SB : 將 x[rs2] 較低的 8 個位元存入位址為 x[rs1] + offset 的 RAM 中

動作	控制訊號
if ((opcode_ == `Opcode_S) && (funct3_ == `F_SB)) 發出 SB 的控制訊號	op_ = `ALUOP_ADD write_ram_ = 1

```
`define Opcode_S    7'b0100011
`define F_SB        3'b000
`define F_SH        3'b001
`define F_SW        3'b010
```

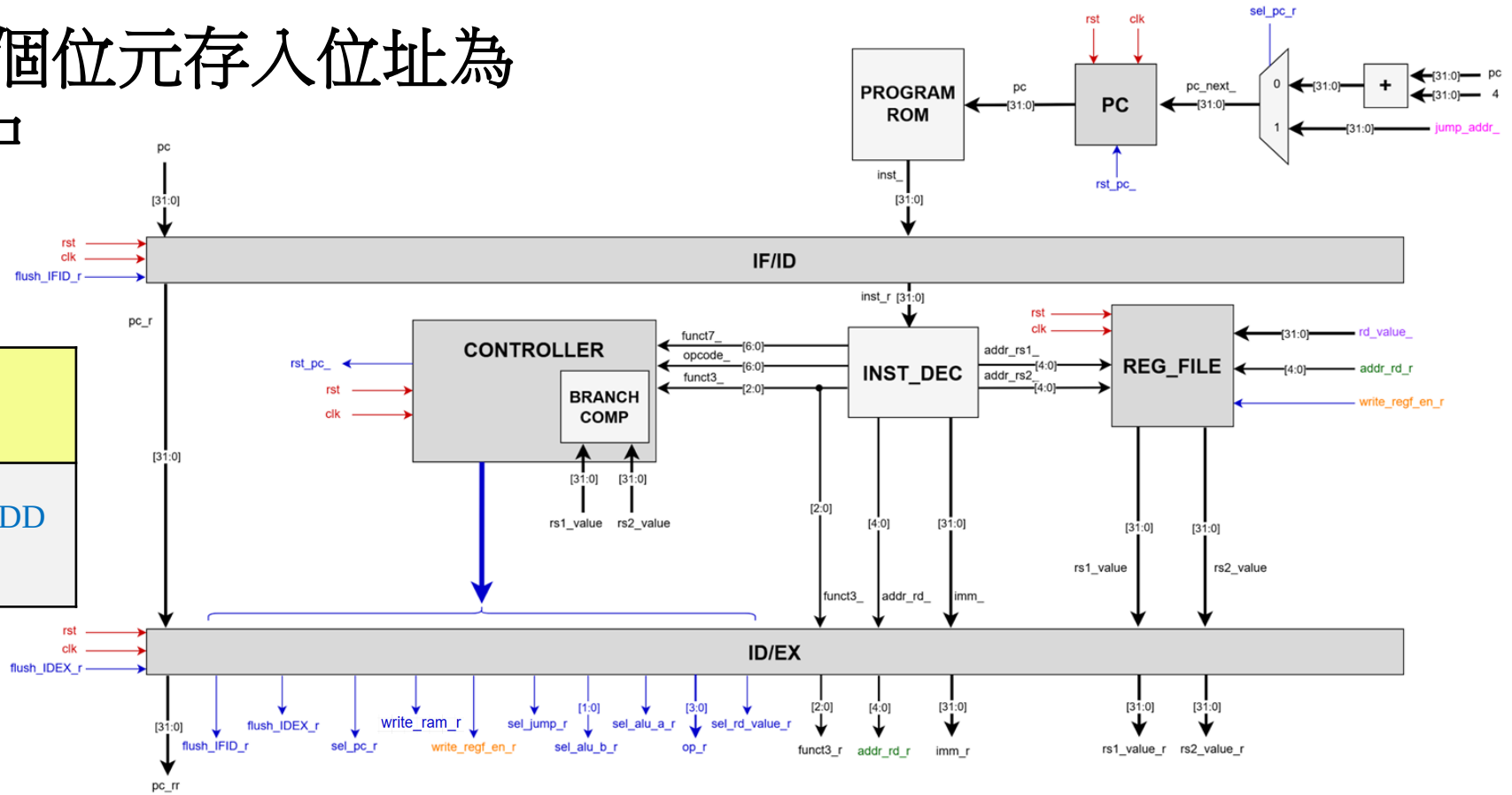
動作	控制訊號
※ addr = rs1_value_r + imm_r M[addr] ← rs2_value_r [7:0]	op_r = `ALUOP_ADD write_ram_r = 1

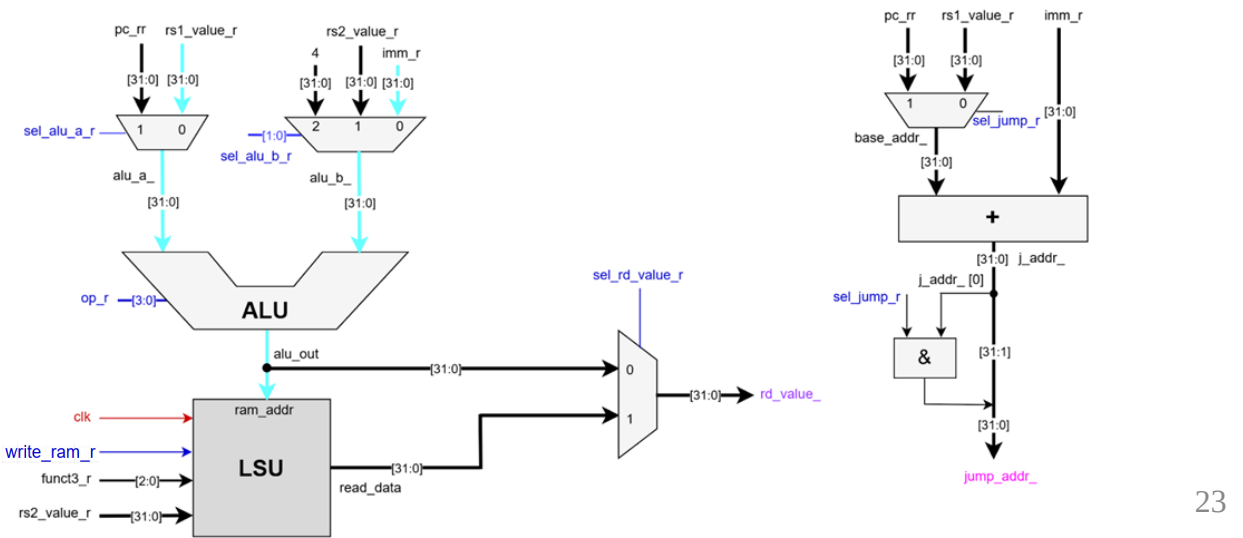


SH :將 x[rs2] 較低的 16 個位元存入位址為 x[rs1] + offset 的 RAM 中

動作	控制訊號
if ((opcode_ == `Opcode_S) && (funct3_ == `F_SH)) 發出 SH 的控制訊號	op_ = `ALUOP_ADD write_ram_ = 1

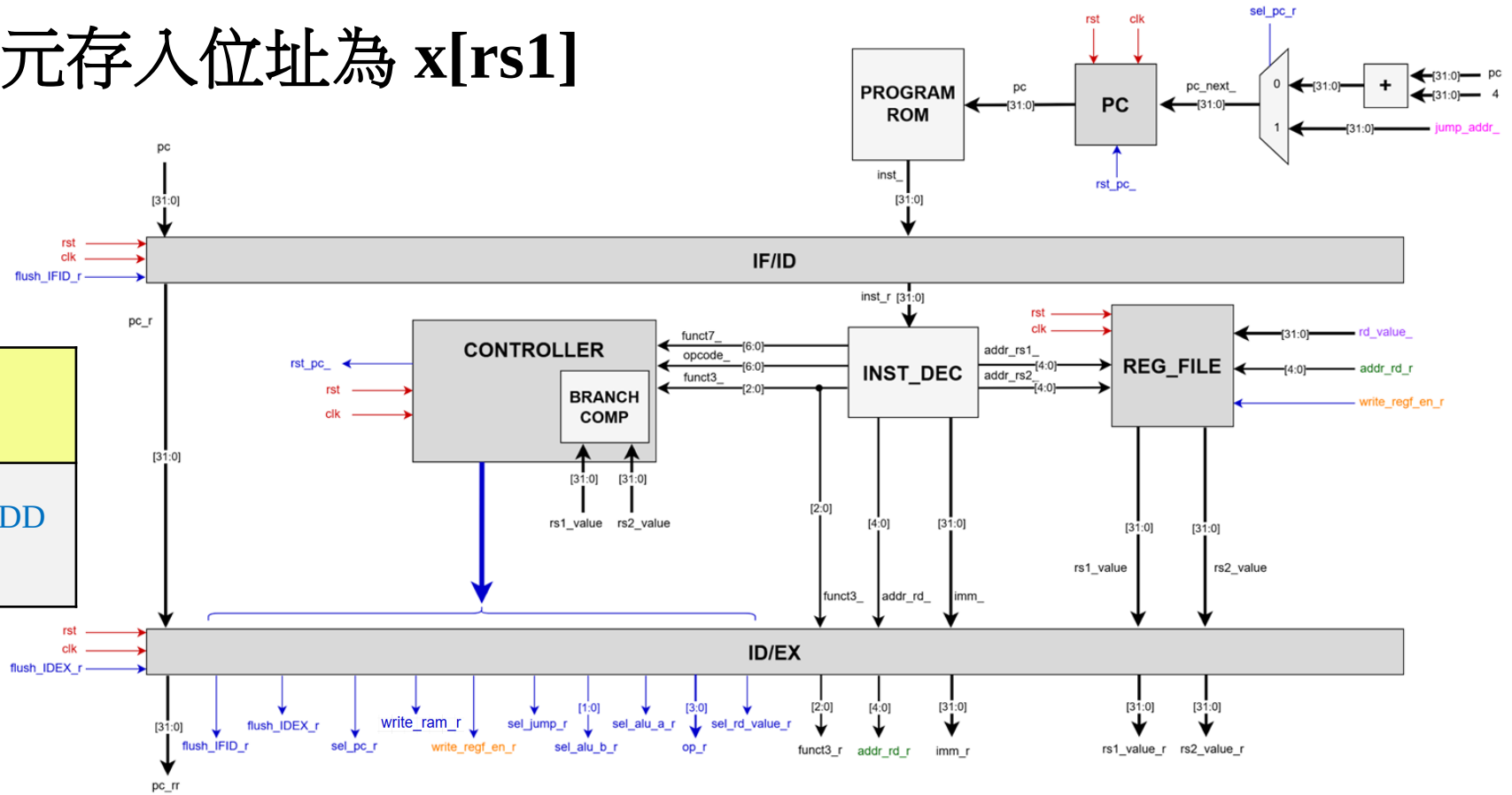
動作	控制訊號
※ addr = rs1_value_r + imm_r M[addr + 1 : addr] ← rs2_value_r [15:0]	op_r = `ALUOP_ADD write_ram_r = 1



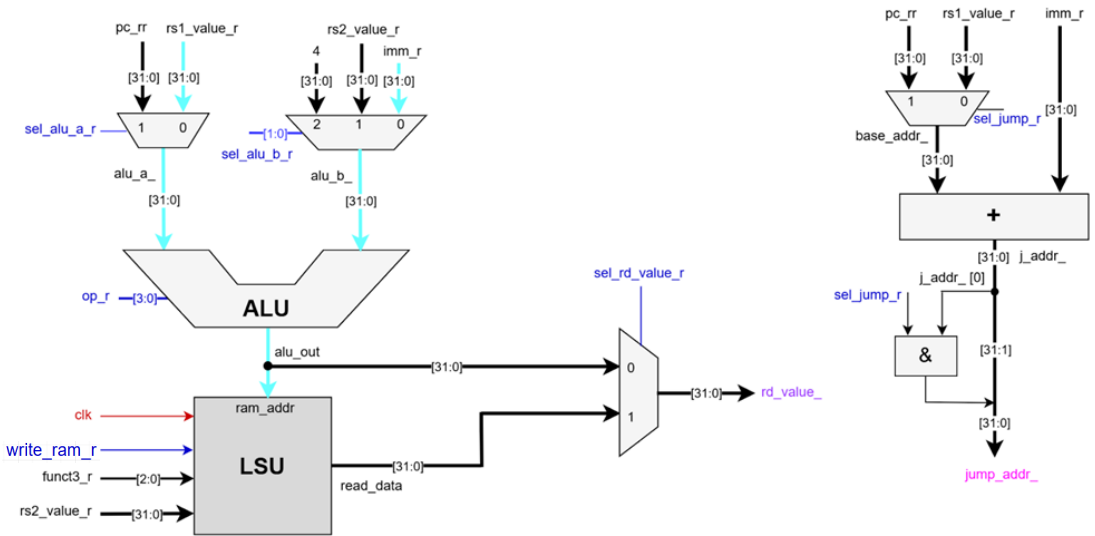


SW :將 x[rs2] 共 32 個位元存入位址為 x[rs1] + offset 的 RAM 中

動作	控制訊號
if ((opcode_ == `Opcode_S) && (funct3_ == `F_SW)) 發出 SW 的控制訊號	op_ = `ALUOP_ADD write_ram_ = 1



動作	控制訊號
※ addr = rs1_value_r + imm_r M[addr + 3 : addr] ← rs2_value_r [31:0]	op_r = `ALUOP_ADD write_ram_r = 1

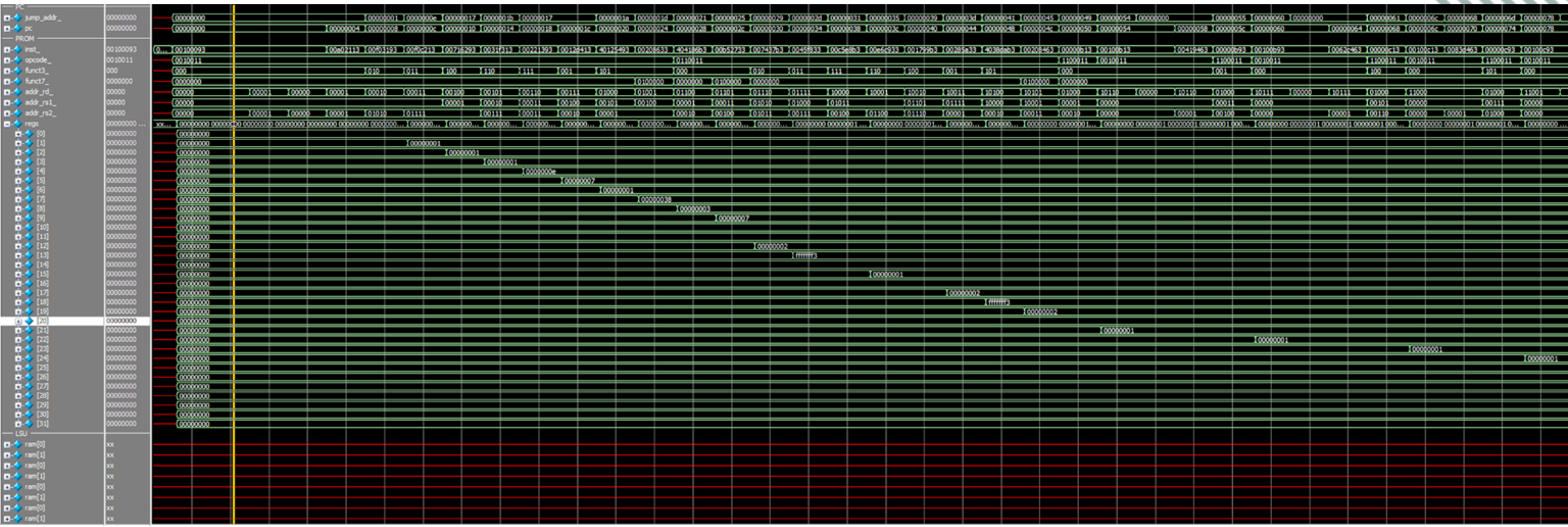


上課實作：ModelSim 模擬

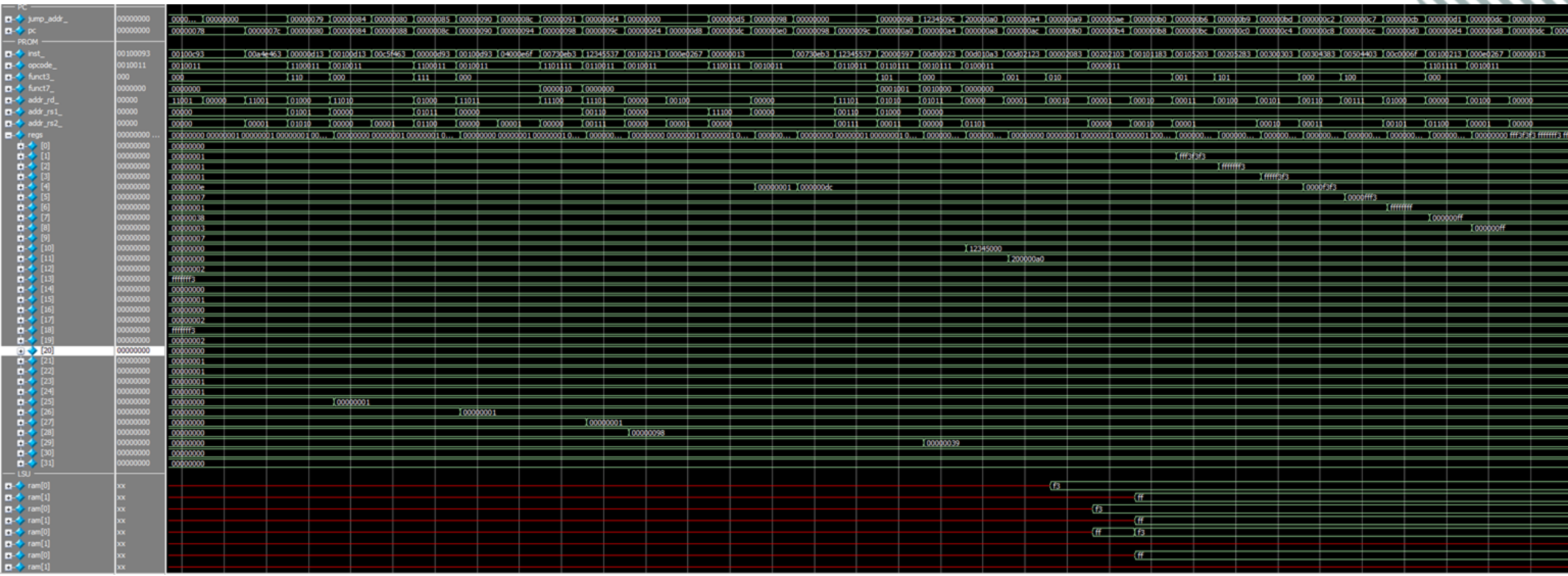
```
#立即值定址
    addi x1, x0, 1
    slti x2, x0, 10
    sltiu x3, x0, 15
    xori x4, x1, 0xF
    ori x5, x2, 0x7
    andi x6, x3, 0x3
    slli x7, x4, 2
    srli x8, x5, 1
    srai x9, x4, 1

#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

#記憶體存取
    sb x13, 0(x0)
    sh x13, 1(x0)
    sw x13, 2(x0)
    lw x1, 0(x0)
    lw x2, 2(x0)
    lh x3, 1(x0)
    lhu x4, 1(x0)
    lhu x5, 2(x0)
    lb x6, 3(x0)
    lbu x7, 3(x0)
    lbu x8, 5(x0)

    j end
jump:
    addi x4, x0, 1
    jalr x4, x28, 0
end:
    nop
```

1	0x00000000	0x00100093	addi x1 x0 1
2	0x00000004	0x00A02113	slti x2 x0 10
3	0x00000008	0x00F03193	sltiu x3 x0 15
4	0x0000000C	0x00F0C213	xori x4 x1 15
5	0x00000010	0x00716293	ori x5 x2 7
6	0x00000014	0x0031F313	andi x6 x3 3
7	0x00000018	0x00221393	slli x7 x4 2
8	0x0000001C	0x0012D413	srli x8 x5 1
9	0x00000020	0x40125493	srai x9 x4 1
10	0x00000024	0x00208633	add x12 x1 x2
11	0x00000028	0x404186B3	sub x13 x3 x4
12	0x0000002C	0x00B52733	slt x14 x10 x11
13	0x00000030	0x007437B3	sltu x15 x8 x7
14	0x00000034	0x0045F833	and x16 x11 x4
15	0x00000038	0x00C5E8B3	or x17 x11 x12
16	0x0000003C	0x00E6C933	xor x18 x13 x14
17	0x00000040	0x001799B3	sll x19 x15 x1
18	0x00000044	0x00285A33	srl x20 x16 x2
19	0x00000048	0x4038DAB3	sra x21 x17 x3
20	0x0000004C	0x00208463	beq x1 x2 8
21	0x00000050	0x00000B13	addi x22 x0 0
22	0x00000054	0x00100B13	addi x22 x0 1
23	0x00000058	0x00419463	bne x3 x4 8
24	0x0000005C	0x00000B93	addi x23 x0 0
25	0x00000060	0x00100B93	addi x23 x0 1
26	0x00000064	0x0062C463	blt x5 x6 8
27	0x00000068	0x00000C13	addi x24 x0 0
28	0x0000006C	0x00100C13	addi x24 x0 1
28	0x0000006C	0x00100C13	addi x24 x0 1
29	0x00000070	0x0083D463	bge x7 x8 8
30	0x00000074	0x00000C93	addi x25 x0 0
31	0x00000078	0x00100C93	addi x25 x0 1
32	0x0000007C	0x00A4E463	bltu x9 x10 8
33	0x00000080	0x00000D13	addi x26 x0 0
34	0x00000084	0x00100D13	addi x26 x0 1
35	0x00000088	0x00C5F463	bgeu x11 x12 8
36	0x0000008C	0x00000D93	addi x27 x0 0
37	0x00000090	0x00100D93	addi x27 x0 1
38	0x00000094	0x04000E6F	jal x28 64
39	0x00000098	0x00730EB3	add x29 x6 x7
40	0x0000009C	0x12345537	lui x10 74565
41	0x000000A0	0x20000597	auipc x11 131072
42	0x000000A4	0x00D00023	sb x13 0(x0)
43	0x000000A8	0x00D010A3	sh x13 1(x0)
44	0x000000AC	0x00D02123	sw x13 2(x0)
45	0x000000B0	0x00002083	lw x1 0(x0)
46	0x000000B4	0x00202103	lw x2 2(x0)
47	0x000000B8	0x00101183	lh x3 1(x0)
48	0x000000BC	0x00105203	lhu x4 1(x0)
49	0x000000C0	0x00205283	lhu x5 2(x0)
50	0x000000C4	0x00300303	lb x6 3(x0)
51	0x000000C8	0x00304383	lbu x7 3(x0)
52	0x000000CC	0x00504403	lbu x8 5(x0)
53	0x000000D0	0x00C0006F	jal x0 12
54	0x000000D4	0x00100213	addi x4 x0 1
55	0x000000D8	0x000E0267	jalr x4 x28 0
56	0x000000DC	0x00000013	addi x0 x0 0

上課實作：ModelSim 模擬



上課實作：ModelSim 模擬



上課實作：FPGA 燒錄

- 本次使用記憶體位址 0x0 來儲存上數計數器中計數的數值，每次迴圈使用 LW 載入數值並在累加完後使用 SW 存回記憶體。

- 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。

```
init:
    # 初始化計數器
    sw x0, 0(x0)

loop:
    lw x31, 0(x0)
    call delay

    # 計數器加 1
    addi x31, x31, 1
    sw x31, 0(x0)

    # 無窮迴圈
    j loop

# 延遲副程式
delay:
    # 外層迴圈次數
    li x10, 5000       # 外層迴圈次數 (5000)
outer_loop:
    # 內層迴圈次數
    li x11, 1000      # 內層迴圈次數 (5000)
    nop
    nop
inner_loop:
    addi x11, x11, -1 # 內層迴圈遞減
    nop
    nop
    bnez x11, inner_loop # 如果 x11 不為零，繼續內層迴圈
    addi x10, x10, -1 # 外層迴圈遞減
    nop
    nop
    bnez x10, outer_loop # 如果 x10 不為零，繼續外層迴圈

    ret                # 返回主程式
```

```
0x00000000    0x00002023    sw x0 0(x0)
0x00000004    0x00002F83    lw x31 0(x0)
0x00000008    0x00000317    auipc x6 0
0x0000000C    0x014300E7    jalr x1 x6 20
0x00000010    0x001F8F93    addi x31 x31 1
0x00000014    0x01F02023    sw x31 0(x0)
0x00000018    0xFEDFF06F    jal x0 -20
0x0000001C    0x00001537    lui x10 1
0x00000020    0x38850513    addi x10 x10 904
0x00000024    0x3E800593    addi x11 x0 1000
0x00000028    0x00000013    addi x0 x0 0
0x0000002C    0x00000013    addi x0 x0 0
0x00000030    0xFFF58593    addi x11 x11 -1
0x00000034    0x00000013    addi x0 x0 0
0x00000038    0x00000013    addi x0 x0 0
0x0000003C    0xFE059AE3    bne x11 x0 -12
0x00000040    0xFFF50513    addi x10 x10 -1
0x00000044    0x00000013    addi x0 x0 0
0x00000048    0x00000013    addi x0 x0 0
0x0000004C    0xFC051CE3    bne x10 x0 -40
0x00000050    0x00008067    jalr x0 x1 0
```

THANK YOU