學期實驗大綱

1. MIPS assembly programming
2. FC-DNN design using PyTorch
3. FC-DNN implementation on Zynq
4. Verilog modeling & simulation
5. Master/slave coprocessing on Zynq
6. Mips-Core accelerator
7. Trace-driven cache simulation

Mips-Core accelerator (Hardware based)

助教: 郭帛霖、廖柏宸

授課大綱

- **Lab6:**

  A. Software (11/29)

  1. Zynq與Mips Core的交互方式
  2. 如何在Mips Core上運行FC-DNN
  3. 如何優化C code，讓執行時間變更少

  B. Hardware (12/4)

  1. Mips Core code structure
  2. 如何做Simulation
  3. 新指令的添加方式

實驗目標

- **優化Mnist FC-DNN手寫辨識專案的執行速度：**

1. 新增自定義指令集 (MAC指令d = a * b + c)
2. 完善Pipeline
   a) 新增Pipeline interlock機制 (Stall取代NOP)
   b) 新增Forwarding機制
   c) 新增Branch prediction機制

Outline

- **Mips-Core**
  1. 系統規格
  2. 系統架構
  3. Memory map
  4. 程式說明 (5 stage pipeline rtl code)
  5. Simulation
- **Exercise**
- **Homework**

Mips Core – 系統規格

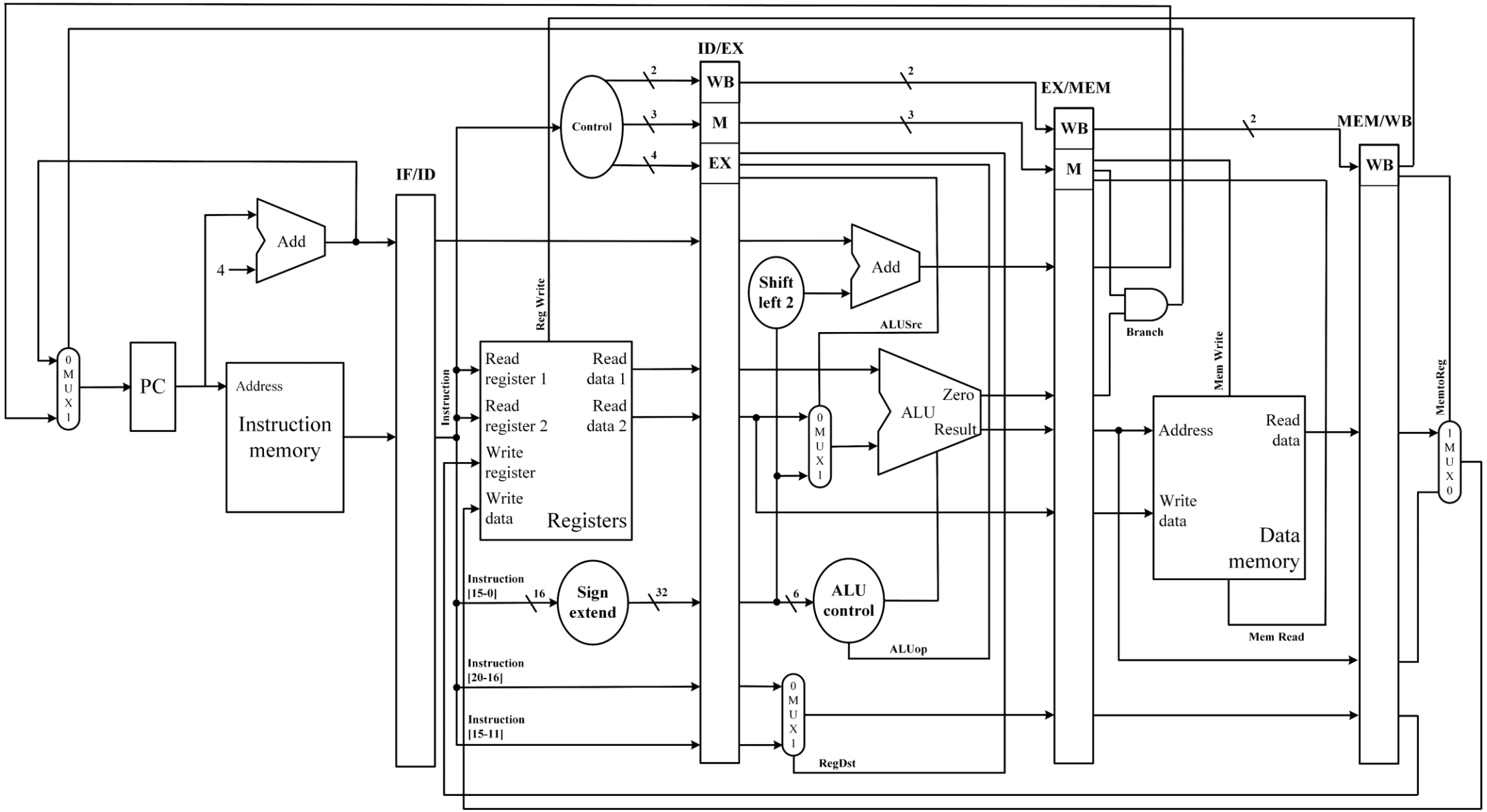
- **Mips-Core：**
  1. **Pipeline:** 5 Stage (IF, ID, EXE, MEM, WB), without forwarding unit & branch prediction
  2. **IM/DM size:** 8KB/8KB (2048 * 32 bits)
  3. **Register:** 32 integer registers & 32 floating point register (IEEE754)
  4. **Instruction support:**
     1. **Integer:**
        1. R type: add, sub, and, or, slt
        2. I type: addi, lw, sw, beq, bne
        3. J type: j
     2. **Floating point:**
        1. R type: add.s, mul.s
        2. I type: lwc1, swc1

Mips Core – Memory map

➢ **Memory map：**

	Address (multiples of 4)
IM (8KB)	0x40000000 (IM[0]) - 0x40001FFC (IM[2047])
DM (8KB)	0x40002000 (DM[0]) - 0x40003FFC (DM[2047])
RF	0x40004000 (RF[0]) - 0x4000407C (RF[31])
MIPS_RSTN	0x40008004

Mips Core – 系統架構

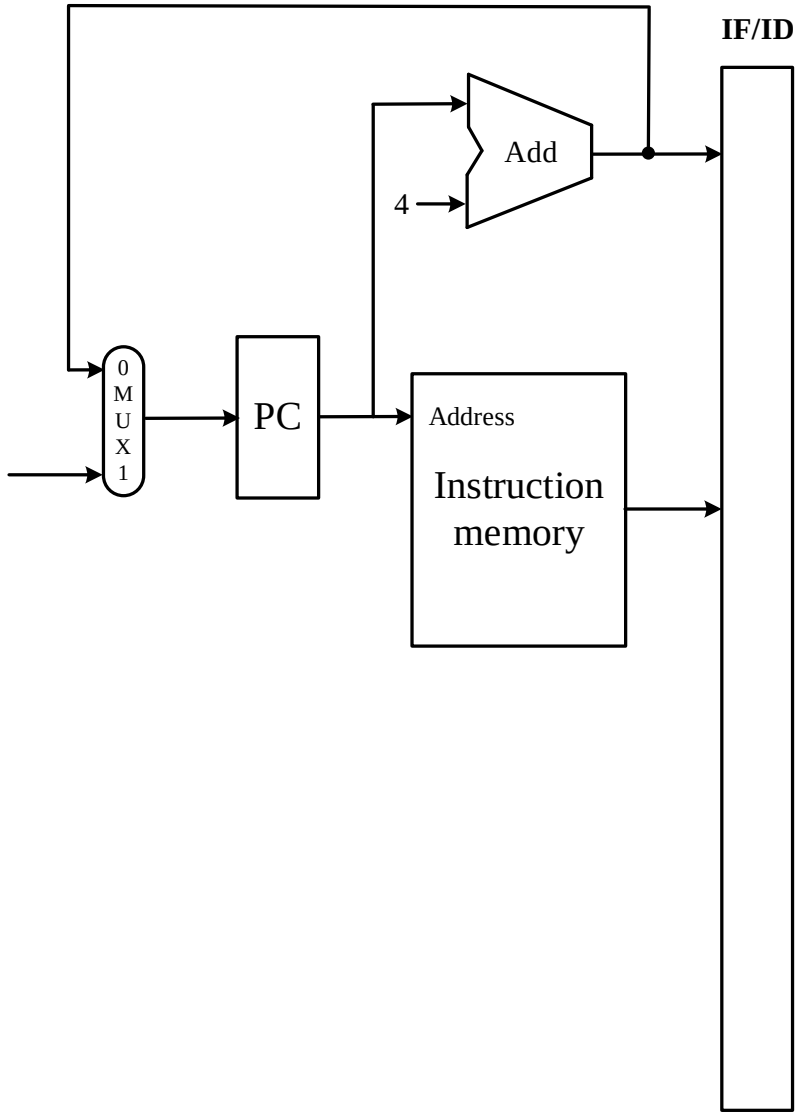


Mips Core – 檔案結構

- ahb_tb
  - cpu_ahb_if
    - ahb_ctrl
    - **cpu_top**
      - **if_pipe** (instruction fetch)
        - **instr_mem** (instruction memory)
      - **id_pipe** (instruction decode)
        - **rf** (register file)
        - **fp_rf** (floating point register file)
        - **idu** (instruction decode unit)
      - **ex_pipe** (execution)
        - **fp_add** (floating point adder)
        - **fp_mul** (floating point multiplier)
      - **mem_pipe** (memory)
        - **data_mem** (data memory)

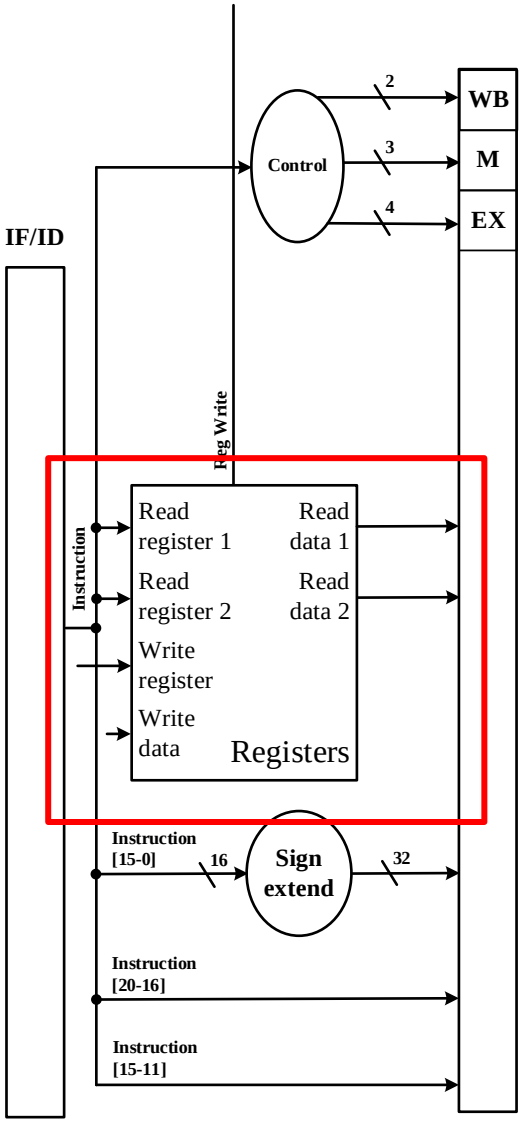
主要修改的地方

程式說明 – Instruction Fetch (取指 + PC計算)



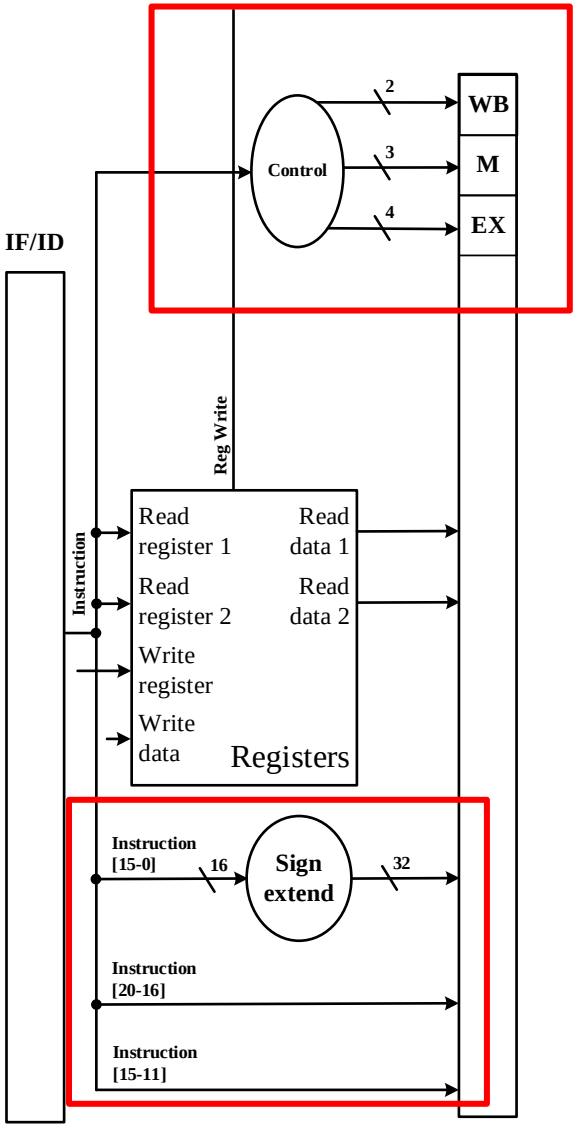

```
always @(*)
begin
    if(!rstn)
        fetch_instr <= 32'd0;
    else
        fetch_instr <= instr_mem_dout;
end

always @(posedge clk or negedge rstn)
begin
    if(!rstn)
        fetch_pc <= 32'd0;
    else
        fetch_pc <= (branch_xm) ? branch_addr_xm :
            ((jump_dx) ? jump_addr : (fetch_pc + 4)) ;
end

sram instr_mem( … );
```

程式說明 – Instruction Decode (RF讀取)



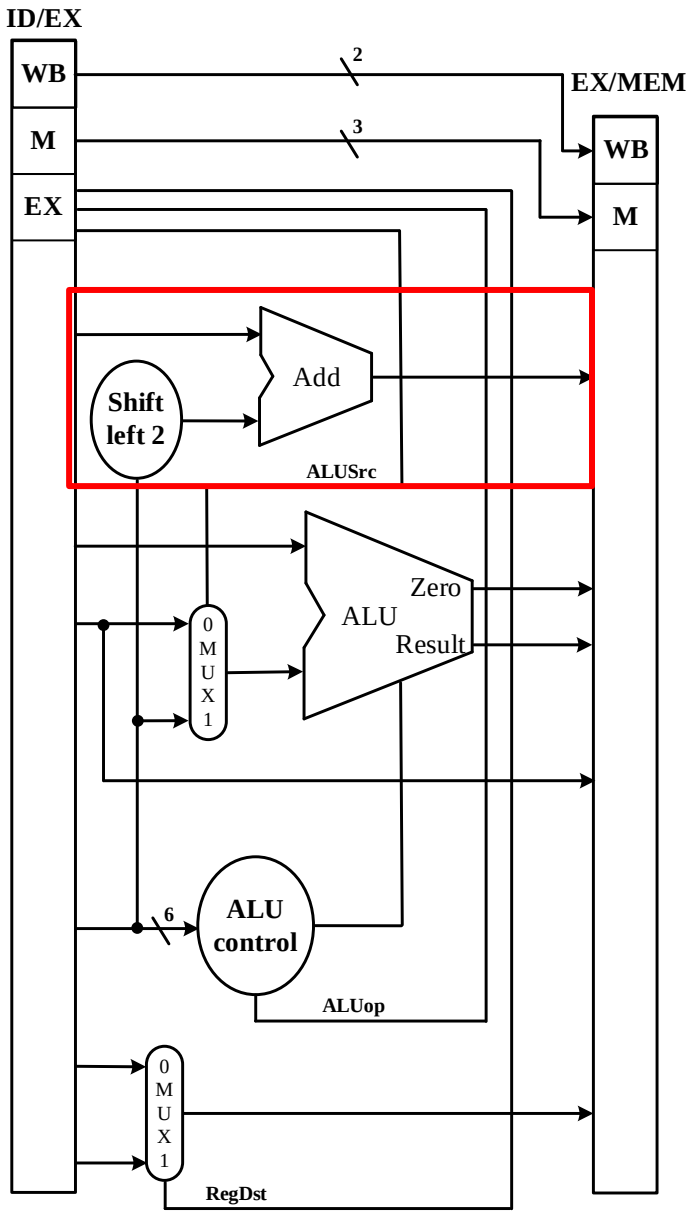

```
reg             [31:0] REG_I [0:31];

//read data to rs
always @* begin
    rs_data = REG_I[rs_addr];
end

//read data to rt
always @* begin
    rt_data = REG_I[rt_addr];
end
```

程式說明 – Instruction Decode (Control訊號解碼)



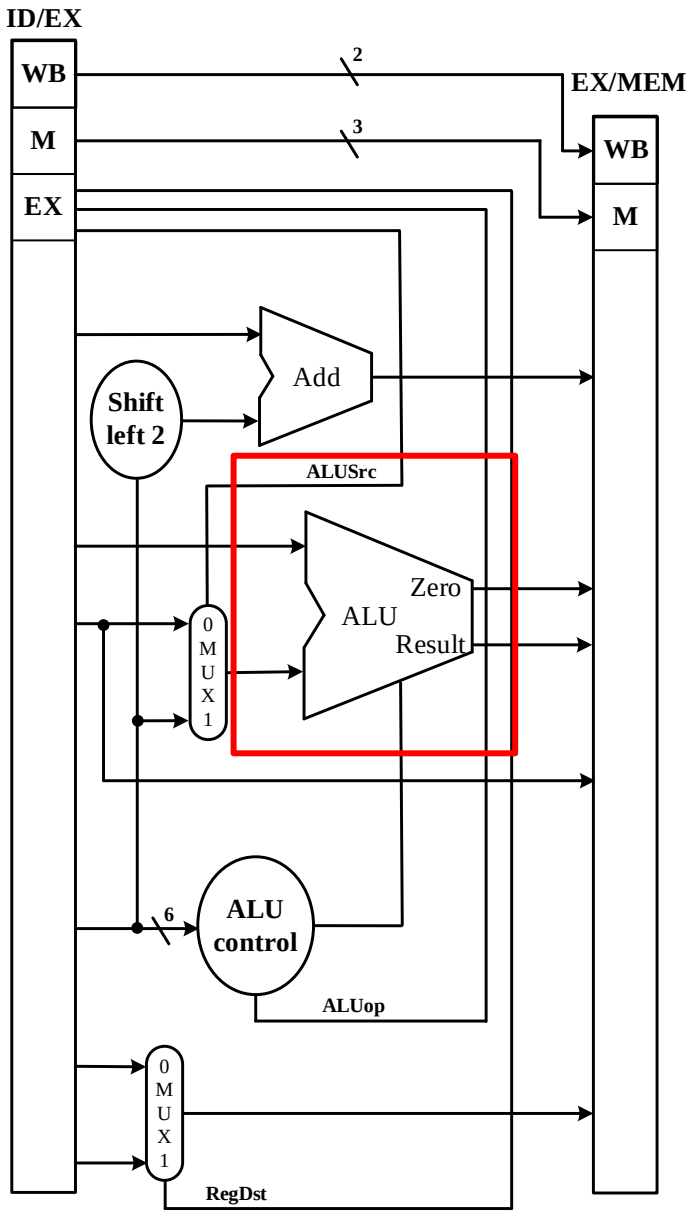

```
case( instr[31:26] )
        R_TYPE: …
        ADDI: …
        LW: begin
                alu_src2            <= {{16{instr[15]}}, instr[15:0]};
                rd_addr_dx          <= instr[20:16];
                mem_to_reg_dx       <= 1'b1;
                reg_write_dx        <= 1'b1;
                mem_read_dx         <= 1'b1;
                mem_write_dx        <= 1'b0;
                branch_dx           <= 1'b0;
                alu_ctrl            <= 4'd2;
                fp_operation_dx     <= 1’b0;
            end
        …
        …
        F_R_TYPE:
endcase
```

程式說明 – Execution (Branch位址計算)



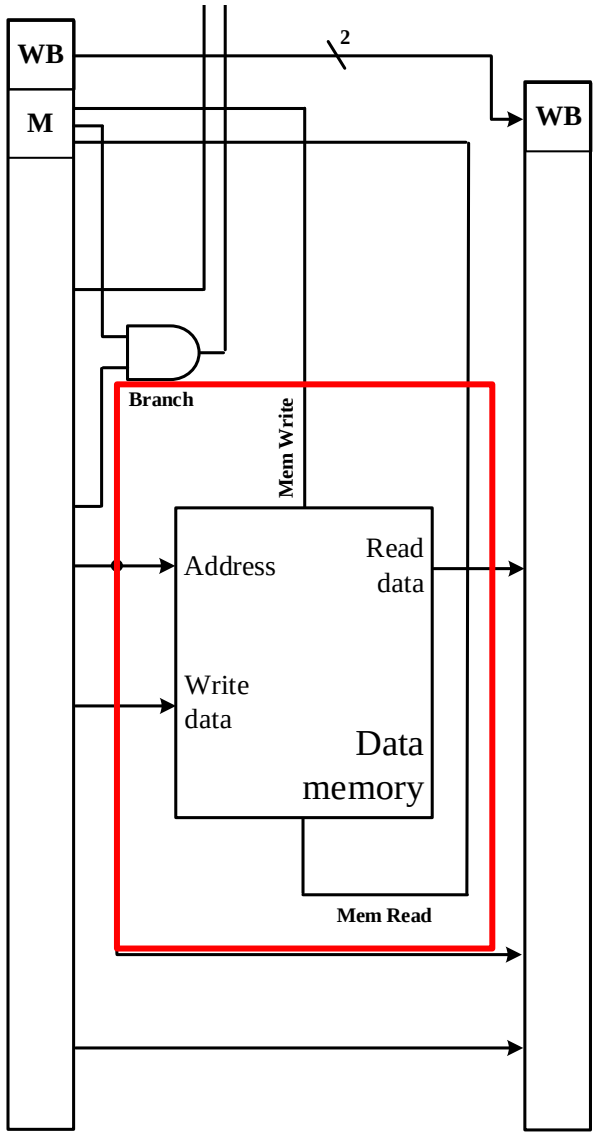

```
always @(posedge clk or negedge rstn)
begin
    if(!rstn) begin
        …
        end else begin
        …
        ...
        branch_addr_xm    <= pc_dx + {{15{imm[15]}}, imm, 2'b0};
    end
end
```

程式說明 – Execution (ALU邏輯)



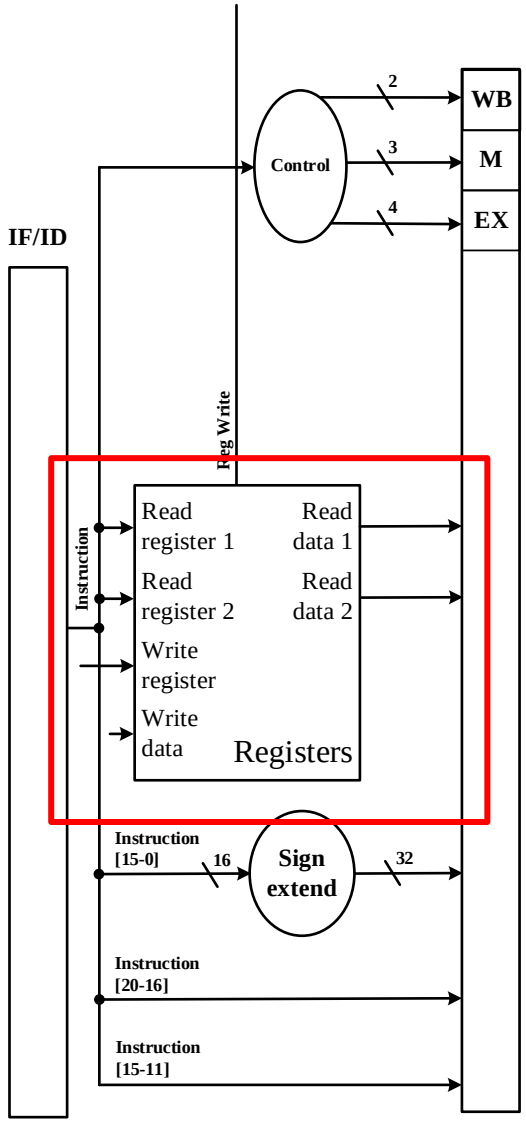

```
case(alu_ctrl)
        4'd0:
            alu_out_xm <= alu_src1 & alu_src2;
        4'd1:
            alu_out_xm <= alu_src1 | alu_src2;
        4'd2:
            alu_out_xm <= alu_src1 + alu_src2;
        4'd6:
            alu_out_xm <= alu_src1 - alu_src2;
        4'd7:
            alu_out_xm <= alu_src1 < alu_src2 ? 1 : 0;
        default:
            alu_out_xm <= alu_out_xm;
endcase
```

程式說明 – Memory (Data Memory寫入、讀取)




```
//write data
always @(*) begin
        …
        data_mem_addr <= alu_out_xm[12:2];
        data_mem_din   <= mem_data_xm;
        data_mem_we    <= mem_write_xm;
        …
end
//read data
assign mem_data_to_reg = mem_read_xm ? data_mem_dout :
                                 mem_data_to_reg_tmp;
sram data_mem(
    .addra(data_mem_addr),
    .clka(clk),
    .dina(data_mem_din),
    .douta(data_mem_dout),
    .ena(1'b1),
    .wea(data_mem_we)
);
```

程式說明 – Write back(RF寫入)




```
reg             [31:0] REG_I [0:31];

//write back
always @(posedge clk or negedge rstn)
…
if(reg_write_mw && !fp_operation_mw && rd_addr != 5'd0) begin
   REG_I[rd_addr] <= (mem_to_reg_mw) ? mem_data_to_reg : alu_out_mw;
end
```

Simulation – 測試資料產生流程 (Machine code)

- 撰寫Mips assembly code
- 在發生Hazard的指令間插入NOP (add $0, $0, $0)
- 用MARS輸出Machine code，並複製到im_data/im.txt中
- beq與j指令的Machine code需手動轉換 (Memory base與MARS不同)

```
acc:
        beq $5, $4, exit        10a40018//IM[10]
        ...
        j acc                   0800000a//IM[32]
        ...
exit:
        swc1 $f4, 4($4)         e4840004//IM[35]
```

offset: 35 – 12 + 1 = 24 = 0x18

- 儲存im_data/im.txt檔

Simulation – Testbench使用與修改方式

- **ahb_tb.v**

1. **寫入 Mips相對應的Memory map位址 (IM or DM…)**

```
ahb_write(addr, data);
```

2. **讀取 Mips相對應的Memory map位址**

```
ahb_read(addr, data);
```

3. **若是im.txt有修改，則必須修改讀取的指令數**

```
//write im
for(i = 0; i < 40; i = i + 1) begin
    ahb_write(im_addr, instr[i]);
    im_addr = im_addr + 32'd4;
end
```

Simulation – 執行結果

- Terminal中會顯示出暫存器與DM的值以及計算結果
- 下圖為模擬的結果 (memory_dump可將Memory中的值顯示出來)

```
R00-R07: 00000000 00000000 00000000 00000000 00001ff8 00001ff8 000004d2 00000000
R08-R15: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
R16-R23: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
R24-R31: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

F00-F07: 00000000 3f800000 3f800000 3f800000 447fc000 00000000 00000000 00000000
F08-F15: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
F16-F23: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
F24-F31: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

2024   : 3f800000 3f800000 3f800000 3f800000 3f800000 3f800000 3f800000 3f800000
2032   : 3f800000 3f800000 3f800000 3f800000 3f800000 3f800000 3f800000 3f800000
2040   : 3f800000 3f800000 3f800000 3f800000 3f800000 3f800000 3f800000 447fc000

ans: 447fc000
```

一般暫存器

浮點數暫存器(IEEE754)

Data Memory

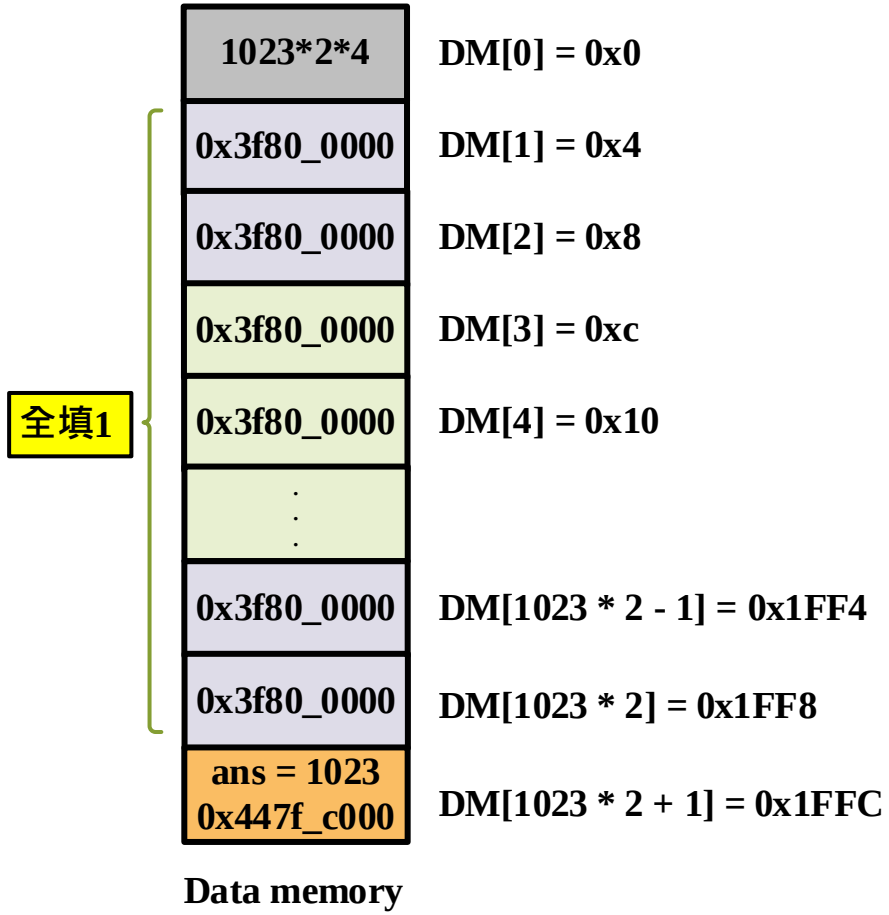
計算結果

Simulation – FC-DNN單顆神經元計算驗證

- 將DM的1~2046，全部填1 (IEEE754 = 0x3f80_0000)
- 計算公式 (累乘加)
  - 1*1 + 1*1 + … + 1*1 = 1023 (IEEE754 = 0x447f_c000)
- 確認輸出是否為1023 (IEEE754 = 0x447f_c000)

```
2024   : 3f800000 3f800000 3f800000 3f800000 3f800000 3f800000 3f800000 3f800000
2032   : 3f800000 3f800000 3f800000 3f800000 3f800000 3f800000 3f800000 3f800000
2040   : 3f800000 3f800000 3f800000 3f800000 3f800000 3f800000 3f800000 447fc000

ans: 447fc000
```

▲ Testbench模擬結果



IEEE754 base converter
https://baseconvert.com/ieee-754-floating-point

Exercise - 新增MUL指令

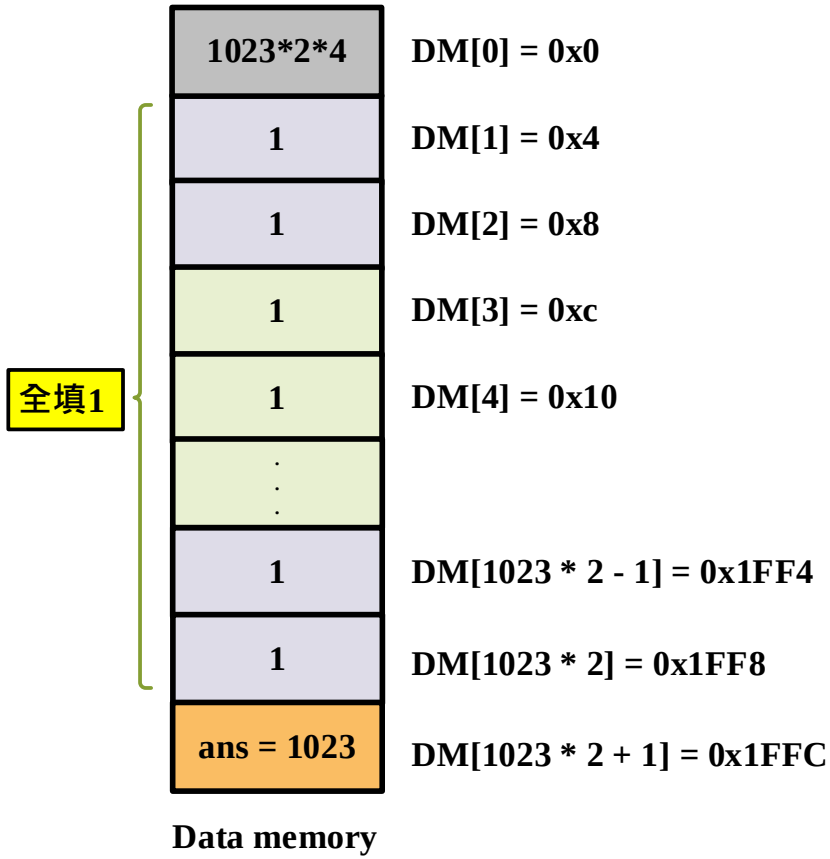
- **說明：**
  1. 計算方式同單顆神經元的FC-DNN，差別為計算整數而非浮點數

31 – 26	25 – 21	20 – 16	15 – 11	10 – 6	5 – 0
SPECIAL2 011100	rs	rt	rd	00000	MUL 000010
6	5	5	5	5	6

▲ MUL指令格式

- **預期結果**
  1. Terminal中會顯示整數乘加後的整數運算結果

     ans: 000003ff (1023)



Exercise - 新增MUL指令

- **指令添加方式(與R-type指令雷同)：**
  - **修改id_dcu.v**
    1. 在 (case(instr[31:26]))，添加SPETIAL2 OP code 6’b011100的解碼
    2. 設置控制訊號

```
alu_src2          //ALU SRC2的來源
rd_addr_dx        //RD的位址
mem_to_reg_dx     //讀memory是否需要寫入暫存器
reg_write_dx      //是否需要寫入暫存器
mem_read_dx       //是否要讀memory
mem_write_dx      //是否要寫memory
branch_dx         //是否為branch指令
fp_operation_dx   //是否為浮點運算指令
```

    3. 添加case(instr[5:0])判斷是否為MUL指令，是的話 alu_ctrl <= 4'd3;

Exercise - 新增MUL指令

◆ **修改ex_pipe.v**

1. 在alu_ctrl解碼處新增MUL運算

```
case(alu_ctrl)
    4'd0:
        alu_out_xm <= alu_src1 & alu_src2;
    4'd1:
        alu_out_xm <= alu_src1 | alu_src2;
    4'd2:
        alu_out_xm <= alu_src1 + alu_src2;
    4'd3:
        alu_out_xm <= alu_src1 * alu_src2;
    4'd6:
        alu_out_xm <= alu_src1 - alu_src2;
    4'd7:
        alu_out_xm <= alu_src1 < alu_src2 ? 1 : 0;
    default:
        alu_out_xm <= alu_out_xm;
endcase
```

Exercise - 新增MUL指令

- **Simulation：**
  1. **編譯：**

     iverilog –o dio ahb_tb.v
  2. **執行：**

     vvp dio
  3. **查看波型：**

     gtkwave cpu_hw.vcd

- **驗收方式：**
  1. 在Terminal顯示出累加結果ans: 000003ff ，請助教檢查

Homework

- **作業內容：**

1. **優化Mnist FC-DNN手寫辨識專案的執行速度，可能的優化方向為(盡力做就好)：**
   a) 新增自定義指令集 (MAC指令d = a * b + c)
   b) 完善Pipeline
      1) 新增Pipeline interlock機制 (Stall取代NOP)
      2) 新增Forwarding機制
      3) 新增Branch prediction機制
2. **將實作結果撰寫成5頁內的A4紙本報告，內容可自由發揮**

Homework

- **紙本報告內容建議：**
  1. 簡述Mnist FC-DNN
  2. 執行FC-DNN計算的效能瓶頸 (Mips core)
  3. 優化方式 (硬體、軟體)
  4. 優化結果 (Perf數據前後對比)
  5. 結論與心得
- **作業繳交方式：**
  1. 實體Demo與繳交紙本報告
- **Office hour：**
  1. 時間：每週三、四 下午2:00 – 5:00
  2. 地點：資工館 501A實驗室

Homework

- **Demo及報告繳交時間：**

  2023/12/27 (三) 下午2:00 – 5:00 (資工館 501A實驗室)

  ※ 當天有課請提早通知助教 ，逾時不接受補demo

- **Demo內容：**

  優化後的Perf執行時間

- Demo注意事項：

  1. 請攜帶紙本報告繳交
  2. 需歸還Zedboard，並請記得帶借用單

APPENDIX

浮點指令格式 – lwc1、swc1

Load Word to Floating Point **LWC1**

31–26	25–21	20–16	15–0
LWC1 110001	base	rt	offset
6	5	5	16

Format: `LWC1 ft, offset(base)` **MIPS32 (MIPS I)**

Store Word from Floating Point **SWC1**

31–26	25–21	20–16	15–0
SWC1 111001	base	ft	offset
6	5	5	16

Format: `SWC1 ft, offset(base)` **MIPS32 (MIPS I)**

浮點指令格式 – add.s、mul.s

Floating Point Add **ADD.fmt**

31–26	25–21	20–16	15–11	10–6	5–0
COP1 010001	fmt	ft	fs	fd	ADD 000000
6	5	5	5	5	6

Format: `ADD.S fd, fs, ft` **MIPS32 (MIPS I)**

Floating Point Multiply **MUL.fmt**

31–26	25–21	20–16	15–11	10–6	5–0
COP1 010001	fmt	ft	fs	fd	MUL 000010
6	5	5	5	5	6

Format: `MUL.S fd, fs, ft` **MIPS32 (MIPS I)**

教學影片

- **從零開始創建Vivado專案**

  https://youtu.be/tggUh-G34hU

- **Vivado快速包裝IP**

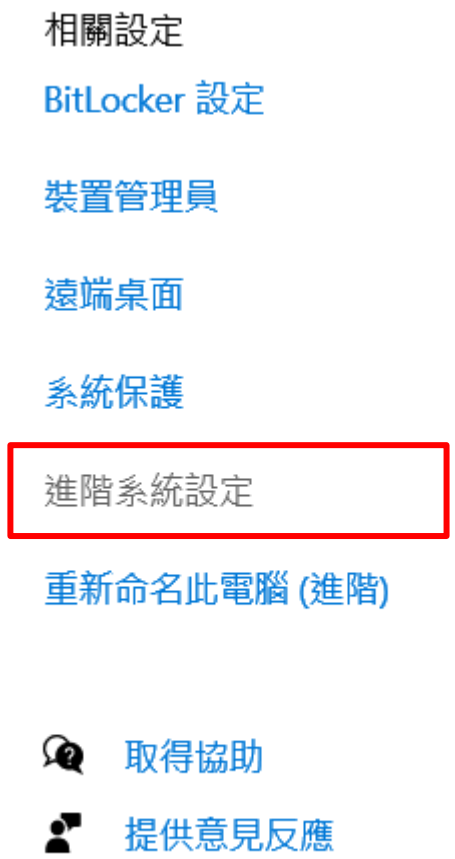
  https://youtu.be/bBvuOrZBHaM

iverilog安裝

- 雙擊iverilog-v12-20220611-x64_setup.exe 並進行安裝
- 開始icon點擊右鍵，並選取系統

iverilog安裝

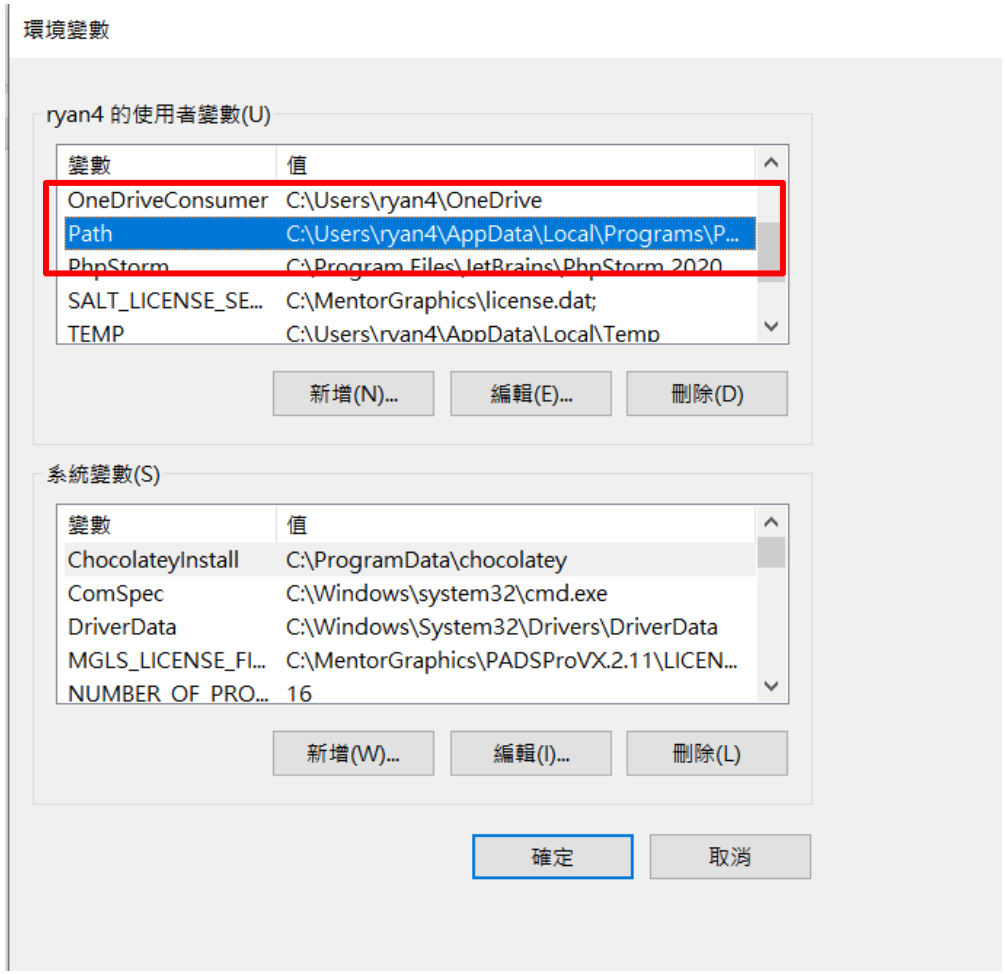
➢ 點擊進階系統設定



➢ 選取環境變數

iverilog安裝

- 雙擊Path



- 新增這兩個檔案路徑，點擊確定