

Tiny MIPS on FPGA

COLAB3

Outline

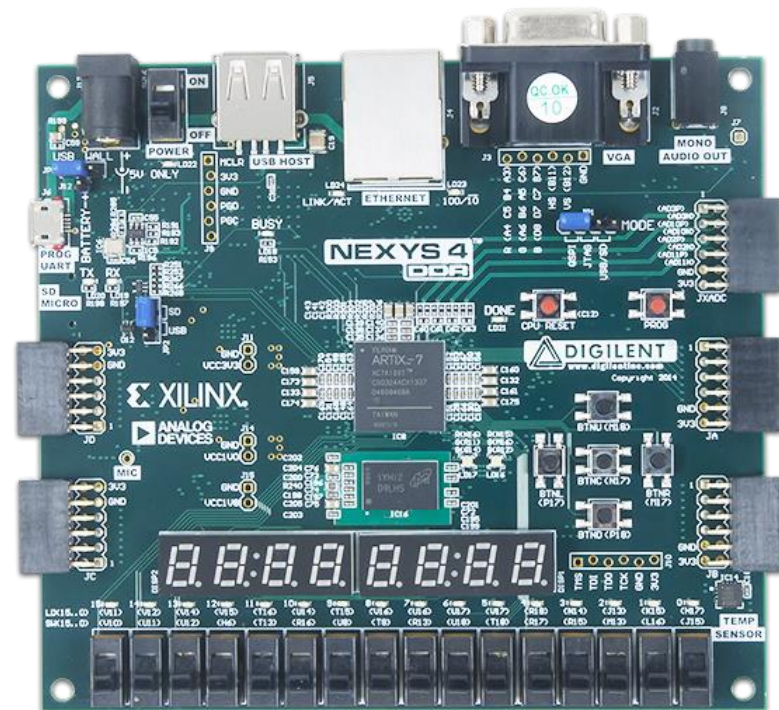
- 實驗目的
- 實驗環境
- MIPS介紹
- 課堂練習
- 回家作業
- 評分標準

實驗目的

- 在前面兩堂實驗課，相信同學已經學會如何使用 Verilog 撰寫組合語言，並使用 FPGA 進行功能的驗證
- 本堂實驗課將運用前面所學，以 Verilog 實作精簡指令集（RISC）處理器中的 MIPS CPU，了解各種指令在 RISC 的運行方式

實驗環境

- 在本實驗中同學將使用Xilinx的系統單晶片設計套件(Vivado Design Suite)及Nexys4 DDR FPGA進行驗證



Nexys-4 DDR FPGA

- 請同學珍惜使用
- 本堂課程結束前請填寫並繳回財產借用單

Nexys4-DDR Artix-7 FPGA 開發板 學習板 Xilinx XUP Digilent



◎ 支援付款快手

商品編號： 11090613953389

[檢舉](#)

商品備註

- 物品狀況：全新
- 物品所在地：台灣-台北市
- 上架時間：2009-06-13 13:15:39
- 物品開始價格：\$14,200元
- 可能會提前結束販售

直購價：**\$14,200**

數量： 庫存 1 件

[加入購物車](#)

[直接購買](#)

[加入追蹤](#)

已賣數量：**4**

付款方式：銀行或郵局轉帳
郵局無摺存款
貨到付款

運送方式：郵寄寄送 **40 元**
宅配/快遞 **40 元**
貨到付款 **80 元**

[合併運費規則](#)

賣家資訊

[加入最愛](#)

[lilan-1 \(18072\)](#)

[賣場首頁](#)



全部商品：**13893**

評價分數：**18072** [查看](#)

[關於我](#)

喜歡這商品嗎?按讚及+1推薦給你的朋友吧!

[分享](#)

[讚 0](#)

商品價格與運費更新時間: 2016-06-13 20:18:11

MIPS介紹

- MIPS（Microprocessor without Interlocked Pipeline Stages）是一種採取精簡指令集的指令集架構
- MIPS被廣泛應用在許多電子產品、網路裝置、個人娛樂裝置與商業裝置上
- 歷史介紹
 - 1980年代為精簡指令集（RISC: Reduced Instruction Set Computer）對以x86為代表的複雜指令集（CISC: Complex Instruction Set Computer）發起挑戰的時間點，而MIPS為RISC陣營中最早的挑戰者之一
 - 1981年，由當時史丹佛大學的教授－John Hennessy領導團隊，完成了第一個MIPS架構處理器
 - 1984年，John Hennessy創立MIPS科技公司，並在成立的第二年推出第一個晶片設計R2000
 - 1988年，MIPS推出了R3000。這款產品很快大獲成功，銷售超過百萬顆
 - 2021年，全球三大晶片架構之一MIPS走入歷史。MIPS科技公司放棄繼續設計MIPS架構，全心投入RISC-V陣營



▲ John LeRoy Hennessy



▲ MIPS科技公司



▲ 當初SONY的PS也使用了R3000晶片

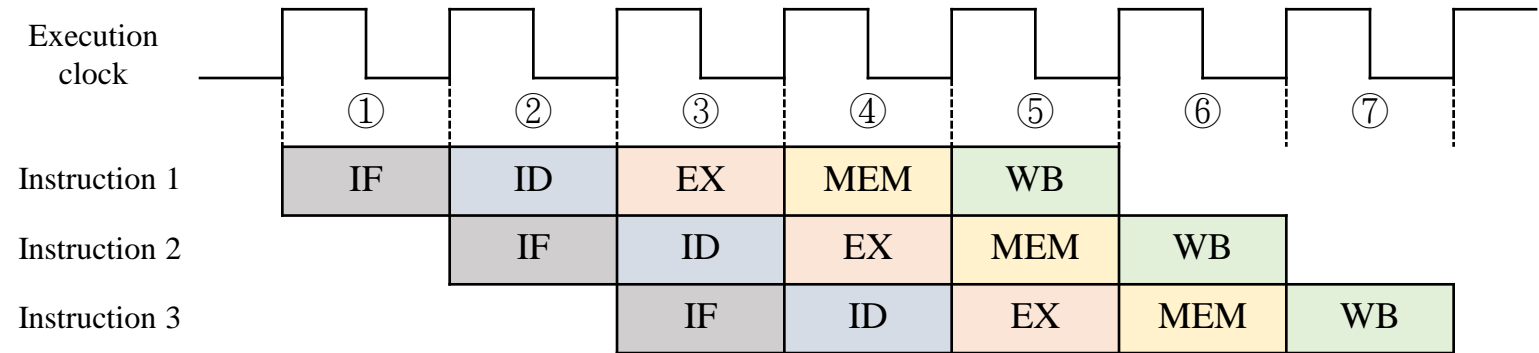
流水線介紹

- RISC架構下的指令data path可拆解成5 stage完成，並於pipeline中執行
- 每個stage完成的動作可視為一組micro-operation，各有其對應的micro-architecture

IF	ID	EX	MEM	WB
----	----	----	-----	----

stage縮寫	全名	作用
IF	Instruction fetch	取得指令
ID	Instruction decode & register read	指令解碼和讀取 register
EX	Execute or address calculation	執行或位址運算
MEM	Memory access	記憶體讀取
WB	Write back to registers	寫回register

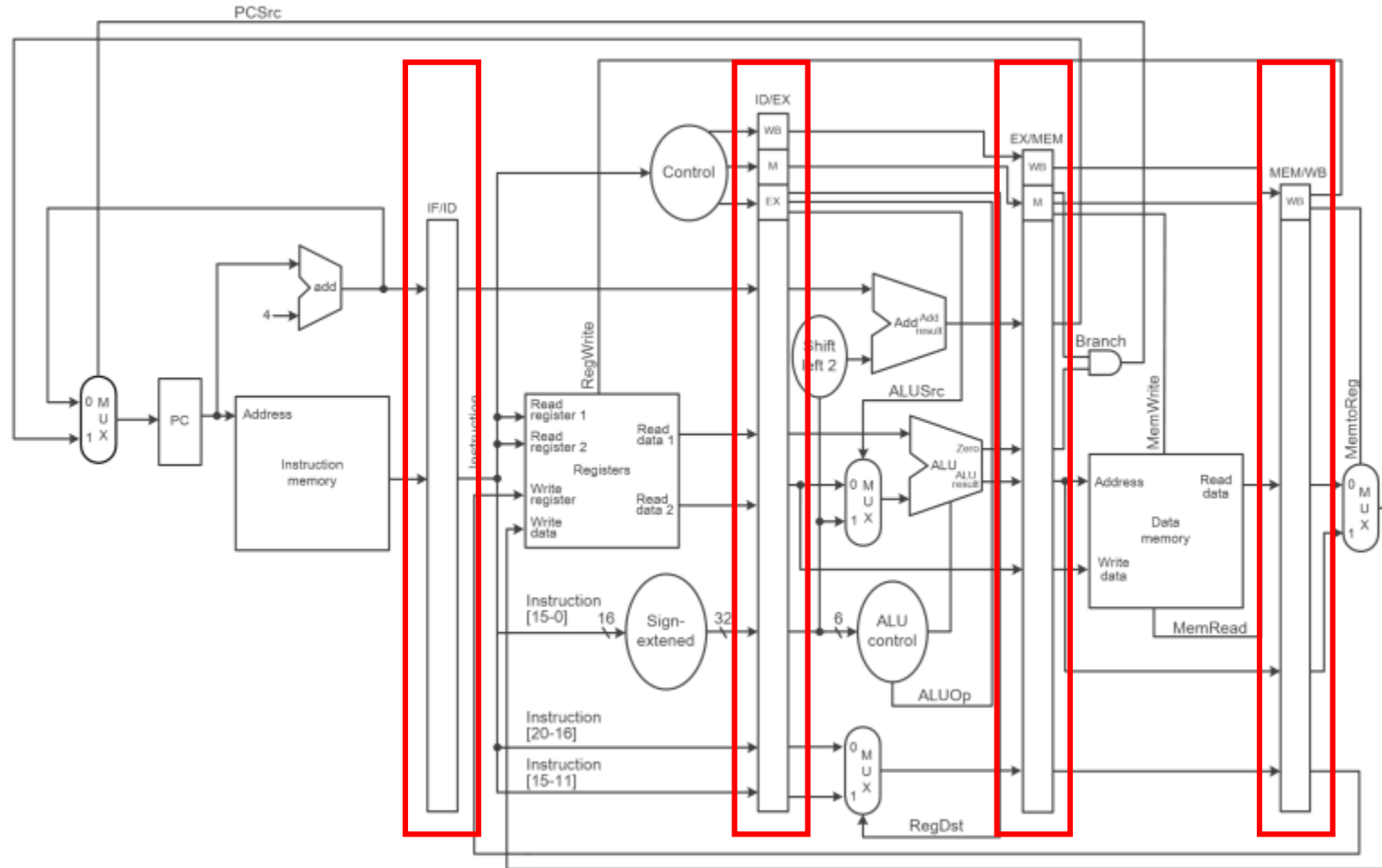
- 在同個clock下，允許多條指令在不同的stage中執行



流水線介紹 (cont'd)

- 管線暫存器（pipeline register）可保存不同stage執行的值
- 基本RISC pipeline架構下的管線暫存器有四個
 1. IF/ID
 2. ID/EX
 3. EX/MEM
 4. MEM/WB
- 各stage之I/O相關性：管線在執行過程中，會將訊號於管線暫存器中逐級傳送，故上一級的output通常為下一級的input
- 定義好各stage之input與output，即完成初步pipeline硬體架構規劃

Pipeline Structure

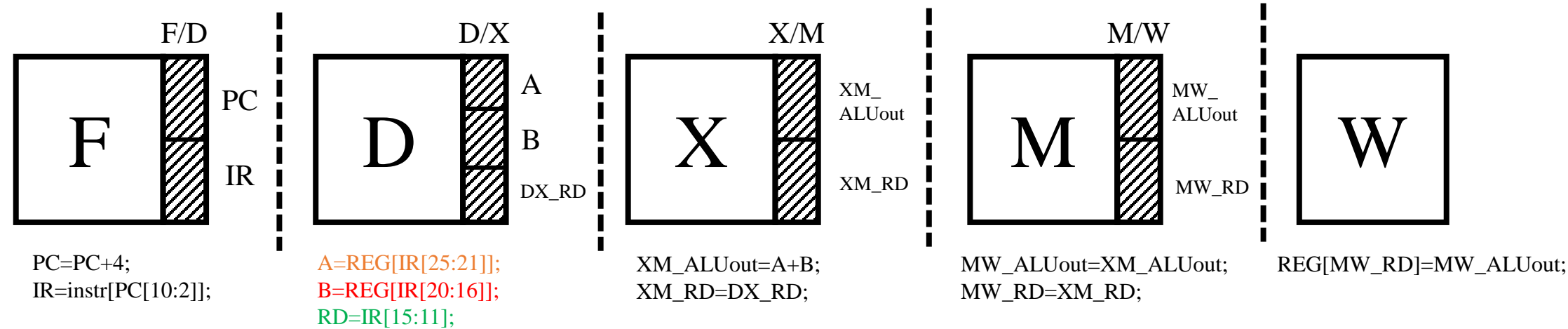


pipeline register

Pipeline Design (EX. add)

add rd, rs, rt \Rightarrow $\text{reg(rd)} = \text{reg(rs)} + \text{reg(rt)}$

Description	Add the value of two registers and stores the result in a register
Operation	$\$d = \$s + \$t$; advance_pc (4);
Syntax	add \$d, \$s, \$t
Encoding	0000 00ss ssst tttt dddd d000 0010 0000



* IR: 32-bit instruction

Modularization (EX. add)

F/D

```
module INSTRUCTION_FETCH(  
    clk,  
    rst,  
  
    PC,  
    IR  
);  
  
input clk, rst;  
output reg [31:0] PC, IR;  
  
// instructions  
reg [31:0] instr [127:0];  
  
always @(posedge clk)  
begin  
    if(rst)begin  
        PC <= 32'd0;  
        IR <= 32'd0;  
    end else begin  
        PC <= PC+4;  
        IR <= instr[PC[10:2]];  
    end  
end  
endmodule
```

D/X

```
module INSTRUCTION_DECODE(  
    clk,  
    rst,  
    PC,  
    IR,  
    MW_RD,  
    MW_ALUout,  
  
    A, B, RD  
);  
  
input clk, rst;  
input [31:0] IR, PC, MW_ALUout;  
input [4:0] MW_RD;  
  
output reg [31:0] A, B;  
output reg [4:0] RD;  
  
// register files  
reg [31:0] REG [0:31];  
  
always @(posedge clk)  
    REG[MW_RD] <= MW_ALUout;  
  
always @(posedge clk)  
begin  
    A <= REG[IR[25:21]];  
    B <= REG[IR[20:16]];  
    RD <= IR[15:11];  
end  
endmodule
```

X/M

```
module EXECUTION(  
    clk,  
    rst,  
    A,  
    B,  
    DX_RD,  
  
    ALUout,  
    XM_RD,  
);  
  
input clk, rst;  
input [31:0] A, B;  
input [4:0] DX_RD;  
  
output reg [31:0] ALUout;  
output reg [4:0] XM_RD;  
  
always @(posedge clk)  
begin  
    ALUout <= A + B;  
    XM_RD <= DX_RD;  
end  
endmodule
```

M/W

```
module MEMORY(  
    clk,  
    rst,  
    ALUout,  
    XM_RD,  
  
    MW_ALUout,  
    MW_RD,  
);  
  
input clk, rst;  
input [31:0] ALUout;  
input [4:0] XM_RD;  
  
output reg [31:0] MW_ALUout;  
output reg [4:0] MW_RD;  
  
// data memory  
reg [31:0] Mem [0:127];  
  
always @(posedge clk)  
begin  
    MW_ALUout <= ALUout;  
    MW_RD <= XM_RD;  
end  
endmodule
```

(WB=ID)

CPU.v

```
`timescale 1ns/1ps

`include "INSTRUCTION_FETCH.v"
`include "INSTRUCTION_DECODE.v"
`include "EXECUTION.v"
`include "MEMORY.v"

module CPU(
    clk,
    rst
);
input clk, rst;
/*===== Wire =====*/
// INSTRUCTION_FETCH wires
wire [31:0] FD_PC, FD_IR;
// INSTRUCTION_DECODE wires
wire [31:0] A, B;
wire [4:0] DX_RD;
wire [2:0] ALUctr;
// EXECUTION wires
wire [31:0] XM_ALUout;
wire [4:0] XM_RD;
// DATA_MEMORY wires
wire [31:0] MW_ALUout;
wire [4:0] MW_RD;

/*===== INSTRUCTION_FETCH =====*/
INSTRUCTION_FETCH IF(
    .clk(clk),
    .rst(rst),

    .PC(FD_PC),
    .IR(FD_IR)
);
```

宣告各Stage之前傳值所需要的連接線

接續

```
/*===== INSTRUCTION_DECODE =====*/
INSTRUCTION_DECODE ID(
    .clk(clk),
    .rst(rst),
    .PC(FD_PC),
    .IR(FD_IR),
    .MW_RD(MW_RD),
    .MW_ALUout(MW_ALUout),

    .A(A),
    .B(B),
    .RD(DX_RD),
    .ALUctr(ALUctr)
);

/*===== EXECUTION =====*/
EXECUTION EXE(
    .clk(clk),
    .rst(rst),
    .A(A),
    .B(B),
    .DX_RD(DX_RD),
    .ALUctr(ALUctr),

    .ALUout(XM_ALUout),
    .XM_RD(XM_RD)
);

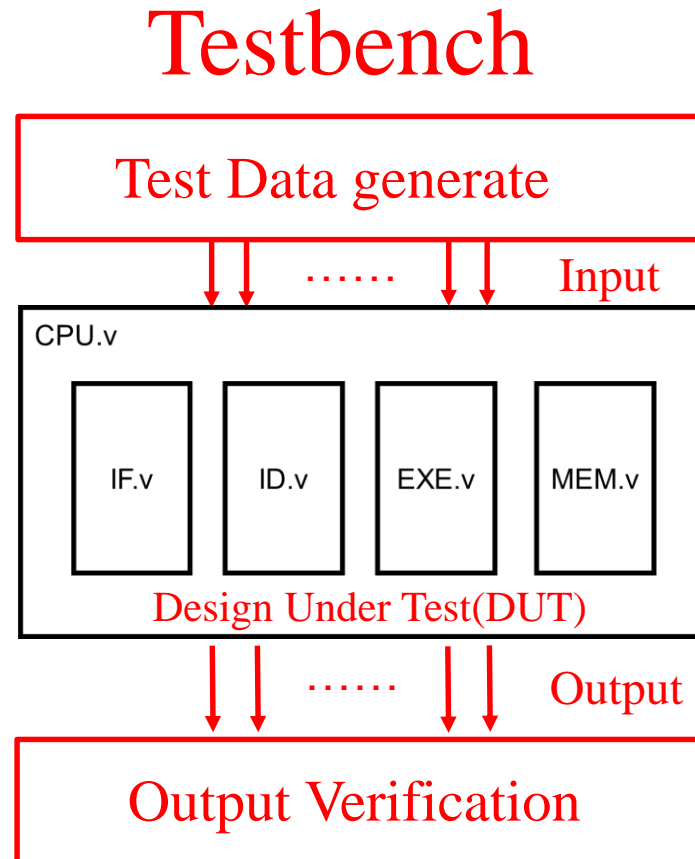
/*===== DATA_MEMORY =====*/
MEMORY MEM(
    .clk(clk),
    .rst(rst),
    .ALUout(XM_ALUout),
    .XM_RD(XM_RD),

    .MW_ALUout(MW_ALUout),
    .MW_RD(MW_RD)
);

endmodule
```

RISC Processor RTL Simulation

- 利用 structural modeling 的方式設計RTL並進行模擬，將需要驗證的設計 (Design under verification, DUV) 包在 top-module 之下，並以 high-level 與法產生測試 pattern 以及觀察結果



testbench.v

```
// Instruction DM initialilation
initial
begin
/*===== write down your program =====*/
cpu.IF.instruction[ 0] = 32'b000000 00001 00010 00011 00000 100000; //add $3, $1, $2      $3 = 1 + 2 = 3
cpu.IF.instruction[ 1] = 32'b000000 00000 00000 00000 00000 100000; //NOP(add $0, $0, $0)
cpu.IF.instruction[ 2] = 32'b000000 00000 00000 00000 00000 100000; //NOP(add $0, $0, $0)
cpu.IF.instruction[ 3] = 32'b000000 00000 00000 00000 00000 100000; //NOP(add $0, $0, $0)
for (i=4; i<128; i=i+1) cpu.IF.instruction[ i] = 32'b000000 00000 00000 00000 00000 100000; //NOP(add $0, $0, $0)
cpu.IF.PC = 0;
end
```

輸入code的機械碼

※ 插入NOP處理Hazard問題

因為CPU沒有做任何處理hazard的硬體，故只能透過插入NOP指令或是調整指令順序的方式節省cycle數。

什麼時候插入NOP?

EX. add \$3, \$1, \$2

add \$5, \$3, \$4

第一行的\$1+\$2還未寫回\$3，故下一行的\$3內並非預期的值，故插入3個NOP等待

testbench.v

顯示所有register
及data memory內
容

```
//display all Register value and Data memory content
always @(posedge Clk) begin
    cycles <= cycles + 1;
    if (cycles == `INSTRUCTION_NUMBERS) $finish; // Finish when excute the 24-th instruction (End label).
    $display("PC: %d cycles: %d", cpu.FD_PC>>2 , cycles);
    $display(" R00-R07: %08x %08x %08x %08x %08x %08x %08x %08x", cpu.ID.REG[0], cpu.ID.REG[1], cpu.ID.REG[2], cpu.ID.REG[3],cpu.ID.REG[4],cpu.ID.REG[5],cpu.ID.REG[6],cpu.ID.REG[7]);
    $display(" R08-R15: %08x %08x %08x %08x %08x %08x %08x %08x", cpu.ID.REG[8], cpu.ID.REG[9], cpu.ID.REG[10], cpu.ID.REG[11],cpu.ID.REG[12],cpu.ID.REG[13],cpu.ID.REG[14],cpu.ID.REG[15]);
    $display(" R16-R23: %08x %08x %08x %08x %08x %08x %08x %08x", cpu.ID.REG[16], cpu.ID.REG[17], cpu.ID.REG[18], cpu.ID.REG[19],cpu.ID.REG[20],cpu.ID.REG[21],cpu.ID.REG[22],cpu.ID.REG[23]);
    $display(" R24-R31: %08x %08x %08x %08x %08x %08x %08x %08x", cpu.ID.REG[24], cpu.ID.REG[25], cpu.ID.REG[26], cpu.ID.REG[27],cpu.ID.REG[28],cpu.ID.REG[29],cpu.ID.REG[30],cpu.ID.REG[31]);
    $display(" 0x00 : %08x %08x %08x %08x %08x %08x %08x %08x", cpu.MEM.DM[0],cpu.MEM.DM[1],cpu.MEM.DM[2],cpu.MEM.DM[3],cpu.MEM.DM[4],cpu.MEM.DM[5],cpu.MEM.DM[6],cpu.MEM.DM[7]);
    $display(" 0x08 : %08x %08x %08x %08x %08x %08x %08x %08x", cpu.MEM.DM[8],cpu.MEM.DM[9],cpu.MEM.DM[10],cpu.MEM.DM[11],cpu.MEM.DM[12],cpu.MEM.DM[13],cpu.MEM.DM[14],cpu.MEM.DM[15]);
end
```

```
//generate wave file, it can use gtkwave to display
initial begin
    $dumpfile("cpu_hw.vcd");
    $dumpvars;
end
endmodule
```

產生波形檔

testbench輸出結果說明

目前執行cycle數

```
PC:      47 cycles: 47
R00-R07: 00000000 00000001 00000002 00000003 00000000 00000000 00000000 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
0x00    : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x08    : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
PC:      48 cycles: 48
R00-R07: 00000000 00000001 00000002 00000003 00000000 00000000 00000000 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
0x00    : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x08    : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
PC:      49 cycles: 49
R00-R07: 00000000 00000001 00000002 00000003 00000000 00000000 00000000 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
0x00    : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x08    : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

register (R00~R31) 內數值

data memory(0xx0~0x16)
內數值

Register 以及 Memory的初始化

INSTRUCTION_DECODE.v

```
//write back
always @(posedge clk or posedge rst)
    if(rst) begin
        REG[0] <= 32'd0;
        REG[1] <= 32'd1;
        REG[2] <= 32'd2;

        for (i=3; i<32; i=i+1) REG[i] <= 32'b0;
    end
    else if(MW_RegWrite)
        REG[MW_RD] <= (MW_MemtoReg)? MDR : MW_ALUout;
```

Register 初始值 (\$0~\$31)

MEMORY.v

```
always @(posedge clk or posedge rst)
    if(rst)begin
        for (i=0;i<128;i=i+1)
            DM[i] <= 32'b0;
        end
    else if(XM_MemWrite)
        DM[ALUout[6:0]] <= XM_MD;
```

Memory初始值 (0x00000000-0x11111111)

課堂練習一

- 修改提供的壓縮檔內的testbench，使用事先定義好的加法功能，在下圖紅色區塊中加入適當的指令，讓程式做連續加法，使得 $\$4 = 9$ （※Hint: 請注意hazard問題）
- 初始化時給定暫存器初始值： $\$0 = 0$ 、 $\$1 = 1$ 、 $\$2 = 2$
- 向助教展示demo結果 (20%)

請在此區塊加入適當的指令

```
// Instruction DM initialilaton
initial
begin
```

```
/*===== write down your program =====*/
cpu.IF.instruction[ 0] = 32'b000000_00001_00010_00011_00000_100000; //add $3, $1, $2      $3 = 1 + 2 = 3
cpu.IF.instruction[ 1] = 32'b000000_00000_00000_00000_00000_100000; //NOP(add $0, $0, $0)
cpu.IF.instruction[ 2] = 32'b000000_00000_00000_00000_00000_100000; //NOP(add $0, $0, $0)
cpu.IF.instruction[ 3] = 32'b000000_00000_00000_00000_00000_100000; //NOP(add $0, $0, $0)
for (i=4; i<128; i=i+1) cpu.IF.instruction[ i] = 32'b000000_00000_00000_00000_00000_100000; //NOP(add $0, $0, $0)
cpu.IF.PC = 0;
end
```

testbench.v

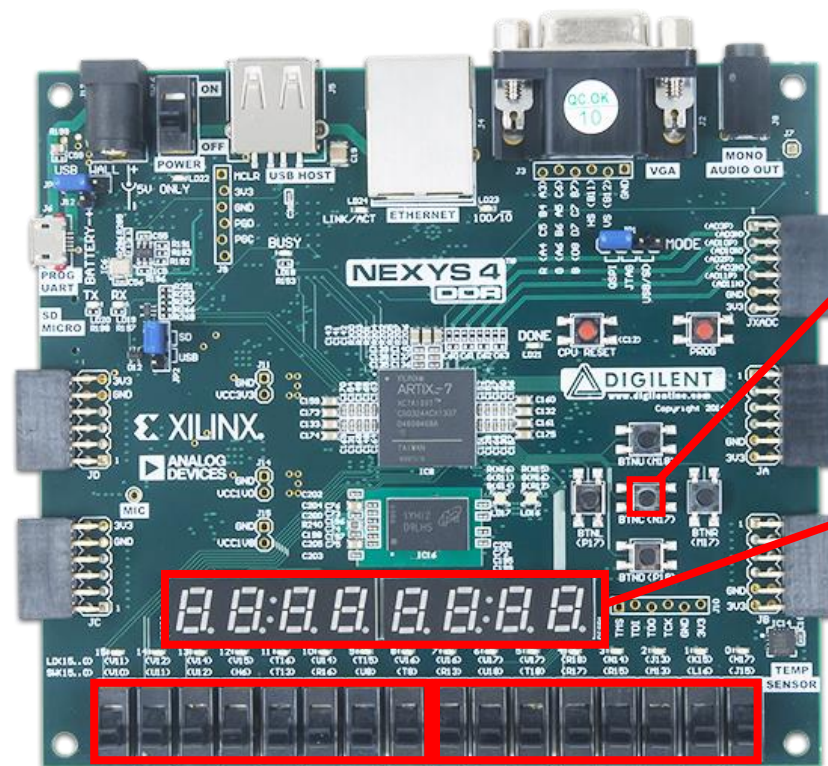
```
//write back
always @(posedge clk or posedge rst)
begin
    if(rst) begin
        REG[0] <= 32'd0;
        REG[1] <= 32'd1;
        REG[2] <= 32'd2;

        for (i=3; i<32; i=i+1) REG[i] <= 32'b0;
    end
    else if(MW_RegWrite)
        REG[MW_RD] <= (MW_MemtoReg)? MDR : MW_ALUout;
end
```

INSTRUCTION_DECODE.v

課堂練習二

- 以FPGA進行課堂練習一的功能驗證
 - 將課堂練習一所撰寫的機械碼放置到COLAB3-2專案testbench對應的位置，以進行FPGA的功能驗證
- 向助教展示demo結果 (20%)



執行程式

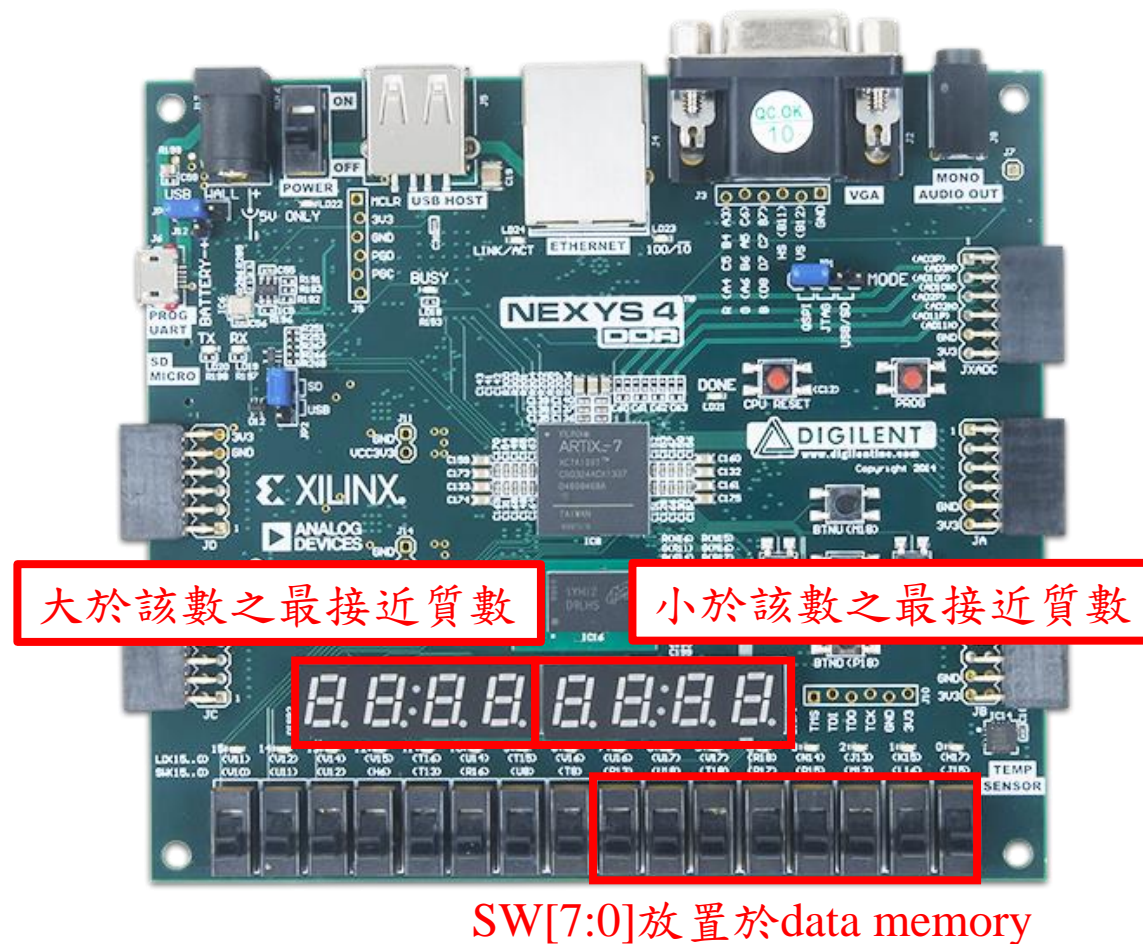
輸出Tiny MIPS中\$4的值，
以七段顯示器進行功能驗證

SW[15:8]放置於\$2 SW[7:0]放置於\$1

作業說明

- 新增RISC指令 (30%)
 - R-type : add, sub, and, or, slt
 - I-type : lw, sw, beq
 - J-type : j
- 修改testbench，使其能執行COLAB1的找質數程式 (20%)
 - 從MEM中讀取(lw) 一個輸入定值進行運算，並接運算完的兩筆值存回(sw) MEM
- 使用FPGA進行功能驗證 (10%)
 - Input: **SW[7:0]**放置於data memory
 - Output: 以七段顯示器進行功能驗證（左邊四位輸出大於該數之最接近質數、右邊四位輸出小於該數之最接近質數）
- Demo時間：11/17（四）下午2:00-5:00

※ 當天有課請**提早**通知助教 竣尹或唐興（請使用facebook messenger告知），**逾時不接受補demo**



參考資料

- MIPS Instruction Reference: <http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html>

附錄

Icarus Verilog教學

- 編譯RISC CPU檔案



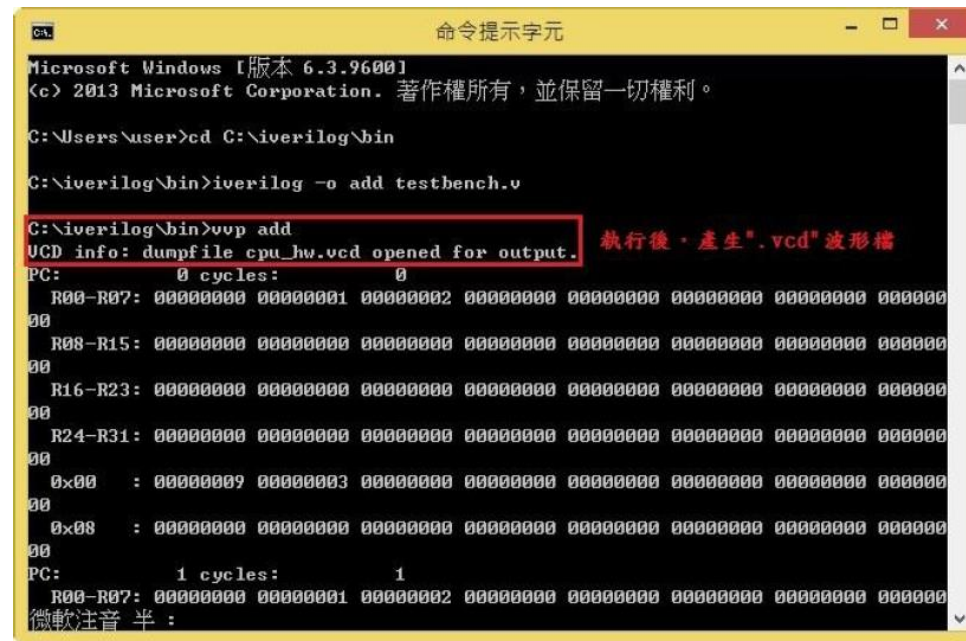
```
Microsoft Windows [版本 6.3.9600]
(c) 2013 Microsoft Corporation. 著作權所有，並保留一切權利。

C:\Users\user>cd C:\iverilog\bin
C:\iverilog\bin>iverilog -o add testbench.v_
```

鍵入此段指令，編譯RISC CPU之testbench

微軟注音 半：

- 執行後，產生波形檔(cpu_hw.vcd)



```
Microsoft Windows [版本 6.3.9600]
(c) 2013 Microsoft Corporation. 著作權所有，並保留一切權利。

C:\Users\user>cd C:\iverilog\bin
C:\iverilog\bin>iverilog -o add testbench.v
C:\iverilog\bin>vvp add
VCD info: dumpfile cpu_hw.vcd opened for output. 執行後，產生".vcd"波形檔
PC: 0 cycles: 0
R00-R07: 00000000 00000001 00000002 00000000 00000000 00000000 00000000 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
0x00 : 00000007 00000003 00000000 00000000 00000000 00000000 00000000 00000000
0x08 : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
PC: 1 cycles: 1
R00-R07: 00000000 00000001 00000002 00000000 00000000 00000000 00000000 00000000
微軟注音 半：
```

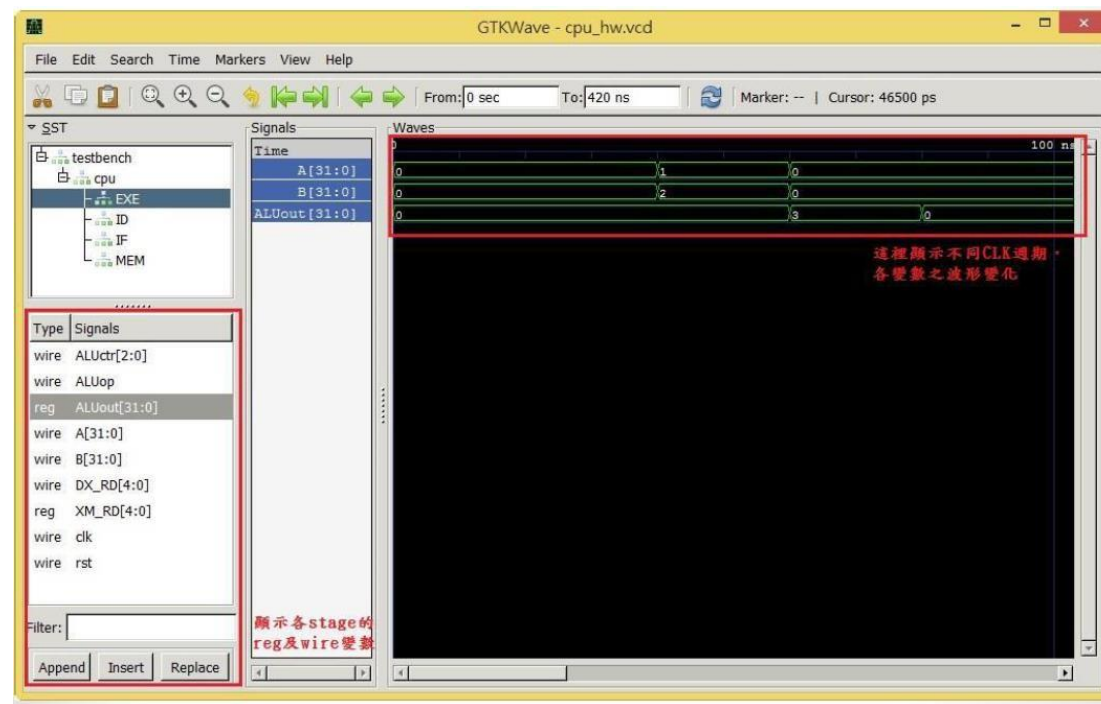
gtkwave教學

- 執行gtkwave，顯示波形檔

```
C:\iverilog\bin>cd C:\iverilog\gtkwave\bin
C:\iverilog\gtkwave\bin>gtkwave cpu_hw.vcd
```

使用gtkwave程式，顯示波形檔

微軟注音 半:



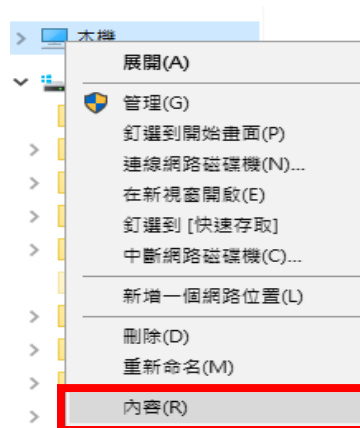
Icarus Verilog進階環境設定

➤ 避免同學將程式全放在bin資料夾編譯、執行，請同學依照下面步驟操作：

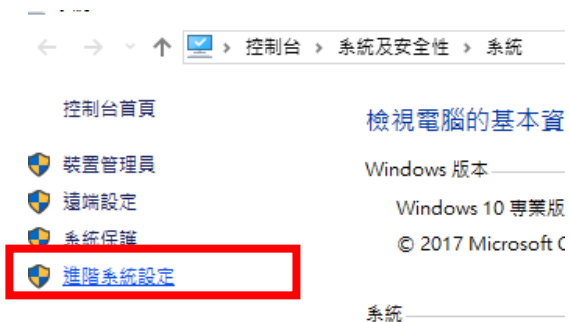
➤ 打開檔案總管



➤ 在本機圖示點擊右鍵，選擇內容



➤ 點擊進階系統設定

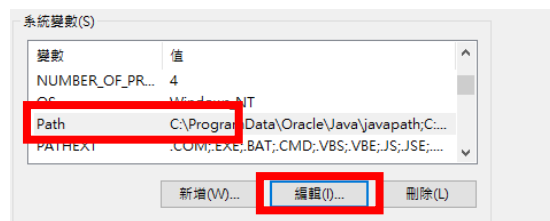


Icarus Verilog進階環境設定

➤ 點擊環境變數



➤ 點擊path並按下編輯



➤ 新增並輸入bin資料夾路徑，按下確定

※路徑為iverilog與gtkwave下的bin資料夾，已將資料放置同處，同學只需新增一個環境變數

