Use of ISS and C2A

實驗目的

1. MIPS ISA 的基本認識
2. RISC & CISC的認識與差異
3. MIPS暫存器簡介
4. MIPS 指令集架構
5. 認識 MIPS ISS Platform and MIPS ISA
6. 實驗環境架設

Background

- MIPS 原始的概令是透過指令管線化來增加CPU運算的速度，最早的MIPS架構是32位元,最新的版本已經變成64位元, 32位元向上相容MIPS64位元架構。
- MIPS 是精簡指令集架構(RISC)。

RISC & CISC

RISC: Reduce Instruction Set Computer

Uses registers for internal computations and only load/store instructions are used to access memory.

Major processors now use RISC-like ISA.

- **MIPS** (PlayStation, PlayStation 2 , PSP…)
- **ARM** (Apple iPods/iPhone/iPad, NDS..)

RISC & CISC

CISC: Complex Instruction Set Computer

- Early computers use CISC ISA.
- One of the reasons is to reduce code size due to expensive memory cost in the early days.

Simple Comparisons: RISC & CISC

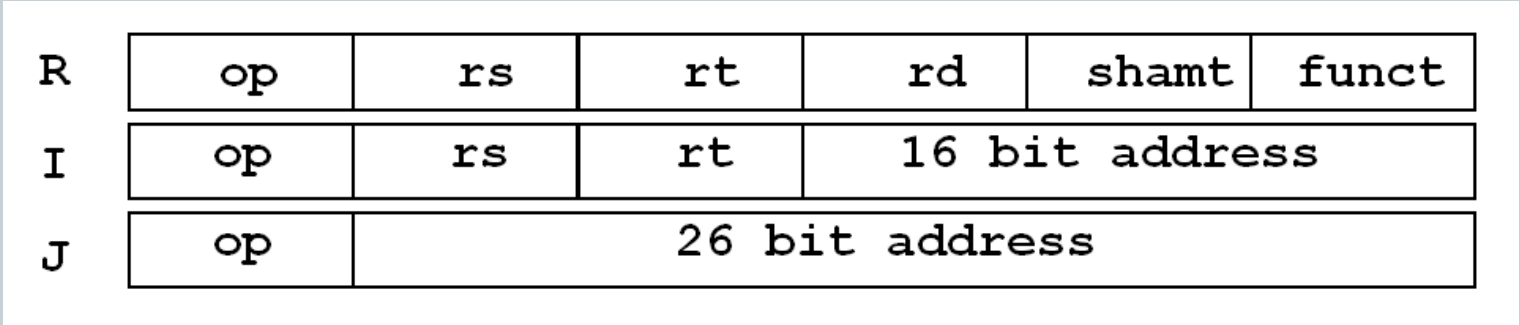
架構	RISC	CISC
指令集	精簡,指令數少	複雜,指令較多
指令能力	因指令精簡,能力較差與CISC相比	能力強,因擁有額外指令在相同情況下,可用較少指令來達成任務
指令長度	長度固定,所有指令長度相同	不同指令可能有不同的長度
編譯上	較輕鬆因長度相同	相對困難處理較複雜
暫存器	多	少
定址方式	少	多

Overview of MIPS ISA: registers

register	assembly name	Comment
r0	$zero	Always 0
r1	$at	Reserved for assembler
r2-r3	$v0-$v1	Stores results
r4-r7	$a0-$a3	Stores arguments
r8-r15	$t0-$t7	Temporaries, not saved
r16-r23	$s0-$s7	Contents saved for use later
r24-r25	$t8-$t9	More temporaries, not saved
r26-r27	$k0-$k1	Reserved by operating system
r28	$gp	Global pointer
r29	$sp	Stack pointer
r30	$fp	Frame pointer
r31	$ra	Return address

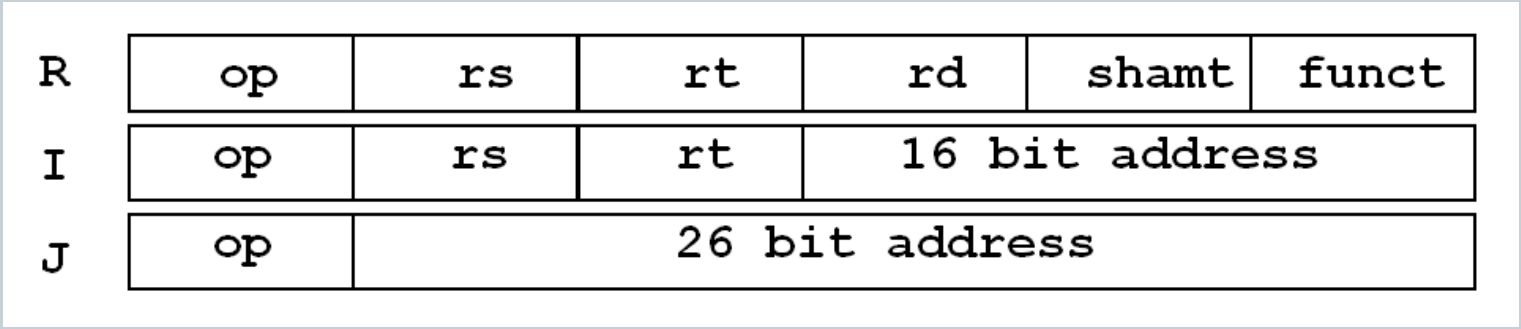
Overview of MIPS ISA

- Simple instruction formats, all are 32 bits wide.
- Very structured
- Instruction formats only have three types



Overview of MIPS ISA: Instruction format

- Op 與funct 有6 bits用來判斷指令的type
- rs,rt 有5 bits,是來源暫存器欄位
- rd有5 bits,是目的暫存器欄位
- shamt有5 bits,是偏移量欄位



MIPS Arithmetic/Logic

- All instructions have 3 operands
- Operand order is fixed (destination 1st)

如何把高階語言轉換成組合語言?

Ex.

in C code ： a = b + c

MIPS' code ：add a, b, c

R type

- R type的指令,功能主要是負責運算作用
- 例如:add 與 sub 皆為R type指令
- 當指令是R type時,op code皆為0
- add指令的funct為32, sub為34

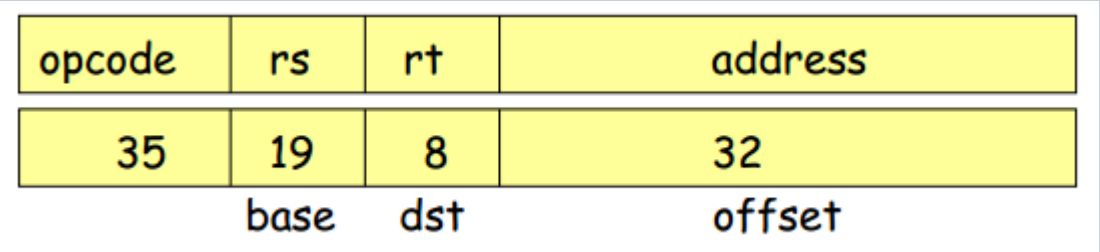
Ex： add $s1, $s2, $t0

opcode	rs	rt	rd	shift amt	function
0	18	8	17	0	32

I type

- I type的指令,功能之一主要是負責存取記憶體
- 例如:lw (load word) 與sw (store word)
- 當指令是lw時,op code為35

Ex： lw $t0, 32($s3)



I type

- beq (Branch on equal)和 bne (branch not equal)是一種判斷指令,常用於loop和if else指令。

- beq $s1 , $s2 , L :

  表示是判斷 s1和s2暫存器的內存值是否相同,若是相同則跳躍到L的位置執行。

- bne $s1 , $s2 , L :

  表示是判斷 s1和s2暫存器的內存值是否相同,若是不相同則跳躍到L的位置執行。

I type

Example:

C code A[12] = h + A[8];

MIPS code

Var.	Reg.
h	$s2
A_{base}	$s3

```
lw   $t0,32($s3)
add  $t0,$s2,$t0
sw   $t0,48($s3)
```

J type

- J type的指令,功能主要是跳到某個位置執行例如: Jump
- 當指令是jump時, op code為2

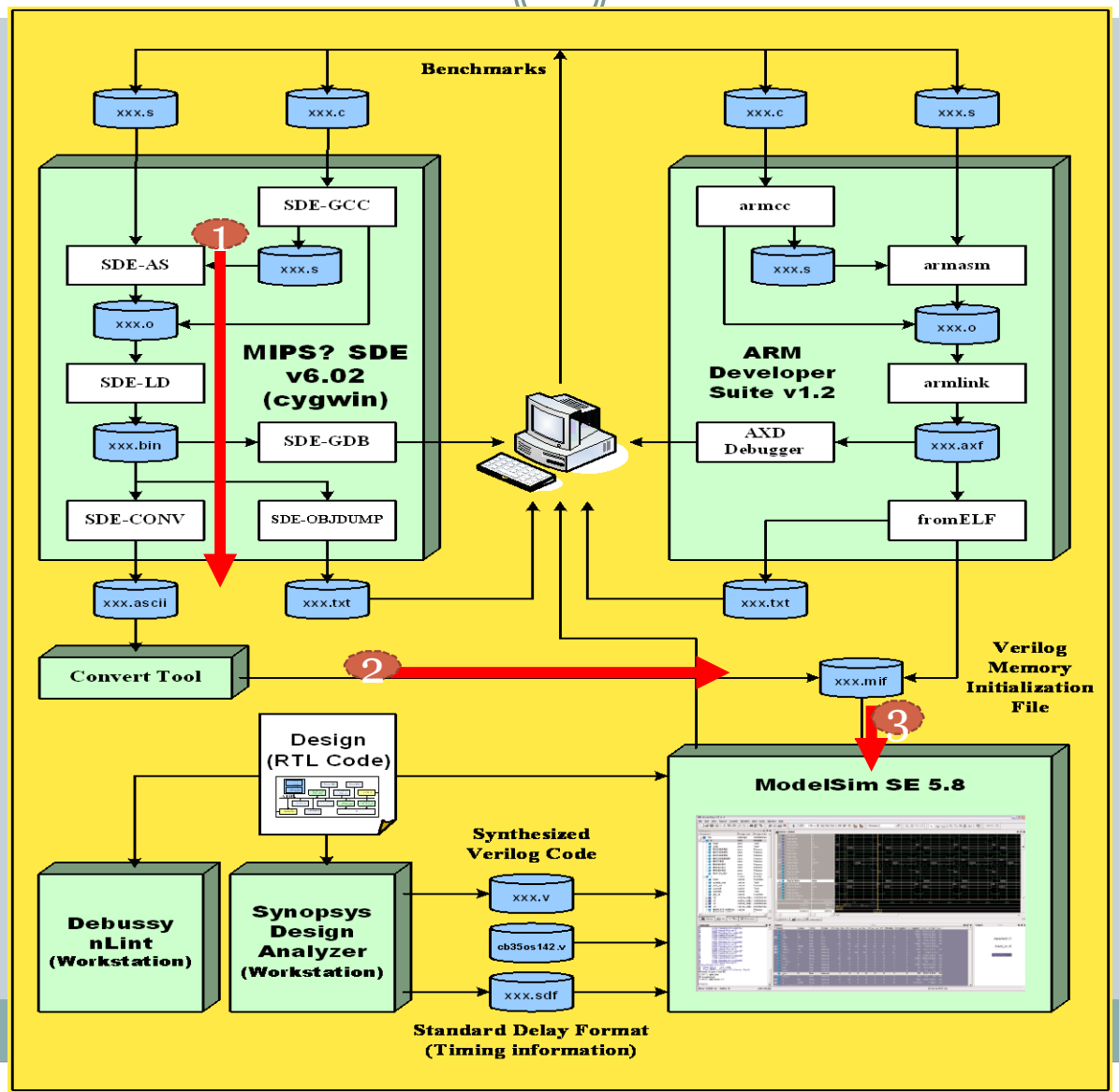
Ex： jump to L　　*L的位置是10000

opcode	address
2	2500

實驗目的

1. MIPS ISA 的基本認識
2. RISC & CISC的認識與差異
3. MIPS暫存器簡介
4. MIPS 指令集架構
5. 認識 MIPS ISS Platform and MIPS ISA
6. 實驗環境架設

MIPS/ARM/Modelsim Simulation Tool Trains



實驗環境

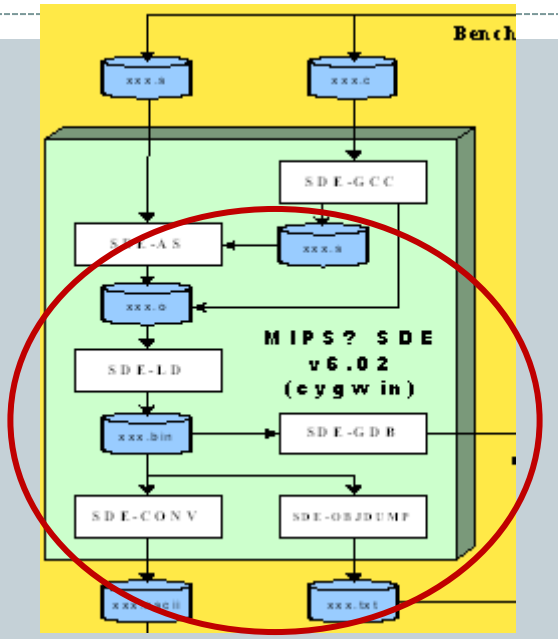
1. Linux (此處以VM上，安裝Ubuntu OS)
   - 因為使用的 MIPS Cross compiler需要在Linux下才能執行
2. MIPS Cross compiler (Compile program)
   - 使用的是**MIPS® SDE**
   - 主要是將C轉成ASCII,以便後續的處理
3. Modelsim (Run RTL CPU simulator)
   - 結合wave來驗證C code or assembly code是否正確
4. Java
   - Some utility tools are run in Java.
   - http://www.oracle.com/technetwork/java/index.html

Transform assembly code to binary code

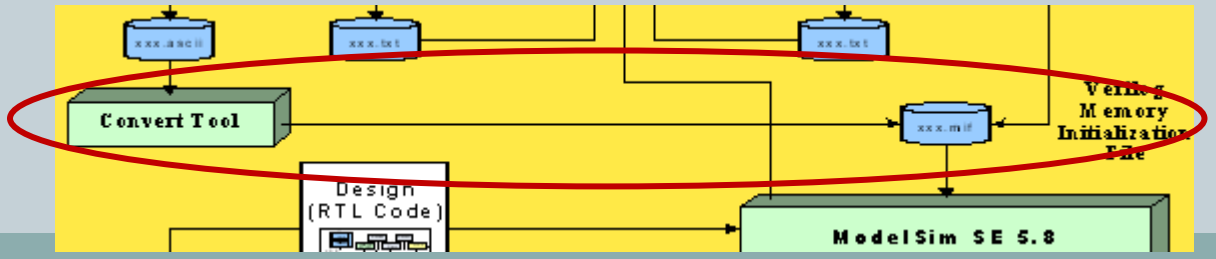
Transform assembly code to binary code

- **Use MIPS® SDE Lite (base on linux)**
  - **Step.1:** transform assembly code to object file
  - **Step.2:** link object file to image file
  - **Step.3:** transform binary to ASCII format
  - **Step.4:** generate a file for debugging

    (Step1~4: sh make_sh.sh)



- **Use converter & combine**
  - **Step.5:** Convert assembly code to Memory Initialization File (for Verilog input)
  - **Step.6:**與系統的初始值combine,讓系統能夠開機



Step.1: transform assembly code to object file

- 在linux下打開Terminal
- 輸入：sde-as add.s -g -o add.o
  1. sde-as:目的是讓Assembly code轉成Object file
  2. -g :目的是讓Object file 添加輔助Debug的資訊
  3. -o :讓產生的目標檔

Step.1-執行圖

➢ add.s

```
1  li $8,8
2  li $9,9
3  add $10,$8,$9
4  move $11,$10
```

```
mips-ta@ubuntu: ~/CPU_LAB/TEST
mips-ta@ubuntu:~/CPU_LAB/TEST$ ls
add.s
mips-ta@ubuntu:~/CPU_LAB/TEST$ sde-as add.s -g -o add.o
mips-ta@ubuntu:~/CPU_LAB/TEST$ ls
add.o  add.s
mips-ta@ubuntu:~/CPU_LAB/TEST$
```

➢ 腳本中，對應到的指令

File Edit View Search Tools Documents Help

Open Save Undo

make_sh.sh

```
sde-as add.s -g -o add.o
sde-ld add.o -Ttext 0x00008000 -o add.image
sde-objdump add.image -D > add.txt
sde-conv add.image -o add.conv
```

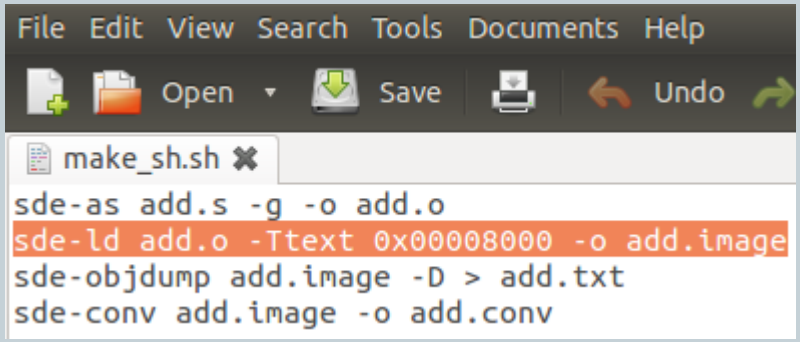
Step.2: link object file to image file

- 輸入 : sde-ld add.o -Ttext 0x00008000 -o add.image
  1. sde-ld : 目的是可將多個Object file link壓縮成為一個 image檔
  2. Ttext :目的是設定code的起始位置

Step.2-執行圖

```
mips-ta@ubuntu:~/CPU_LAB/TEST$ ls
add.o  add.s
mips-ta@ubuntu:~/CPU_LAB/TEST$ sde-ld add.o -Ttext 0x00008000 -o add.image
sde-ld: warning: cannot find entry symbol __start; defaulting to 000000000000800
0
mips-ta@ubuntu:~/CPU_LAB/TEST$ ls
add.image  add.o  add.s
mips-ta@ubuntu:~/CPU_LAB/TEST$
```

- 腳本中，對應到的指令



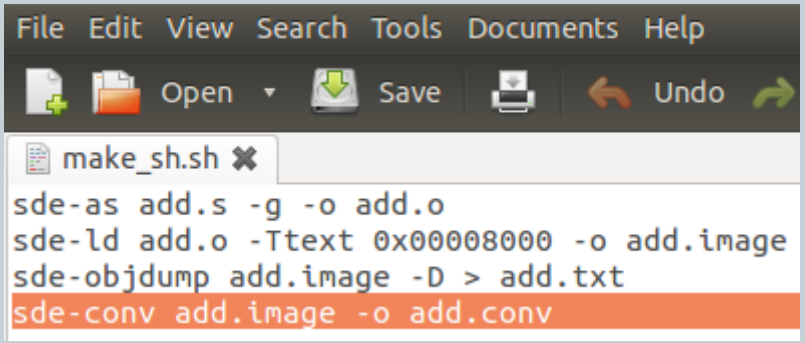
Step.3: transform binary to ASCII format

- 輸入: sde-conv add.image -o add.conv
  1. sde-conv : 將binary的格式轉成ASCII的格式(add.conv)，因為modelsim所input的檔案是ASCII 的格式.
  2. The LAB can use RTL-based MIPS simulator for the ease of debugging and signal observations.

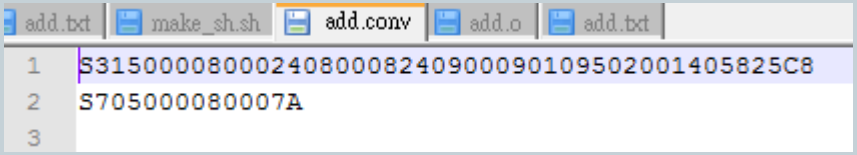
Step.3-執行圖

```
mips-ta@ubuntu:~/CPU_LAB/TEST$ ls
add.image  add.o  add.s
mips-ta@ubuntu:~/CPU_LAB/TEST$ sde-conv add.image -o add.conv
mips-ta@ubuntu:~/CPU_LAB/TEST$ ls
add.conv  add.image  add.o  add.s
mips-ta@ubuntu:~/CPU_LAB/TEST$
```

- 腳本中，對應到的指令



- add.conv



Step.4: generate a file for debugging

- 輸入: sde-objdump add.image -D > add.txt
  1. sde-objdump :目的產生.txt 檔一種讓程式設計者容易debug的格式，會在添加local和global address 等資訊
  2. -D:拆開所有指令

Step.4-執行圖

```
mips-ta@ubuntu:~/CPU_LAB/TEST$ ls
add.conv  add.image  add.o  add.s
mips-ta@ubuntu:~/CPU_LAB/TEST$ sde-objdump add.image -D > add.txt
mips-ta@ubuntu:~/CPU_LAB/TEST$ ls
add.conv  add.image  add.o  add.s  add.txt
mips-ta@ubuntu:~/CPU_LAB/TEST$
```

- 腳本中，對應到的指令

make_sh.sh

```
sde-as add.s -g -o add.o
sde-ld add.o -Ttext 0x00008000 -o add.image
sde-objdump add.image -D > add.txt
sde-conv add.image -o add.conv
```

- add.txt

add.txt | make_sh.sh

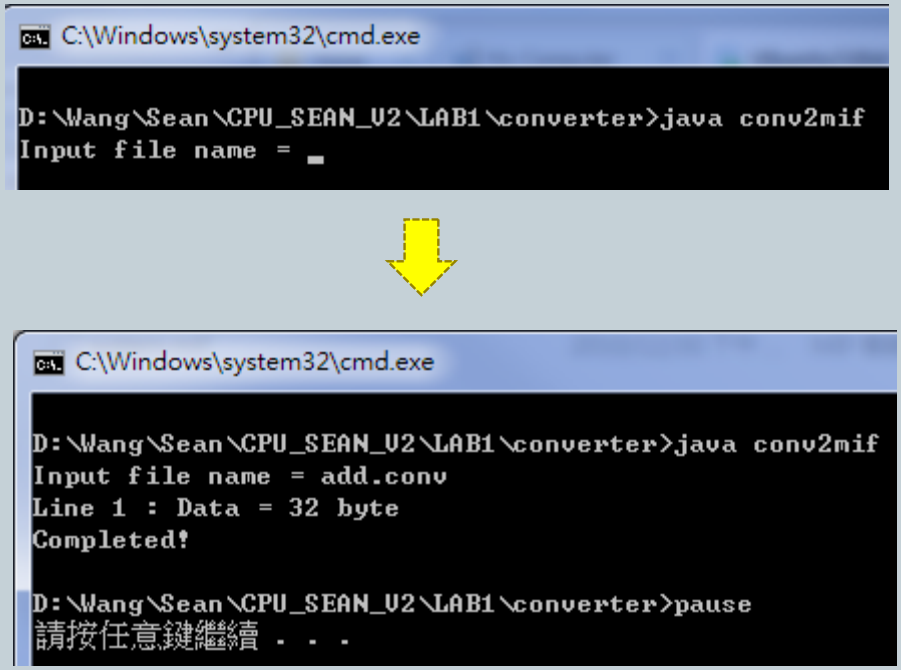
```
1
2  add.image:     file format elf32-bigmips
3
4  Disassembly of section .text:
5
6  0000000000008000 <_ftext>:
7      8000:   24080008    li   $t0,8
8      8004:   24090009    li   $t1,9
9      8008:   01095020    add $t2,$t0,$t1
10     800c:   01405825    move    $t3,$t2
11 Disassembly of section .data:
```

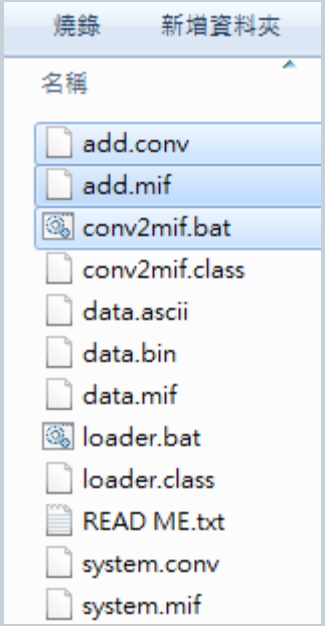
Step.5: Convert assembly code to Memory Initialization File (for Verilog input)

- 把cross complier出來的add.conv放到windows下的converter資料夾，並執行以下步驟:
  1. 點選conv2mif.bat，將add.conv 轉成 add.mif

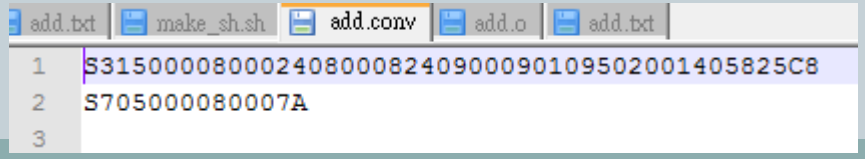
- 使用conv2mif.bat將 add.conv轉成add.mif 檔，目的在於可以將ASCII檔轉成Verilog想要的記憶體格式，以便程式(指令)能夠有序在CPU中被執行。

Step.5-執行圖

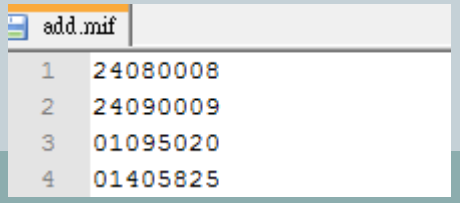




- add.conv



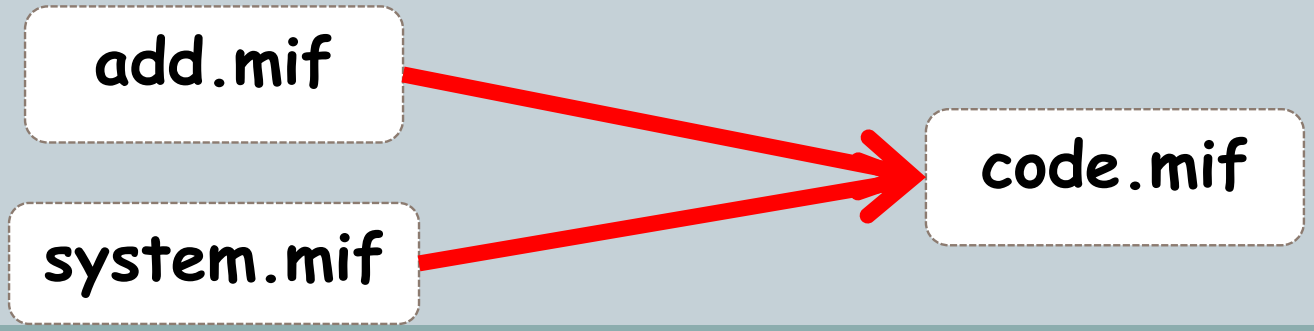
- add.mif



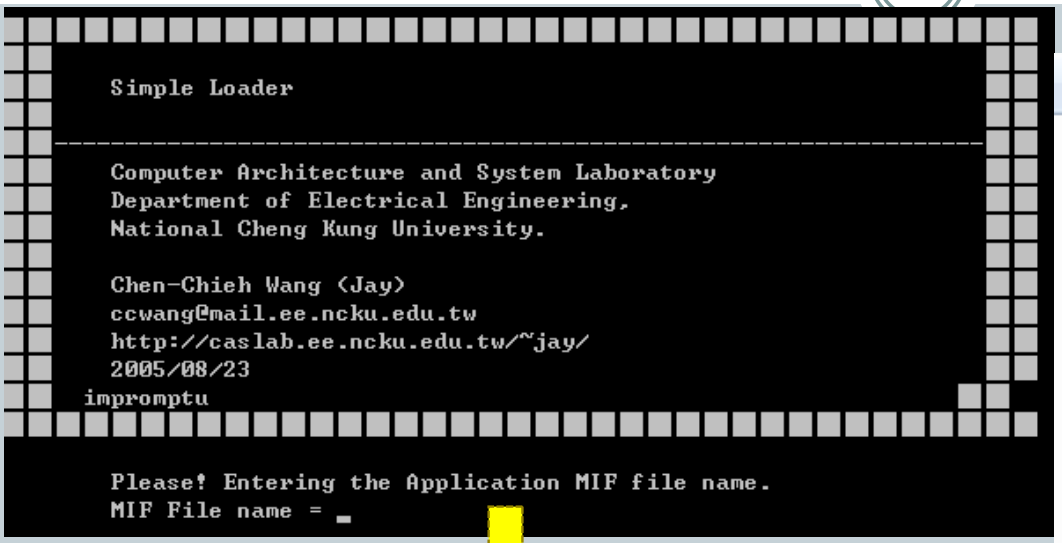
Step.6:與系統的初始值combine,讓系統能夠開機

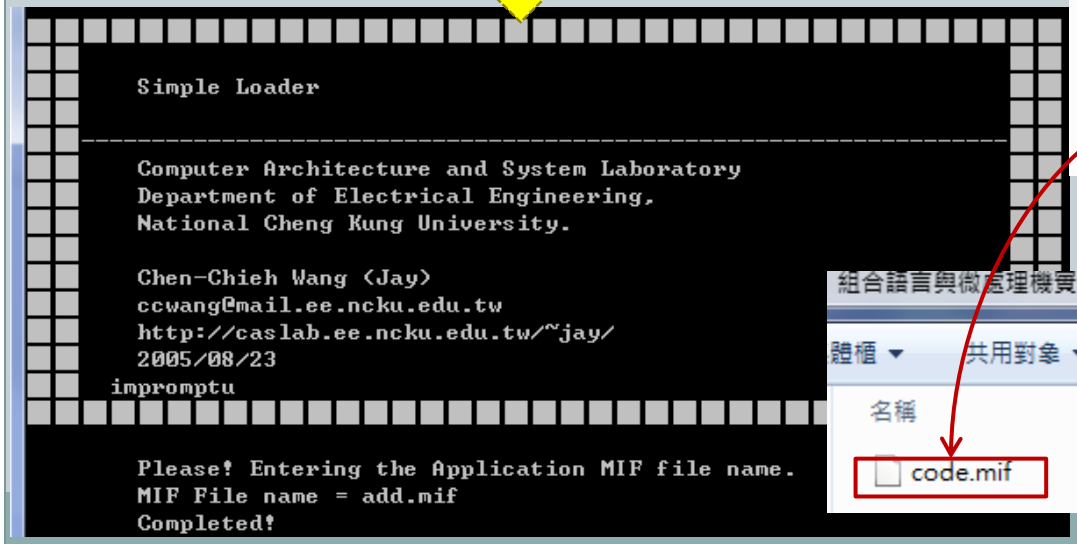
- 產生完add.mif檔之後
  1. 點擊loader.bat輸入add.mif
  2. 產生出code.mif檔,此時將此檔案放置到MIPS_cpu_test\software_mips目錄下
     可以自己命名

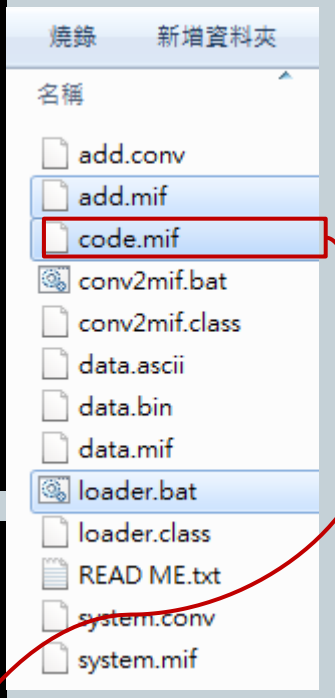
- Combine with system.mif
  - 此時只有編譯過我們的程式是無法讓CPU正常的運作，因為缺少給系統初值的動作，因此執行loader就是將程式(add.mif)和系統初值設定程式(system.mif)結合起來，這樣才能順利讓CPU執行

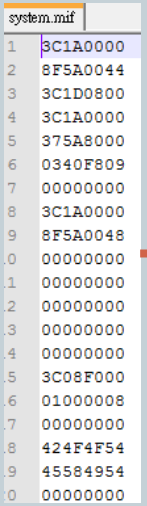


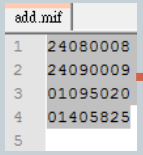
Step.6-執行圖

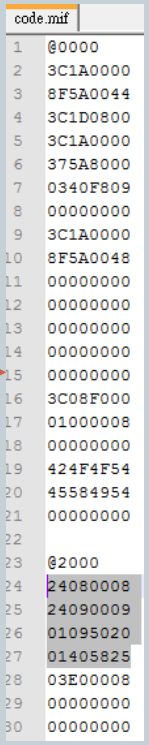

Simple Loader
Computer Architecture and System Laboratory
Department of Electrical Engineering,
National Cheng Kung University.
Chen-Chieh Wang (Jay)
ccwang@mail.ee.ncku.edu.tw
http://caslab.ee.ncku.edu.tw/~jay/
2005/08/23
impromptu
Please! Entering the Application MIF file name.
MIF File name =

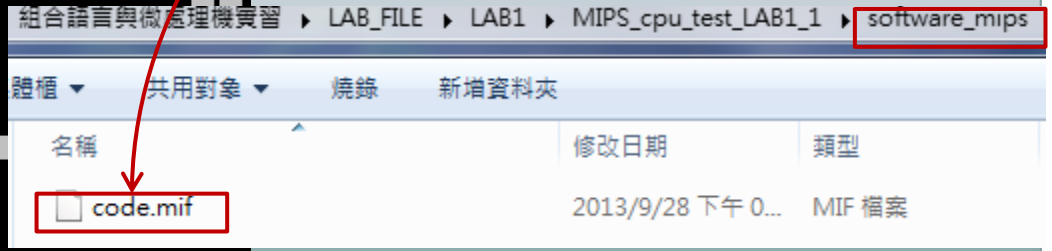

Simple Loader
Computer Architecture and System Laboratory
Department of Electrical Engineering,
National Cheng Kung University.
Chen-Chieh Wang (Jay)
ccwang@mail.ee.ncku.edu.tw
http://caslab.ee.ncku.edu.tw/~jay/
2005/08/23
impromptu
Please! Entering the Application MIF file name.
MIF File name = add.mif
Completed!


燒錄
新增資料夾
名稱
add.conv
add.mif
code.mif
conv2mif.bat
conv2mif.class
data.ascii
data.bin
data.mif
loader.bat
loader.class
READ ME.txt
system.conv
system.mif


system.mif
1 3C1A0000
2 8F5A0044
3 3C1D0800
4 3C1A0000
5 375A8000
6 0340F809
7 00000000
8 3C1A0000
9 8F5A0048
10 00000000
11 00000000
12 00000000
13 00000000
14 00000000
15 00000000
16 3C08F000
17 01000008
18 00000000
19 424F4F54
20 45584954
21 00000000

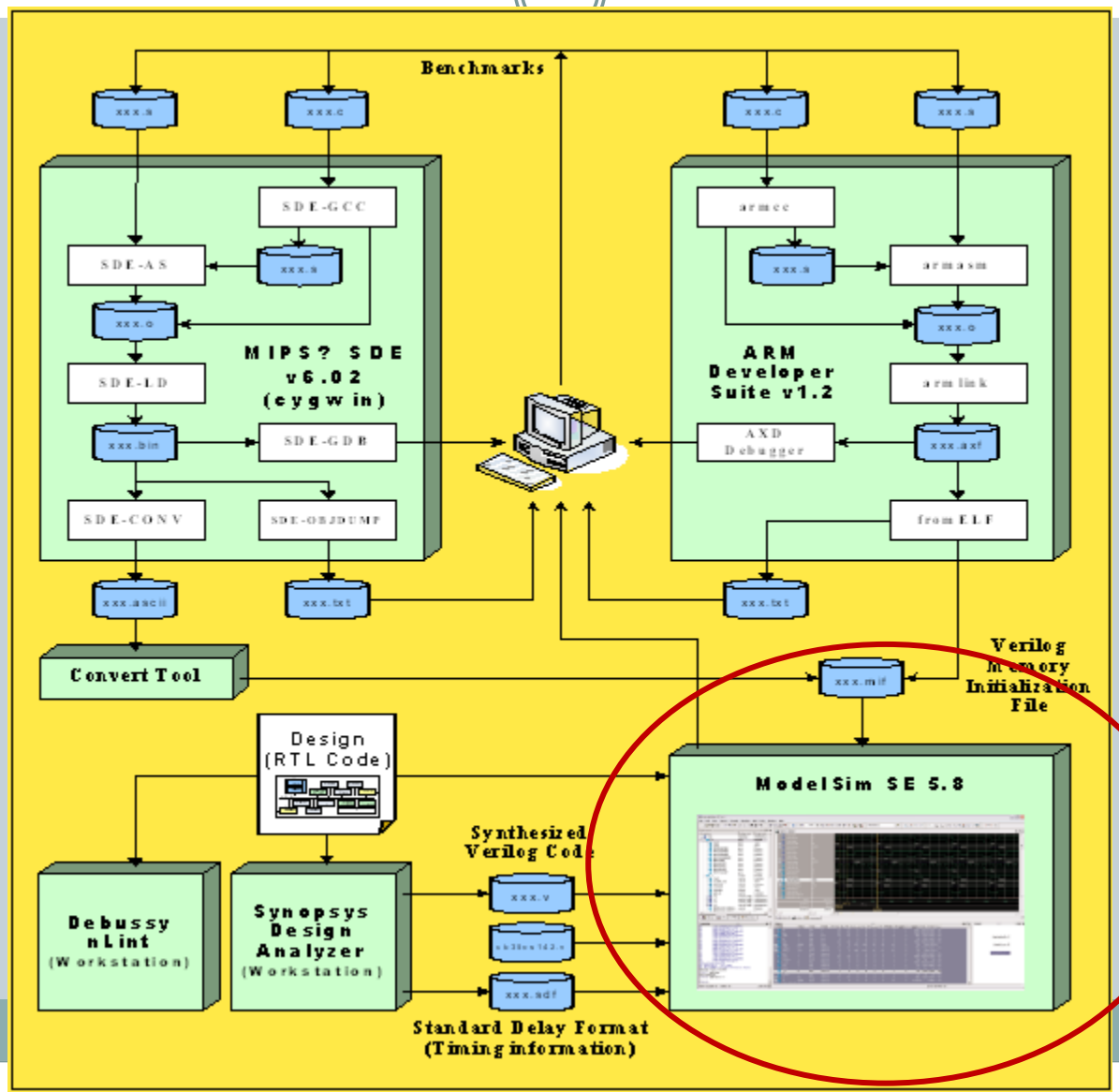

add.mif
1 24080008
2 24090009
3 01095020
4 01405825
5


code.mif
1 @0000
2 3C1A0000
3 8F5A0044
4 3C1D0800
5 3C1A0000
6 375A8000
7 0340F809
8 00000000
9 3C1A0000
10 8F5A0048
11 00000000
12 00000000
13 00000000
14 00000000
15 00000000
16 3C08F000
17 01000008
18 00000000
19 424F4F54
20 45584954
21 00000000
22
23 @2000
24 24080008
25 24090009
26 01095020
27 01405825
28 03E00008
29 00000000
30 00000000


組合語言與微處理機實習
LAB_FILE
LAB1
MIPS_cpu_test_LAB1_1
software_mips
體檔
共用對象
燒錄
新增資料夾
名稱
修改日期
類型
code.mif
2013/9/28 下午 0...
MIF 檔案

Modelsim 驗證教學

Modelsim 驗證教學

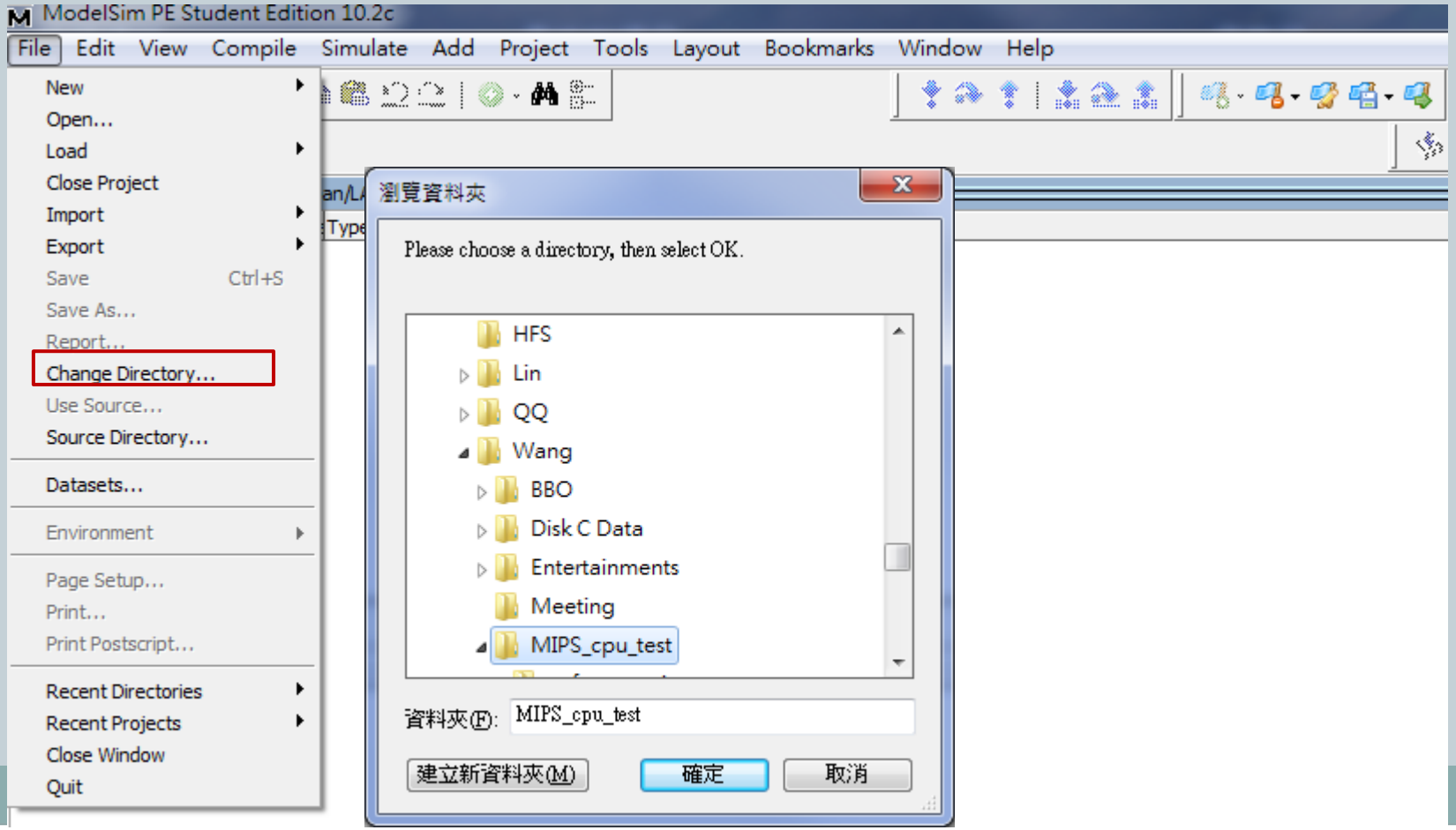


Modelsim 驗證教學

- **Step.1: change to LAB1 file location**

可以自己命名

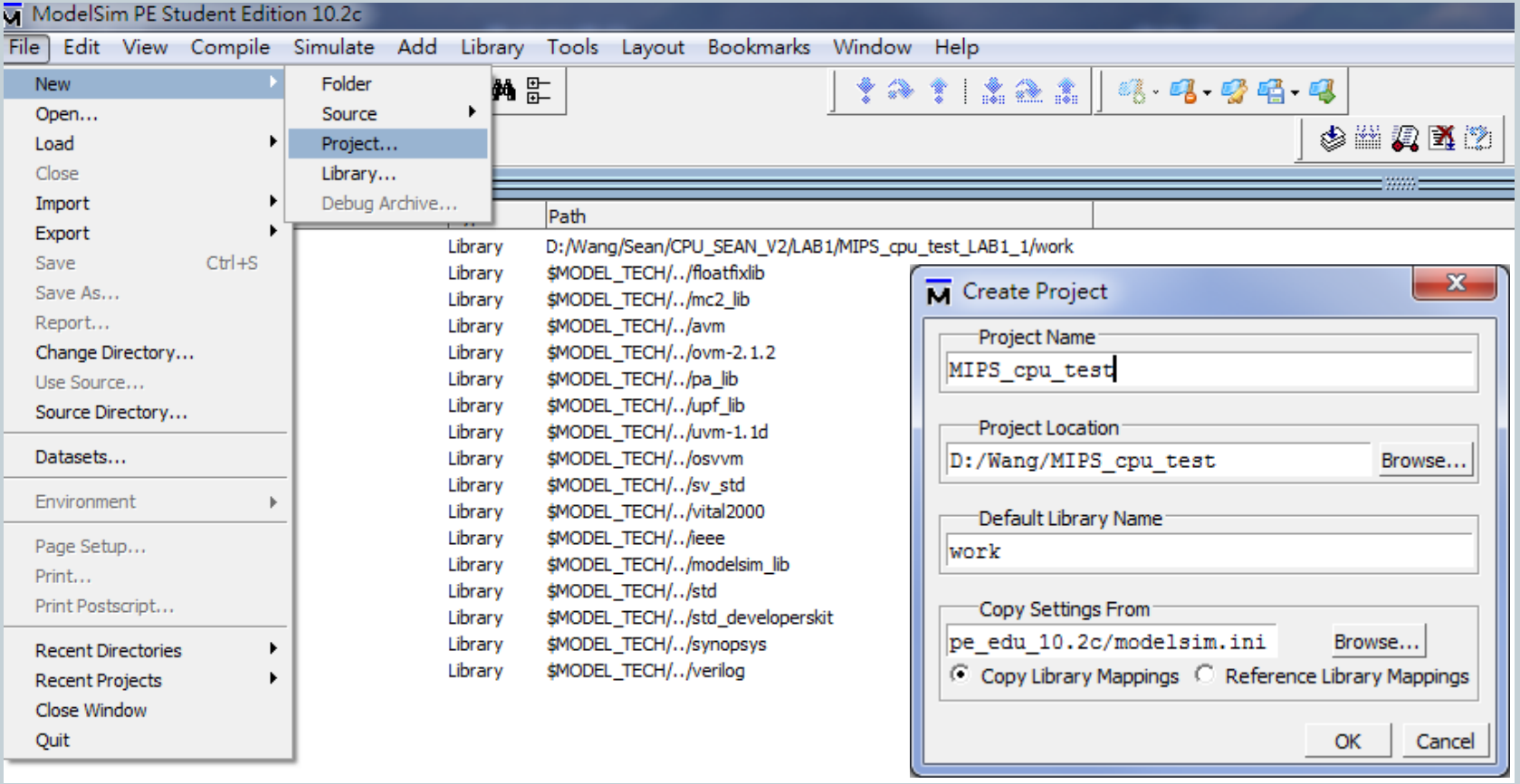
開modelsim後,在File下選擇change directory到你放\...\MIPS_cpu_test 的地方



Modelsim 驗證教學

- **Step.2: new project**

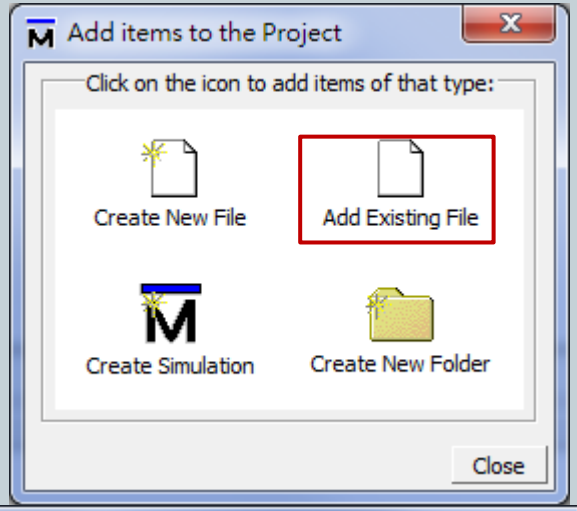
接著new一個project(名稱自訂)

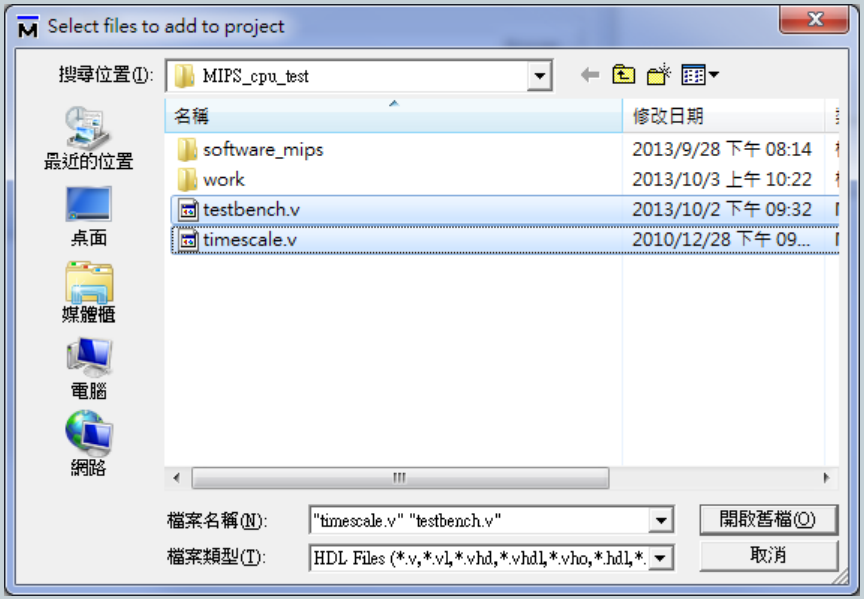


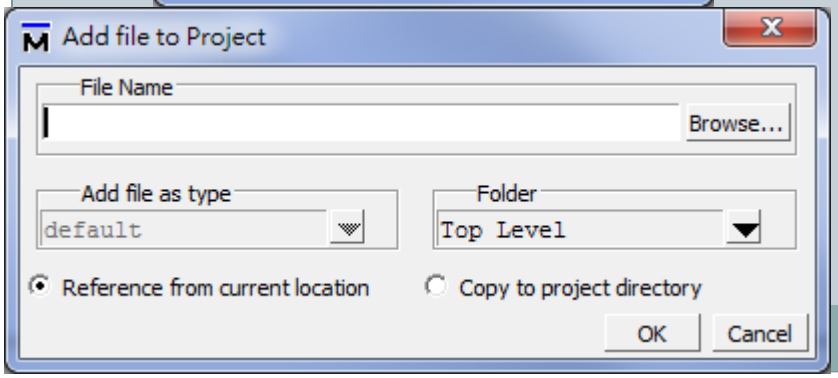
Modelsim 驗證教學

- **Step.3: add source code**

  新增完project後,會跳出一個視窗(如圖),點選Add Existing File將VHDL/Verilog source code加入到這個project中。



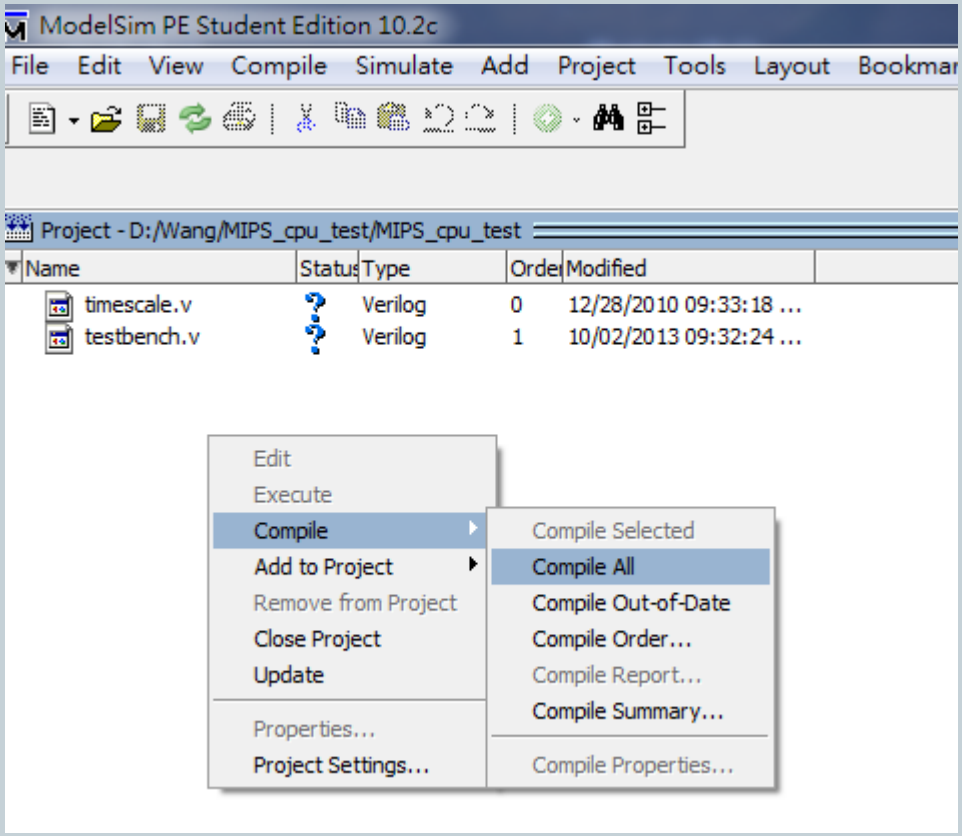




Modelsim 驗證教學

- **Step.4: compiler source code**

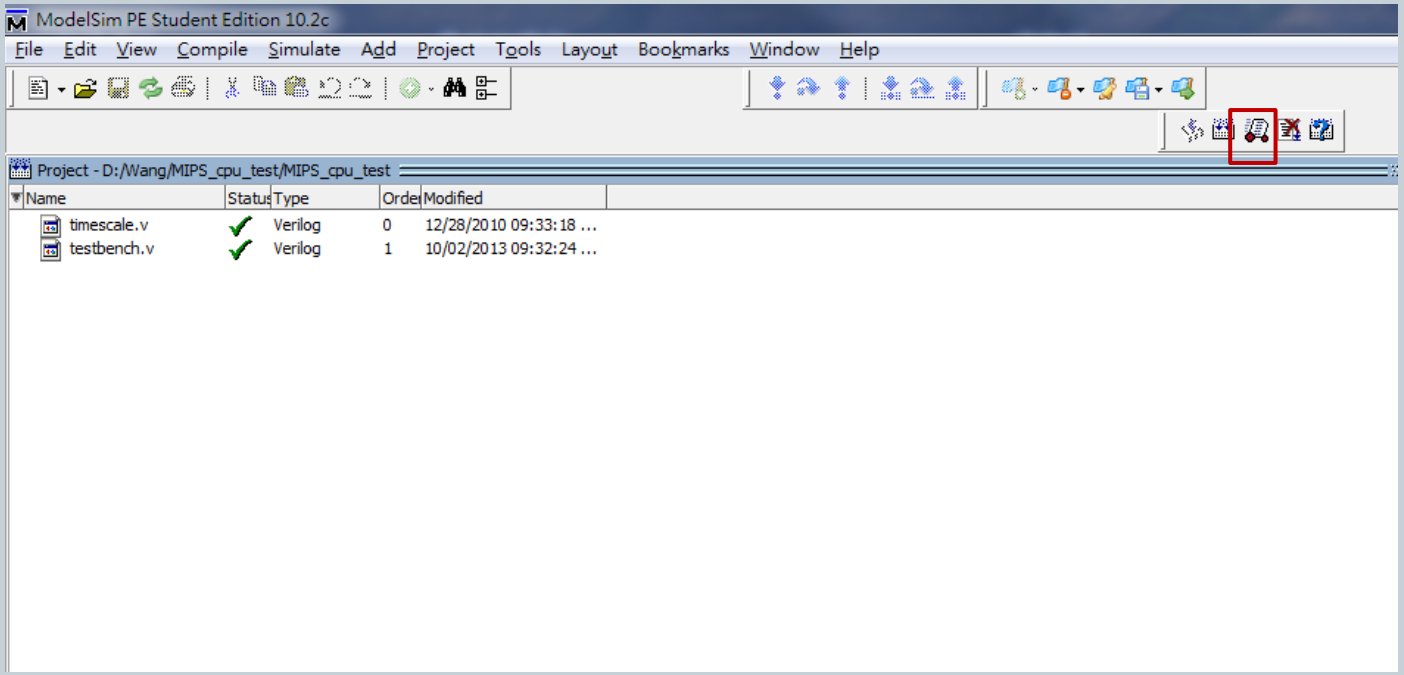
  點擊compiler all



Modelsim 驗證教學

- **Step.5: simulate source code**

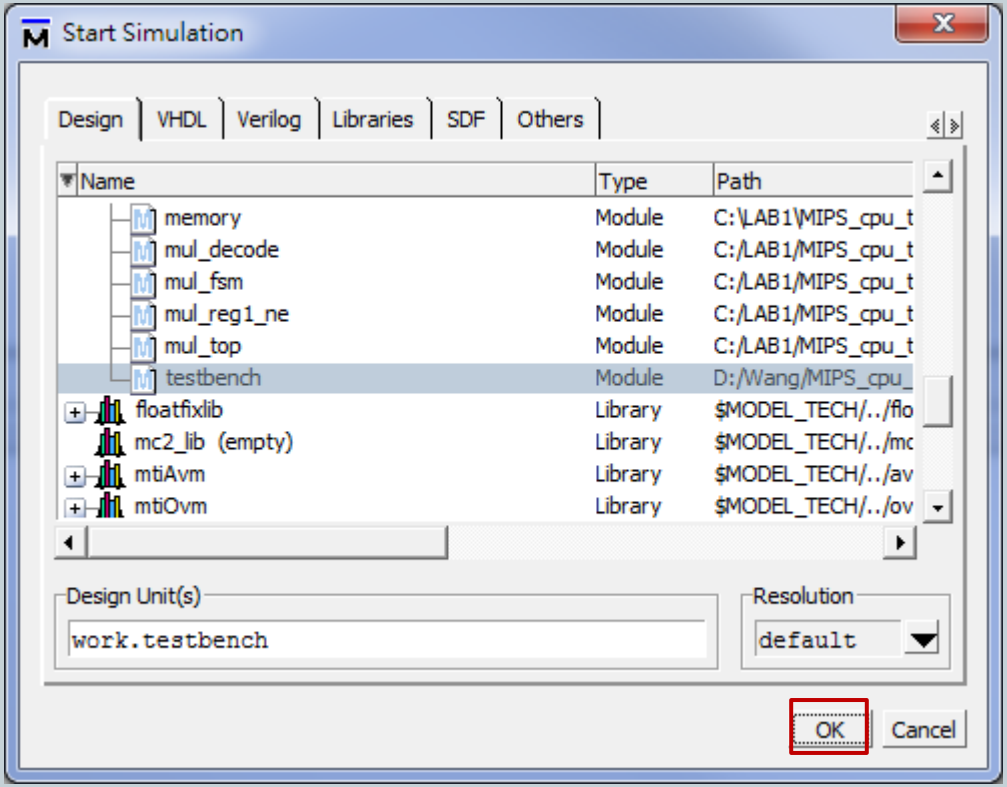
  點擊simulate



Modelsim 驗證教學

- **Step.6: choose testbench and enable optimization**

打開work後選testbench後，按OK。

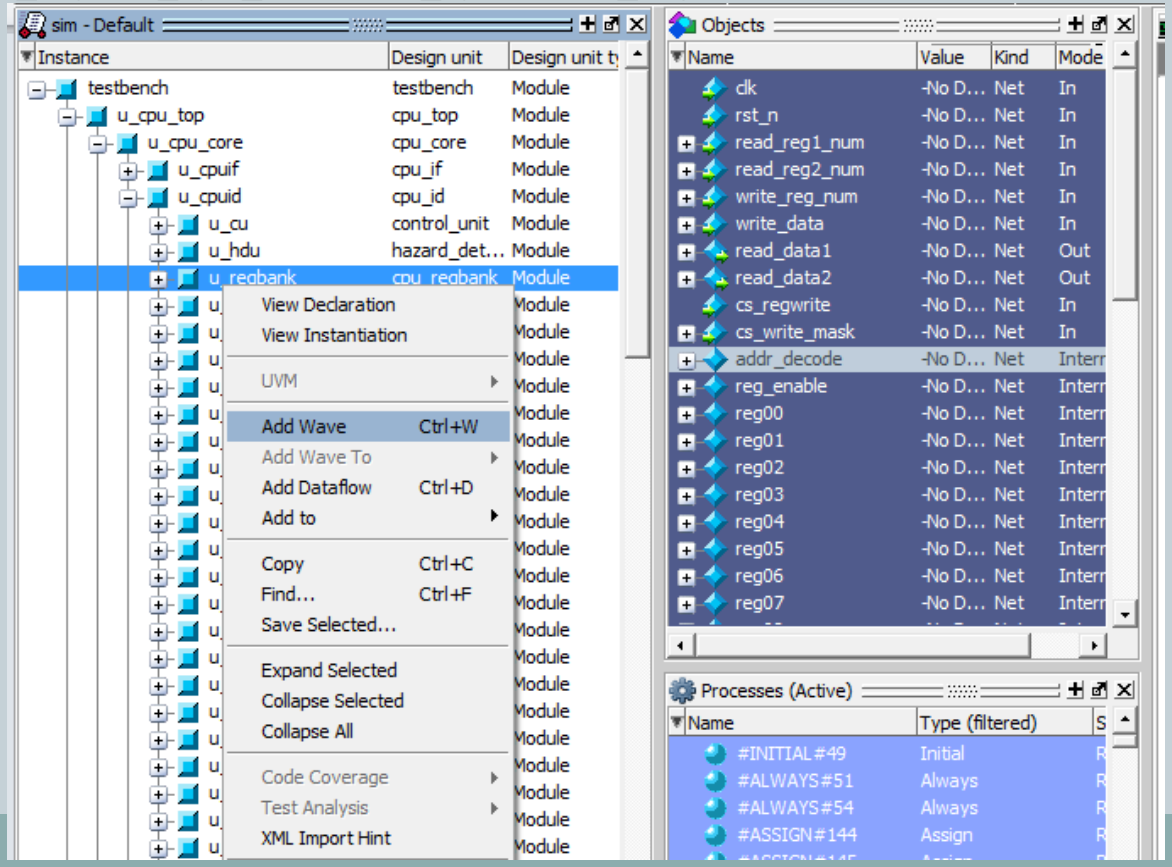


Modelsim 驗證教學

- **Step.7: add signal to wave**

  將你要看的訊號線按右鍵add wave,在這裡我們只看register

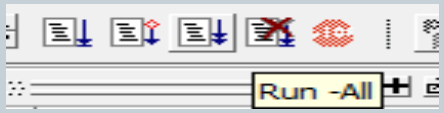
  (u_cpu_top→u_cpu_core → u_cpuid → u_regbank)



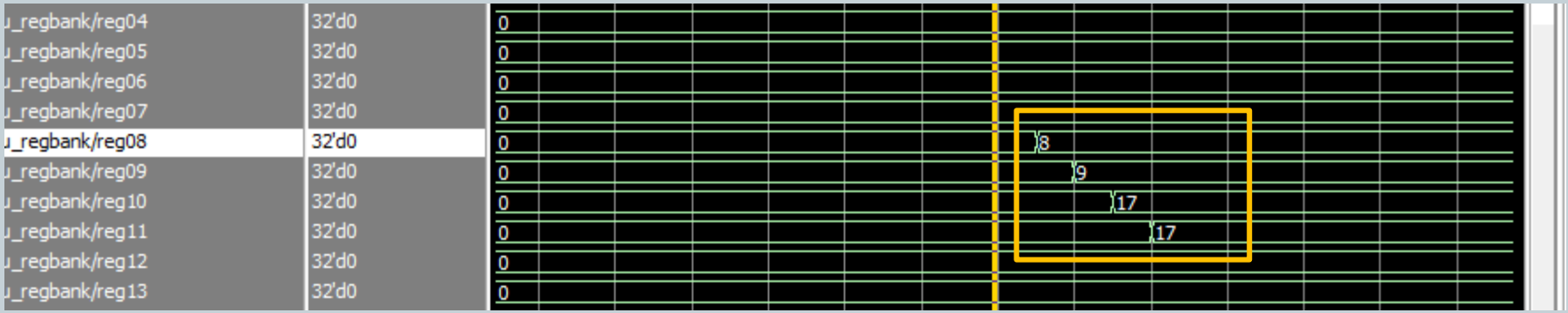
Modelsim 驗證教學

- **Step.8: check result**

  點選run-all檢視波形圖,即可看到結果,這邊是我們預設的簡單加法程式,code內容可至MIPS_cpu_test\software_mips目錄下查看 。



➢ Wave



➢ add.s

```
1  li $8,8
2  li $9,9
3  add $10,$8,$9
4  move $11,$10
```

➢ code.mif

```
21  00000000
22
23  @2000
24  24080008
25  24090009
26  01095020
27  01405825
28  03E00008
29  00000000
30  00000000
```

實作題目

實作(一)

- 下面為一個簡單的for迴圈程式，請同學練習用組合語言描述它，並依照上述step1~7步驟，完成在modelsim驗證結果。

```
int i;

  int sum=0;

  for (i=1 ; i<=10 ; i++)
   {
    sum = sum + i;
   }
```

✓ 備註:

i使用$8

sum使用$9

實作(一) ---參考範例

- 一個簡單的if-else C code(左圖)，共有f,g,h,i,j共5個暫存器,若用$s0~$4的register表示，可得到類似(右圖)的組合語言。

```
if( i == j)
    f = g+h;
else
    f = g-h;
```

```
      Bne  $s3, $s4,Else    # go to Else if (s3 != s4)
      add  $s0, $s1,$s2     # s0 = s1 + s2
      J    Exit             # go to Exit
Else: sub  $s0, $s1,$s2     # s0 = s1 - s2

Exit :
```

實作(二)

- 請同學練習存取記憶體的指令(load/store)，C程式如下請完成它的組合語言後，一樣跑上述step1~7步驟，完成在modelsim驗證結果。

```
int main(void)
{
    volatile int* a=(int*)0x20000000;      //陣列a是從記憶體 0x20000000開始存data
    volatile int* b=(int*)0x20000008;      //陣列b是從記憶體 0x20000008開始存data



    a[0]=1;                                //對記憶體位址0x20000000寫入1的值

    a[1]=2;                                //對記憶體位址0x20000004寫入2的值

    b[0]=a[0]+a[1];                        //將記憶體位置 0x20000000和0x20000004中
                                           的值讀出來做相加後，將結果寫入0x20000008
                                           的位置
    return 0;
```

實作(二) ---參考範例

- For C:

```
volatile int* a=(int*)0x40000000;
a[0]=8;
```

- For Assembly Code:

```
li   $3 , 1073741824    // 讓某暫存器存 0x40000000的(10進位)
li   $2 , 8             // 讓某暫存器先存8
sw  $2 , 0($3)          // 把記憶體 0x40000000  存8
```

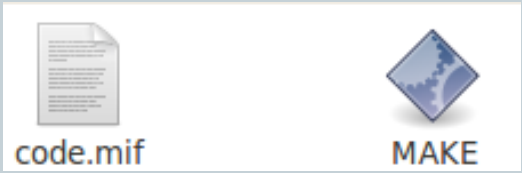
實作(三)---Run C-based MIPS ISS

- 請同學將實作(二)的code.mif拿到LINUX的環境下，並利用名叫MAKE的執行檔來執行此mif檔，之後會產生一個output.txt，同學可利用此output檔來看每個指令執行的過程。

- 執行方式：

  到相對應的目錄下，開啟終端機，並且輸入./MAKE code.mif

  (若不能執行，先輸入chmod 777 MAKE)



實作(三)---觀察結果

- 開始執行完之後產生的output.txt，可以看每一個指令的說明以及變化。(是instruction count)

```
**************************************************
This function i going to do : ORI
And the instruction is : 34420003
The fuction description is : GPR[rt] ← GPR[rs] or immediate
Result:
imm is : 3
rt is 2 AND $2 is 3
rs is 2 AND $2 is 3
**************************************************
**************************************************
This function i going to do : ADD
And the instruction is : 00012020
The fuction description is : GPR[rd] ← GPR[rs] + GPR[rt]
Result:
rs is 0 AND $0 is 0
rt is 1 AND $1 is 2
rd is 4 AND $4 is 2
**************************************************
```

→ 目前執行到的指令以及十六進位碼

挑戰題

- 請同學利用組合語言實現Fibonacci數列(n=10)的結果，並將答案寫入記憶體位置0x20000000的位置，最後用Modelsim看register和Memory中的結果。

- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$

F0 = 0
F1 = 1
F2 = F1 + F0
F3 = F2 + F1
F4 = F3 + F2
F5 = F4 + F3
F6 = F5 + F4
F7 = F6 + F5
.
.
.
F10 = F9 +F8

MIPS組合語言暫存器的規定&
MIPS指令參考

MIPS組合語言暫存器的規定

名稱	暫存器號碼	用途	程序呼叫中是否被保留
$zero	0	Always 0	n.a.
$at	1	Reserved for assembler	No
$v0-$v1	2-3	Stores results	No
$a0-a3	4-7	Stores arguments	Yes
$t0-$t7	8-15	Temporaries,not saved	No
$s0-$s7	16-23	Contents saved for use later	Yes
$t8-$t9	24-25	More temporaries,not save	No
$ko-$k1	26-27	Reserved by operating system	No
$gp	28	Global pointer	Yes
$sp	29	Stack pointer	Yes
$fp	30	frame pointer	Yes
$ra	31	Return address	Yes

Arithmetic/Logic Operations

Arithmetic Operations			
ADD	R_D, R_S, R_T	$R_D = R_S + R_T$	(overflow trap)
ADDI	R_D, R_S, const16	$R_D = R_S +$ const16$^{\pm}$	(overflow trap)
ADDIU	R_D, R_S, const16	$R_D = R_S +$ const16$^{\pm}$	
ADDU	R_D, R_S, R_T	$R_D = R_S + R_T$	
CLO	R_D, R_S	$R_D =$ CountLeadingOnes(R_S)	
CLZ	R_D, R_S	$R_D =$ CountLeadingZeros(R_S)	
LA	R_D, label	$R_D =$ Address(label)	
LI	R_D, imm32	$R_D =$ imm32	
LUI	R_D, const16	$R_D =$ const16 << 16	
MOVE	R_D, R_S	$R_D = R_S$	
NEGU	R_D, R_S	$R_D = -R_S$	
SEB[R2]	R_D, R_S	$R_D = R_{S_{7:0}}^{\pm}$	
SEH[R2]	R_D, R_S	$R_D = R_{S_{15:0}}^{\pm}$	
SUB	R_D, R_S, R_T	$R_D = R_S - R_T$	(overflow trap)
SUBU	R_D, R_S, R_T	$R_D = R_S - R_T$	

Condition Testing and Conditional Move Operations

Condition Testing and Conditional Move Operations		
MOVN	R_D, R_S, R_T	IF $R_T \neq 0$, $R_D = R_S$
MOVZ	R_D, R_S, R_T	IF $R_T = 0$, $R_D = R_S$
SLT	R_D, R_S, R_T	$R_D = (R_S^{\pm} < R_T^{\pm})\ ?\ 1 : 0$
SLTI	R_D, R_S, const16	$R_D = (R_S^{\pm} < \text{const16}^{\pm})\ ?\ 1 : 0$
SLTIU	R_D, R_S, const16	$R_D = (R_S^{\varnothing} < \text{const16}^{\varnothing})\ ?\ 1 : 0$
SLTU	R_D, R_S, R_T	$R_D = (R_S^{\varnothing} < R_T^{\varnothing})\ ?\ 1 : 0$

Jump And Branch

Jumps And Branches (Note: One Delay Slot)		
B	$\text{OFF}18$	PC += $\text{OFF}18^{\pm}$
BAL	$\text{OFF}18$	R_A = PC + 8, PC += $\text{OFF}18^{\pm}$
BEQ	R_S, R_T, $\text{OFF}18$	IF R_S = R_T, PC += $\text{OFF}18^{\pm}$
BEQZ	R_S, $\text{OFF}18$	IF R_S = 0, PC += $\text{OFF}18^{\pm}$
BGEZ	R_S, $\text{OFF}18$	IF $R_S \geq 0$, PC += $\text{OFF}18^{\pm}$
BGEZAL	R_S, $\text{OFF}18$	R_A = PC + 8; IF $R_S \geq 0$, PC += $\text{OFF}18^{\pm}$
BGTZ	R_S, $\text{OFF}18$	IF $R_S > 0$, PC += $\text{OFF}18^{\pm}$
BLEZ	R_S, $\text{OFF}18$	IF $R_S \leq 0$, PC += $\text{OFF}18^{\pm}$
BLTZ	R_S, $\text{OFF}18$	IF $R_S < 0$, PC += $\text{OFF}18^{\pm}$
BLTZAL	R_S, $\text{OFF}18$	R_A = PC + 8; IF $R_S < 0$, PC += $\text{OFF}18^{\pm}$
BNE	R_S, R_T, $\text{OFF}18$	IF $R_S \neq R_T$, PC += $\text{OFF}18^{\pm}$
BNEZ	R_S, $\text{OFF}18$	IF $R_S \neq 0$, PC += $\text{OFF}18^{\pm}$
J	$\text{ADDR}28$	PC = $PC_{31:28}$ :: $\text{ADDR}28^{\varnothing}$
JAL	$\text{ADDR}28$	R_A = PC + 8; PC = $PC_{31:28}$ :: $\text{ADDR}28^{\varnothing}$
JALR	R_D, R_S	R_D = PC + 8; PC = R_S
JR	R_S	PC = R_S

Multiply and Divide Operations & Accumulator Access Operation

Multiply and Divide Operations		
DIV	Rs, Rt	Lo = Rs$^{\pm}$ / Rt$^{\pm}$; Hi = Rs$^{\pm}$ mod Rt$^{\pm}$
DIVU	Rs, Rt	Lo = Rs$^{\varnothing}$ / Rt$^{\varnothing}$; Hi = Rs$^{\varnothing}$ mod Rt$^{\varnothing}$
MADD	Rs, Rt	Acc += Rs$^{\pm}$ × Rt$^{\pm}$
MADDU	Rs, Rt	Acc += Rs$^{\varnothing}$ × Rt$^{\varnothing}$
MSUB	Rs, Rt	Acc −= Rs$^{\pm}$ × Rt$^{\pm}$
MSUBU	Rs, Rt	Acc −= Rs$^{\varnothing}$ × Rt$^{\varnothing}$
MUL	Rd, Rs, Rt	Rd = Rs$^{\pm}$ × Rt$^{\pm}$
MULT	Rs, Rt	Acc = Rs$^{\pm}$ × Rt$^{\pm}$
MULTU	Rs, Rt	Acc = Rs$^{\varnothing}$ × Rt$^{\varnothing}$

Accumulator Access Operations		
MFHI	Rd	Rd = Hi
MFLO	Rd	Rd = Lo
MTHI	Rs	Hi = Rs
MTLO	Rs	Lo = Rs

Lord and Store Operations

Load and Store Operations		
LB	R_D, off16(R_S)	$R_D = mem8(R_S + off16^{\pm})^{\pm}$
LBU	R_D, off16(R_S)	$R_D = mem8(R_S + off16^{\pm})^{\varnothing}$
LH	R_D, off16(R_S)	$R_D = mem16(R_S + off16^{\pm})^{\pm}$
LHU	R_D, off16(R_S)	$R_D = mem16(R_S + off16^{\pm})^{\varnothing}$
LW	R_D, off16(R_S)	$R_D = mem32(R_S + off16^{\pm})$
LWL	R_D, off16(R_S)	$R_D = LoadWordLeft(R_S + off16^{\pm})$
LWR	R_D, off16(R_S)	$R_D = LoadWordRight(R_S + off16^{\pm})$
SB	R_S, off16(R_T)	$mem8(R_T + off16^{\pm}) = R_{S_{7:0}}$
SH	R_S, off16(R_T)	$mem16(R_T + off16^{\pm}) = R_{S_{15:0}}$
SW	R_S, off16(R_T)	$mem32(R_T + off16^{\pm}) = R_S$
SWL	R_S, off16(R_T)	$StoreWordLeft(R_T + off16^{\pm}, R_S)$
SWR	R_S, off16(R_T)	$StoreWordRight(R_T + off16^{\pm}, R_S)$
ULW	R_D, off16(R_S)	$R_D = unaligned\_mem32(R_S + off16^{\pm})$
USW	R_S, off16(R_T)	$unaligned\_mem32(R_T + off16^{\pm}) = R_S$

Logical and Bit-Field Operations

Logical and Bit-Field Operations		
AND	R_D, R_S, R_T	$R_D = R_S \,\&\, R_T$
ANDI	R_D, R_S, const16	$R_D = R_S \,\&\, \text{const16}^{\varnothing}$
EXT[R2]	R_D, R_S, P, S	$R_S = R_{S_{P+S-1:P}}{}^{\varnothing}$
INS[R2]	R_D, R_S, P, S	$R_{D_{P+S-1:P}} = R_{S_{S-1:0}}$
NOP		No-op
NOR	R_D, R_S, R_T	$R_D = \sim(R_S \mid R_T)$
NOT	R_D, R_S	$R_D = \sim R_S$
OR	R_D, R_S, R_T	$R_D = R_S \mid R_T$
ORI	R_D, R_S, const16	$R_D = R_S \mid \text{const16}^{\varnothing}$
WSBH[R2]	R_D, R_S	$R_D = R_{S_{23:16}} :: R_{S_{31:24}} :: R_{S_{7:0}} :: R_{S_{15:8}}$
XOR	R_D, R_S, R_T	$R_D = R_S \oplus R_T$
XORI	R_D, R_S, const16	$R_D = R_S \oplus \text{const16}^{\varnothing}$