處理器設計與實作

實習講義

編撰者
成大電機已計算機架構與系統研究室CASLAB

國立成功大學電機系與電機所

大綱

1. 實驗目的
2. RISC-V組合語言(assembly language)介紹
   - 暫存器規定
   - RISC-V定址方法
   - 記憶體存取相關指令
   - 無條件跳轉指令
3. 實驗：用RISC-V組合語言描述 C code
   - 練習題（一）
   - 練習題（二）
   - 進階題
   - 實驗結報
4. 附件
   - 附件一：Lab1 modelsim驗證教學
   - 附件二：RV32I指令表

實驗目的

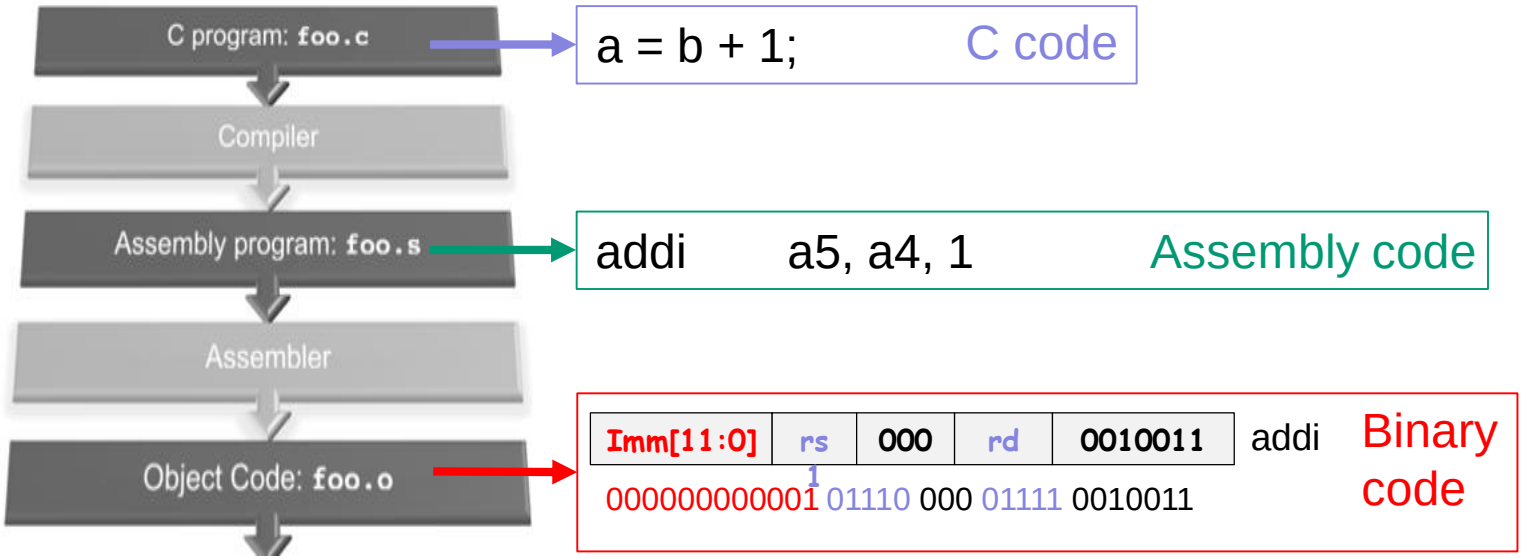
1. 了解 RISC-V 組合語言
2. 學習使用RISC-V 組合語言來實現對應的 C code，並在練習中使用組合語言來實作呼叫函式(function) 與存取結構(struct)
3. 練習在ModelSim 上驗證和檢查結果

RISC-V組合語言 (assembly language) 介紹

RISC-V 組合語言介紹

- **組合語言**(assembly language)：
  - 不同的硬體架構會有不同的指令集架構(ISA)
  - 而一種指令集架構，對應著一種組合語言
- 本次實驗所使用的RISC-V指令類型主要為：RV32I
  - 為 32 位元基礎整數指令集，有40多條指令
  - 支持 32 位元尋訪記憶體地址、32 位元整數暫存器

32bits integer



RV32I暫存器規定

Register name :
暫存器在CPU內的名字，
數字為該暫存器的編號

Symbolic name :
ABI 名字，寫assembly code 主要用這個名字。

Owner:
誰需要負責保存這個register的值。
Caller:呼叫者(上一層函數)
Callee:被呼叫者(現正執行的函數)

Register name	Symbolic name	Description	Owner
32 integer registers			
x0	Zero	Always zero	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5	t0	Temporary / alternate return address	Caller
x6–7	t1–2	Temporary	Caller
x8	s0/fp	Saved register / frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function argument / return value	Caller
x12–17	a2–7	Function argument	Caller
x18–27	s2–11	Saved register	Callee
x28–31	t3–6	Temporary	Caller

RV32I暫存器規定(cont.)

Saved register($s) &
Temporary register($t)

本質上都是儲存資料的暫存器。但使用上有一些差別：
Temporary register呼叫函式前後值會不一致，需要由caller來儲存；
而Saved register則由callee負責保存它的值。

舉例來說：
函式 A 中，有使用暫存器$t和$s，當它呼叫另一個函式B。
當B結束後回傳時，B會確保$s的值和原本A呼叫時一樣，但$t的值可能會改變。

Register name	Symbolic name	Description	Owner
32 integer registers			
x0	Zero	Always zero	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5	t0	Temporary / alternate return address	Caller
x6–7	t1–2	Temporary	Caller
x8	s0/fp	Saved register / frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function argument / return value	Caller
x12–17	a2–7	Function argument	Caller
x18–27	s2–11	Saved register	Callee
x28–31	t3–6	Temporary	Caller

RV32I暫存器規定(cont.)

x1/ra(Return address):
當函式結束後，便會返回到這個暫存器所儲存的位址。

x2/sp(Stack pointer):
當進入一個新的作用範圍(scope)，在記憶體中需要空出一段新的堆疊空間(stack)。此暫存器紀錄目前所空出的堆疊在記憶體中的頂部(下限)。

x8/s0(frame pointer):
此暫存器紀錄目前函式所使用的堆疊在記憶體中的底部(上限)。若不想記錄則可做為一般的save register。

Register name	Symbolic name	Description	Owner
32 integer registers			
x0	Zero	Always zero	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5	t0	Temporary / alternate return address	Caller
x6–7	t1–2	Temporary	Caller
x8	s0/fp	Saved register / frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function argument / return value	Caller
x12–17	a2–7	Function argument	Caller
x18–27	s2–11	Saved register	Callee
x28–31	t3–6	Temporary	Caller

RV32I暫存器規定(cont.)

Register name	Symbolic name	Description	Owner
32 integer registers			
x0	Zero	Always zero	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5	t0	Temporary / alternate return address	Caller
x6–7	t1–2	Temporary	Caller
x8	s0/fp	Saved register / frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function argument / return value	Caller
x12–17	a2–7	Function argument	Caller
x18–27	s2–11	Saved register	Callee
x28–31	t3–6	Temporary	Caller

x10-11/a0-1 (Function argument /return value):
呼叫函式時用來傳遞參數，和函式結束要返回時存入此暫存器。

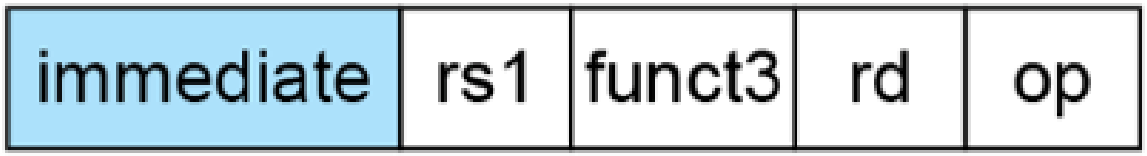
x12-17/a2-7 (Function argument):
呼叫函式時用來傳遞參數。

RISC-V定址模式

- 定址模式(addressing modes)：
  - 會決定指令所找到的運算數(operand)
  - 透過暫存器中的值或指令中的常數來找到記憶體位址
- RISC-V定址模式主要分為四種方法：
  - Immediate addressing
  - Register addressing
  - Base displacement addressing
  - PC-relative addressing

RISC-V定址模式(cont.)

- 定址模式(addressing modes) ：
  - 會決定指令所找到的運算數(operand)
  - 透過暫存器中的值或指令中的常數來找到記憶體位址
- RISC-V定址模式主要分為四種方法：
  - Immediate addressing
  - Register addressing
  - Base displacement addressing
  - PC-relative addressing

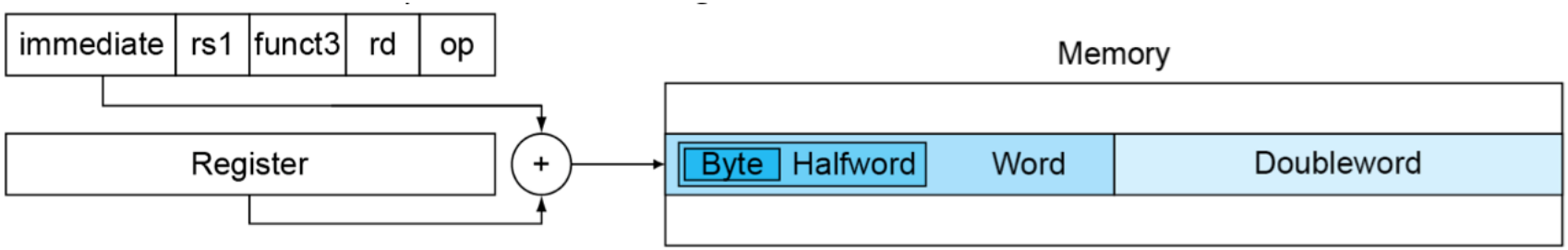


RISC-V定址模式(cont.)

- 定址模式(addressing modes) ：
  - 會決定指令所找到的運算數(operand)
  - 透過暫存器中的值或指令中的常數來找到記憶體位址
- RISC-V定址模式主要分為四種方法：
  - Immediate addressing
  - Register addressing
  - Base displacement addressing
  - PC-relative addressing

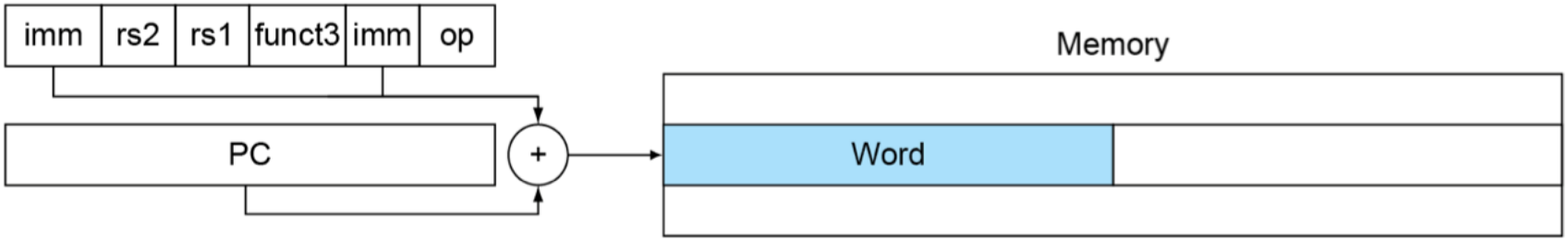
RISC-V定址模式(cont.)

- 定址模式(addressing modes) ：
  - 會決定指令所找到的運算數(operand)
  - 透過暫存器中的值或指令中的常數來找到記憶體位址
- RISC-V定址模式主要分為四種方法：
  - Immediate addressing
  - Register addressing
  - Base displacement addressing
  - PC-relative addressing



RISC-V定址模式(cont.)

- 定址模式(addressing modes) ：
  - 會決定指令所找到的運算數(operand)
  - 透過暫存器中的值或指令中的常數來找到記憶體位址
- RISC-V定址模式主要分為四種方法：
  - Immediate addressing
  - Register addressing
  - Base displacement addressing
  - PC-relative addressing



RV32I記憶體存取相關指令

- Load/Store 指令：
  - 利用Load/Store指令在記憶體和暫存器之間傳輸數據
  - 其它指令都是在暫存器中和立即數(immediate)之間運算
- 使用注意事項：
  - 目標暫存器不能為x0（恆為0）
  - 記憶體位址：將立即數展開為32位與rs1中的值相加所得
    - 立即數的負數表示為2進位的補數
  - Load/Store指令要盡量對齊地址
    - Eg. lw(load word)指令所訪問的地址應該與4 byte對齊

32-bit RISC-V instruction formats

Format	Bit																													
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode					
Immediate	imm[11:0]												rs1					funct3			rd					opcode					
Upper immediate	imm[31:12]																				rd					opcode					
Store	imm[11:5]							rs2					rs1					funct3			imm[4:					opcode					
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode					
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode					

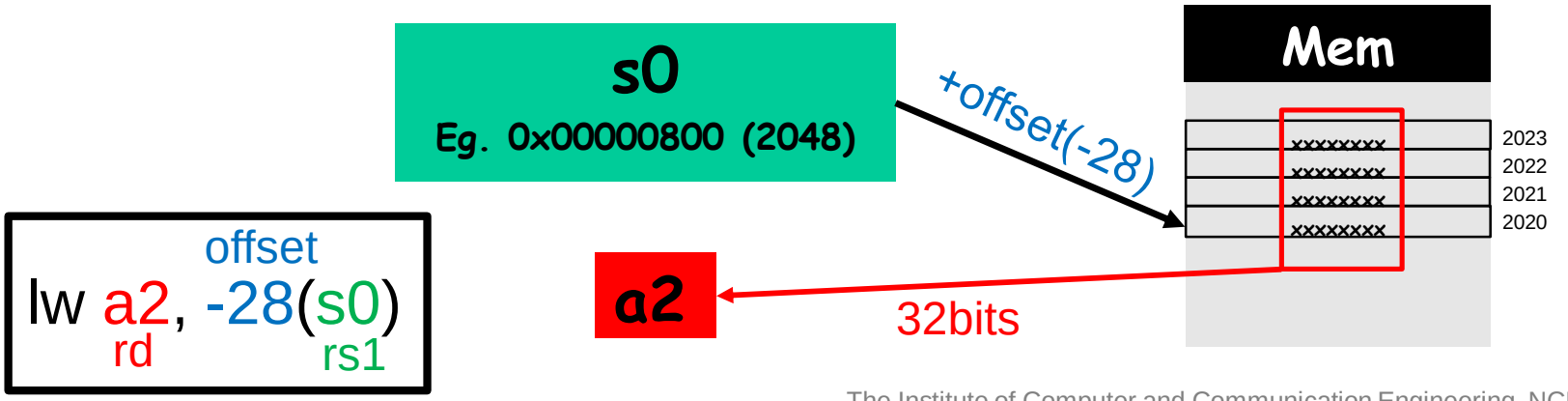
Load

Store

RV32I記憶體存取相關指令-Load

12	5	3	5	0000011
Imm(offset)	rs1	func3	rd	opcode

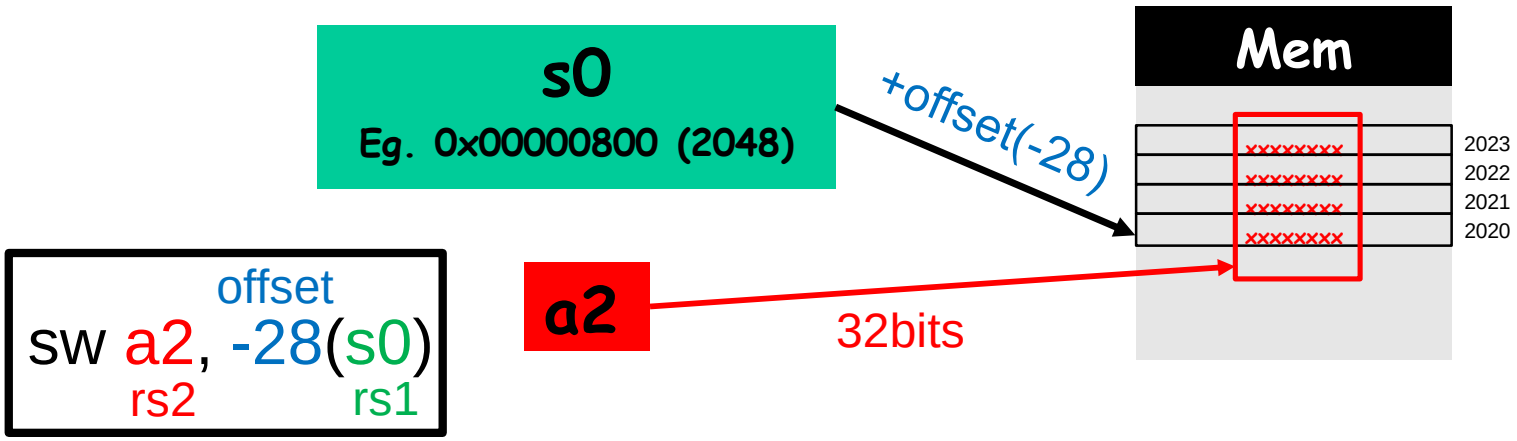
func3	Assembly	解釋
000	lb rd, offset(rs1)	Load byte 讀8bits符號位擴展寫回rd
001	lh rd, offset(rs1)	Load half word 讀16bits符號位擴展寫回rd
010	lw rd, offset(rs1)	Load word 讀32bits寫回rd
100	lbu rd, offset(rs1)	Load byte unsigned 讀8bits高位補0寫回rd
101	lhu rd, offset(rs1)	Load half word unsigned 讀16bits高位補0寫回rd



RV32I記憶體存取相關指令-Store

7	5	5	3	5	0100011
Imm(offset)	rs2	rs1	func3	Imm(offset)	opcode

func3	Assembly	解釋
000	sb rs2, offset(rs1)	Store byte 將rs2中最低8bits寫回記憶體
001	sh rs2, offset(rs1)	Store half word 將rs2中最低16bits寫回記憶體
010	sw rs2, offset(rs1)	Store word 將rs2中最低32bits寫回記憶體



RV32I無條件跳轉指令

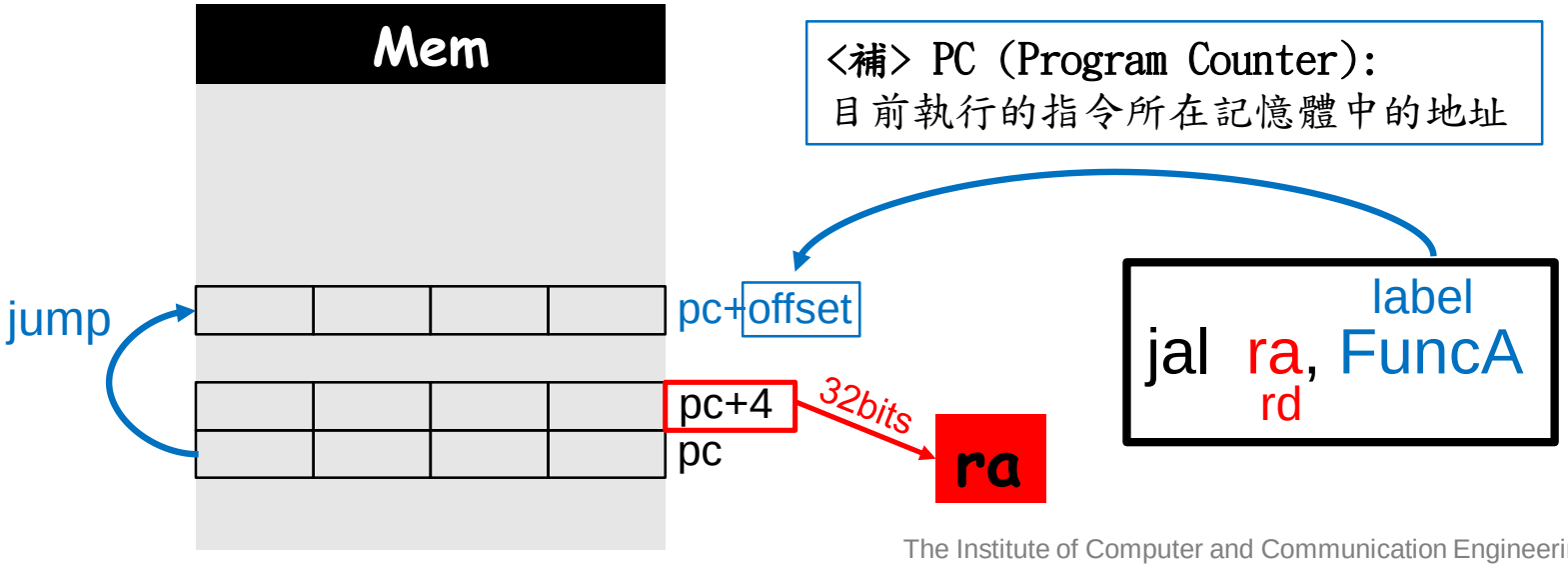
- RV32I 提供兩類型跳轉指令：
  - 無條件跳轉指令(Unconditional Jumps)
  - 有條件跳轉指令(Conditional branches)(附件)
- 無條件跳轉指令：
  - 顧名思義，該指令一定會發生跳轉(jump)
  - 使用時機通常為函式呼叫
  - 呼叫函式時要注意呼叫慣例(Calling Convention)
- 有兩個指令：
  - jal (jump and link)
  - jalr (jump and link register)

<補>呼叫慣例(Calling Convention):
也就是前面所提到暫存器的owner。
呼叫者(caller)和被呼叫者(callee)之間有一套規定，如何傳遞參數、誰該保存暫存器的值等等

RV32I無條件跳轉指令-jal

offset[20:1]				5	1101111
Imm[20]	Imm[10:1]	Imm[1	Imm[19:1	rd	opcode

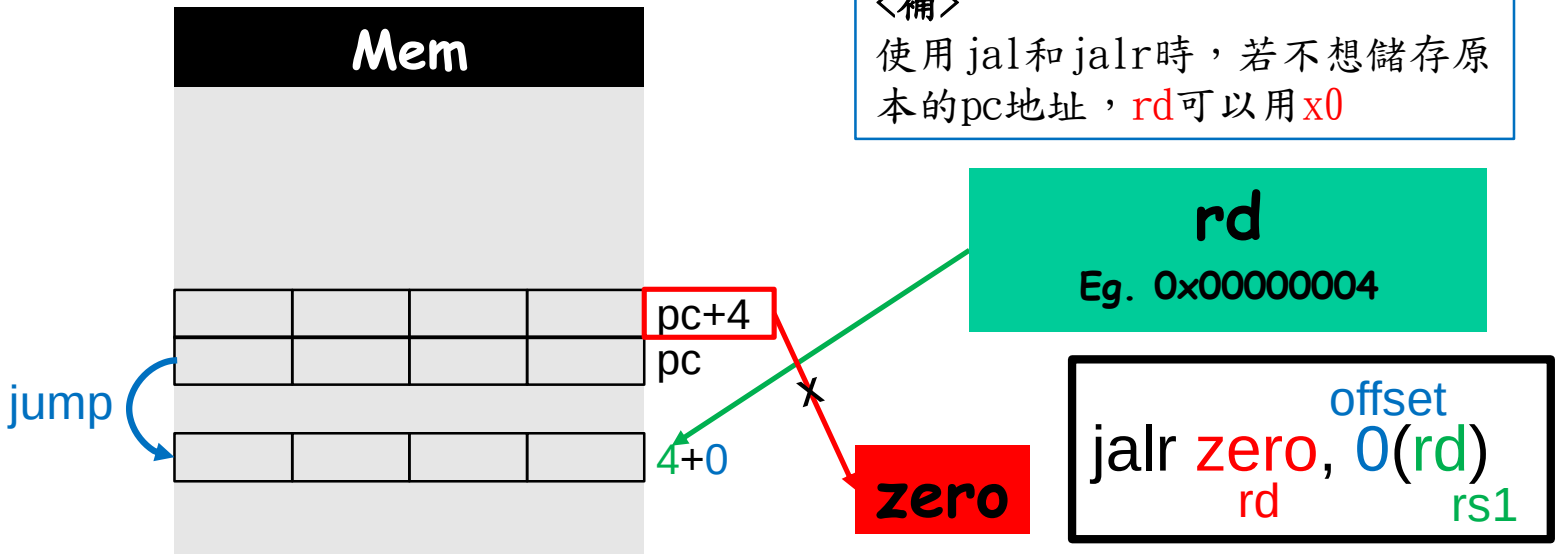
Assembly	解釋
jal rd, label	1. 將offset 符號位擴展，與該指令的pc相加，即為跳轉的目標地址 2. 將下一條指令的PC(當前PC+4)寫入rd (通常用ra儲存) 指令長度為一個32bits(4 bytes) 注意：在實際編寫組合語言中，跳轉的目標通常是使用label，組譯器(assembler)會根據label所在的地址算出offset，再轉成二進位指令編碼



RV32I無條件跳轉指令-jalr

12	5	3	5	1100111
Imm(offset)	rs1	func3	rd	opcode

func3	Assembly	解釋
000	jalr rd, (offset)rs1	1. 將立即數(imm)作為signed offset符號位擴展與暫存器rs1相加，並將最低有效位設為0，得到跳轉目標地址 2. 同jal，將下一條指令的PC(當前PC+4)寫入rd (通常用ra儲存)



實驗：用 RISC-V 描述 C code

實驗環境

- Windows
  - Modelsim (Run CPU simulator)
  - VirtualBox (Linux Ubuntu18.04 x86)
    - RISC-V Toolchain (Compile program)

實驗概述

- 使用RISC-V 組合語言來撰寫練習題所提供的 C 程式
- 遵照 Lab1 的流程將檔案編譯
- 將編譯後的程式碼使用 Modelsim 進行模擬
- 將 RISC-V Assembly code 儲存下來並將結果截圖

實驗步驟

- Windows
  1. 在實驗室網站下載Lab2.zip並解壓縮
  2. 打開VirtualBox
  3. 將Lab2放入共享資料夾
- VirualBox
  4. 找到Lab2，進到practice_x(練習1 or 2)
  5. 會有一個檔名.c檔
     - (p1:max_return.c / p2:data_struct.c)
  6. 照著 C code 編寫自己的Assembly code
  7. 將寫好的Assembly code 存檔
     - 存成：檔名.s

檔名
practice_1: max_return
practice_2: data_struct

實驗步驟(cont.)

- VirualBox
  8. 按照Lab1的步驟編譯.s檔
     1) $ riscv32-unknown-elf-as -mabi=ilp32 檔名.s -o 檔名.o
     2) $ riscv32-unknown-elf-ld -b elf32-littleriscv -T link.ld 檔名.o -o 檔名
     3) $ riscv32-unknown-elf-objdump -dC 檔名 > 檔名.dump
     4) $ riscv32-unknown-elf-objcopy -O binary 檔名 檔名.bin
     5) $ python3 bin2mem.py --bin 檔名.bin (產生MEM檔)
- Windows
  9. 確認共享資料夾practice_x有檔名的MEM檔
  10. 接著請參考Lab1 “Modelsim驗證教學”
      - 放在最後面的附件一

練習題(一) C code

```
#include<stdio.h>

int sum(int,int,int);
int main()
{
    int a=44,b=87,c=2;
    volatile int* n = (int*) 0x00000800;
    *n=sum(a,b,c);
     return 0;
}

int sum(int a,int b,int c)
{
    int n;
    n=a+b+c;
    return n;
}
```

函式sum的運算結果會回傳存到記憶體位置2048

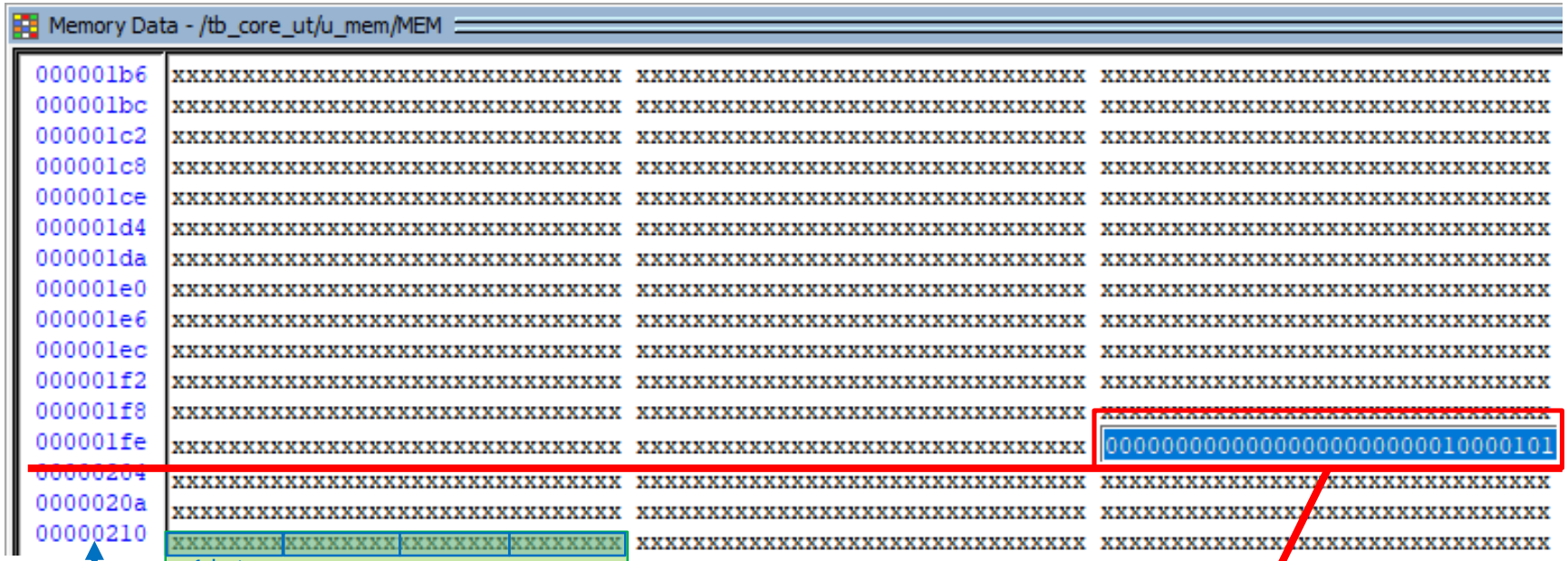
練習題(一) Assembly code hint

```
1   main:
2
3       #### 剛進入main function ####
4       addi    sp,sp,-32   #進入main function, 空出一個stack的空間(sp作為main stack的底部)
5       sw      ra,28(sp)   #將ra存起來(owner:caller -> 之後return 0要用)
6       sw      s0,24(sp)   #將s0存起來(owner:callee -> callee要負責維持save register的值)
7       addi    s0,sp,32    #拿s0作為frame pointer(s0作為main stack的頂部)
8
9       #### 宣告 int a ####
10      addi    t1,zero,44  #
11      sw      t1,-20(s0)  # int a 被我放到(s0-20)這個記憶體位址
12
13      #### 宣告 int b,c ####
14
15
16      #### 宣告 int *n ####
17
18
19      #### 準備function arguments ####
20
21
22      #### 進入sum function (使用跳轉指令) ####
23
24
25      #### 將拿到的回傳值(放在a0)放進*n ####
26
27
28      #### 結束main function ####
29      lw      ra,28(sp)   # 拿回ra(owner:caller -> 拿回自己保存的值)
30      lw      s0,24(sp)   # 拿回s0(owner:callee -> callee要負責維持save register的值)
31      addi    sp,sp,32    # 釋出main的stack
32      jalr    zero,0(ra)  # 跳出main(使用跳轉指令)
33
```

練習題(一) Assembly code hint (cont.)

```
34  sum:
35      #### 剛進入sum function ####
36
37
38      #### 儲存function arguments ####
39
40
41      #### 跑 sum function (n = a+b+c), 並將回傳值n放到a0 ####
42
43
44      #### 結束sum function ####
45
46
```

練習題(一) 結果



位址：0x00000200(512)
數值：133($2^7+2^2+2^0$)

<補>
我們在寫assembly code計算記憶體位址是以byte(8bits)為單位，而Modelsim中的memory是以word(32bits)為單位。而因此在判斷位址的時候要除以4

練習題(二) C code

```
#include <stdio.h>

struct student{
    int mathGrade;
    int csGrade;
    int englishGrade;
};
int main()
{
    volatile struct student* A =(struct student*) 0x00000800;
    volatile struct student* B =(struct student*) 0x00000820;
    struct student s1 = {60,70,70};
    struct student s2 = {70,50,80};

    *A = s1;
    *B = s2;
    return 0;
}
```

將宣告的兩個struct分別放在記憶體位址2048和2080

練習題(二) Assembly code hint

```
1  main:
2          #### 剛進入main function(注意stack的大小)####
3
4
5          #### 宣告stuct student* A ####
6          addi    a5,zero,1024
7          addi    a5,a5,1024
8          sw      a5,-20(s0)
9
10         #### 宣告stuct student* B ####
11
12
13         #### 宣告stuct student s1 ####
14         addi    a5,zero,60
15         sw      a5,-36(s0)
16         addi    a5,zero,70
17         sw      a5,-32(s0)
18         addi    a5,zero,70
19         sw      a5,-28(s0)
20
21         #### 宣告stuct student s2 ####
22
23
```

練習題(二) Assembly code hint (cont.)

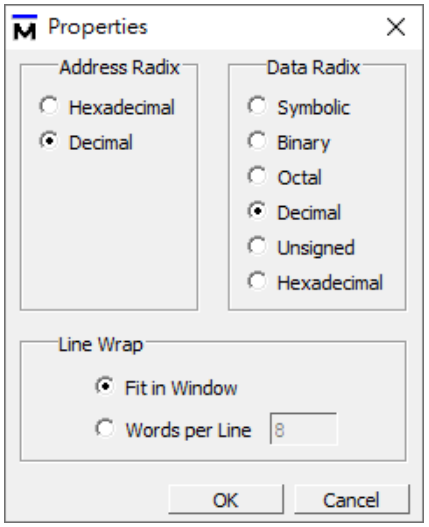
```
24          #### *A = s1 ####
25          lw      a5,-20(s0)      #拿到*A
26          lw      a4,-36(s0)      #將s1的值一個一個丟進去
27          sw      a4,0(a5)
28          lw      a4,-32(s0)
29          sw      a4,4(a5)
30          lw      a4,-28(s0)
31          sw      a4,8(a5)
32
33          #### *B = s2 ####
34
35
36          #### 結束 main function ####
37
38
```

練習題(二) 結果

Memory Data - /tb_core_ut/u_mem/MEM - Default

374	x	x	x	x	x	x	x	x	x	x	x	x	x
391	x	x	x	x	x	x	x	x	x	x	x	x	x
408	x	x	x	x	x	x	x	x	x	x	x	x	x
425	x	x	x	x	x	x	x	x	x	x	x	x	x
442	x	x	x	x	x	x	x	x	x	x	x	x	x
459	x	x	x	x	x	x	x	x	x	x	x	x	x
476	x	x	x	x	x	x	x	x	x	x	x	x	x
493	x	x	x	x	x	x	x	x	x	x	x	x	x
510	x	x	60	70	70	x	x	x	x	x	70	50	80
527	x	x	x	x	x	x	x	x	x	x	x	x	x
544	x	x	x	x	x	x	x	x	x	x	x	x	x

<補>
如果一直看二進位覺得麻煩，可以選擇memory視窗後，點選：
View -> Properties 可以改進位

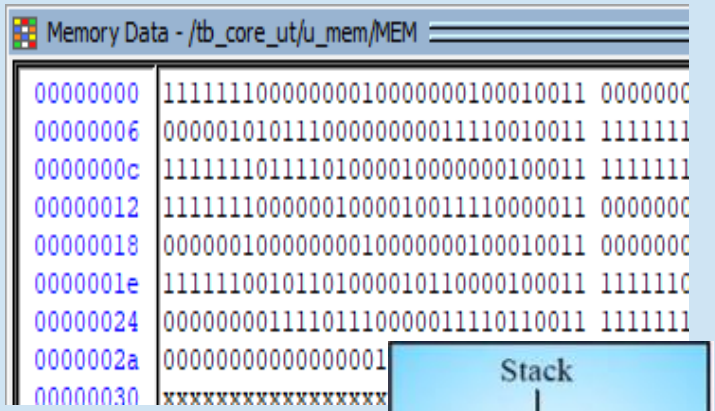


其它觀察

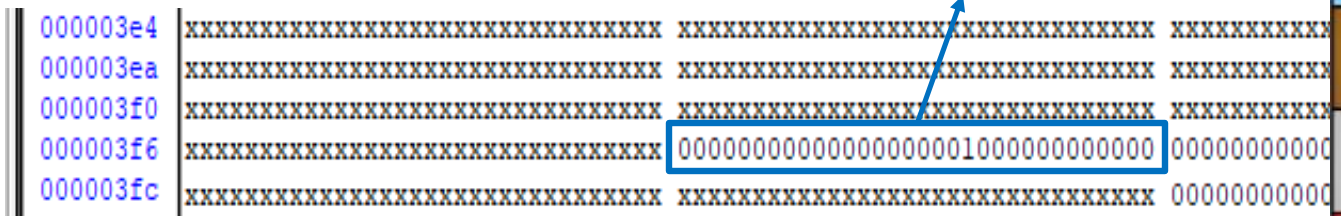
1. 在testbench.sv裡面有對CPU內的register做一些事前設定

```
// Setup Return Address (ra/x1)
`REG[x1] = 944;

// Store while(1); at 944
// 944 = 0x3B0, word 236
u_mem.MEM[944>>2] = rv32_jal(x0, 0); // j 1b

// Setup Frame Pointer (s0/x8)
`REG[x8] = 0;

// Setup Stack Pointer (sp/x2) to the end of the memory (4KB).
`REG[x2] = 4096;
```

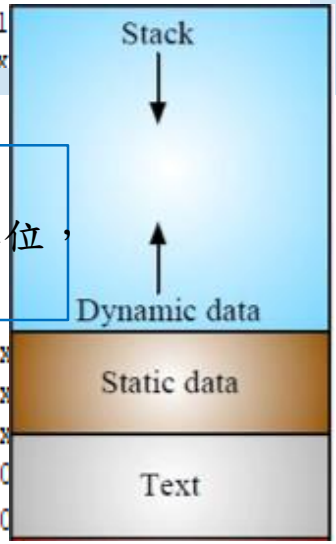
2. 指令(instruction)在記憶體中是從位址小的地方開始儲存



3. 函式的區域變數則是從位址大的地方儲存



<補> 位元組順序(Endianness)：
此處為大端序(big-endian)，以byte為單位，高位byte儲存在最低的記憶體位址



進階題

請trace以下assembly code

1. 寫出每次迴圈執行後memory的變化(共四次)
2. 簡單描述此code意義為何

challenge.s

```
main:
        addi    sp,sp,-48
        sw      s0,44(sp)
        addi    s0,sp,48
        li      a5,4096
        addi    a5,a5,-2048
        sw      a5,-28(s0)
        li      a5,5
        sw      a5,-32(s0)
        lw      a5,-28(s0)
        li      a4,10
        sw      a4,0(a5)
        lw      a5,-28(s0)
        addi    a5,a5,4
        li      a4,92
        sw      a4,0(a5)
        lw      a5,-28(s0)
        addi    a5,a5,8
        li      a4,55
        sw      a4,0(a5)
        lw      a5,-28(s0)
        addi    a5,a5,12
        li      a4,1
        sw      a4,0(a5)
        lw      a5,-28(s0)
        addi    a5,a5,16
        li      a4,46
        sw      a4,0(a5)
        sw      zero,-20(s0)
        j       .L2
.L6:
        sw      zero,-24(s0)
        j       .L3
.L5:
        lw      a5,-24(s0)
        addi    a5,a5,1
        slli    a5,a5,2
        lw      a4,-28(s0)
        add     a5,a4,a5
        lw      a4,0(a5)
        lw      a5,-24(s0)
        slli    a5,a5,2
        lw      a3,-28(s0)
        add     a5,a3,a5
        lw      a5,0(a5)
        bge     a4,a5,.L4
        lw      a5,-24(s0)
        slli    a5,a5,2
        lw      a4,-28(s0)
        add     a5,a4,a5
        lw      a5,0(a5)
        sw      a5,-36(s0)
        lw      a5,-24(s0)
        addi    a5,a5,1
        slli    a5,a5,2
        lw      a4,-28(s0)
        add     a4,a4,a5
        lw      a5,-24(s0)
        slli    a5,a5,2
        lw      a3,-28(s0)
        add     a5,a3,a5
        lw      a4,0(a4)
        sw      a4,0(a5)
        lw      a5,-24(s0)
        addi    a5,a5,1
        slli    a5,a5,2
        lw      a4,-28(s0)
        add     a5,a4,a5
        lw      a4,-36(s0)
        sw      a4,0(a5)
.L4:
        lw      a5,-24(s0)
        addi    a5,a5,1
        sw      a5,-24(s0)
.L3:
        lw      a5,-32(s0)
        addi    a4,a5,-1
        lw      a5,-20(s0)
        sub     a5,a4,a5
        lw      a4,-24(s0)
        blt     a4,a5,.L5
        lw      a5,-20(s0)
        addi    a5,a5,1
        sw      a5,-20(s0)
.L2:
        lw      a5,-32(s0)
        addi    a5,a5,-1
        lw      a4,-20(s0)
        blt     a4,a5,.L6
        li      a5,0
        mv      a0,a5
        lw      s0,44(sp)
        addi    sp,sp,48
        jr      ra
```

實驗結報

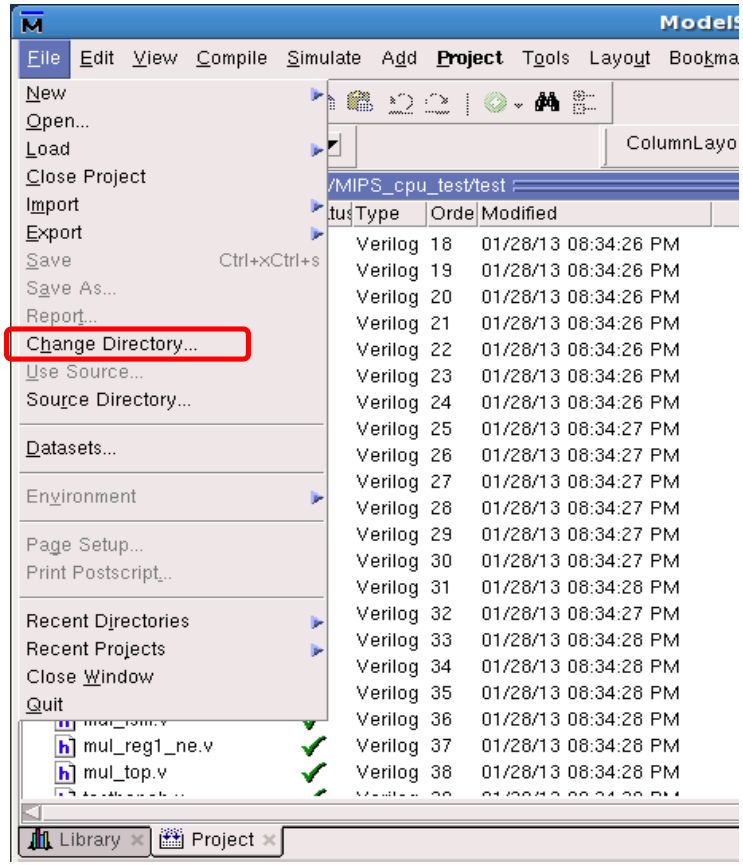
- 結報格式(每組一份)
  - 封面
  - 實驗內容(程式碼註解、結果截圖)
  - 實驗心得
- 繳交位置
  - ftp : 140.116.164.225 port : 21
  - 帳號/密碼 : ca_lab / Carch2020
- DeadLine: 10/12 18:00前
- TA Contact Information:
  - 助教信箱 : ericwang0911@gmail.com
  - Rm 92617
  - Office hour : (Tuesday) 8:00pm~10:00pm

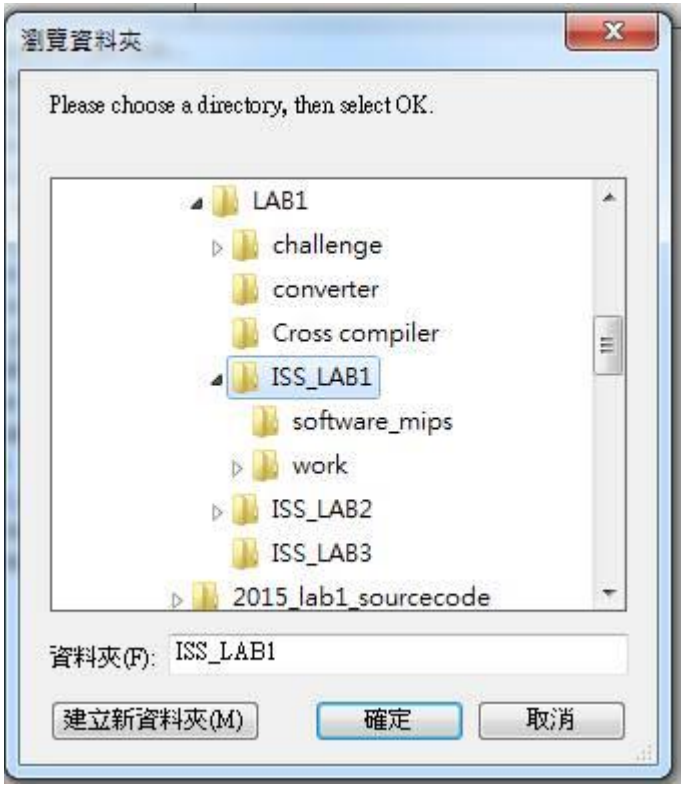
附件一： Labl modelsim驗證教學

Modelsim 驗證教學

Step.1: change to your file location

打開modelsim後,在File下選擇change directory到你放*software_mips*跟*work*資料夾的地方

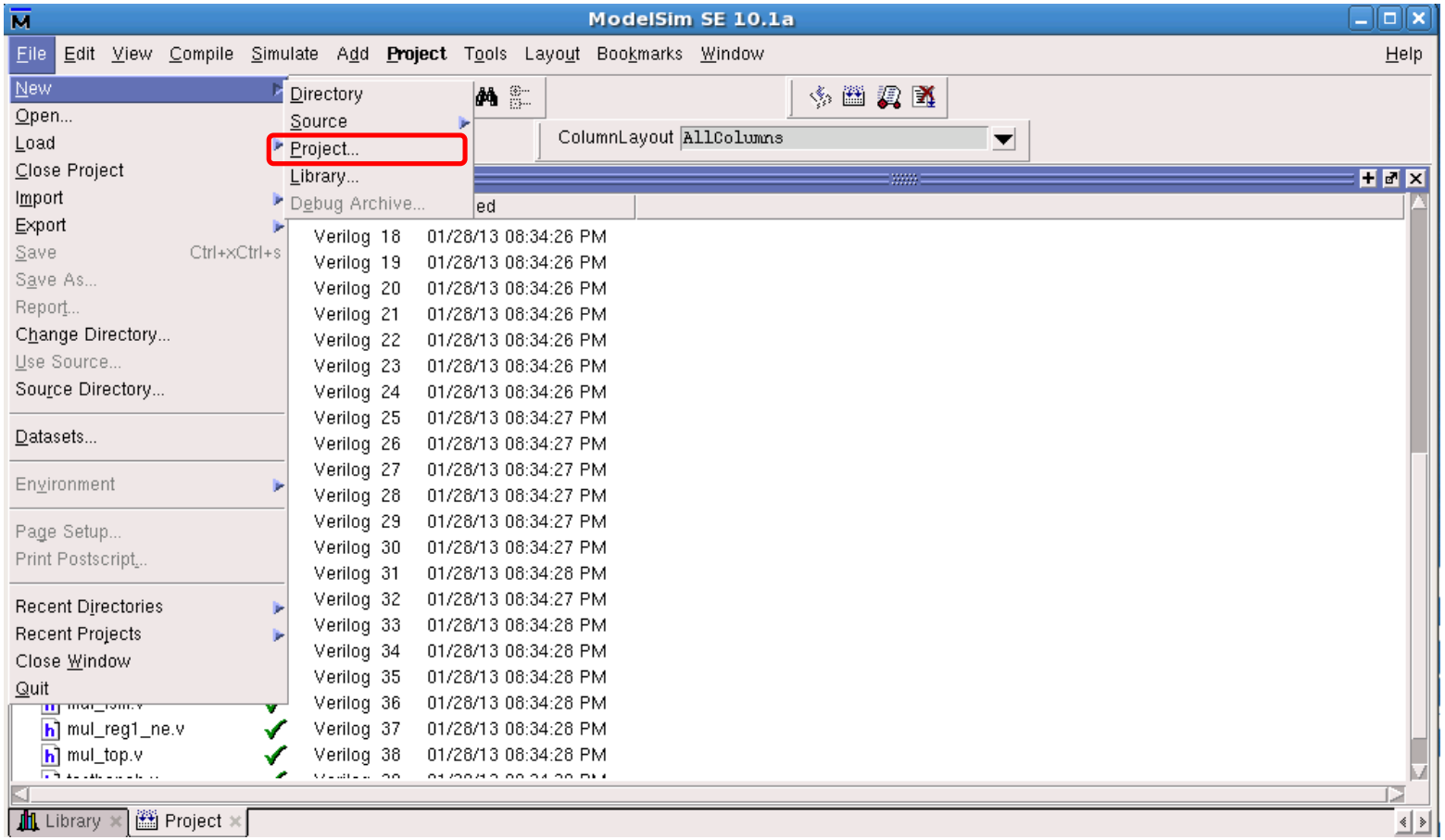




Modelsim 驗證教學

Step.2: new project

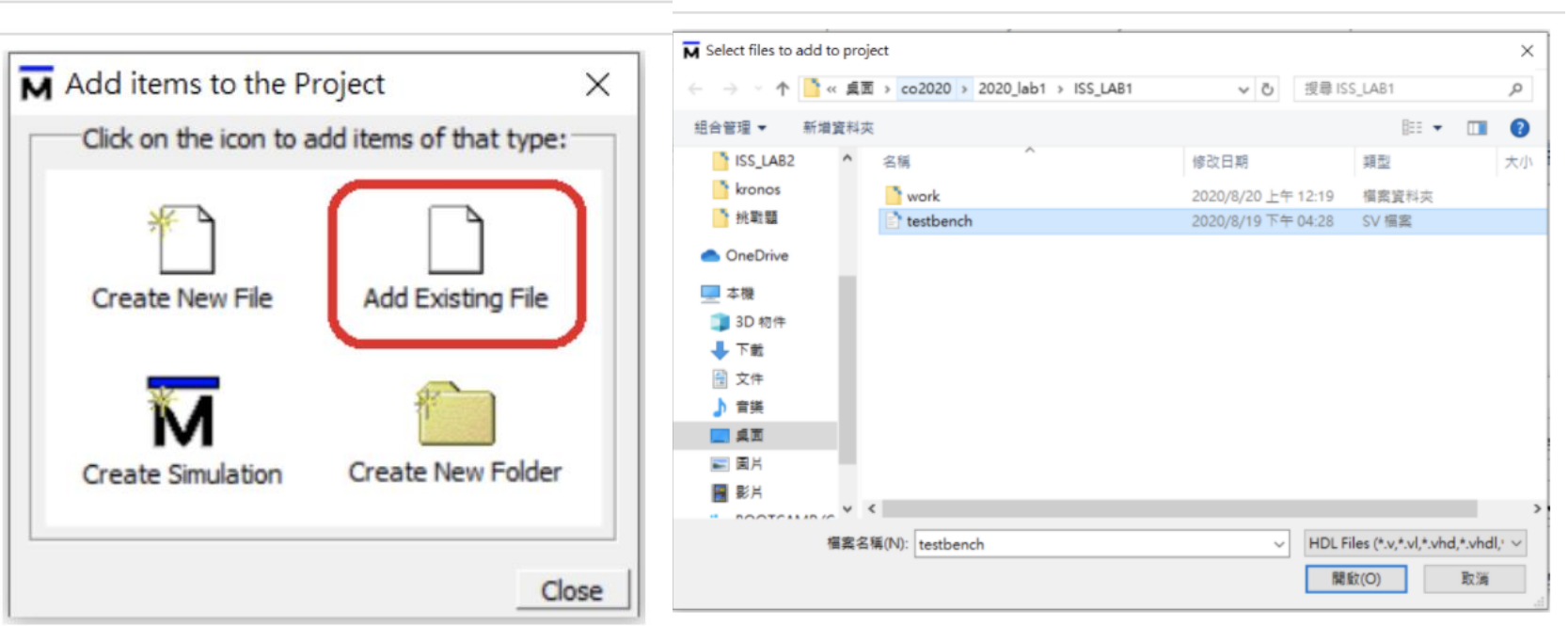
接著new一個project(名稱自訂)



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

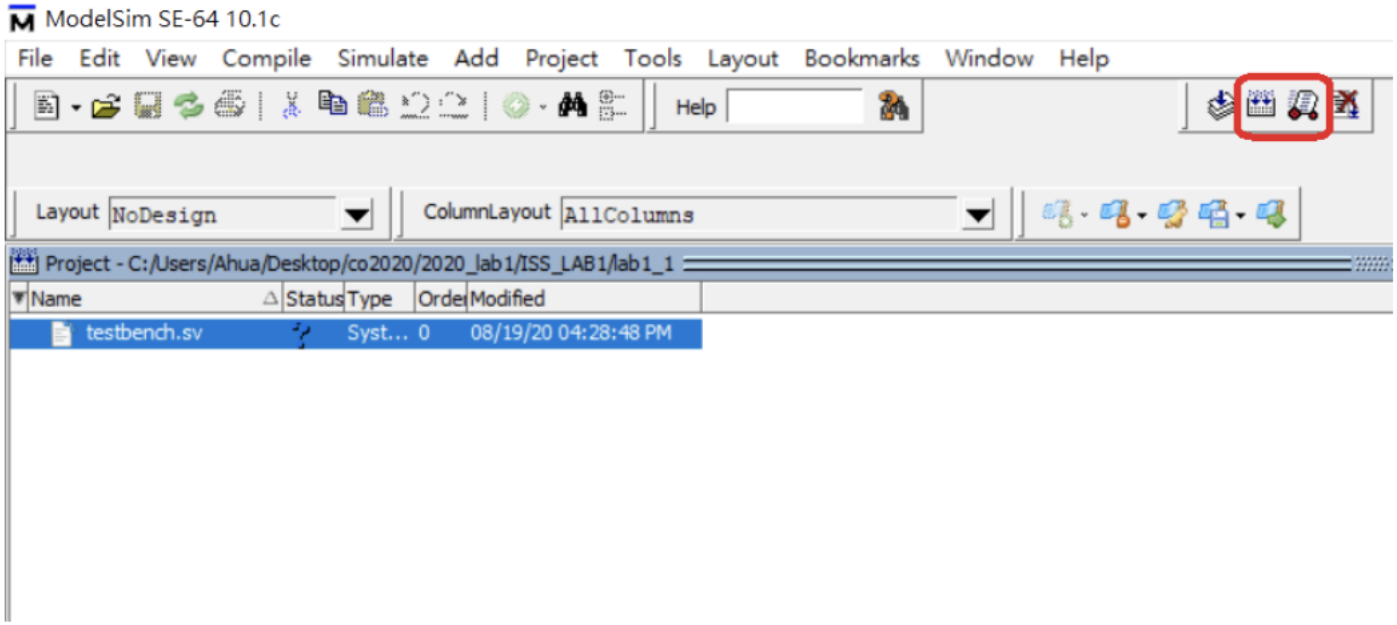


Modelsim 驗證教學

Step.4: compile and simulate source code

點擊compile 

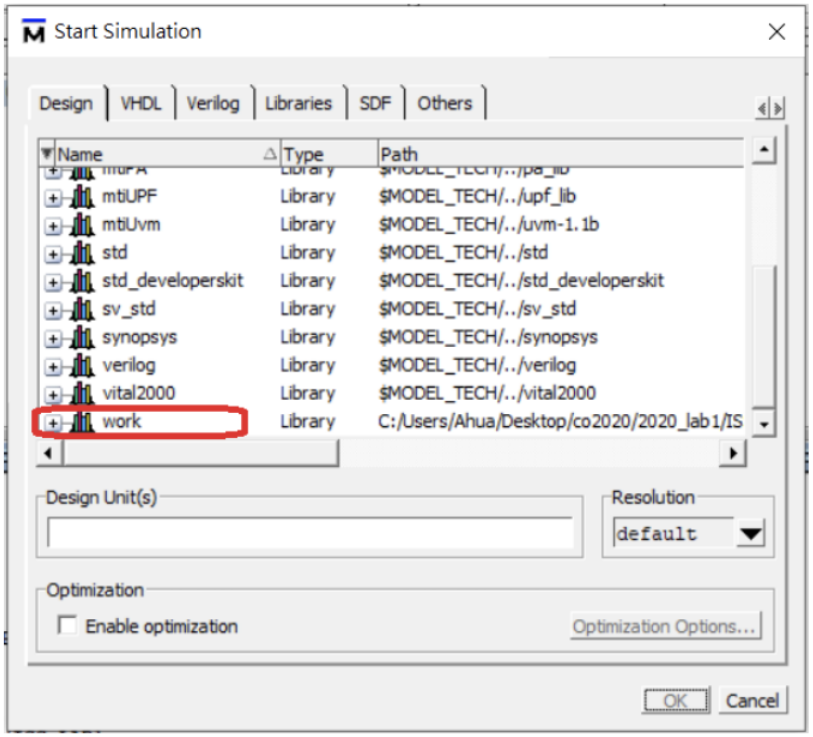
接著點擊*simulate* 

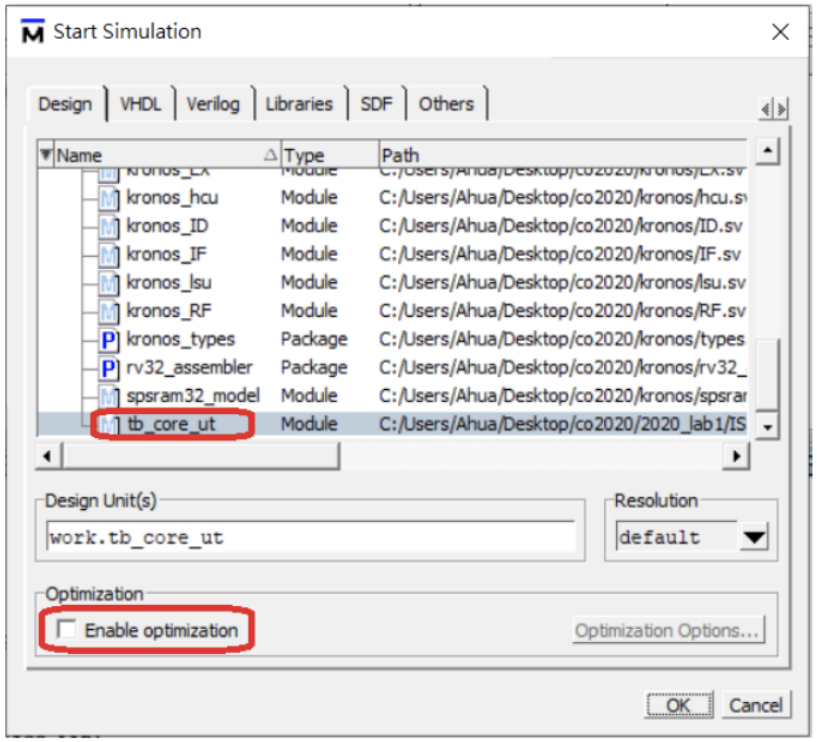


Modelsim 驗證教學

Step.5: choose testbench and disable optimization

打開work後選testbench並取消最佳化



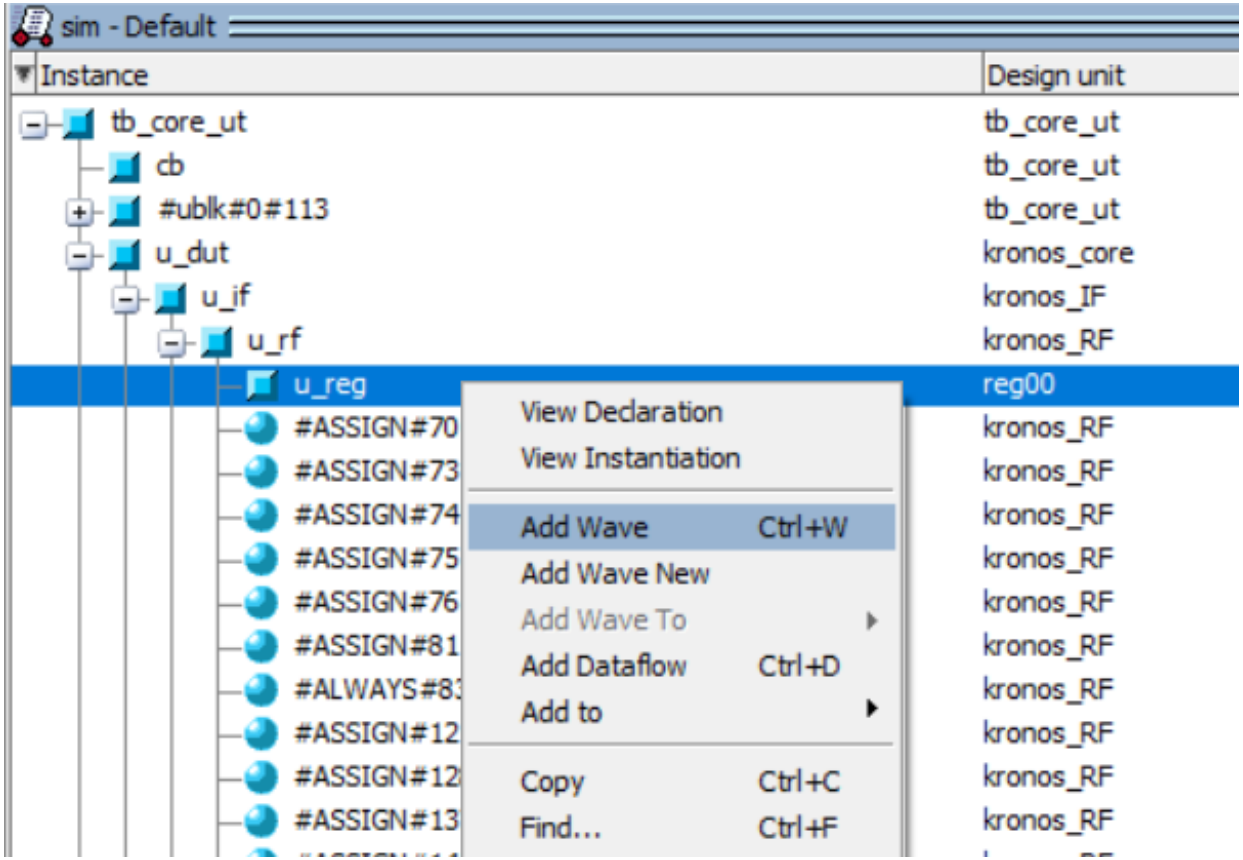


Modelsim 驗證教學

Step.6: add signal to wave

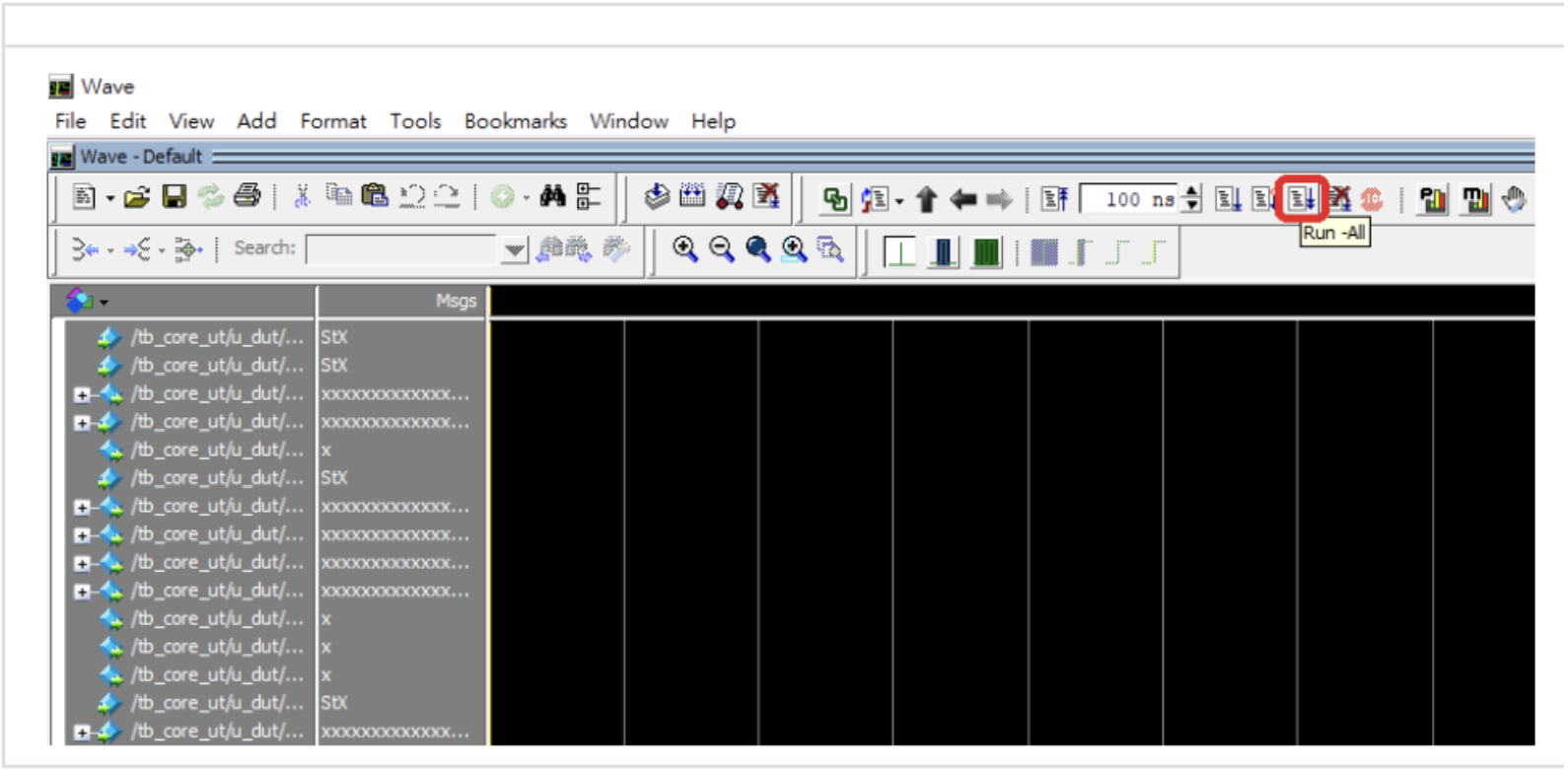
將你要看的訊號線按右鍵add wave

tb_core_ut > u_dut > u_if > u_rf > u_reg 右鍵 Add Wave



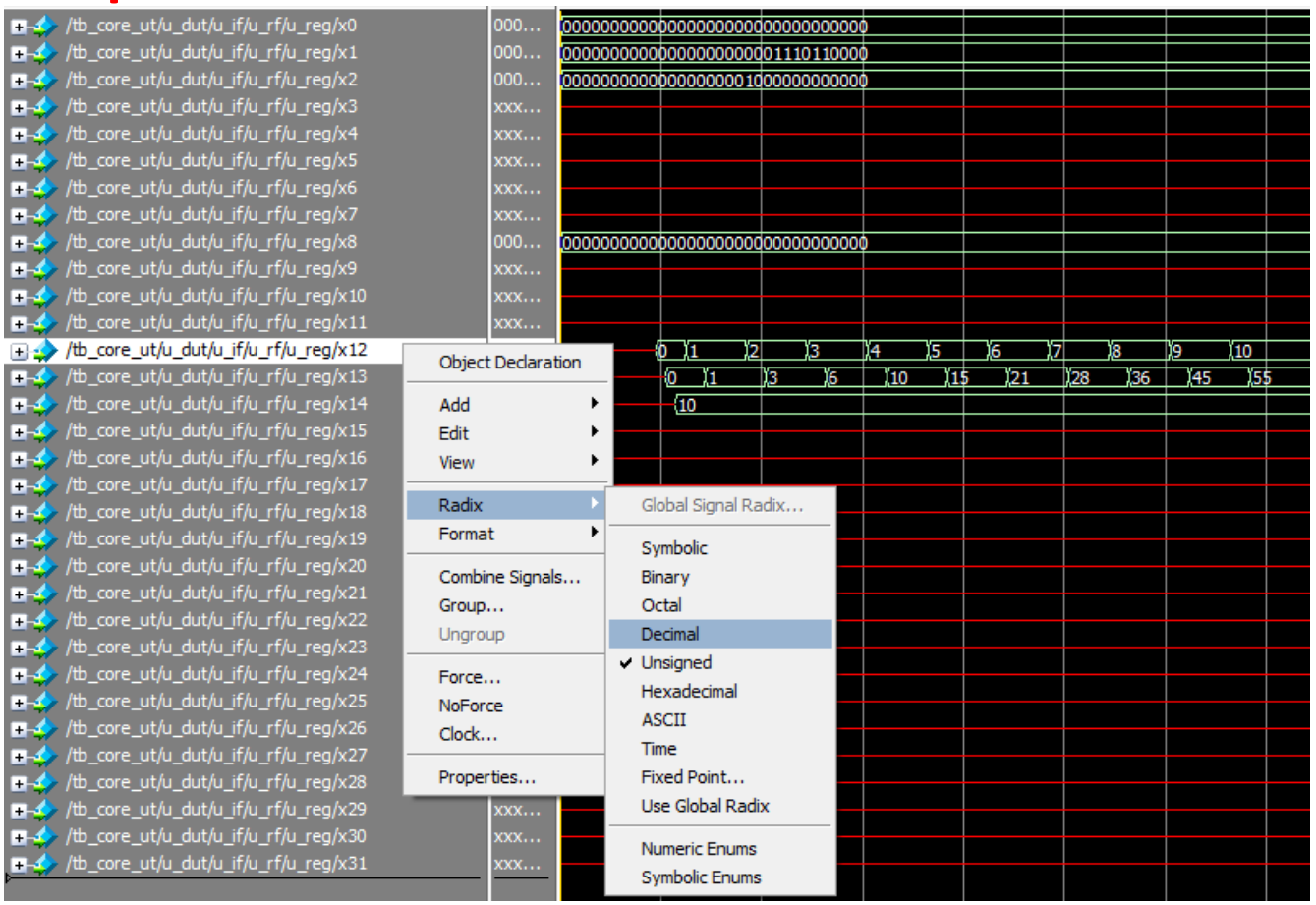
Modelsim 驗證教學

Step.7: run all



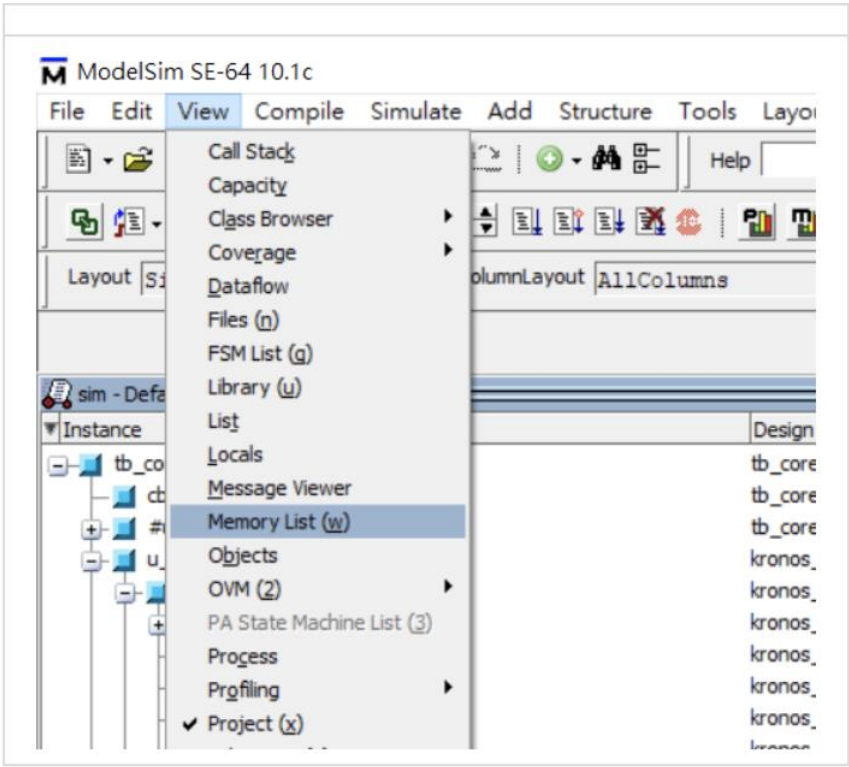
Modelsim 驗證教學

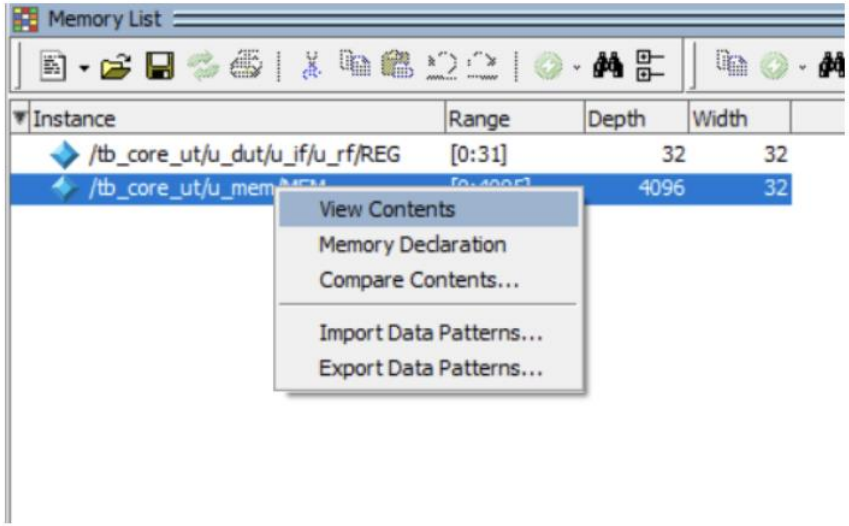
Step.8: check wave result



Modelsim 驗證教學

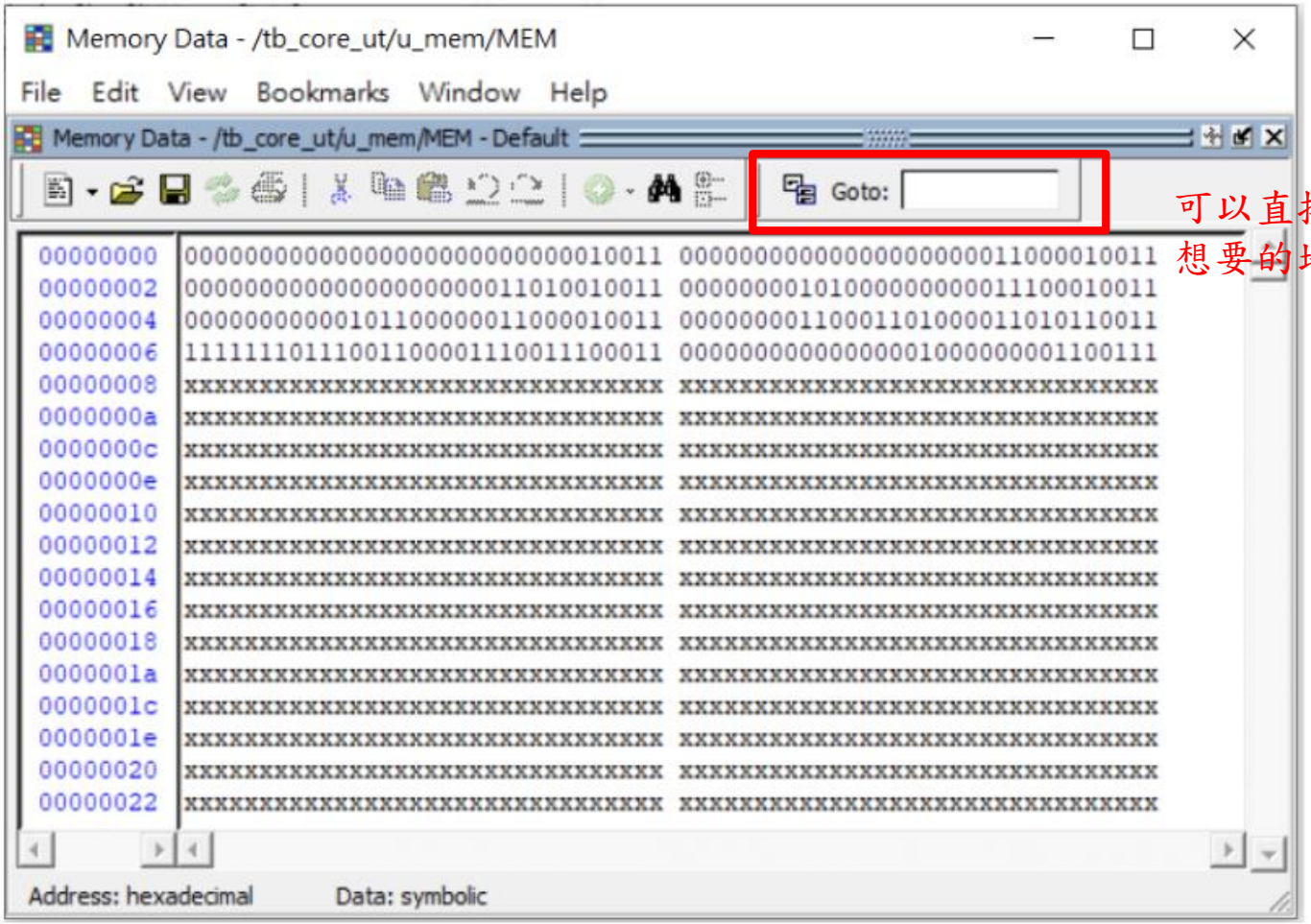
Step.9: view->memory list





Modelsim 驗證教學

Step.10: check memory result



可以直接跳到想要的地址

附件二： RV32I指令表

可能用到的指令

Category	Name	Fmt	RV32I Base	
Shifts	Shift Left Logical	R	SLL	rd,rs1,rs2
	Shift Left Log. Imm.	I	SLLI	rd,rs1,shamt
	Shift Right Logical	R	SRL	rd,rs1,rs2
	Shift Right Log. Imm.	I	SRLI	rd,rs1,shamt
	Shift Right Arithmetic	R	SRA	rd,rs1,rs2
	Shift Right Arith. Imm.	I	SRAI	rd,rs1,shamt
Arithmetic	ADD	R	ADD	rd,rs1,rs2
	ADD Immediate	I	ADDI	rd,rs1,imm
	SUBtract	R	SUB	rd,rs1,rs2
	Load Upper Imm	U	LUI	rd,imm
	Add Upper Imm to PC	U	AUIPC	rd,imm
Logical	XOR	R	XOR	rd,rs1,rs2
	XOR Immediate	I	XORI	rd,rs1,imm
	OR	R	OR	rd,rs1,rs2
	OR Immediate	I	ORI	rd,rs1,imm
	AND	R	AND	rd,rs1,rs2
	AND Immediate	I	ANDI	rd,rs1,imm
Compare	Set <	R	SLT	rd,rs1,rs2
	Set < Immediate	I	SLTI	rd,rs1,imm
	Set < Unsigned	R	SLTU	rd,rs1,rs2
	Set < Imm Unsigned	I	SLTIU	rd,rs1,imm

有條件跳轉指令(Conditional branches)

Branches	Branch =	B	BEQ	rs1,rs2,imm
	Branch ≠	B	BNE	rs1,rs2,imm
	Branch <	B	BLT	rs1,rs2,imm
	Branch ≥	B	BGE	rs1,rs2,imm
	Branch < Unsigned	B	BLTU	rs1,rs2,imm
	Branch ≥ Unsigned	B	BGEU	rs1,rs2,imm
Jump & Link	J&L	J	JAL	rd,imm
	Jump & Link Register	I	JALR	rd,rs1,imm
Synch	Synch thread	I	FENCE	
	Synch Instr & Data	I	FENCE.I	
Environment	CALL	I	ECALL	
	BREAK	I	EBREAK	

官方文件：
https://riscv.org//wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

偽指令(pseudoinstructions)

Pseudoinstruction	Base Instruction(s)	Meaning
`la rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load address
`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]` `l{b|h|w|d} rd, symbol[11:0](rd)`	Load global
`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `s{b|h|w|d} rd, symbol[11:0](rt)`	Store global
`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fl{w|d} rd, symbol[11:0](rt)`	Floating-point load global
`fs{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fs{w|d} rd, symbol[11:0](rt)`	Floating-point store global
`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if ≠ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if < zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if > zero
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copy single-precision register
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Single-precision absolute value
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Single-precision negate
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copy double-precision register
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Double-precision absolute value
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Double-precision negate
`beqz rs, offset`	`beq rs, x0, offset`	Branch if = zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if ≠ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if ≤ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if ≥ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if < zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if > zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if >
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if ≤
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if >, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if ≤, unsigned
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, rs, 0`	Jump register
`jalr rs`	`jalr x1, rs, 0`	Jump and link register
`ret`	`jalr x0, x1, 0`	Return from subroutine
`call offset`	`auipc x6, offset[31:12]` `jalr x1, x6, offset[11:0]`	Call far-away subroutine
`tail offset`	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`	Tail call far-away subroutine
`fence`	`fence iorw, iorw`	Fence on all memory and I/O

Table 20.2: RISC-V pseudoinstructions.

組譯器(assembler)除了產生機器碼之外，還可以翻譯一些擴展指令。

這些擴展指令即為偽指令:

1. 標準指令的特殊情況
2. 由許多標準指令組合而成。

主要就是方便組語的Coding