處理器設計與實作

實習講義

編撰者

成大電通所計算機架構與系統研究室CASLAB

國立成功大學電機系與電腦與通信工程研究所

Outline

1. RISC-V 的基本認識
   A. RISC & CISC的認識與差異
   B. RISC-V Register Sets
   C. RISC-V 指令格式 (Instruction format)
2. 實驗內容

RISC-V 的基本認識

1. RISC-V Background

- Started by students from UC Berkeley in May 2010 as part of the Parallel Computing Laboratory (Par Lab), of which Prof. **David Patterson** was Director. The worldwide interest in RISC-V is its **free and open standard** to which SW can be ported, and which **allows anyone to freely develop their own HW to run the software.**
- Open ISA delivers easier support from a broad range of OS, SW vendors and tool developers. And no other ISA is architected like RISC-V, allowing for user extensibility of the architecture without breaking existing extensions or incurring software fragmentation

2. RISC & CISC

RISC (Reduced Instruction Set Computing):

RISC is a CPU design strategy based on the insight that a simplified instruction set (as opposed to a complex set) provides higher performance when combined with a microprocessor architecture capable of executing those instructions using fewer microprocessor cycles per instruction.

Major processors now use RISC ISA :

- **MIPS** (PlayStation, PlayStation 2 , PSP…)
- **ARM** (Apple iPods/iPhone/iPad, 3DS…)

CISC (Complex Instruction Set Computing) :

CISC is the opposing architecture compared to RISC

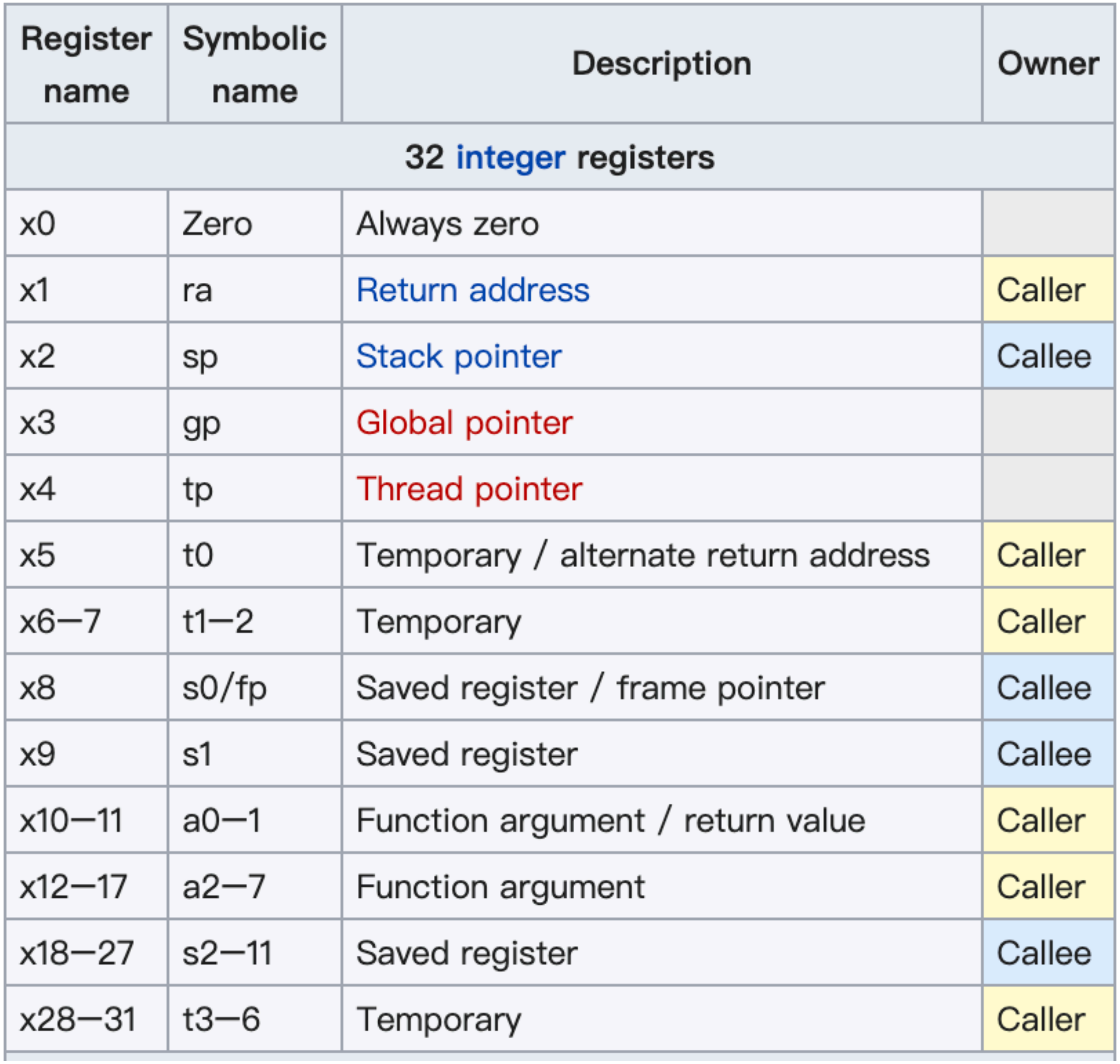
Early computers use CISC ISA. One of the reasons is to reduce code size due to expensive memory cost in the early days.

CISC tends to increase the performance of an individual instruction and has variable instruction length. The famous x86 instructions are CISC-based from the ISA point of view.

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 與CISC相比能力較差	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	因長度相同較輕鬆	相對困難,處理較複雜
暫存器	多	少
定址方式	少	多

3. Overview of RISC-V Integer Register Sets



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

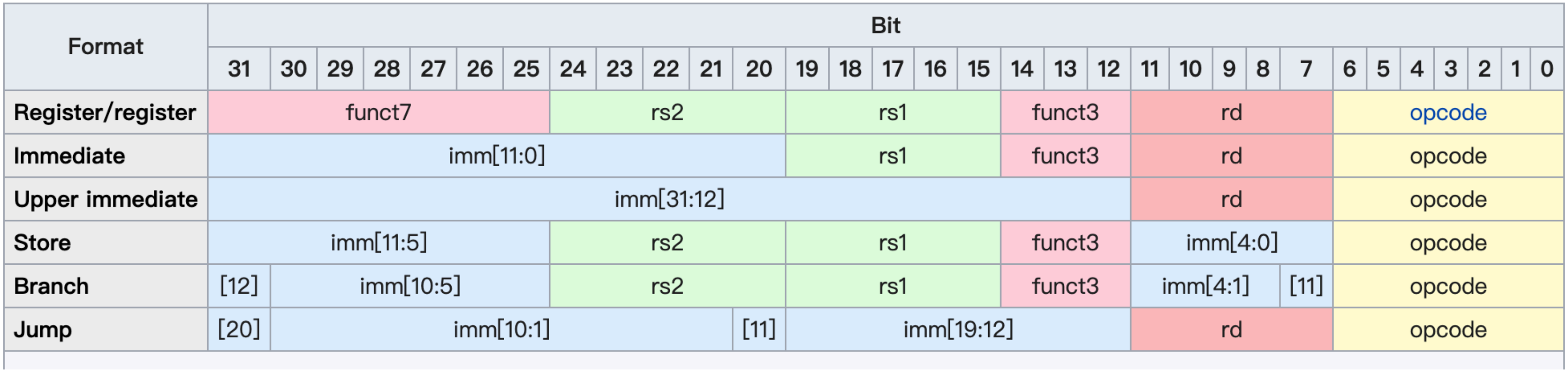
4. Instruction formats of RISC-V

- There are 6 types for instruction formats in RISC-V(32bit).
  - **R-type :** register-register
  - **I-type :** short immediates and load
  - **S-type :** stores
  - **B-type :** conditional branch, a variation of S-type
  - **U-type :** long immediates
  - **J-type :** unconditional jumps, a variation of U-type
- Regularity, all types are 32 bits wide.

Well structures of Instruction formats

- opcode(7 bits): Partially specifies which of the 6 types of instruction formats.
- funct7, and funct3 (10 bits): These two fields, further than the opcode field, specify the operation to be performed.
- rs1 (5 bits): Specifies, by index, the register containing first operand (i.e., source register).
- rs2 (5 bits): Specifies the second operand register.
- rd (5 bits): Specifies the destination register to which the computation result will be directed.

32–bit RISC–V instruction formats



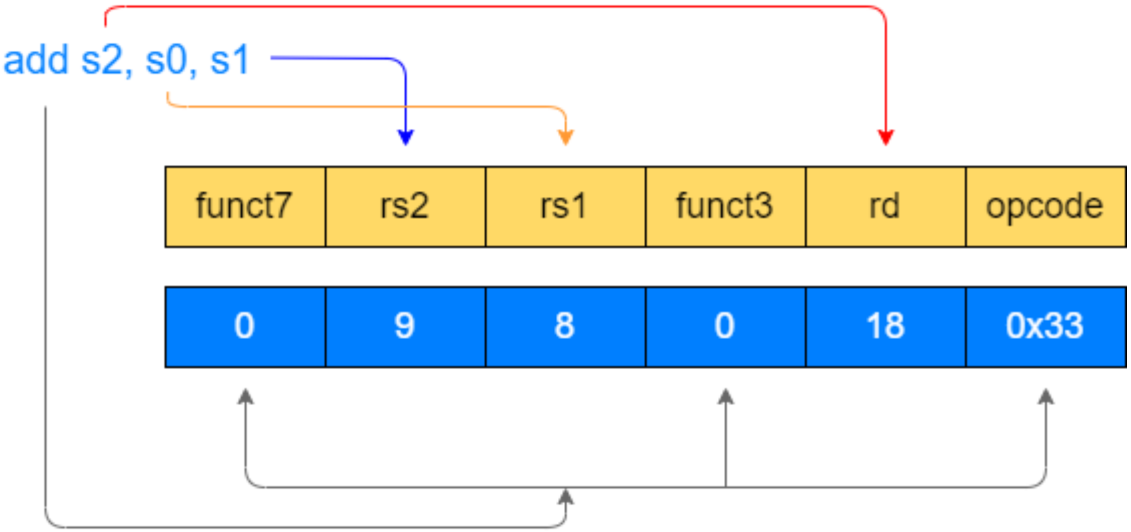
Format	Bit																															
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Upper immediate	imm[31:12]	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

算術與邏輯指令

- add, sub,…皆為此種類型的指令
- 這部分的指令皆為R-type

Ex:

C code	Assembly code
a = b + c	add s2, s0, s1
b = c - a	sub s1, s2, s0



funct7	rs2	rs1	funct3	rd	opcode
0	9	8	0	18	0x33

讀取與儲存記憶體

- sw, lw, ...皆為此種類型的指令
- 讀取的指令皆為I-type,儲存的指令皆為S-type

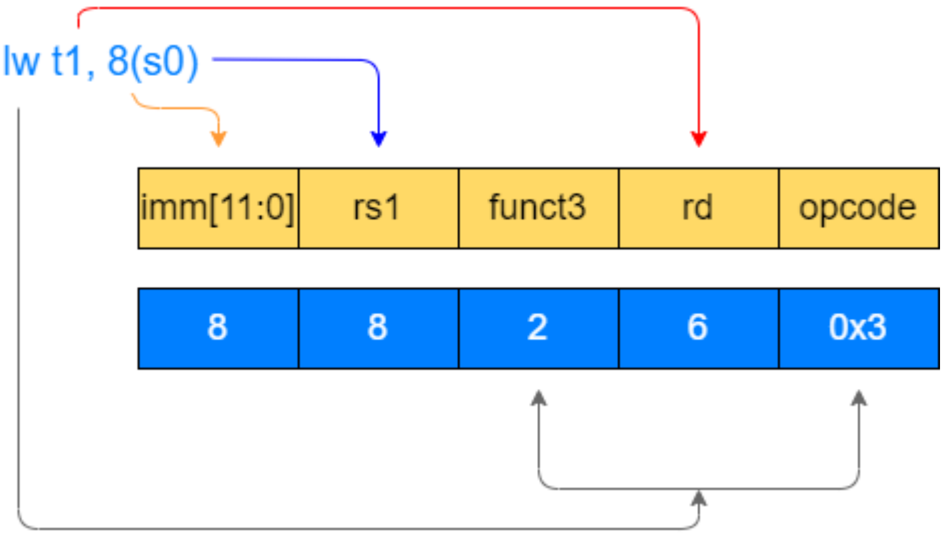
Ex:

C code

```
arr[1] = arr[0]+arr[2]
```

Assembly code

```
lw t0, 0(s0)     // t0 = arr[0]
lw t1, 8(s0)     // t1 = arr[2]
add t2, t0, t1   // t2 = t0+t1
sw t2, 4(s0)     // arr[1] = t2
```



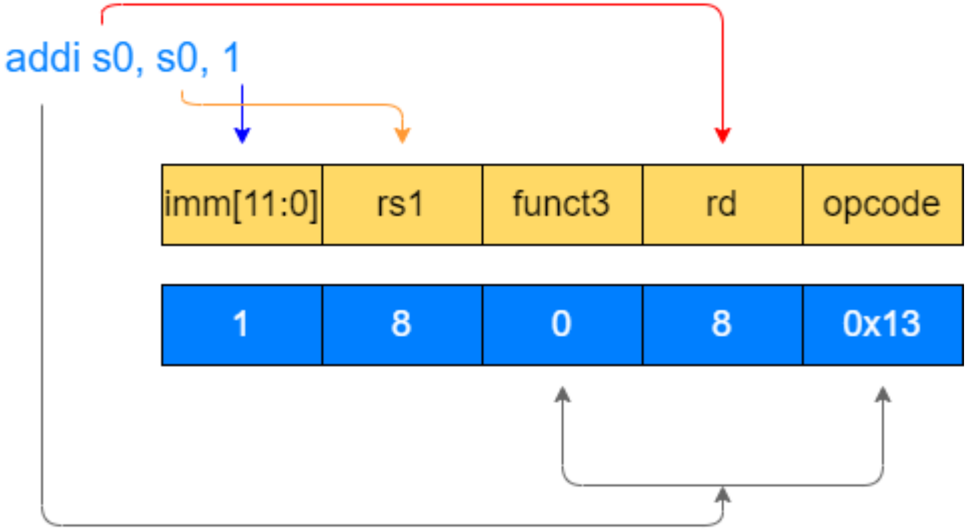
imm[11:0]	rs1	funct3	rd	opcode
8	8	2	6	0x3

邏輯算術指令：一個運算元是 12-bit 整數

- e.g. rd = rs1<某種運算> imm的指令
- addi, ori, slti, slli,... 皆為此種指令的運算
- 這部分的指令有一部分I-type,一部分R-type

Ex:

C code	Assembly code
a = a + 1	addi s0, s0, 1



imm[11:0]	rs1	funct3	rd	opcode
1	8	0	8	0x13

條件指令

- beq, bne, …為此種類型的指令

- 這部分的指令皆為B-type

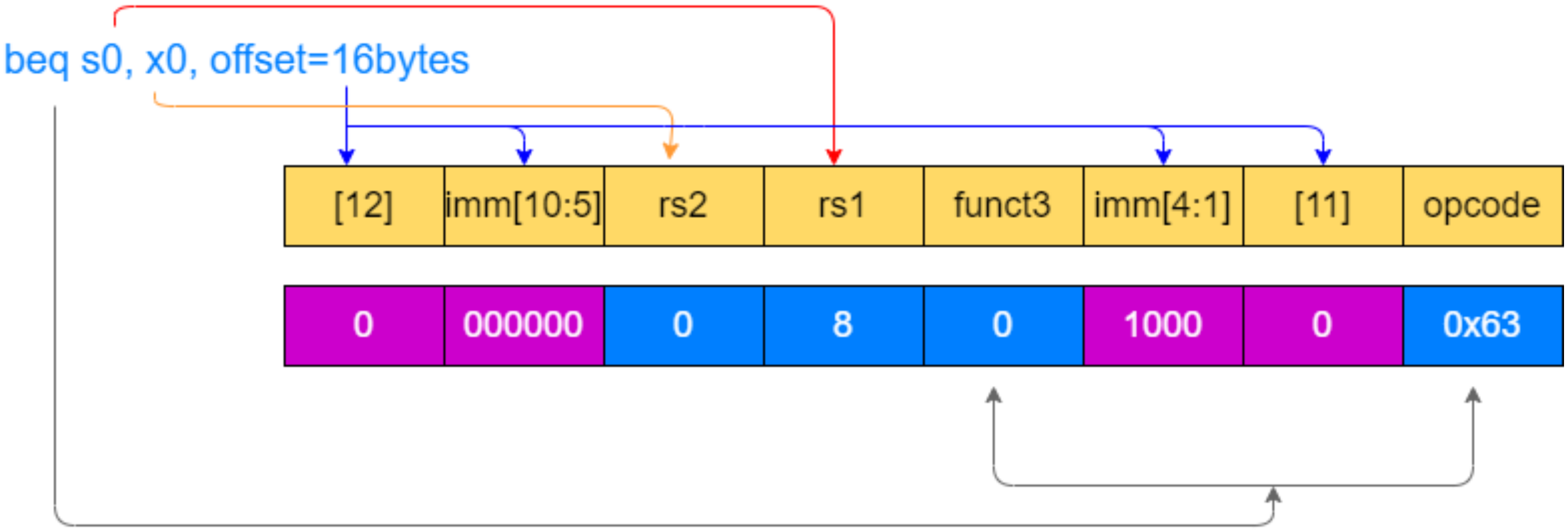
Ex:

```
C code                              Assembly code

if(x == 0) x = x + 1;                        beq s0, x0, EQUAL
else                                EQUAL:
                                             addi s0, s0, 1
```

Ex:

```
C code                              Assembly code

if(x != 0) x = x + 1;                        bne s0, x0, EQUAL
else                                EQUAL:
                                             addi s0, s0, 1
```

條件指令



beq s0, x0, offset=16bytes

[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	[11]	opcode
0	000000	0	8	0	1000	0	0x63

非條件跳躍

- jal, jalr為此種類型的指令
- jal 為J-type; jalr為I-type

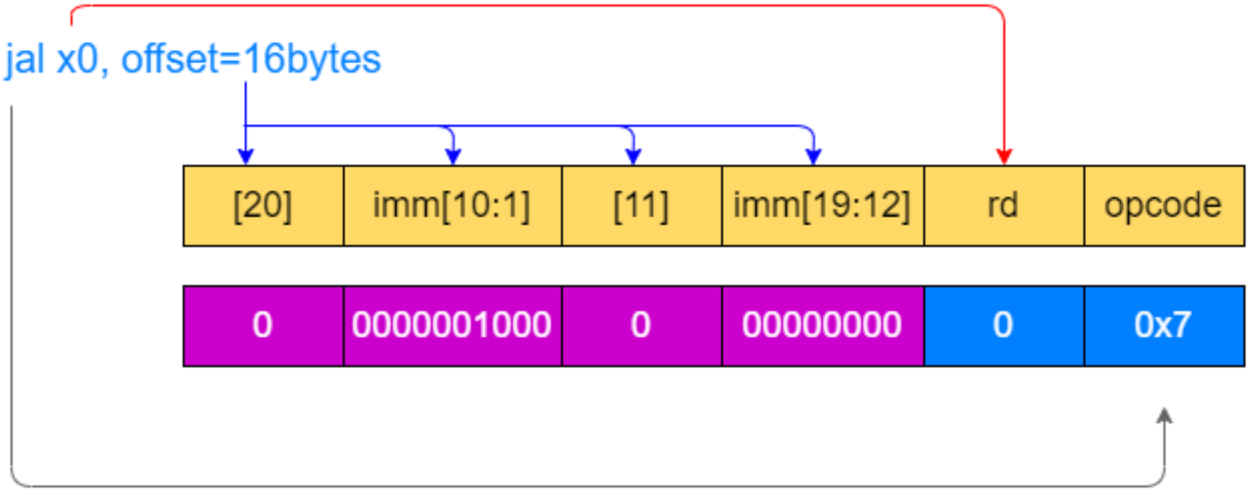
Ex:

C code

```
while(x != 0){
        x--;
}
```

Assembly code

```
LOOP:   bne s0, x0, END
        subi s0, s0, 1
        jal x0, LOOP
END:
```



jal x0, offset=16bytes

[20]	imm[10:1]	[11]	imm[19:12]	rd	opcode
0	0000001000	0	00000000	0	0x7

實驗內容

實作(一)

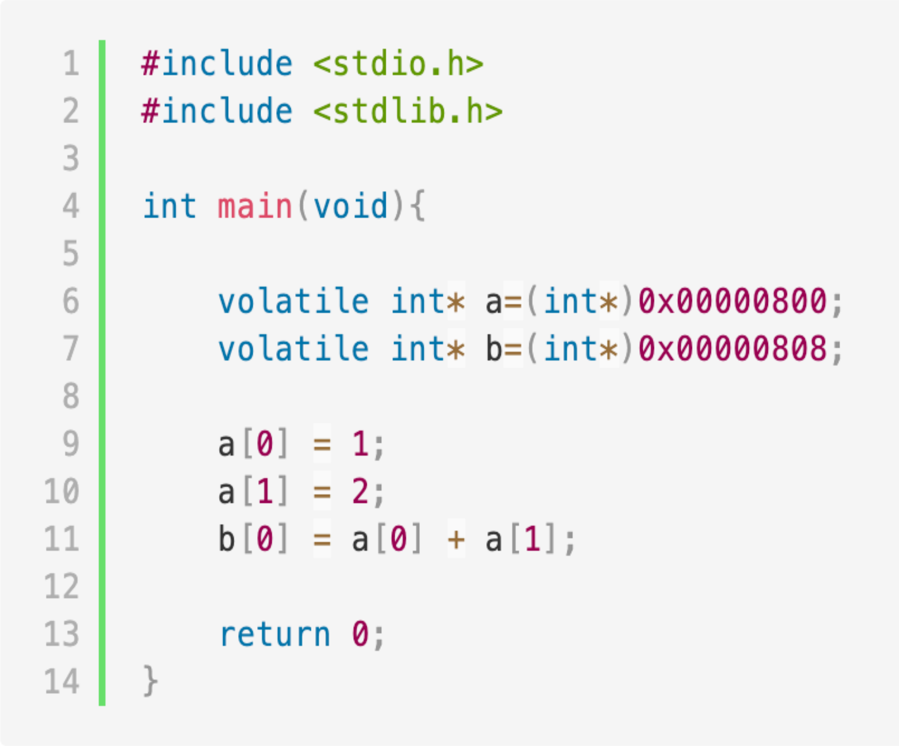
- 下面為一個簡單的for迴圈程式，請同學練習用組合語言描述它，並依照上述步驟，完成在modelsim驗證結果。

```
int i;

int sum=0;

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

備註: i使用x12　sum使用x13

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，一樣跑上述步驟，完成在modelsim驗證結果。



```c
#include <stdio.h>
#include <stdlib.h>

int main(void){

    volatile int* a=(int*)0x00000800;
    volatile int* b=(int*)0x00000808;

    a[0] = 1;
    a[1] = 2;
    b[0] = a[0] + a[1];

    return 0;
}
```

//陣列a是從記憶體 0x00000800開始存data

//陣列b是從記憶體 0x00000808開始存data

//對記憶體位址0x00000800寫入1的值

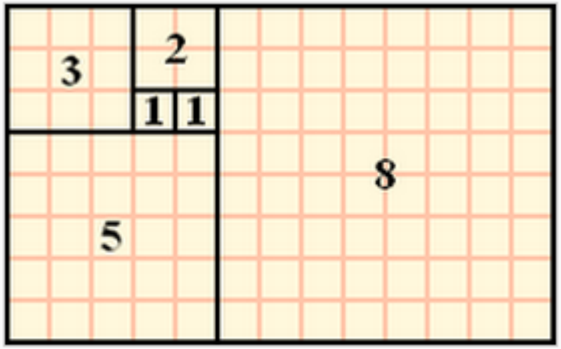
//對記憶體位址0x00000804寫入2的值

//將記憶體位置 0x00000800和0x00000804中的值讀出來做相加後，將結果寫入0x00000808的位置

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0x00000800的位置，最後用Modelsim看register和Memory中的結果。

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



以費波那西數為邊的正方形拼成的長方形

實驗結報

- 結報格式(每組一份)
  - 封面 (第幾組+組員)
  - 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: 10/5 18:00pm
- TA Contact Information:
  - 助教信箱：anita19961013@gmail.com
  - Lab：92617
  - Office hour：(Tuesday)8:00pm~10:00pm