第3章 組合語言

在電腦發展的早期，許多人都會使用組合語言撰寫程式。但在電腦發達的今天，組合語言通常只被用來撰寫非常低階的程式，像是啟動載入器、配置中斷向量、作業系統的行程切換、驅動程式、以及少數為了速度原因而採用組合語言的程式，例如像繪圖卡的顯示程式等。

雖然如此，學習組合語言的概念仍然有其重要性。因為組合語言除了是計算機結構的核心領域，可以幫助理解電腦的硬體架構之外，還可以幫助讀者學習系統軟體的設計方式，是理解系統軟體的捷徑。正因為如此，組合語言可以說是系統程式課程的核心，

在本章當中，我們將以範例導向的方式，說明組合語言的寫法。然後，在後續的幾個章節中 (第 4, 5, 6, 7, 8, 9, 10, 11 章)，我們將透過組合語言，引領讀者進入系統軟體的領域。這些領域包含組譯器、連結與載入、巨集處理器、編譯器、虛擬機器、作業系統、嵌入式系統等主題。因此，組合語言可以說是本書的核心章節，請讀者務必詳細閱讀。

3.1 基本範例

在本書中，CPU0 的組合語言一律採用目標在前的撰寫方式，也就是前置表示法。像是 ADD R1, R2, R3 指令，所代表的意思是 R1 = R2+R3。其中的第一個參數 R1 是加法運算的目標暫存器。

在本節當中，我們將從最簡單的組合語言指令開始，逐步進入組合語言的世界，最後寫出一個完整的程式。希望讓讀者於閱讀完本節之後，能具備基本的組合語言程式概念。

現在，就讓我們從最簡單的資料移動範例開始，學習組合語言程式的撰寫方法。

資料移動

在一般的程式語言當中，指定語句是很常見的，像是在範例 3.1 的 程式中，y = x 就是一個指定語句的範例，其結果會造成 x 變數的內容流入到 y 變數當中。

範例 3.1 C 語言當中的指定語句 - 資料流動指令

```
void main() {
   int x = 3, y;
   y = x;
}
```

在 CPU0 的組合語言當中，也有類似的語法，那就是 MOV 指令。然而，MOV 指令的兩個參數都只能是暫存器，所以只能將資料從暫存器移動到另一個暫存器。範例 3.2 便是一個使用 MOV 移動暫存器資料的範例。

範例 3.2 組合語言當中的資料移動指令 - MOV

(a) 組合語言	(b) C 語言
MOV R1, R2	R1 = R2

如果要移動記憶體當中的資料，則必須改用載入與儲存指令。舉例而言，範例 3.3 中的 LD R1, B 指令，會將記憶體變數 B 中的資料，先載入到暫存器 R1 當中。然後再利用 ST R1, A 指令，將暫存器 R1 的內容儲存到變數 A 中。這兩行程式的合成效果相當於 C 語言當中的指定語句 A = B。

範例 3.3 以組合語言模擬 C 語言當中的指定語句 A=B

(a) 組合語言	(b) C 語言 (對照版)	(c) C 語言 (簡化版)
LD R1, B ST R1, A	R1 = B A = R1	A = B

基本數學運算

除了進行資料的移動之外，組合語言還可以用來做基本的數學運算，像是加法、減法運算，或者是乘法與除法運算。然而，這些數學運算同樣是在暫存器當中進行的。因此，要進行運算之前，必須先將記憶體內容載入到暫存器當中，才能開始進行數學運算。

舉例而言，如果我們想達成 C 語言運算式 A=B + C – D 的效果，則必須先用 LD 指令將變數 B、C、D 分別載入到暫存器 R2, R3, R4 當中，然後再使用 ADD、 SUB 等指令，進行加減運算，最後，將運算的結果 R1 存回變數 A 當中。其對應的組合語言程式如範例 3.4(a) 所示。

範例 3.4 以組合語言模擬 C 語言當中的數學運算式 A = B + C - D

(a) 組合語言	(b) C 語言 (對照版)	(c) C 語言 (簡化版)

LD R2, B
LD R3, C
LD R4, D
ADD R1, R2, R3
SUB R1, R1, R4
ST R1, A
	R2 = B
	R3 = C
	R4 = D
	R1 = R2 + R3
	R1 = R1 – R4
	A = R1
		A = B + C – D

模擬條件判斷

if 語句在 C 語言當中是很重要的條件判斷指令，但是在組合語言當中並沒有 if 指令。還好，我們可以利用比較指令 CMP 與條件跳躍指令 (例如 JEQ、JGT、JLT 等)，達成與 if 語句相同的功能。

在範例 3.5 (a) 的組合語言中，為了將變數 C 設定為 A 與 B 當中較大者，我們使用了 CMP 與 JGT 兩個指令，模擬出了語句 if (A>B) 的功能。

首先，我們利用 LD 指令將 A , B 分別載入到 R1, R2 當中。然後再利用 CMP 指令比較 R1 與 R2 的內容。接著，根據比較的結果，利用 JGT 指令，決定到底要將哪一個暫存器存入變數 C 當中。根據 CMP 的結果，如果 R1 > R2 (也就是 A>B)，則 JGT 指令將會跳到 IF 標記，將 C 設為 R1 (也就是 A)，否則，會先將 C 設為 R2 (也就是 B) 之後，再跳到 EXIT 離開程式。

範例 3.5 以組合語言模擬 C 語言當中的 if 判斷式

(a) 組合語言	(b) C 語言 (對照版)	(c) C 語言 (簡化版)

LD R1, A
LD R2, B
CMP R1, R2
JGT IF
ST R2, C
JMP EXIT
IF: ST R1, C
EXIT:RET
	R1 = A;
	R2 = B;
	If (R1 > R2)
	goto IF;
	C = R2;
	goto EXIT;
	IF: C = R1;
	EXIT: RET;
		
		If (A>B)
		C = A;
		else
		C = B;
		
		return;

模擬迴圈

迴圈是 C 語言當中語法較為複雜的指令，要模擬迴圈功能，仍然必須藉助跳躍指令。舉例而言，範例 3.6 當中的無窮迴圈，即可使用組合語言當中的 JMP 跳躍功能達成。

範例 3.6 以組合語言實作無窮迴圈

(a) 組合語言	(b) C 語言 (對照版)

LOOP:
ADD R1, R1, R2
JMP LOOP
	while (1) {
	R1 = R1 + R2;
	}

然而，如果要模擬的不是無窮迴圈，就必須同時使用比較指令與條件式跳躍，讓程式有機會跳出迴圈。舉例而言，我們在範例 3.7 的組合語言中，寫出了一個可以從 1 加到 10 的程式，並將加總結果存放在變數 sum 當中。

由於範例 3.7 的程式較長，我們將不再逐行進行文字上的解說，請讀者直接看程式當中的 C 語言對照版，應該可以輕易的理解組合語言的內容。我們將只針對某些特別的指令進行說明。

範例 3.7 組合語言的完整程式範例，加總迴圈，從 1 加到 10。

	(a) 組合語言	(b) C 語言 (對照版)	(c) C 語言 (真實版)
1	LD R1, sum	R1 = sum;	int sum=0;
2	LD R2, i	R2 = i;	int i;
3	LDI R3, 10	R3 = 10;	for (i=1; i<=10; i++)
4	LDI R4, 1	R4 = 1;	sum += i;
5	FOR:CMP R2, R3	if (R2 > R3)//(i > 10)	
6	JGT EXIT	goto EXIT;	
7	ADD R1, R2, R1	R1 = R1 + R2;	
8	ADD R2, R4, R2	R2 = R2 + R4;	
9	JMP FOR	goto FOR;	
10	EXIT:RET	EXIT:return;	
11	i: WORD 1	int i = 1;	
12	sum:WORD 0	int sum = 0;	

在 CPU0 的組合語言中，當我們要宣告變數時，可以用 RESW、RESB、WORD、BYTE 等指令。其中，WORD、BYTE 是用來宣告具有初值的變數，而 RESW、RESB 則用來宣告未設定初值的變數或陣列。

舉例而言，範例 3.7 中的 i: RESW 1 這個指令，就保留了一個字組的 (Word) 空間給 i 變數，而 sum: WORD 0 則宣告了一個初值為 0 的變數 sum。

另外，當範例 3.7 中的 for 迴圈被翻譯成組合語言時，其中的比較條件 i<=10 是

迴圈繼續的條件，但是我們想要翻譯成類似 if 的跳離指令，於是利用相反的條件 i > 10 判斷是否跳出迴圈。這只是為了邏輯上的方便性所作的修改，並不影響整體程式的正確性。

在範例 3.7 (b) 的 C 語言對照版中，有一些不合 C 語言語法之處。舉例而言，我們將變數 sum 與 i 的定義放在程式的最後，而非程式的開頭，這是為了要能與組合語言的寫法一致才如此撰寫的，並非 C 語言真的可以這樣撰寫，請讀者不要誤解。

3.2 陣列存取

接下來，我們將示範如何使用 CPU0 中的相對定址方式，進行字串複製的動作。我們將模仿兩種 C 語言的實作方式，一種是使用指標進行字串複製，另一種是使用陣列進行字串複製。

字串複製 (指標版)

在範例 3.8 的 C 語言程式當中，展示了以 while 迴圈撰寫字串複製功能的程式。在此範例中，由於 C 語言的字串是以 '\0' 字元作為結尾，因此，該程式在 while 迴圈當中，只要發現 b[i] 是 '\0' 這個結尾字元時，就會跳開迴圈，結束複製行為。

範例 3.8 字串複製 (指標版)

(a) 組合語言	(b) C 語言 (對照版)	(c) C 語言 (真實版)

LD R2, aptr
LD R3, bptr
LDI R7, 1
while:
LDB R4, [R3]
STB R4, [R2]
ADD R2, R2, R7;
ADD R3, R3, R7;
CMP R4, R0
JEQ endw
JMP while
endw:
RET
	R2=*aptr;
	R3=*bptr;
	R7=1;
	while:
	R4 = [R3]; // R4 = *bptr
	[R2] = R4; //*aptr = R4
	R2= R2+1;
	R3=R3+1;
	if (R4!= R0) // b[i] != '\0'
	goto endw;
	goto while;
	endw:
	return;
		char a[10];
		char b[] = "Hello !";
		char *aptr=a, *bptr=b;
		
		while (1) {
		*aptr = *bptr;
		
		aptr++;
		bptr++;
		if (*bptr == '\0')
		break;
		}
		return;

a:	RESB 10	char a[10];	
b:	BYTE "Hello !",0	char b[] = "Hello !"	
aptr:	WORD a	int *aptr = a;	
bptr:	WORD b	int *bptr = b;	

在範例 3.8 的組合語言程式中，我們先利用 LD 指令將變數 i, a, b，分別載入到 R1, R2, R3 暫存器。此後，所有對這些變數的運算行為，都改用暫存器進行運算，最後再將結果存回對應的變數即可。

當我們將範例 3.8 (c) 的 C 語言改寫成組合語言時，其中的兩個指標變數 aptr 與 bptr，分別對應到 R2 與 R3 暫存器。因此，可以透過 LDB R4, [R3] 指令載入 bptr 所指的字元，然後再透過 STB R4, [R2] 將該字元存入 aptr 中。

在範例 3.8 的組合語言版本當中，除了使用 JMP 跳躍指令模擬 C 語言當中的 while 迴圈之外，還利用了 CMP 與 JEQ 指令模擬了 if … break 語句，在發現結束字元 '\0' 時，利用 JEQ 跳到迴圈結尾的 endw 標記上，以離開迴圈。

接著，讓我們採用陣列進行字串複製的程式，看看這種程式與指標的作法有何不同之處。

字串複製 (索引版)

當我們不使用指標進行字串複製，而改以陣列的方式實作時，程式將會如範例 3.9 所示。這個程式與範例 3.8 的不同點是沒有使用指標，而是改用 a[i] = b[i] 的方式，進行複製動作。

為了模擬 a[i]=b[i] 這樣的指令，我們採用了 CPU0 當中的索引定址模式。首先我們用 LD R1, i; LD R2, aptr; LD R3, bptr; 這些指令，將變數 I 與 a, b 的位址分別載入到 R1, R2, R3 當中，接著利用 LDB R4, [R3+R1] 與 STB R4, [R2+R1] 等索引定址法，模擬出 a[i]=b[i] 的功能，然後同樣用 CMP 與 JEQ 指令，決定是否要跳出 while 迴圈。

範例 3.9 字串複製 (索引版)

(a) 組合語言	(b) C 語言 (對照版)	(c) C 語言 (真實版)
LD R1, i	R1=i;	char a[10];
LD R2, aptr	R2=*aptr;	char b[] = "Hello !";
LD R3, bptr	R3=*bptr;	int i = 0;
LDI R7, 1	R7=1;	

while:
LBR R4, [R3+R1]
SBR R4, [R2+R1]
CMP R4, R0
JEQ endw
ADD R1, R1, R7
JMP while
endw: RET
a: RESB 10
b: BYTE "Hello !",0
aptr: WORD a
bptr: WORD b
i: WORD 0
	while:
	R4 = [R3+R1]; // R4 = b[i]
	[R2+R1] = R4; // a[i] = R4
	if (R4!= R0) // b[i] != '\0'
	goto endw;
	R1 = R1 + R7; // i++;
	goto while;
	endw: return;
	char a[10];
	char b[] = "Hello !"
	int *aptr=a;
	int *bptr=b;
	int i=0;
		while (1) {
		a[i] = b[i];
		
		if (b[i] == '\0')
		break;
		i++;
		}
		return;

請讀者仔細比較範例 3.8 與範例 3.9，應當可以看出這兩種字串複製方法的相同與相異點。假如不採用索引定址的模式，則 LBR R4, [R3+R1] 指令可改用 ADD R5, R3, R1; LDB R4, [R5] 這兩個指令完成同樣的功能，這會讓程式變得較大，而且多用了一個 R5 暫存器，因此效率會較差。同樣的，我們也可以用 ADD R6, R2, R1; 與 SDB R4, [R6] 取代 SBR R4, [R2+R1] 指令，但是會讓效率變得更差。因此，處理器的定址模式會影響到程式的效率，所以商業上所使用的處理器，有時會採用相當複雜的定址模式。

整數陣列的複製

在範例 3.10 中，我們示範了整數陣列的複製方法，其運作原理與範例 3.9 相當類似，但是仍然有些不同點，我們將一一說明。

範例 3.10 陣列複製 (索引版)

(a) 組合語言	(b) C 語言 (對照版)	(c) C 語言

LD R2, aptr
LD R3, bptr
LDI R8, 1
LDI R9, 100
ST R0, i
LD R1, i
FOR:
SHL R5, R1, 2
LDR R6, [R5+R2]
	R2 = *aptr;
	R3 = *bptr;
	R8 = 1;
	R9 = 100;
	i = 0;
	R1 = i;
	FOR:
	R5 = R1 << 2; // R5=i*4;
	R6 = [R5+R2]; // R6 = a[i]
		int a[100], b[100];
		int i;
		for (i=0; i<100; i++) {
		b[i] = a[i];
		}

STR R6, [R5+R3]
ADD R1, R1, R8
CMP R1, R9
JLE FOR
RET
a: RESW 100
b: RESW 100
aptr:WORD a
bptr:WORD b
i: RESW 1
	[R5+R3] = R6; // b[i] = R6;
	R1 = R1+1; // i++;
	If (R1 <= R9) // i<=100
	goto FOR;
	return
	int a[100];
	int b[100];
	int *aptr=a;
	int *bptr=b;
	int i;	

第一個不同之處在 i=0 這個指令上，由於範例 3.9 中 int i = 0 語句直接在宣告中設定初值，因此可以用 i WORD 0 指令模擬其語義。然而，在範例 3.10 中，由於先宣告了 int i，然後才在 for 迴圈中指定 i=0 的值。因此，組合語言在宣告 i 時以 i RESW 1 指令保留一個字組空間給 i，然後才利用 ST R0, i 指令將 0 填入到變數 i 當中[1]。

第二個不同點主要在 SHL R5, R1, 2 這一行上，這個動作其實是為了將 R1 的值乘以 4 而做的。這種作法雖然怪異但卻很常見，因為移位指令的速度通常比乘法更快，所以我們可以利用 SHL R5, R1, 2 指令將 R1 當中的 i 值左移兩位，以達成將 i 值乘以 4 的目的。

第三個不同點在於迴圈結束的判斷，由於本範例的 C 語言使用 FOR 迴圈，這與範例中採用精簡寫法的組合語言並無法一一對照，因此兩者只在行為模式上一至，但在比較運算 < 與 JLE 上並不相同，寫出 C 語言的原因只是為了讓讀者較容易理解組合語言的意義而已，兩者並沒有 100% 的對應關係。

3.3 副程式呼叫

在本節當中，我們將說明副程式的呼叫方法。本節分為兩個部分，第一個部分說明單層次的副程式呼叫，以及參數傳遞方法，第二個部份則是說明多層次的副程式呼叫，以及參數傳遞方法。

單層次的副程式呼叫

在附錄 A 的指令表中，有兩個與副程式呼叫相關的指令，分別是 CALL 與 RET。

[1] 暫存器 R0 不需要設定為零，因為 R0 是永遠為常數 0 的唯讀暫存器。

在 CPU0 當中要呼叫副程式時，必須使用 CALL 指令進行呼叫，然後再使用 RET 指令返回。

CALL 其實是一個特製的跳躍指令，CALL 指令會將返回點儲存到 LR 中，然後才進行跳躍動作，其過程如下圖所示。

必須注意的是，LR 所儲存的返回點，並非 CALL 指令的位址，而是下一個指令的位址。因為當 CALL 指令被提取後，PC 值就立刻被加上 4，接著控制單元才根據 IR 中的 CALL 指令將 PC 存入 LR 當中，因此 LR 才能儲存到正確的返回點。

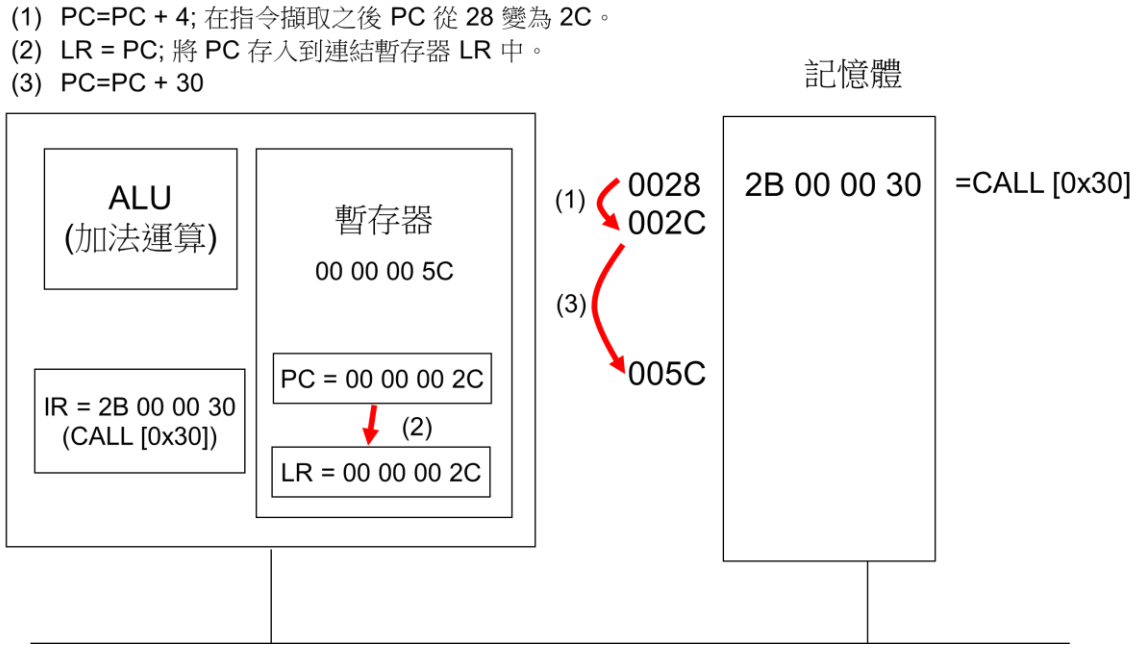

圖 3.1 指令 CALL [0x30] 的執行過程

然後，在副程式結束時，我們會利用 RET 指令，將原本儲存的 LR 值送回 PC 中，以返回 CALL 指令的下一行，完成副程式呼叫的動作。

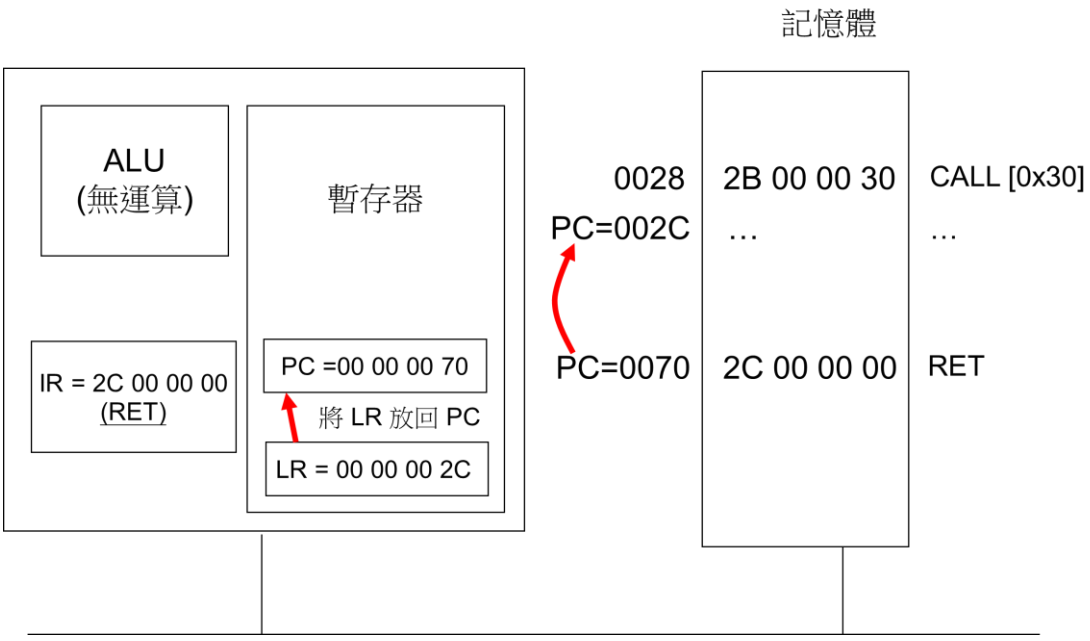

圖 3.2 指令 RET 的執行過程

圖 3.2 顯示了 RET 指令的執行過程，RET 指令會將 LR 送回 PC 中，於是 CPU 會接著執行 CALL 的下一個指令。因此，利用 CALL 與 RET 指令，我們就能在組合語言程式當中呼叫副程式，並且能正確的返回。

參數的傳遞方法

當有參數要傳遞給副程式時，有兩種方法，一種是透過暫存器傳遞參數，另一種是透過堆疊傳遞參數。其中，透過暫存器傳遞參數的方式速度較快，但是卻會受限於暫存器的數量。

在範例 3.11 當中，我們利用了 R2 來傳遞 x 參數給函數 f，然後將計算結果放入 R1 傳回給主程式。在本節中，我們都會使用暫存器 R1 儲存傳回值，這種習慣對程式的一致性會有所幫助

範例 3.11 單層副程式的呼叫-以暫存器傳遞參數

(a) C 語言程式	(b) 組合語言程式

void main() {
　　int x = 1, y;
　　y = f(x);
}
	LD R2, x
	　　CALL f
	　　ST R1, y
	　　RET
	x: WORD 1
	y: RESW 1

int f(int a) {
  return a+a;
}
	f:
	  ADD R1, R2, R2
	  RET

多層次的副程式呼叫

以暫存器傳遞參數的方式既簡單又快速，但是由於暫存器通常不多，可能無法容納太多參數。若要能容納更多的參數，可以透過堆疊傳遞參數，以避開暫存器數量的限制。

使用堆疊傳遞參數，在呼叫副程式之前，可以先透過 PUSH 指令，將參數推入到堆疊中，接著以 CALL 指令呼叫副程式。然後，在副程式當中，也必須配合上一層的程式，以進行參數的處理。首先，在副程式的開頭，必須透過 POP 指令將參數取回，然後利用 PUSH 指令保存 LR 暫存器 (因為下一層的 CALL 指令會覆蓋掉 LR，所以必須先儲存到堆疊中)，接著才開始撰寫程式內容。最後，在副程式結束前，必須使用 POP 指令取回 LR，以便恢復原先的返回點，然後才用 RET 指令返回上一層程式。

範例 3.12 顯示了多層次的呼叫方式，其中 f1() 函數會呼叫 f2() 函數，因此總共有兩層呼叫。這兩個函數的傳回值都是使用 R1 暫存器，以便與範例 3.11 一致。

範例 3.12 多層副程式的呼叫的組合語言程式

(a) 組合語言	(b) C 語言 (對照版)	(c) C 語言 (真實版)

LD R2, x
  PUSH R2
  CALL f1
  ST R1, y
  RET
x: WORD 1
y: RESW 1
 	
f1:
  POP R2
  PUSH LR
  ST R2, t
  LD R3, pt
  PUSH R3
	R2 = x;
	push R2; //push parameter x
	f1();
	y = R1; // y = f1(x)
	return;
	int x = 1;
	int y;
	
	f1() {
	POP R2; // R2=t=x;
	PUSH LR //保留連結暫存器
	t = R2;
	R3 = pt;
	PUSH R3; // push &t
		int main() {
		  int x = 1;
		  int y;
		  y = f1(x);
		  return 1;
		}
		int f1(int t) {
		  int b = f2(&t);
		  return b+b;
		}
		int f2(int *p) {
		  int r= *p+5;
		  return r;
		}

CALL f2
ST R1, b
ADD R1, R1, R1
POP LR
RET
t: RESW 1
b: RESW 1
pt: WORD t
 	
f2:
POP R2
LD R3, [R2]
LDI R4, 5
ADD R1, R3, R4
ST R1, r
RET
r: RESW 1
	f2();
	b = R1; // b = f2(&t)
	R1 = R1 + R1; // = b + b;
	POP LR // 回復連結暫存器
	return; // return b+b;
	int t;
	int b;
	int *pt = &t;
	}
	f2() {
	POP R2; // R2=p
	R3=[R2]; // R3 = *p
	R4 = 5;
	R1 = R3 + R4; // R1=*p+5
	r = R1; // r = *p+5
	return;
	int r;	

由於我們使用 R1 作為傳回值，因此在 f1() 與 f2() 函數中，我們都從 R2 開始將參數取回，以避免使用到 R1 暫存器。函數 f1() 先以 PUSH LR 保存返回點，然後在結束前用 POP LR 恢復返回點，這讓 RET 指令得以正確的返回上一層，避免因為 f1() 呼叫了 f2() 而造成 LR 被覆蓋的現象。

在函數 f2() 當中，我們沒有用 PUSH LR 指令保存 LR，這是因為 f2() 沒有再使用 CALL 指令了，因此不需要擔心 LR 被破壞的問題。如果 f2() 還有呼叫副程式時，就必須使用保存與恢復 LR，這是相當重要的一件事，否則程式將會因為返回點被覆蓋而產生難以預期的行為。

3.4 進階語法

在本節當中，我們將討論組合語言的進階語法，像是初始值的設定、Literal 的使用、以及假指令的用法等。這些都是為了讓程式設計師更容易撰寫組合語言而設計的語法，而且在許多組譯器當中都會支援這些功能。

定址範圍的問題

由於單一指令的 CPU 定址範圍通常無法涵蓋所有的位址空間，因此，組譯器有

時必須自行決定定址格式，甚至在無法正確定址時提示錯誤訊息，以告知程式設計師，讓程式設計師適時的採用特定的定址方式。

舉例而言，在範例 3.13 當中，我們用 X: RESW 100000 語句宣告了一個大小為十萬的 Word 型態之陣列，由於 CPU0 的一個 Word 佔用 4 bytes，因此 X 陣列佔據四十萬 bytes 的空間。這已經大於 LD 指令當中 16 位元常數 Cx 的定址範圍 (-32768-32767)。此時，組譯器應該提示錯誤訊息，例如，『Line 1: LD R1, B <-- Error, instruction B address out of range.』以便告知程式設計師應該修改程式。

範例 3.13 存取位址超過可定址範圍的組合語言程式碼

(a) 程式碼 (錯誤版)

```
        LD      R1, B
        ST      R1, A
X:      RESW    100000
A:      RESW    1
B:      WORD    29
```

(b) 程式碼 (修正後版本)

```
        LD      R9, APtr
        LD      R1, [R9]
        ST      R1, [R9+4]
APtr:   WORD    A
X:      RESW    100000
A:      RESW    1
B:      WORD    29
```

此時，一個有經驗的組合語言程式設計師，就應該會發現這個問題，然後，更改定址模式，不要再用相對於 PC 的預設定址方式，而改用某個暫存器 (例如 R9) 當作基底，以避免超出定址範圍的問題。舉例而言，程式設計師可以將範例 3.13 (a) 的錯誤版改寫，改成 (b) 的修正後版本。該修正版利用基底定址法，先將基底值 Base，載入到暫存器 R9 當中，接著所有的定址都強制相對於 R9 進行，如此，就能避開指令與變數距離過遠的問題了。

初始值

CPU0 的組合語言採用的類似 C 語言的語法，例如用 0x 開頭代表十六進位數值，像是範例 3.14 中的 oDev 就具有一個十六進位的初始值。同樣的，我們可以用字串符號 "..." 將某個字串內容框住，以宣告 byte 陣列的初始值。如範例 3.14 中的 EOF 所示。

範例 3.14 常數值表示法的組合語言程式範例

```
...
        LD R1, EOF
WLOOP:  ST R1, oDev
```

```
…
EOF:    BYTE "EOF"
oDev:   WORD 0xFFFFFF00
```

Literal

所謂的 Literal 是內嵌於指令當中的初始值，像是 **Error! Reference source not found.** (a) 當中的 "EOF"，就是 Literal 的範例。

雖然 LD R1, "EOF" 這個指令很像是直接把字串 "EOF" 載入到暫存器 R1 當中，但實際上並非如此。組譯器會將 Literal 展開成變數，就像範例 3.15 (b) 中的 $L1 的作法一樣。實際上，Literal 只是一種方便的寫法而已。

範例 3.15 包含 Literal 的組合語言程式範例

(a) 具有 Literal 的組合語言			(b) 將 Literal 展開後的結果		
	LD	R1, "EOF"		LD R1, $L1	
	ST	R1, Ptr		ST R1, ptr	
Ptr:	RESW	1	Ptr:	RESW	1
X:	RESW	100000	X:	RESW	100000
			$L1:	BYTE "EOF"	

通常在使用 Literal 的時候，組譯器會將 Literal 宣告在程式的最後，這就有可能造成定址範圍過大的問題。舉例而言，在範例 3.15 當中，我們用 X: RESW 100000 語句宣告了一個大小為十萬的字組陣列，由於 CPU0 的一個字組佔用 4 bytes，因此 X 陣列佔據了四十萬 bytes 的空間。這已經大於 LD 指令當中 16 位元常數 Cx 的定址範圍 (-32768-32767)。此時，組譯器應該提示錯誤訊息，例如，『Line 1: LD R1, "EOF" <= Error, address out of range.』以便告知程式設計師應該修改程式。

為了避免此種情況，程式師可以用 LTORG 這種指令，要求組譯器提早展開 Literal。範例 3.16 就利用 LTORG 指令在 X 陣列宣告前先展開 Literal，因而縮小了 LD R1, $L1 指令到 $L1 之間的距離，如此就能避免巨大陣列 X 所造成的定址範圍問題。

範例 3.16 以 LTORG 提早展開 Literal 的範例

(a) 具有 Literal 的組合語言			(b) 將 Literal 展開後的結果
	LD	R1, "EOF"	LD R1, $L1
	ST	R1, Ptr	ST R1, ptr

	LTORG		$L1:	BYTE "EOF"	
Ptr:	RESW	1	Ptr:	RESW	1
X:	RESW	100000	X:	RESW	100000

假指令

除了一般的指令之外，組合語言當中還會有許多組譯指引，像是 ORG, EQU 等，這些並非 CPU 的真實指令，因此也被稱為『假指令』 (Pseudo Instruction)。

以下，我們將介紹 EQU、ORG 等組合語言當中常見的假指令，以及分段用的假指令，像是 .text, .data 等，讓讀者熟悉這些常見的語法功能。

假指令- EQU

在 C 語言當中，我們常透過 #define 指令定義常數、公式或函數，在進階的組合語言語法當中，通常也會提供類似的功能。

在範例 3.17 中，我們就利用 EQU 這個假指令，定義了 MAXLEN 這個符號為 4096，並且將 PC 這個符號為 R15，LR 這個符號為 R14，這些都是 EQU 指令的範例，這種寫法讓組合語言更有彈性，同時也能用來提高程式的可讀性。

範例 3.17 使用 EQU 假指令的組合語言程式片段

(a) 具有 EQU 的組合語言		(b) 將 EQU 展開後的結果
MAXLEN	EQU 4096	
PC	EQU R15	
LR	EQU R14	
	LDI R1, MAXLEN	LDI R1, 4096
	MOV LR, PC	MOV R14, R15

在範例 3.18 當中，我們利用 EQU 指令，定義出 person 記錄中 name 與 age 欄位的公式。這樣就能在組合語言當中，模擬出與 C 語言的結構 struct 類似的效果。

範例 3.18 使用 EQU 進行相對位址模擬 C 語言的 struct 結構

(a) 組合語言		(b) C 語言
person:	RESB 24	struct person {
name	EQU person	char name[20];
age	EQU person + 20	int age;

…	}

在 EQU 指令當中，我們可以用錢字號 $ [2] 取得目前的位址。適當的使用錢字號 $ 可以達成許多令人驚訝的功能。

舉例而言，範例 3.19 當中的 person EQU $ 指令，會將其位址記錄在 person 這個符號當中，接著利用 name RESB 20 與 age RESW 1 分別保留 person 記錄所需要的欄位空間，這種寫法比範例 3.18 更為清楚，是相當常用的組合語言技巧。

範例 3.19 使用 EQU 與錢字號 $ 模擬 C 語言的 struct 結構

(a) 組合語言	(b) C 語言

person EQU $
name: RESB 20
age: RESW 1
…
	struct person {
	char name[20];
	int age;
	}

假指令- ORG

ORG 的功能是用來重新設定組譯器的目前位址，強制設定該行的位址為特定數值。在很多時後，ORG 與 EQU 都能解決同樣的問題，但是其做法卻完全不同。

舉例而言，範例 3.20 就利用 ORG 指令，達成了類似範例 3.18 的效果。因為第一個 ORG person 指令強制目前位址回到 person 的起頭，使得欄位 name 與 person 重疊在一起，因而讓後續欄位 age 的位址落在 person + 20 上，這相當於範例 3.18 中的 age EQU person + 20 之效果。

範例 3.20 使用 ORG 假指令的組合語言程式片段

	(a) 組合語言		(b) C 語言
1	person:	RESB 240	struct {
2		ORG person	
3	name:	RESB 20	char name[20];
4	age:	RESW 1	int age;
5	size	EQU $-person	
6		ORG	} person[10];
7	sum:	WORD 0	int sum = 0;

[2] 錢字號 $ 代表目前位址，也有組譯器用星號 * 代表。

沒有參數的 ORG 指令，其效果是讓『目前位址』回到上一個 ORG 指令前面。舉例而言，在範例 3.20 當中，第 6 行的 ORG 指令，是讓位址回到 ORG person 之前，如此才不會導致 sum 變數被置入 person 陣列中，造成變數空間重疊的問題。

必須注意的是，範例 3.20 當中的第 1 行 person RESB 240 保留了 240 個 byte 的空間，但是欄位內容 name 與 age 總共只占用了 24 個 byte。因此第 1 行相當於宣告了 10 個 person 的記錄空間，也就相當於右半部的 C 語言中所宣告的 person[10] 之效果，這也是組合語言中常見的技巧之一。

當然，我們也可以直接將第二個 ORG 指令改為 ORG person + 240，但是這樣的寫法必須前後一致，否則就會很容易產生錯誤。因此，使用不具參數的 ORG 指令回復位址，是比較方便且不容易出錯的寫法。

運算式

在範例 3.20 當中，第 5 行的 size EQU $-person 是一個可以計算出 person 記錄大小的運算式。因為 $ 所代表的是『目前位址』，由於前面有 name RESB 20 與 age RESW 1 這兩個指令讓組譯器前進了 24 個 byte，因此，size 的內容將會是 24。採用這種寫法，可以讓我們輕鬆的在組合語言當中計算出 person 結構的大小。

範例 3.21 是另一個使用 EQU 的範例，該範例使用 BUFEND 記住 BUFFER 緩衝區的結尾，然後用 BUFEND-BUFFER 這個運算式，計算出 BUFFER 的大小。利用這種技巧，我們可以計算任何結構的大小。

範例 3.21 使用運算式計算陣列大小的組合語言程式片段

```
BUFFER: RESB 4096
BUFEND EQU $
BUFLEN EQU BUFEND-BUFFER
```

從上面幾個範例可以看出，程式設計師若能善用 ORG、EQU、運算式與錢字號 $，將會對組合語言的設計有很大的幫助，因此，一個好的組合語言設計師必須懂得善用這些語法。

分段

在今日的組合語言當中，我們常會對程式進行分段，這種作法可以解決某些定址

範圍的問題，並且對作業系統的保護功能會有所幫助。

在現今的組合語言程式當中，通常會分為程式段 (.text) 與資料段 (.data)，程式段用來儲存指令，而資料段用來放置變數，如範例 3.22 所示。

範例 3.22 採用分段機制的組合語言程式

組合語言	說明
.text LD R1, EOF WLOOP: ST R1, oDev RET .data BUFFER: RESB 65536 EOF: RESB "EOF"	程式段開始 資料段開始

透過這種分段方式，作業系統就能根據段落的屬性，禁止任何指令存取程式段的記憶體，如此就能防止駭客竊取記憶體中的資料。同時也能防止駭客寫入 JMP 指令到程式段，企圖取得 CPU 的控制權。

3.5 實務案例

閱讀至此，相信讀者已經瞭解組合語言的基本概念了，接著，讓我們使用 GNU 工具，真正在個人電腦上撰寫幾個組合語言程式。

在本節中，我們將利用 gcc 編譯器與 as 組譯器，組譯我們所撰寫的組合語言，實際進行組合語言的程式練習，在 x86 系列的 IA32 平台上面，撰寫幾個簡單的組合語言，以便印證本章的理論，讓理論轉化為實務。

3.5.1 IA32 的組合語言

在 IA32 處理器上，目前常見的組譯器有微軟的 MASM、GNU 的 as 與開放原始碼的 NASM 等，這些組譯器各自採用不同的組合語言語法， GNU as 採用的是 AT&T 的語法，MASM 與 NASM 則是以 Intel 語法為基礎，但卻各自進行了擴充。

AT&T 與 Intel 的組合語言語法有明顯的差異，舉例而言，Intel 語法的目標暫存器會放在第一個參數中，採用的是前置式語法，而 AT&T 語法的目標暫存器會放在最後一個參數中，採用的是後置式語法。另外在語法細節上也有許多的差異。

範例 3.23 顯示了這兩種語法的差異性，在 MASM 所使用的 Intel 語法中，中括號 [] 代表位址，像是 mov eax, [ebx+3] 就是將 ebx+3 所指向的記憶體內容取出，放入 eax 暫存器當中。但是在 GNU 所使用的 AT&T 語法中，圓刮號 () 才代表位址，因此必須使用 mov 3(%ebx), eax 指令才能達成同樣的功能。

範例 3.23 兩種 IA32 的組合語言語法 (Intel 與 AT&T 語法)

(a) MASM 組合語言 (Intel 語法)		(b) GNU 組合語言 (AT&T 語法)	
mov	eax,1	movl	$1,%eax
mov	ebx,0ffh	movl	$0xff,%ebx
int	80h	int	$0x80
mov	ebx, eax	movl	%eax, %ebx
mov	eax,[ecx]	movl	(%ecx),%eax
mov	eax,[ebx+3]	movl	3(%ebx),%eax
mov	eax,[ebx+20h]	movl	0x20(%ebx),%eax
add	eax,[ebx+ecx*2h]	addl	(%ebx,%ecx,0x2),%eax
lea	eax,[ebx+ecx]	leal	(%ebx,%ecx),%eax
sub	eax,[ebx+ecx*4h-20h]	subl	-0x20(%ebx,%ecx,0x4),%eax

在學習 IA32 組合語言的路上，會遭遇到許多的障礙，這些障礙絕大部分都是因為 x86 處理器的歷史所造成的。在過去的 DOS 時代，x86 組合語言的輸出入通常是透過 BIOS 或 DOS 的中斷指令進行的，但是這種方式在 Windows 系統當中將無法運作。在 Windows 當中必須透過系統函數進行輸出入，但是對於組合語言的初學者而言，這兩種方式都太過複雜了。為了解決這個問題，在本節中，我們將採用 C 與組合語言連結的方式，以避免牽涉到複雜的輸出入介面等問題。

本書主要以 GNU 工具進行示範，因此以下的範例會使用 GNU 版的 AT&T 語法，您可以用 Dev C++ 當中的 gcc 與 as 工具進行組譯動作，以實際體會組合語言的用法。

首先，請讀者撰寫如範例 3.24 的 C 語言程式 (main.c) 與範例 3.25 的組合語言程式 (gnu_add.s[3])。我們在範例 3.24 的 C 程式中用 asmMain() 的呼叫方式，呼叫了範例 3.25 中的 _asmMain 程式 [4]。

[3] 在微軟的 Windows 作業系統中，組合語言程式的附檔名通常用 *.asm 表示，但在 Linux 與 GNU 的習慣中，通常用 *.s 作為組合語言程式的附檔名。

[4] 當 C 語言的函數被轉換為組合語言時，會在函數名稱前面加上底線 "_"。因此範例 3.24 中的 asmMain() 呼叫，在組合語言當中必須以 _asmMain 的標記定義。

範例 3.24 可呼叫組合語言的 C 程式

檔案：ch03/main.c	說明

#include <stdio.h>
 	
int main(void) {
  printf("eax=%d\n", asmMain());
}
	
	呼叫 asmMain() 組合語言函數，最後存入 eax 的值會被傳回印出。

範例 3.25 被 C 語言所呼叫的組合語言程式 (gnu_add.s)

檔案：ch03/gnu_add.s	說明

.text
.globl _asmMain
.def _asmMain; .scl 2; .type 32; .endef
_asmMain:
movl $1, %eax
addl $4, %eax
subl $2, %eax
ret
	程式段開始
	宣告 _asmMain 為全域變數，以方便 C 語言主程式呼叫。
	asmMain() 函數開始。
	eax = 1
	eax = eax + 4
	eax = eax – 2
	return

當您順利撰寫完上述兩個程式後，可以利用 gcc 編譯器，同時編譯 main.c 並組譯 gnu_add.s，此時 gcc 會順便將兩者連結，直接輸出執行檔。

範例 3.26 使用 gcc 編譯、組譯並且連結 (gnu_add)

編譯指令	說明
C:\ch03>gcc main.c gnu_add.s -o add C:\ch03>add asmMain()=3	編譯 main.c、組譯 gnu_add.s 並連結為 add.exe 執行 add.exe 輸出結果為 3

範例 3.26 是筆者的執行過程，最後會印出 eax=3 這樣的訊息。由於 gcc 編譯器會利用 eax 暫存器儲存傳回值，因此在組合語言 _asmMain 最後一行的 ret 指令執行後，就會將 eax 當作 asmMain() 的傳回值，因此 printf("eax=%d\n", asmMain()) 這行程式就會印出 eax=3 的結果。

透過這種以 C 語言呼叫組合語言的方法，我們就可以很容易的在個人電腦上測試組合語言程式，而不會被輸出入的問題卡住。您可以在任何的組合語言程式中加入 _asmMain: 標記，然後就能與 main.c 連結後組譯執行，隨時測試您對組合

語言的想法。

舉例而言，若我們將 asmMain 模組由 gnu_add.s 換成範例 3.27的 gnu_sum.s，那麼，採用相似的編譯程序，就可以改為計算 1+2+…+10 的結果。其編譯與執行過程如範例 3.28 所示。

範例 3.27 被 C 語言所呼叫的組合語言程式 (gnu_sum.s)

檔案：ch03/gnu_sum.s	說明
.data	資料段開始
sum: .long 0	int sum = 0
.text	程式段開始
.globl _asmMain	宣告 _asmMain 為全域變數，以
.def _asmMain; .scl 2; .type 32; .endef	方便 C 語言主程式呼叫。
_asmMain:	asmMain() 函數開始。
movl $1, %eax	eax = 1;
FOR1:	FOR1:
addl %eax, sum	sum = sum + eax;
addl $1, %eax	eax = eax + 1;
cmpl $10,%eax	if (eax <= 10)
jle FOR1	goto FOR1;
movl sum, %eax	eax = sum;
ret	return;

範例 3.28 使用 gcc 編譯、組譯並且連結 (gnu_sum)

編譯指令	說明
C:\ch03>gcc main.c gnu_sum.s -o sum	編譯 main.c、組譯 gnu_sum.s 並連結為 sum.exe
C:\ch03>sum	執行 sum.exe
asmMain()=55	輸出結果為 55

雖然上述的範例相當簡單，但是應該足夠作為讀者學習組合語言的起點了。有興趣的讀者可以試著撰寫基本的組合語言程式，像是陣列元素的搜尋等等，以便練習組合語言的邏輯概念。甚至進一步研讀組合語言的相關書籍，或者研究 Linux 核心等進階的主題，這些都是深入學習組合語言的好方法。

習題

3.1 請寫出一個 CPU0 的組合語言程式，可以計算 a=b*3+c-d 的算式。

3.2 請寫出一個 CPU0 的組合語言副程式 swap，可以將暫存器 R1 與 R2 的內容交換。

3.3 請寫出一個 CPU0 的組合語言副程式 isPrime，可以判斷暫存器 R2 當中的值是否為質數，如果是就將 R1 設為 1 傳回，否則就將 R1 設為 0。

3.4 請寫出一個 CPU0 的組合語言程式，可以計算出 2*2+4*4…+100*100 的結果，並將結果儲存在變數 sum 當中。

3.5 請以圖解的方式，說明在 IA32 處理器的 eax 暫存器中，為何會有 eax, ax, ah, al 等不同名稱，這些名稱代表的是哪個部分？

3.6 請寫出一個 IA32 的組合語言副程式 swap，可以將暫存器 eax 與 ebx 的內容交換。

3.7 請寫出一個 IA32 的組合語言副程式 isPrime，可以判斷暫存器 ebx 當中的值是否為質數，如果是就將 eax 設為 1 傳回，否則就將 eax 設為 0。

3.8 請撰寫一個 IA32 的組合語言程式，可以計算 2*2+4*4…+100*100 的結果後傳回，然後仿照 3.5.1 節的作法，使用 GNU 的 gcc 編譯連結該程式，並且執行看看結果是否正確。