# 第4章　組譯器

組譯器的主要功能，是將組合語言轉換為機器碼。在本章中，我們將說明組譯器的工作原理。

我們將在 4.1 節當中，利用範例導向的方式，說明組譯器的運作原理，看看組合語言是如何被轉換成機器指令的。然後，在 4.2 節當中，詳細說明組譯器的演算法與資料結構，以及這些資料結構在組譯器當中是如何被使用的。然後，在 4.3 節當中，以一個較完整的組合語言範例，搭配 4.2 節的演算法，對照說明組譯器的詳細運作方式。

## 4.1 組譯器簡介

組譯器乃是將組合語言轉換為目的檔的工具。有時，組譯器也會直接將組合語言轉換為可執行檔。因此，組譯器是組合語言程式師所使用的主要工具。

圖 4.1 顯示了組譯器的運作過程，當程式師寫完組合語言程式後，可以利用組譯器，將該程式轉換為二進位的目的檔，然而，由於二進位的表示法過於冗長，通常我們在範例中會改寫為 16 進位，以方便讀者閱讀。

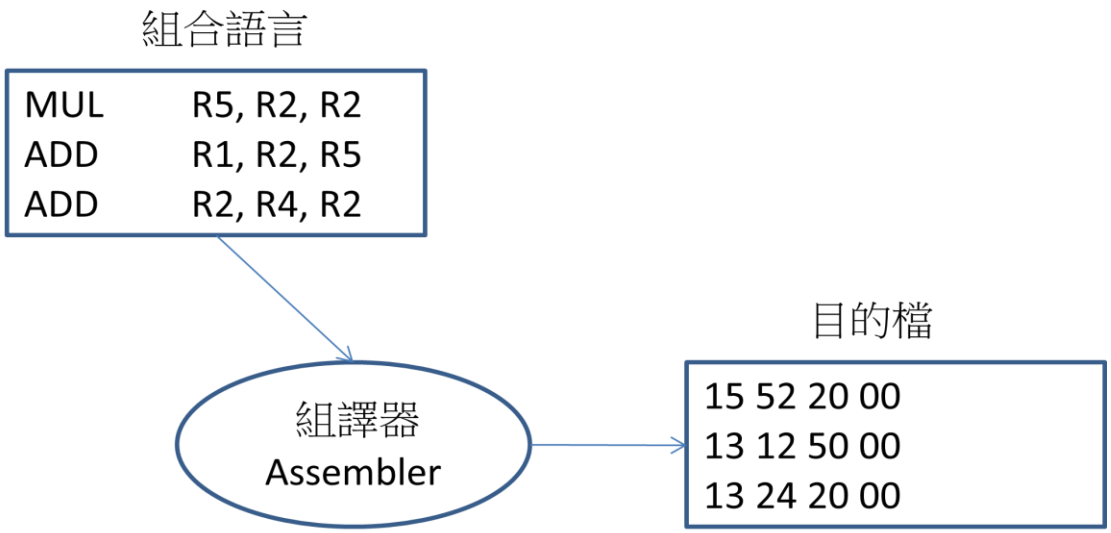


圖 4.1 組譯器的過程示意圖

範例 4.1 是一個簡單的 CPU0 組合語言程式，指令 LD R1, B 是將記憶體變數 B

的值載入到暫存器 R1 當中，而 ST R1, A 則是將暫存器 R1 儲存到記憶體變數 A 當中，整個程式的功能相當於執行 C 語言中的 A = B 指令。

範例 4.1 簡單的組合語言程式

| 組合語言程式碼 | 說明 |
|---|---|
| LD R1, B | 載入記憶體變數 B 到暫存器 R1 當中 |
| ST R1, A | 將暫存器 R1 存回記憶體變數 A 當中 |
| RET | 返回 |
| A: RESW 1 | 保留一個字組 (Word) 給變數 A |
| B: WORD 29 | 宣告變數 B 為字組，並設初值為 29 |

## 轉譯成目的碼

當範例 4.1 的程式被組譯時，程式中的指令會被轉換為目的碼。範例 4.2 顯示了該程式轉換成目的碼之後的結果。

在範例 4.2 的目的碼 (絕對定址版) 當中，指令 LD R1, B 被轉譯為 00100010，ST R1, A 則被轉譯為 0110000C，RET 被轉譯為 2C000000。接著，資料宣告的指令 A RESW 1 被轉譯為 00000000，但是 B WORD 29 這個指令卻被轉譯為目的碼 0000001D。讀到此處，讀者必然心中有所疑問，這些轉譯動作是如何進行的呢？

範例 4.2 為組合語言程式加上目的碼

| 位址 | 程式碼 | 目的碼 (絕對定址版) | 目的碼 (相對定址版) |
|---|---|---|---|
| 0000 | LD R1, B | 00100010 | 001F000C |
| 0004 | ST R1, A | 0110000C | 011F0004 |
| 0008 | RET | 2C000000 | 2C000000 |
| 000C | A: RESW 1 | 00000000 | 00000000 |
| 0010 | B: WORD 29 | 0000001D | 0000001D |

要能理解目的碼的轉譯過程，最關鍵的部分還是必須回到指令編碼表上，在本書附錄 A 的表格 A.1 中，列出了 CPU0 完整的指令編碼表，以及每一個指令的格式。另外，在附錄 A 的圖 A.2 中，顯示了 CPU0 的指令格式圖。

根據 CPU0 的指令表，我們可以看出 LD 指令的 OP 欄位編碼為 00，ST 的編碼為 01，而且兩者均為 L 型格式。接著從格式圖中，我們可以看到 L 型指令的格式，包含 OP、Ra、Rb、Cx 等四者的位置圖。

舉例而言，在指令 LD R1, B 中，指令 LD 被編為 16 進位的 00、R1 被編為 1，因

此，整個指令應該被編為 001XXXXX，但後續的編碼可能有兩種方法。第一種是採用絕對定址法，第二種是採用相對定址法。

採用絕對定址法時，LD R1, B 其實應該改寫為 LD R1, [R0+B] 才對。根據 L 型指令的格式 (OP Ra, Rb, Cx)，可看出下列對應關係 (OP=LD、Ra=R1、Rb=R0、Cx=B)。由於 R0 永遠為 0，於是組譯器只要在 Cx 的部分填入變數 B 的位址 0010 就行了。

圖 4.2 顯示了指令 LD R1, B 與目的碼 00100010 之間的對應方式。同理，ST R1, A 也就被編為 0110000C，請讀者自行嘗試編碼看看。

| | | | |
|---|---|---|---|
| LD | Ra, | [Rb + | Cx] |
| LD | R1, | | B |
| 00 | 1 | 0 | 0010 |

圖 4.2 將 LD R1, B 指令以絕對定址法編為目的碼

然而，使用絕對定址法有很大的缺點，因為 Cx 欄位的大小只有 16 個 bits，而且採用二補數的格式，最多只能定址 -32768～32767 的範圍。但是既然絕對位址不可能有負數，於是就只剩 0～32767 的部分可用。如果我們堅持採用絕對定址法，那就必須限制這些變數都必須被放在 0～32K 的範圍內，這是相當不合理的[1]。

一個比較好的方式是利用相對定址法編碼，但是要相對於哪一個暫存器呢？一種常見的方式是採用相對於程式計數器 PC 的方式。其原因是指令與被載入的資料定義通常在同一個程式當中，因此，兩者之間的距離不會太大，只要不超過 Cx 的範圍，就可以使用相對於 PC 的定址法。在實務上，很少單一程式模組的大小會超過 32K，因此，相對於 PC 的定址法是不錯的選擇。

採用相對於 PC 的定址法，LD R1, B 的指令，其實應該改寫為 LD R1, [PC+Cx] 的形式，也就是 LR R1, [R15+Cx] 才對，其中的 Cx 應設定為 B-R15，才能正確定址。所以組譯器必須在 Cx 當中放入變數 B 與 R15 的差異值，才能正確的存取變數 B。

在範例 4.2 的相對定址版目的碼當中，我們採用了相對於 PC 的定址法。舉例而言，LD R1, B 指令被編為目的碼 001F000C，其中的 F 是十六進為的 15，代表了 R15 暫存器，也就是 PC。圖 4.3 顯示了我們以相對於 PC 的定址法，將指令 LD

[1] 對於一般的 32 位元電腦而言，定址範圍通常可以達到 4G，這將是 Cx 定址範圍的 65536 *2 倍。

R1, B 編為目的碼的過程。

| 指令編碼 | | | | 計算(Cx=B-PC) | | |
|---|---|---|---|---|---|---|
| LD | Rd, | [Ra + | Cx] | | 0x0010 | (B) |
| LD | R1, | R15 + | (B-PC) | - | 0x0004 | (PC) |
| 00 | 1 | F | 000C | = | 0x000C | (Cx) |

圖 4.3 將 LD R1, B 指令以相對於 PC 的方法編為目的碼

必須注意的一點是，雖然指令 LD R1, B 的位址為 0x0000，但在指令提取動作完成後，PC 已經被加上 4 了，因此圖 4.3 當中的 PC 是 0x0004，而非 0x0000。

所以，Cx 的數值將是 B-PC = 0x0010-0x0004=0x000C，於是 Cx 將被填入 000C，整個 LD R1, B 指令就被編為 001F000C 了。

根據同樣的方式，範例 4.2 中的 ST R1, A 指令，若採用絕對定址的編碼方式，將會被組譯為 0110000C，但是若採用相對於 PC 的編碼方式，將會被組譯成 011F0004，請各位讀者務必自行推導出其對應方式，以便學習組譯器的編碼原理。

範例 4.2 當中的 RET 指令，其編碼方式很簡單，因為並沒有牽涉到定址的問題，於是我們根據 RET 指令的 OP 碼 2C，將指令編為 2C000000。

在範例 4.2 中，還有兩個資料變數 A 與 B，也必須被編為目的碼。資料的編碼相當簡單，只要轉換為 16 進位並且符合格式即可。於是 B WORD 29 的指令被編為 0000001D，這是由於十進位的 29 被轉換成目的碼後，若寫成 16 進位會是 1D，而且 CPU0 當中一個 Word 占據 4 bytes，對應到 16 進位會有 8 個數字。所以，B WORD 29 才會被組譯成 00 00 00 1D。

## 二階段的組譯方式

在範例 4.2 當中，我們除了列出目的碼之外，還列出了每一個指令的位址，這些位址對組譯器而言相當有用。當組譯器對 LD R1, B 指令進行編碼動作時，若不知道變數 B 的位址是 0010，就無法順利的編出欄位 Cx 的目的碼，因此，在真正進行組譯之前，必須先計算每一個變數與標記的位址。

大致上來說，組合語言要轉換成目的碼，必須經過以下步驟：

1. **運算元轉換**：將指令名稱轉換為機器語言，例如 LD 轉為 00，ST 轉為 01

等。

2. **參數轉換**：將暫存器轉為代號，符號轉為記憶體位址，例如 R1 轉為 1，A 轉為 000C，B 轉為 0010 等。
3. **資料轉換**：將原始程式當中的資料常數轉換為內部的機器碼，例如 29 轉換為 001D。
4. **目的碼產生**：根據指令格式，轉換成目的碼，輸出到目的檔中。

在上述步驟當中，除了第 2 個步驟外，都可以用循序的方式來處理，一次處理一行。但是，第 2 步驟的參數可能會有變數名稱，因此必須事先計算出每一個變數(或標記) 的位址。

一般來說，要將組合語言程式轉換為目的碼，必須經過兩道程序，第一道程序是計算每一行指令的位址，並記住每一個標記的位址，第二道程序才是真正將組合語言指令翻譯為機器碼。

範例 4.2 中含有位址、程式與目的碼的輸出型式，其實就是組譯器的輸出報表檔，這是用來提供組譯器設計人員檢視組譯過程有無錯誤的一種檔案。

如果我們將輸出報表中的目的碼集合成一個檔案，這個檔案將是一種最簡單的目的檔，這種目的檔被稱為是『映像檔』。範例 4.2 的中相對定址版的映像檔如範例 4.3 所示，假如範例 4.2 的檔案名稱為 Ex4_1.asm0，則範例 4.3 的映像檔就可命名為 Ex4_1.obj0 或 Ex4_1.img[2]。

範例 4.3 <範例 4.2>的映像檔 (相對定址版)

```
001F000C 011F0004 2C000000 00000000
0000001D
```

必須注意的是，目的檔通常儲存成二進位的格式，範例 4.3 中採用 16 進位的寫法，只是為了表達方便而採用的一種寫法。

在本書的第 12 章中，我們實作了 CPU0 的組譯器，這個組譯器稱為 as0，您可以用該組譯器對 ch12/Ex4_1.asm0 這個檔案進行組譯，其組譯方法與過程如範例 4.4 所示。

範例 4.4 使用 as0 組譯器組譯 Ex4_1.asm0

| 組譯方法與過程 | 說明 |
|---|---|

[2] 在 MS. Windows 當中，目的檔通常以 *.obj 命名，在 Linux 當中，則通常以 *.o 命名，在本書第 12 章的實作當中，我們通常以 *.obj0 的方式命名，以便與處理器 CPU0 相呼應。

```
C:\ch12>as0 Ex4_1.asm0 Ex4_1.obj0                    組譯 Ex4_1.asm0
Assembler:asmFile=Ex4_1.asm0 objFile=Ex4_1.obj0        輸出到 Ex4_1.obj0
==============Assemble============
        LD      R1, B                                 讀入程式並輸出檢視
        ST      R1, A
        RET
A:      RESW    1
B:      WORD    29
================PASS1===============                 組譯的第一階段
0000            LD   R1, B          L  0 (NULL)        計算位址
0004            ST   R1, A          L  1 (NULL)        並建立符號表
0008            RET                 J 2C (NULL)
000C A:         RESW 1              D F0 (NULL)
0010 B:         WORD 29             D F2 (NULL)
==============SYMBOL TABLE=========                  顯示符號表
000C A:         RESW 1              D F0 (NULL)
0010 B:         WORD 29             D F2 (NULL)
=============PASS2=============                      組譯的第二階段
0000            LD   R1, B          L  0 001F000C      編定目的碼
0004            ST   R1, A          L  1 011F0004
0008            RET                 J 2C 2C000000
000C A:         RESW 1              D F0 00000000
0010 B:         WORD 29             D F2 0000001D
==========Save to ObjFile:Ex4_1.obj0==========       將目的碼存入檔案
001F000C011F00042C0000000000000000000001D              Ex4_1.obj0
```

# 4.2 組譯器的演算法

在上一節當中，我們曾經提到組譯器通常需要兩道程序，這種兩道程序的組譯方式，被稱為兩階段組譯法。在本節當中，我們將更詳細的說明這種兩階段組譯器的設計方法，包含其詳細的演算法與資料結構等細節。

在兩階段組譯法當中，第一個階段的任務是定義符號，然後在第二個階段中才真正進行組譯，這兩個階段的工作內容如下所示。

第一階段 (計算符號位址)

1. 決定每一個指令與假指令所佔記憶空間的大小，例如決定 WORD、RESW 等

指令所定義的資料長度，以及 LD、ST 等指令所佔空間的大小。

2. 計算出程式當中每一行的位址。
3. 儲存每一個標籤與變數的位址，以方便第二階段使用。例如範例 4.2 當中的變數 A 為 0x000C，變數 B 為 0x0010 等。

第二階段 (組譯指令與資料)

1. 轉換指令 OP 欄位為機器碼，例如 LD 轉換為 00，ST 轉換為 01 等。
2. 轉換指令參數為機器碼，例如 R1 轉換為 1, B 轉換為 01 0C 等。
3. 轉換資料定義指令為位元值，例如 WORD B 29 轉為 0000001D 等。
4. 產生目的碼並輸出到目的檔當中。

## 組譯器的資料結構

在設計組譯器時，為了儲存指令表與標籤變數等資訊，通常需要兩個表格 -『運算碼表 (Op Table)』 與『符號表 (Symbol Table)』，其中的運算碼表是用來儲存指令的 OP 欄位與其對應的機器碼，以便在第二階段的步驟 1 時能將 OP 欄位轉換為機器碼。這個表格通常內建於組譯器當中，於組譯器啟動後就被載入到記憶體內，以便指令轉換動作發生時使用的。

符號表的用途是儲存標籤 (或變數) 的位址，這個表格一開始通常是空的，於第一階段計算位址後，會將標籤與位址填入該表格當中。如此，在第二階段的步驟 2 當中，就能利用符號表將參數轉換成位址。

這兩個表格通常會儲存在雜湊表中，雜湊表是一種可供快速查詢的資料結構，資料可以很快速的被置入表格中，但刪除時則較為緩慢。

雖然如此，這兩個表格也可以使用樹狀結構儲存。因為，不論採用何種結構，只要能正確的新增資料與搜尋即可，並不一定要使用雜湊的形式。

為了清楚說明組譯的過程，我們在範例 4.5 與範例 4.6 當中，列出了組譯過程中的指令表與符號表。請讀者仔細觀察這兩個表格，以便理解組譯器的編碼原理。

範例 4.5 資料結構 1 - 指令表

```
LD   00
ST   01
LDB  02
....
```

範例 4.6 資料結構 2 – 符號表

| | |
|---|---|
| A | 0000000C |
| B | 00000010 |

在真正的指令表當中，應該包含所有的組合語言指令。但是為了避免過於冗長，範例 4.5 中只列出了開頭的部分。

範例 4.6 當中所列出的，是於第一階段完成後所建立的符號表格。當第一階段的演算法完成後，所有符號的位址，都會被組譯器計算出來，並且填入符號表中，以便第二階段的演算法可以根據符號位址，完成指令編碼的動作。

在理解了組譯器所使用的兩個表格之後，讓我們仔細看看組譯器的兩階段演算法。第一階段的演算法如圖 4.4 所示，其目的是在計算出所有符號的位址。而第二階段的演算法則如圖 4.5 所示，這個階段才真正將組合語言指令轉換為機器碼。這兩個演算法都有中文的註解，請讀者仔細研讀，研讀時務必對照範例 4.2 逐行檢視位址的計算過程與指令的編碼過程，以求真正理解組譯器各個階段的原理。

| 組譯器的第一階段演算法 | 說明 |
|---|---|
| Algorithm AssemblerPass1 | 組譯器的第一階段演算法 |
| input AssmeblyFile | 輸入：組合語言檔 |
| ouptut SymbolTable | 輸出：符號表 |
| begin | |
| SymbolTable = new Table(); | 建立空的符號表 |
| file = open(AssemblyFile) | 開啟組合語言檔 |
| while not file.end | 當檔案還沒結束前，繼續讀檔 |
| line = readLine(file) | 讀下一行 |
| if line is comment | 如果該行為註解 |
| continue | 則忽略此行，繼續下一輪迴圈 |
| label= label(line) | 取得該行中的標記 |
| if label is not null | 如果該行有標記 |
| symbolRecord = symbolTable.search(label) | 於符號表中尋找該標記 |
| if symbolRecord is not null | 如果找到該標記 |
| report error | 則報告錯誤- 標記重複定義 |
| else | 否則 |
| symbolTable.add(label, address) | 將(標記、位址)放入符號表中 |
| end if | |
| op = operator(line) | 取得該行中的指令部分(助憶符號) |
| opRecord = opTable.search(op); | 於指令表中尋找該指令 |

| | |
|---|---|
| if opRecord is not null | 如果找到該指令 |
| address += 4; | 則將位址加 4 |
| else if op is 'BYTE' | 如果指令是BYTE |
| address += 1*length(parameters) | 則將位址加 1* 參數個數 |
| else if op is 'WORD' | 如果指令是WORD |
| address += 4 * length(parameters) | 則將位址加 4* 參數個數 |
| else if op is 'RESB' | 如果指令是RESB |
| address += length(parameter 1) | 則將位址加上參數 1 |
| else if op is 'RESW' | 如果指令是RESW |
| address += 4 * length(parameter 1) | 則將位址加上 4* 參數 1 |
| else | 如果不屬於上述情形之一，則代表 |
| report error | 該指令拼寫有誤，顯示錯誤訊息 |
| end if | |
| end while | |
| end | |

圖 4.4 組譯器的第一階段演算法 - 指定記憶體位址

| 組譯器的第二階段演算法 | 說明 |
|---|---|
| Algorithm AssemblerPass2 | 組譯器的第二階段演算法 |
| input AssmeblyFile, SymbolTable | 輸入：組合語言檔，符號表 |
| ouptut ObjFile | 輸出：目的檔 |
| begin | |
| file = open(AssemblyFile) | 開啟組合語言檔作為輸入 |
| while not file.end | 當檔案還沒結束前，繼續讀檔 |
| line = readLine(file) | 讀下一行 |
| (op, parameter) = parse(line) | 取得指令碼與參數 |
| opRecord = opTable.search(op) | 於指令表中尋找該指令 |
| if opRecord is not null | 如果找到該指令 |
| objCode = translateInstruction (parameter, address, opRecord) | 將指令轉換為目的碼 |
| address += length(objCode) | 計算下一個指令位址 |
| else if op is 'WORD' or 'BYTE' | 如果指令是WORD 或BYTE |
| objCode = translateData(line) | 將資料轉換為目的碼 |
| address += length(objCode) | 計算下一個指令位址 |
| else if op is 'RESB' | 如果指令是RESB |
| address += parameter[1] | 則將位址加上參數 1 |
| else if op is 'RESW' | 如果指令是 RESW |

| | |
|---|---|
| address += 4 * parameter[1] | 則將位址加上 4 * 參數 1 |
| end if | |
| output ObjCode to ObjFile | 將目的碼寫入目的檔當中。 |
| end while | |
| End Algorithm | |
| | |
| Algorithm TranslateInstruction | 轉換指令為目的碼。 |
| Input parameter, pc, opRecord | 輸入：參數、程式計數器、指令記錄 |
| Output objCode | 輸出：目的碼 |
| if (opRecord.type is L) | 如果是 L 型指令 |
| Ra = parameter[1] | 設定 Ra 參數 |
| if (parameter[3] is Constant) | 如果是常數 |
| Rb = 0 | |
| Cx = toInteger(parameter[2]); | 設定 Cx 為該常數。 |
| if (parameter[3] is Variable) | 如果是變數 |
| Cx = address(Variable) – pc | 設定 Cx 為位移 (標記-PC)。 |
| Rb = parameter[2] | 設定 Rb 參數。 |
| end if | |
| objCode = opRecord.opCode | 設定目的碼。 |
| + id(Ra)+id(Rb)+hex(Cx); | |
| else if (op.type is A) | 如果是A型指令 |
| Ra = parameter[1] | 取得Ra |
| Rb = parameter[2] | 取得Rb |
| if (parameter[3] is Register) | 如果參數 3 是暫存器 |
| Rc = parameter[3] | 取得 Rc |
| else if (parameter[3] is Constant) | 如果參數 3 是常數 |
| Cx = toInteger(parameter[3]); | 設定 Cx 為該常數。 |
| end if | |
| objCode = opRecord.opCode | 設定目的碼 |
| + id(Ra)+id(Rb)+id(Rc)+hex(Cx) | |
| else if (op.type is J) | 如果是 J 型指令 |
| Cx = parameter[1] | 取得 Cx |
| objCode = opRecord.opCode+hex(Cx) | 設定目的碼 |
| end if | |
| return objCode | 傳回目的碼 |
| End Algorithm | |

圖 4.5 組譯器的第二階段演算法 – 進行指令與資料編碼

# 4.3 完整的組譯範例

為了更詳細的說明兩階段組譯演算法，在本節當中，我們將使用一個較長的範例，以說明兩階段組譯器當中的第一階段與第二階段之功能。

範例 4.7 包含了一個具有陣列加總功能的副程式，以及呼叫該副程式的主程式。為了方便讀者理解，我們同時列出其組合語言與 C 語言對應程式，以便讀者能輕易的讀懂程式的邏輯。在本節當中，我們會使用兩階段組譯法，將這個範例組譯為目的檔。

範例 4.7 組合語言程式及其 C 語言對照版 (加總功能)

| 行號 | 組合語言 (檔案 ArraySum.asm0) | C 語言 |
|---|---|---|
| 1 | LDI R1, 0 ; R1=0 | int a[] = {3,7,4}; |
| 2 | LD R2, aptr ; R2=aptr | int *aptr = &a; |
| 3 | LDI R3, 3 ; R3=3 | int sum; |
| 4 | LDI R4, 4 ; R4=4 | for (i=3; i>0; i--) { |
| 5 | LDI R9, 1 ; R9=1 | sum += *aptr; |
| 6 | FOR: | aptr += 4; |
| 7 | LD R5, [R2] ; R5=*aptr | } |
| 8 | ADD R1,R1,R5 ; R1+=*aptr | return sum; |
| 9 | ADD R2, R2, R4 ; R2+=4 | |
| 10 | SUB R3, R3, R9 ; R3--; | |
| 11 | CMP R3, R0 ; if (R3!=0) | |
| 12 | JNE FOR ; goto FOR; | |
| 13 | ST R1, sum ; sum=R1 | |
| 14 | LD R8, sum ; R8=sum | |
| 15 | RET | |
| 16 | a: WORD 3, 7, 4 ; int a[]={3,7,4} | |
| 17 | aptr: WORD a ; int *aptr = &a | |
| 18 | sum: WORD 0 ; int sum = 0 | |

在此，我們將利用如圖 4.4 中的組譯器第一階段演算法，對範例 4.8 進行位址計算與標記定義。然後，再利用圖 4.5 中的組譯器第二階段演算法，對此範例進行指令與資料的編碼。

範例 4.8 顯示了這兩階段的編碼結果，其中的第一階段主要是計算記憶體位址並放入符號表中，第二階段則轉換出指令與資料的編碼，然後輸出為目的碼。

範例 4.8 組合語言程式及其目的碼 (加總功能)

```
記憶體位址     組合語言        指令類型 OP 目的碼
0000           LDI  R1, 0          L  8 08100000
0004           LD   R2, aptr       L  0 002F003C
0008           LDI  R3, 3          L  8 08300003
000C           LDI  R4, 4          L  8 08400004
0010           LDI  R9, 1          L  8 08900001
0014 FOR:                            FF
0014           LD   R5, [R2]       L  0 00520000
0018           ADD  R1,R1,R5       A 13 13115000
001C           ADD  R2, R2, R4     A 13 13224000
0020           SUB  R3, R3, R9     A 14 14339000
0024           CMP  R3, R0         A 10 10309000
0028           JNE  FOR            J 21 21FFFFE8
002C           ST   R1, sum        L  1 011F0018
0030           LD   R8, sum        L  0 008F0014
0034           RET                 J 2C 2C000000
0038 a:        WORD 3, 7, 4        D F2 000000030000000700000004
0044 aptr:     WORD A              D F2 00000038
0048 sum:      WORD 0              D F2 00000000
```

在範例 4.8 當中，第一階段的演算法除了計算位址之外，同時也會記住每個標記與變數的位址。

由於 CPU0 當中的每個指令均占 4 bytes，因此，每個指令都會造成位址往上加 4，這是由第一階段演算法 (圖 4.4) 當中的下列程式片段所造成的。

| 組譯器的第一階段演算法 | 說明 |
|---|---|
| …<br>op = operator(line)<br>opRecord = opTable.search(op);<br>if opRecord is not null<br>address += 4;<br>… | …<br>取得該行中的指令部分(助憶符號)<br>於指令表中尋找該指令<br>如果找到該指令<br>則將位址加 4<br>… |

值得注意的是，當組譯器進行到位址 38 時會處理 a 陣列，由於 a 陣列包含了 3, 7, 4 等三個 WORD 型態的數值，此時，位址會一次加上 4*3 = 12 的大小，於

是位址從 16 進位的 0x38 變成 0x44。
這個結果是由第一階段演算法 (圖 4.4) 當中的下列片段所造成的。

| 組譯器的第一階段演算法 | 說明 |
| --- | --- |
| …<br>else if op is 'WORD'<br>address += 4 * length(parameters)<br>… | …<br>如果指令是WORD<br>則將位址加 4 * 參數個數<br>… |

當第一階段執行完之後，除了計算出指令位址之外，最重要的是要將標記與變數的位址記錄到符號表 (Symbol Table) 當中。此時，該符號表的內容應該如圖 4.6 所示。

| 符號名稱 | 位址 |
| --- | --- |
| FOR | 0x14 |
| a | 0x38 |
| aptr | 0x44 |
| sum | 0x48 |

圖 4.6 組合語言程式範例 4.8 於組譯第一階段執行完後的符號表

有了這個符號表之後，就可以進行第二階段的組譯功能，對指令與資料進行編碼，後輸出到目的檔當中。最後，輸出的目的檔格式將如範例 4.9。

範例 4.9 <範例 4.8 >的目的檔

```
08100000 002F003C 08300003 08400004

08900001 00520000 13115000 13224000

14339000 10309000 21FFFFE8 011F0018

008F0014 2C000000 00000003 00000007

0000000400000038000000000
```

# 4.4 實務案例：處理器 IA32 上的 GNU 組譯器

接著，讓我們來看看真實的組譯器設計原理，我們將研究目標鎖定在 IA32 處理器上的 GNU 組譯器 as，以便進一步理解真實組譯器的設計原理。

在本節當中，我們將使用 GNU 的 AS 組譯器觀察 IA32 的指令編碼方式，以學習 IA32 組譯器的設計原理。

透過 –a 參數，可以讓 GNU 的 as 組譯器[3]產生組譯報表檔，範例 4.10 顯示了筆者的組譯過程與輸出報表。

從範例 4.10 中可看出各個變數與標記的位址，變數 sum 位址為 .data 段的 00000000，標記 _asmMain 的位址為 .text 段的 00000000，而 FOR1 的位址為 .text 段的 00000005。

範例 4.10 使用 GNU as 產生組譯報表

```
C:₩ch03>as -a gnu_sum.s

GAS LISTING gnu_sum.s                                   page 1




   1                                    .data
   2 0000 00000000          sum: .long 0
   3 0004 00000000                      .text
   3          00000000
   3          00000000
   4                                .globl _asmMain
```

[3] 有關 GNU as 組譯器的用法可參考下列網址 http://sourceware.org/binutils/docs-2.19/as/。

```
   5                            .def _asmMain; .scl 2; .type 32; .endef
   6                            _asmMain:
   7 0000 B8010000                      mov $1, %eax
   7      00
   8                            FOR1:
   9 0005 01050000                      addl %eax, sum
   9      0000
  10 000b 83C001                        addl $1, %eax
  11 000e 83F80A                        cmpl $10,%eax
  12 0011 7EF2                         jle FOR1
  13 0013 A1000000                      movl sum, %eax
  13      00
  14 0018 C3909090                      ret
  14      90909090
GAS LISTING gnu_sum.s                          page 2


DEFINED SYMBOLS
                                    *ABS*:00000000 fake
```

```
                gnu_sum.s:2          .data:00000000 sum

                gnu_sum.s:6          .text:00000000 _asmMain

                gnu_sum.s:8          .text:00000005 FOR1



NO UNDEFINED SYMBOLS
```

從範例 4.10 當中，我們可以看到 IA32 的指令長度並不固定，有些指令佔 2 bytes (像是 jle FOR1 的機器碼為 7EF2)，有些佔 4bytes (像 addl $1, %eax 的機器碼為 83C001)，有些甚至佔了 6 bytes (像是 addl %eax, sum 的機器碼為 010500000000)，這顯示了 IA32 採用的是變動長度的複雜指令集架構。

即使是同一個指令，IA32 的指令編碼長度也可能會不同 (像是上述的 addl 指令就可能編成 4 bytes 或 6 bytes)。這些都是 IA32 之的機器碼之所以複雜的原因，在本書中我們沒有足夠的篇幅介紹 IA32 的編碼方式，有興趣的讀者可以參考 Kip Irvine 的組合語言一書[4]。

# 習題

4.1 請說明組譯器的輸入、輸出與功能為何？
4.2 請說明組譯器第一階段 (PASS1) 的功能為何？
4.3 請說明組譯器第二階段 (PASS2) 的功能為何？
4.4 請說明組譯器當中的符號表之用途為何？
4.5 請說明組譯器當中的指令表之用途為何？
4.6 請仿照範例 4.4，使用本書第 12 章所實作的 as0 組譯器，組譯 Ex4_1.asm0 組合語言檔，並仔細觀察其輸出結果。
4.7 請閱讀本書第 12 章所附的 Assembler.c 與 Assembler.h 等 C 語言程式，並且對照本章的演算法，以學習 CPU0 組譯器的實作方式。
4.8 請按照 4.4 節的方法，操作 GNU 工具對組合語言進行組譯動作，並檢視組譯報表，找出各個符號的位址。

[4] Kip Irvine, Assembly Language for x86 Processors, 6th edition, ISBN: 0-13-602212-X, Published by: Prentice-Hall (Pearson Education), February 2010.該書第五版有中文翻譯，書名為組合語言，譯者為王國華、白能勝、曾鴻祥，出版社為全華科技，出版日期為 2007 年 11 月 05 日。

4.9 請於 http://kipirvine.com/asm/ 網站下載 Kip Irvine 書籍組合語言程式範例，並以 Visual Studio 進行組譯與執行。