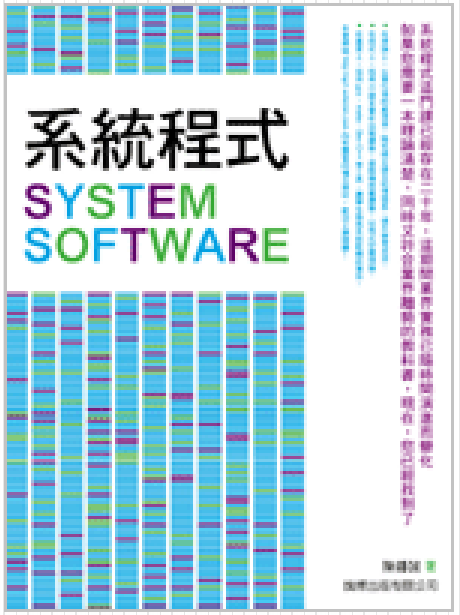
# 第 3 章、組合語言

作者：陳鍾誠

旗標出版社



# 第 3 章、組合語言

- 3.1 基本範例
- 3.2 陣列存取
- 3.3 副程式呼叫
- 3.4 進階語法
- 3.5 實務案例：IA32 的組合語言

# CPU0 的組合語言

- 前置表示法
  - CPU0 的組合語言一律採用目標在前的撰寫方式。

- 範例
  - ADD R1, R2, R3　　相當於　　R1 = R2 + R3

# 3.1 基本範例

- 資料移動
- 基本數學運算
- 模擬條件判斷
- 模擬迴圈
- 一個完整範例 -- 從 1 加到 10

# 資料移動

- C 語言

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;
}
```

- 組合語言

▶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 |

- 以組合語言移動記憶體資料

▶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 | R1 = B | A = B |
| ST R1, A | A = R1 | |

# 基本數學運算

▶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

```
LD   R2, B
LD   R3, C
LD   R4, D
ADD  R1, R2, R3
SUB  R1, R1, R4
ST   R1, A
```

(b) C 語言 (對照版)

```
R2 = B
R3 = C
R4 = D
R1 = R2 + R3
R1 = R1 - R4
A  = R1
```

(c) C 語言 (簡化版)

```
A = B + C - D
```

# 模擬條件判斷

▶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

```
        LD   R1, A
        LD   R2, B
        CMP  R1, R2
        JGT  IF
        ST   R2, C
        JMP  EXIT
IF:     ST   R1, C
EXIT:   RET
```

(b) C 語言 (對照版)

```
        R1 = A;
        R2 = B;
        If (R1 > R2)
        goto IF;
        C = R2;
        goto EXIT;
IF:     C = R1
EXIT:   RET;
```

(c) C 語言 (簡化版)

```

If (A>B)
        C = A;
else
        C = B;

return;
```

# 模擬迴圈

▶範例 3.6 以組合語言實作無窮迴圈

(a) 組合語言

```
LOOP:
      ADD R1, R1, R2
JMP LOOP
```

(b) C 語言 (對照版)

```
while (1) {
  R1 = R1 + R2;
}
```

# 一個完整範例 - 從 1 加到 10

▶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

```
LD     R1,     sum
       LD      R2, i
       LDI     R3, 10
       LDI     R4, 1
FOR:   CMP     R2, R3
       JGT     EXIT
       ADD     R1, R2, R1
       ADD     R2, R4, R2
       JMP     FOR
EXIT:  RET
i:     RESW    1
sum:   WORD    0
```

(b) C 語言 (對照版)

```
       R1 = sum;
       R2 = i;
       R3 = 10;
       R4 = 1;
       if (R2 > R3)//(i > 10)
       goto EXIT;
       R1 = R1 + R2;
       R2 = R2 + R4;
       goto FOR;
EXIT:  return;
       int i;
       int sum = 0;
```

(c) C 語言 (真實版)

```
int sum=0;
int i;
for (i=1; i<=10; i++)
      sum += i;
```

# 3.2 陣列存取

- 字串複製 (指標版)
- 字串複製 (索引版)
- 整數陣列的複製

# 字串複製 (指標版)

▶範例 3.8 字串複製 (指標版)

(a) 組合語言

```
  LD   R1, i
  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
a:     RESB 10
b:     BYTE "Hello !", 0
i:     WORD 0
aptr:  WORD a
bptr:  WORD b
```

(b) C 語言 (對照版)

```
R1=i;
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 !"
int i=0;
int *aptr = a;
int *bptr = b;
```

(c) C 語言 (真實版)

```
char a[10];
char b[] = "Hello !";
char *aptr=a, *bptr=b;

while (1) {
  *aptr = *bptr;

  aptr++;
  bptr++;
  if (*bptr == '\0')
     break;
}
return;
```

# 字串複製 (索引版)

▶範例 3.9 字串複製 (索引版)

(a) 組合語言

```
      LD   R1, i
      LD   R2, aptr
      LD   R3, bptr
      LDI  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
```

(b) C 語言 (對照版)

```
R1=i;
R2=*aptr;
R3=*bptr;
R7=1;
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;
```

(c) C 語言 (真實版)

```
char a[10];
char b[] = "Hello !";
int i = 0;

while (1) {
  a[i] = b[i];

  if (b[i] == '\0')
    break;
  i++;
}
return;
```

# 整數陣列的複製

▶範例 3.10 陣列複製 (索引版)

(a) 組合語言

```
      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]
      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
```

(b) C 語言 (對照版)

```
  R2 = *aptr;
  R3 = *bptr;
  R8 = 1;
  R8 = 100;
  i = 0;
  R1 = i;
FOR:
  R5 = R1 << 2; // R5=i*4;
  R6 = [R5+R2];   // R6 = a[i]
  [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;
```

(c) C 語言

```
int a[100], b[100];
int i;
for (i=0; i<=100; i++) {
 a[i] = b[i];
}
```

# 3.3 副程式呼叫

- 單層次的副程式呼叫
  - 參數的傳遞方法 – 使用暫存器
- 多層次的副程式呼叫
  - 參數的傳遞方法– 使用堆疊

# 單層次的副程式呼叫

- 參數的傳遞方法 – 使用暫存器

▶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

```
void main() {
  int x = 1, y;
  y = f(x);
}


int f(int a) {
  return a+a;
}
```

(b) 組合語言程式

```
      LD R2, x
      CALL f
      ST R1, y
      RET
x:    WORD 1
y:    RESW 1
f:
      ADD R1, R2, R2
      RET
```

# 指令 CALL [0x30] 的執行過程

(1) PC=PC + 4; 在指令擷取之後 PC 從 28 變為 2C。
(2) LR = PC; 將 PC 存入到連結暫存器 LR 中。
(3) PC=PC + 30

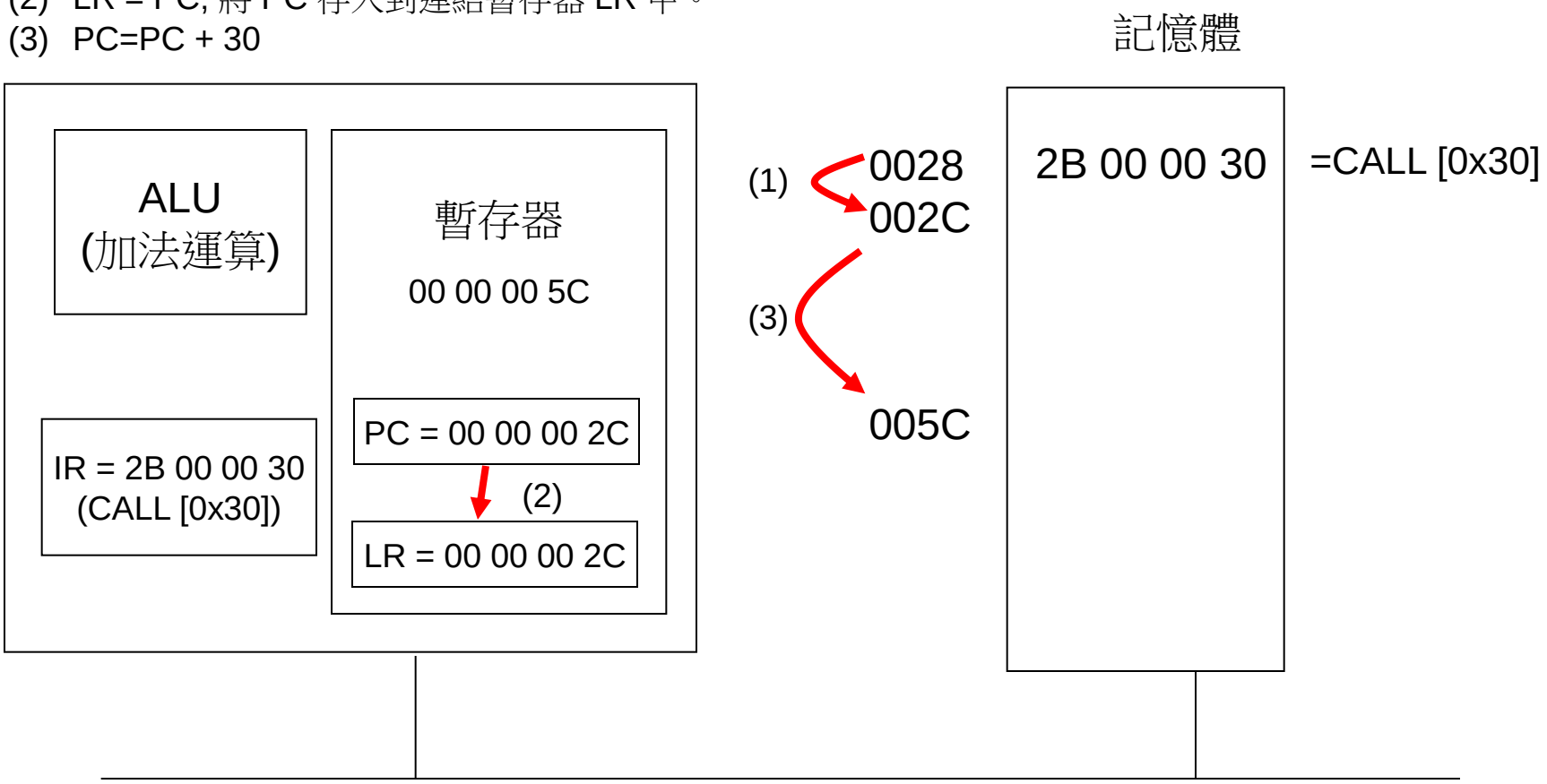



圖 3.1 指令CALL [0x30] 的執行過程

# 指令 RET 的執行過程

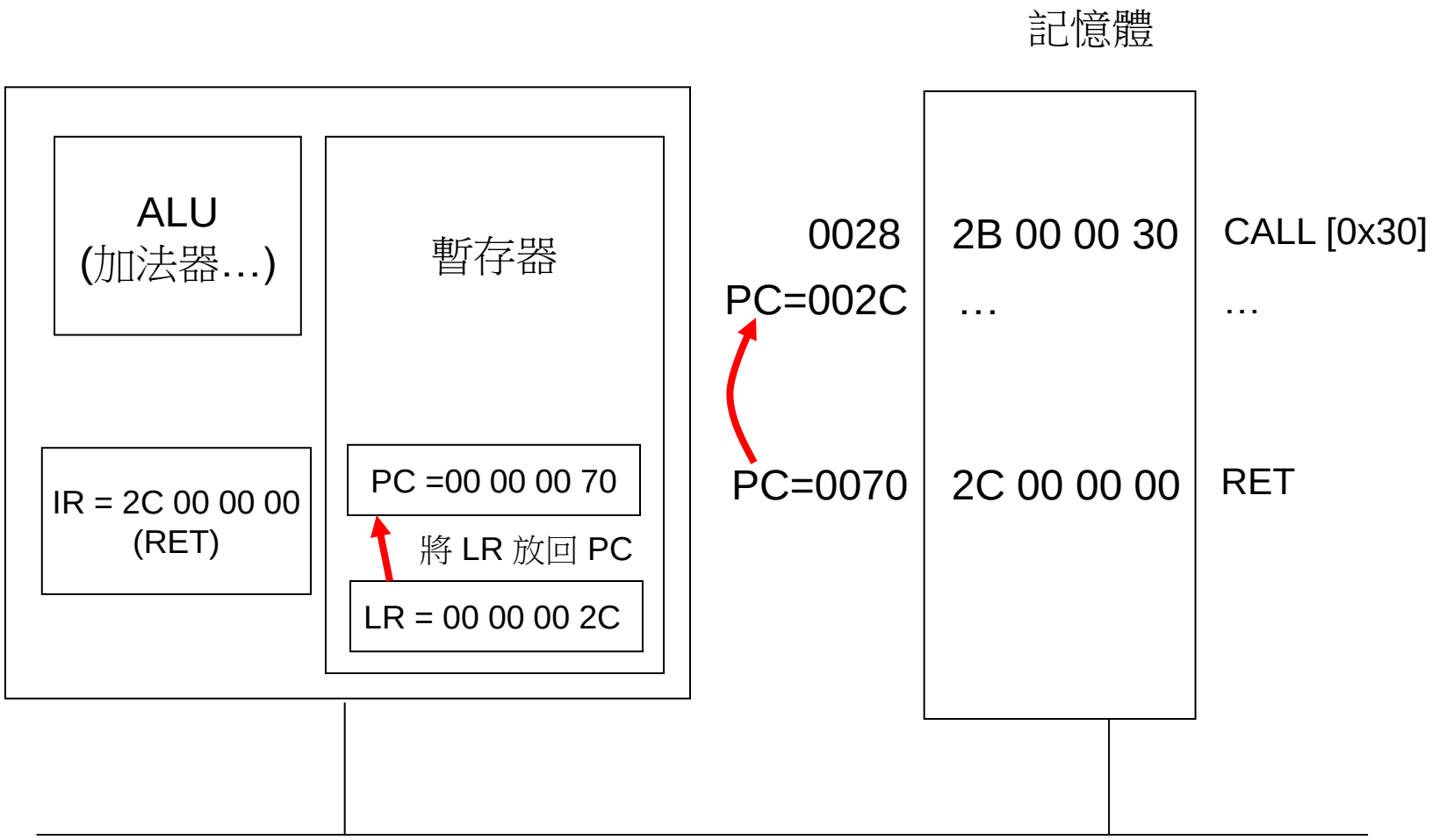



圖 3.2 指令RET 的執行過程

# 多層次的副程式呼叫

- 參數的傳遞方法– 使用堆疊
  - 避免上下層函數用到同一個暫存器，所產生的覆蓋現象。
  - 將 LR 儲存到堆疊中，以免在下一層 CALL 返回位址被覆蓋掉。

範例 3.12 的片段

```
f1:
        POP R2          取得堆疊中的參數
        PUSH LR         保存 LR
        ST R2, t
        LD R3, pt
        PUSH R3
        CALL f2
        ST R1, b
        ADD R1, R1, R1
        POP LR          恢復 LR
        RET             返回
t: RESW 1
b: RESW 1
pt: WORD t
```

```
int f1(int t)
{
  int b = f2(&t);
  return b+b;
}
```

▶範例 3.12 多層副程式的呼叫的組合語言程式

(a) 組合語言

```
   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
   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
```

(b) C 語言 (對照版)

```
R2 = x;
push R2; //push parameter x
f1();
y = R1; // y = f1(x)
return;
int x = 1;
int y;

f1() {
  POP R2; // R2=x;
  PUSH LR //保留連結暫存器
  t = R2;
  R3 = pt;
  PUSH R3; // push &t
  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;}
```

(c) C 語言 (真實版)

```
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;
}
```

# 3.4 進階語法

- 定址範圍的問題
- 初始值
- Literal : 值接將常數嵌入到指令中
- 假指令
  - LTORG : 以 LTORG 提早展開 Literal
  - EQU : 符號定義
  - ORG : 重設位址
- 運算式
- 分段

# 定址範圍的問題

- 避免將巨大陣列放在中間，應該放在最後面，或者用指標的方式解決巨大陣列的問題。

▶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
```

# 初始值

- 範例 3.14 中的 EOF, oDev 等變數都具有初始值。

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 – 直接將常數嵌入到指令中

▶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

```
     LD   R1, "EOF"
     ST   R1, Ptr
Ptr: RESW     1
X:   RESW     100000
```

(b) 將 Literal 展開後的結果

```
      LD R1, $L1
      ST R1, Ptr
Ptr:  RESW   1
X:    RESW   100000
$L1:  BYTE   "EOF"
```

# 以 LTORG 提早展開 Literal 的範例

▶範例 3.16 以 LTORG 提早展開 Literal 的範例

(a) 具有 Literal 的組合語言

```
      LD R1, "EOF"
      ST R1, Ptr
      LTORG
Ptr:  RESW   1
X:    RESW   100000
```

(b) 將 Literal 展開後的結果

```
      LD R1, $L1
      ST R1, ptr
$L1:  BYTE "EOF"
Ptr:  RESW   1
X:    RESW   100000
```

# EQU 假指令

- EQU
  - 是 (Equal)『等於』的意思
  - 我們可以使用 EQU 定義常數，如範例 3.17 所示

▶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

```
MAXLEN   EQU 4096
PC       EQU R15
LR       EQU R14
         LDI R1, MAXLEN
         MOV LR, PC
```

(b) 將 EQU 展開後的結果

```
LDI R1, 4096
MOV R14, R15
```

# 使用 EQU 模擬 struct 結構

- 在範例 3.18 (a) 中，我們將
  - name 定義為 person 的位址
  - age 定義為 person 的位址 + 20
  - 因而模擬了類似 3.18 (b) 當中的功能。

▶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

```
person:     RESB 24
name        EQU person
age         EQU person + 20
...
```

(b) C 語言

```
struct person {
  char name[20];
  int age;
}
```

# 錢字號 ($)

- 錢字號 ($)
  - $ 在組合語言中通常代表目前位址
    - (有些組譯器用星號 *)

▶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

```
person      EQU $
name:       RESB 20
age:        RESW 1
...
```

(b) C 語言

```
struct person {
  char name[20];
  int age;
}
```

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
```

# 使用 EQU 模擬 struct 結構

▶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

```
person      EQU $
name:       RESB 20
age:        RESW 1
...
```

(b) C 語言

```
struct person {
  char name[20];
  int age;
}
```

# ORG 假指令

- ORG 的功能是用來重新設定組譯器的目前位址

▶範例 3.20 使用 ORG 假指令的組合語言程式片段

(a) 組合語言

```
person:     RESB 240
            ORG person
name:       RESB 20
age:        RESW 1
size        EQU $-person
            ORG
sum:        WORD 0
```

(b) C 語言

```
struct {

  char name[20];
  int age;

} person[10];
int sum = 0;
```

# 運算式

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
```

# 分段假指令

- 一個組合語言程式通常可分為
  - 程式段 (.text)
  - 資料段 (.data)
    - 有時會將未設初值的資料放入 BSS 段 (.bss) 中。

▶範例 3.22 採用分段機制的組合語言程式

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

# 3.5 實務案例：IA32 的組合語言

- IA32 是目前 IBM PC 上最常用的處理器

- IBM PC 的組合語言相當複雜，尤其是輸出入部分
  - 使用 BIOS 中斷進行輸出入
  - 使用 DOS 中斷呼叫進行輸出入
  - 使用 Windows 系統呼叫進行輸出入

- 為了避開輸出入的問題，在本節中, 我們將採用
  - C 與組合語言連結的方式

# IA32 的組譯器

- 在 IA32 處理器上, 目前常見的組譯器有
  - 微軟的 MASM (採用 Intel 語法)
  - GNU 的 as 或 gcc (採用 AT&T 語法)
  - 開放原始碼的 NASM (採用 Intel 語法)

- 在本節中，我們將使用 GNU 的 gcc 為開發工具
  - 您可以選用
    - Dev C++ 中的 gcc – (Dev C++ 為本書的主要示範平台)
    - Cygwin 中的 gcc
    - Linux 平台中的 gcc

# Intel 語法 v.s. AT&T 語法

▶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)

```
mov   eax, 1
mov   ebx, 0ffh
int   80h
mov   ebx, eax
mov   eax, [ecx]
mov   eax, [ebx+3]
mov   eax, [ebx+20h]
add   eax, [ebx+ecx*2h]
lea   eax, [ebx+ecx]
sub   eax, [ebx+ecx*4h-20h]
```

(b) GNU 組合語言 (AT&T 語法)

```
movl  $1, %eax
movl  $0xff, %ebx
int   $0x80
movl  %eax, %ebx
movl  (%ecx), %eax
movl  3(%ebx), %eax
movl  0x20(%ebx), %eax
addl  (%ebx, %ecx, 0x2), %eax
leal  (%ebx, %ecx), %eax
subl  -0x20(%ebx, %ecx, 0x4), %eax
```

# C 與組合語言的完整連結範例 (一)

▶範例 3.24 可呼叫組合語言的 C 程式

檔案：ch03/main.c

```
#include <stdio.h>

int main(void) {
  printf("eax=%d\n", asmMain());
}
```

說明

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

```
      .text
.globl _asmMain
      .def _asmMain; .scl  2; .type32; .endef
_asmMain:
      movl $1, %eax
      addl $4, %eax
      subl $2, %eax
      ret
```

說明

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;
```

# 範例 3.25 的執行結果

▶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<br><br>C:\ch03>add<br>asmMain()=3 | 編譯 main.c、組譯 gnu_add.s 並連結為 add.exe<br>執行 add.exe<br>輸出結果為 3 |

# C 與組合語言的完整連結範例 (二)

▶範例 3.24 可呼叫組合語言的 C 程式

檔案：ch03/main.c

```
#include <stdio.h>

int main(void) {
  printf("eax=%d\n", asmMain());
}
```

說明

呼叫 asmMain() 組合語言函數，最後存入 eax 的值會被傳回印出。

▶範例 3.27 被 C 語言所呼叫的組合語言程式 (gnu_sum.s)

檔案：ch03/gnu_sum.s

```
      .data
sum:.long 0
      .text
.globl _asmMain
      .def _asmMain; .scl  2; .type 32; .endef
_asmMain:
      movl $1, %eax
FOR1:
      addl %eax, sum
      addl $1, %eax
      cmpl $10, %eax
      jle FOR1
      movl sum, %eax
      ret
```

說明

```
資料段開始
int sum = 0
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;
FOR1:
      sum = sum + eax;
      eax = eax + 1;
      if (eax <= 10)
            goto FOR1;
      eax = sum;
      return;
```

# 範例 3.27 的執行結果

▶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<br><br>C:\ch03>sum<br>asmMain()=55 | 編譯 main.c、組譯 gnu_sum.s 並連結為 sum.exe<br>執行 sum.exe<br>輸出結果為 55 |

# 習題

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。
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. 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。
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。
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？
6. 請寫出一個 IA32 的組合語言副程式 swap，可以將暫存器 R1 與 R2 的內容交換。
7. 請寫出一個 IA32 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。
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。