

計算機結構 (組合語言與指令集)


陳鍾誠 於金門大學

數位邏輯與程式設計

陳鍾誠 2005 年 5 月 16 日

從 Java 到 組合語言

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



```
WORD sum    0  
WORD i      0  
  
LOAD  R1, sum  
LOAD  R2, i  
LOOP  COMP R2, 10  
      JGT  EXIT  
      INC  R2  
      ADD  R1, R2, R1  
      JMP  LOOP  
EXIT  STORE sum, R1  
      STORE i, R2
```

機器指令（整合範例 – 組譯）

位址

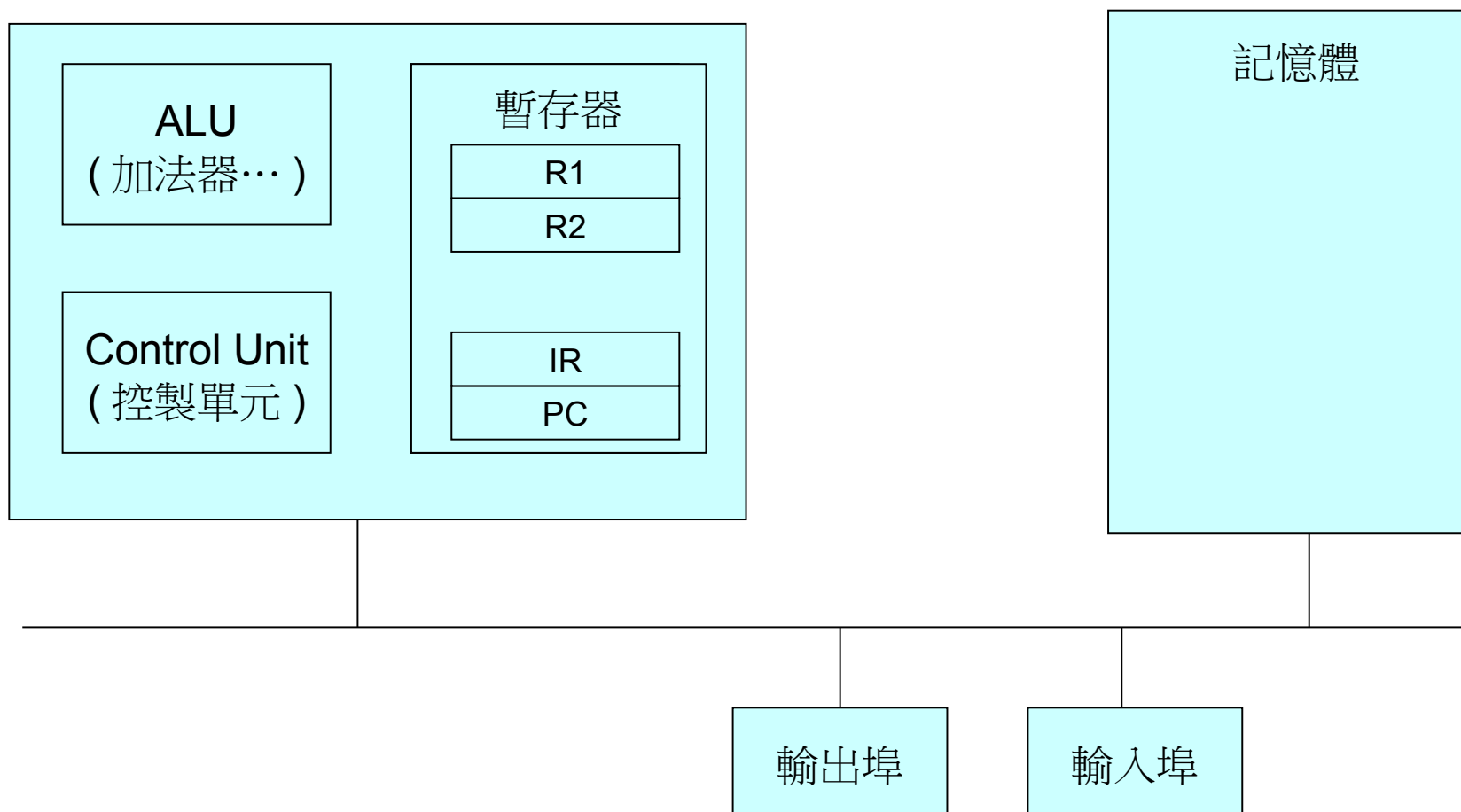
組合語言

機器語言

00 00		LOAD R1, sum
00 04		LOAD R2, i
00 08	LOOP	COMP R2, TEN
00 0C		JGT EXIT
00 10		INC R2
00 14		ADD R1, R2, R1
00 18		JMP LOOP
00 1C	EXIT	STORE R1, sum
00 20		STORE R2, i
00 24		RETURN
00 28		WORD sum 0
00 2C		WORD i 0
00 30		WORD TEN 10

01 01 00 28
01 02 00 2C
04 02 00 30
0A 00 00 1C
1C 02 00 00
03 01 02 01
09 00 00 08
02 01 00 28
02 02 00 2C
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 0A

核心架構（細部）



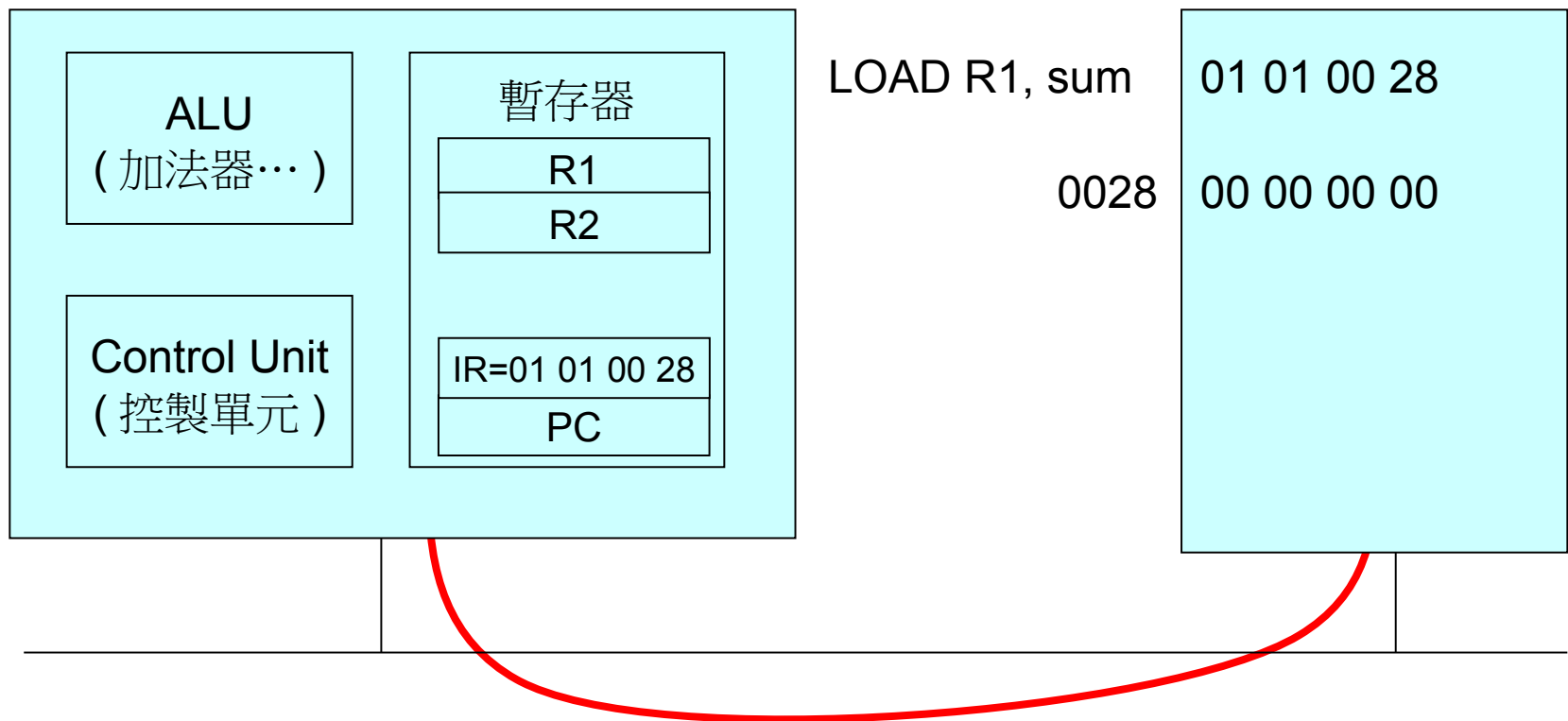
機器指令 LOAD

LOAD R1, sum = 01 01 00 28

LOAD => 01

R1 => 01

40 => 28



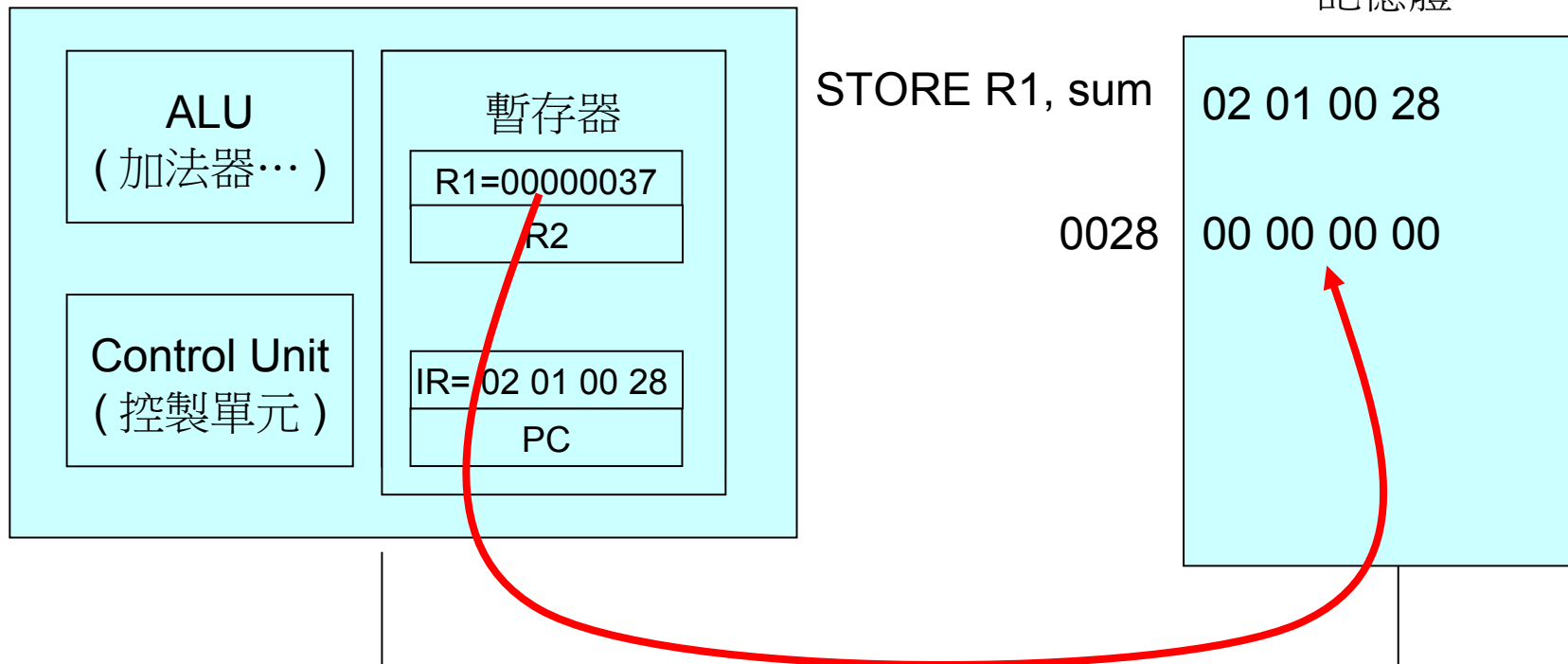
機器指令 STORE

STORE R1, sum = 02 01 00 28

STORE => 02

R1 => 01

sum => 0028



機器指令 ADD R1, R2

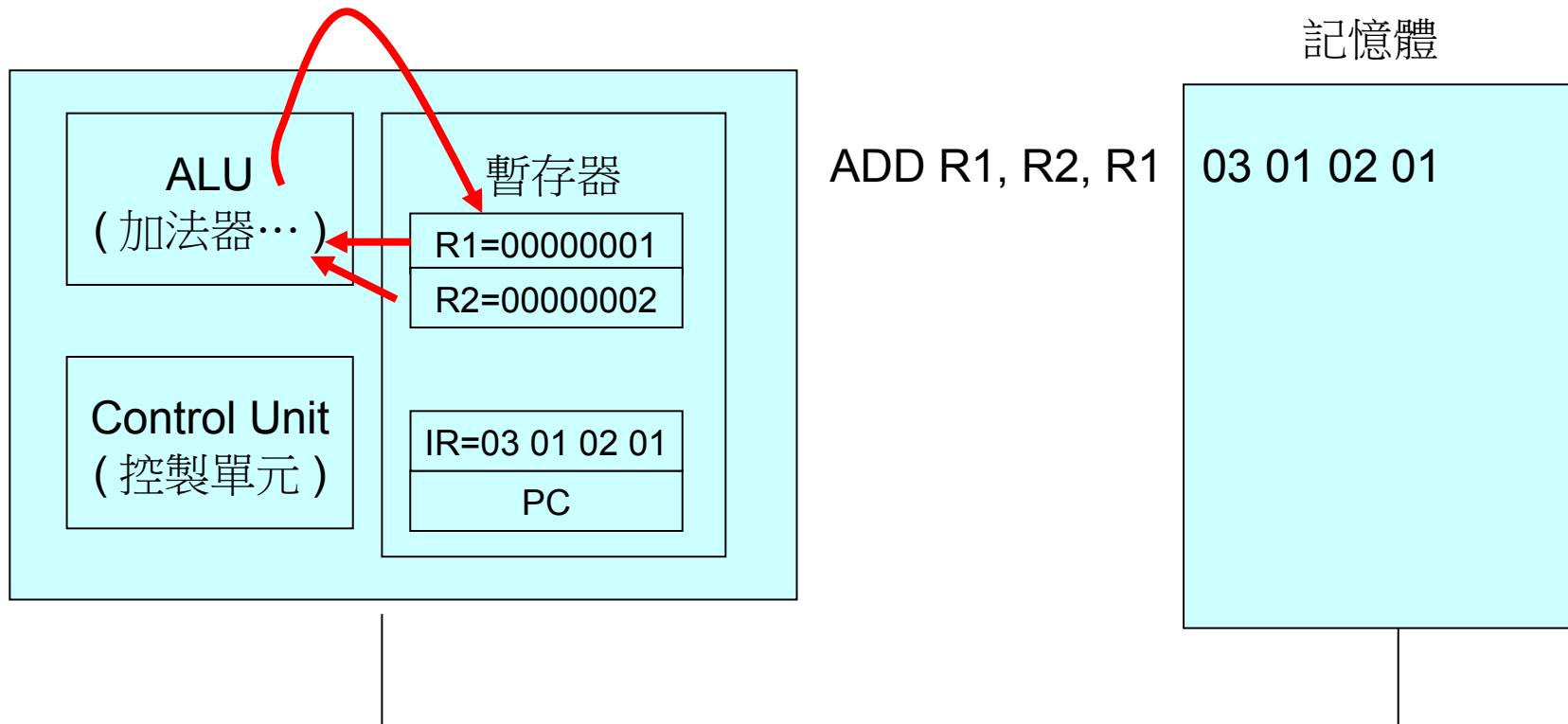
ADD R1, R2, R1 = 03 01 02 01

ADD => 03

R1 => 01

R2 => 02

R1 => 01

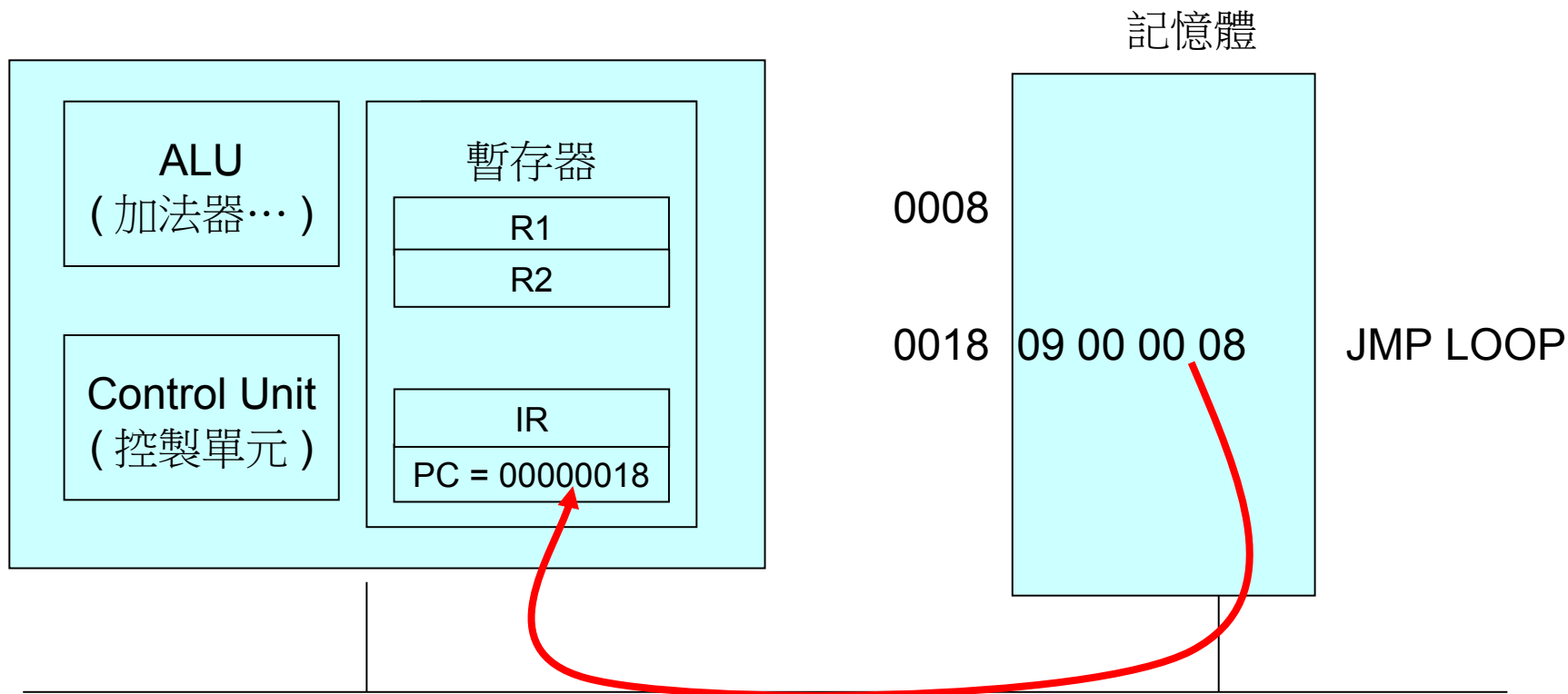


機器指令 JMP

JMP LOOP = 09 00 00 08

JMP => 09

LOOP => 00 08



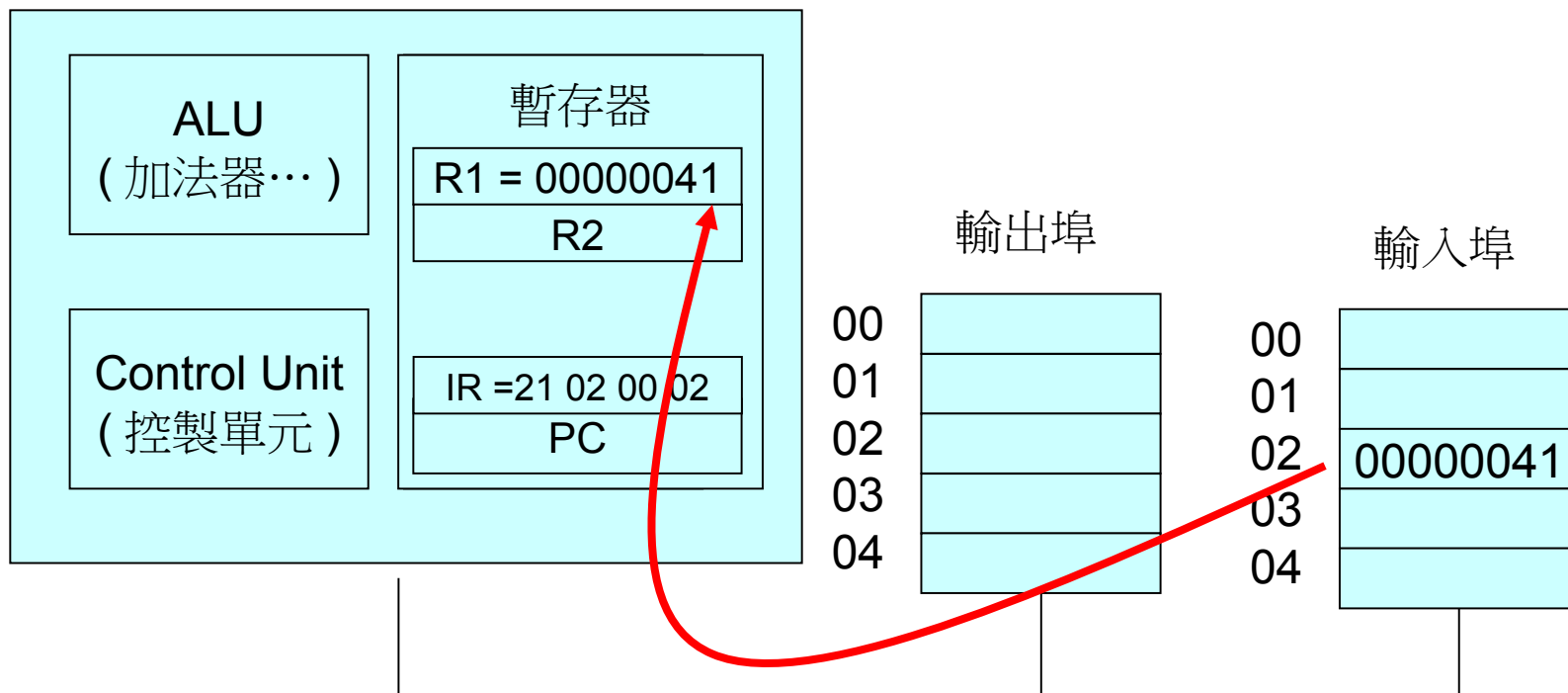
機器指令 READ

READ R2, Keyboard = 21 02 00 02

READ => 21

R2 => 02

Keyboard => 00 02



註：41 是 'A' 的 ASCII 碼

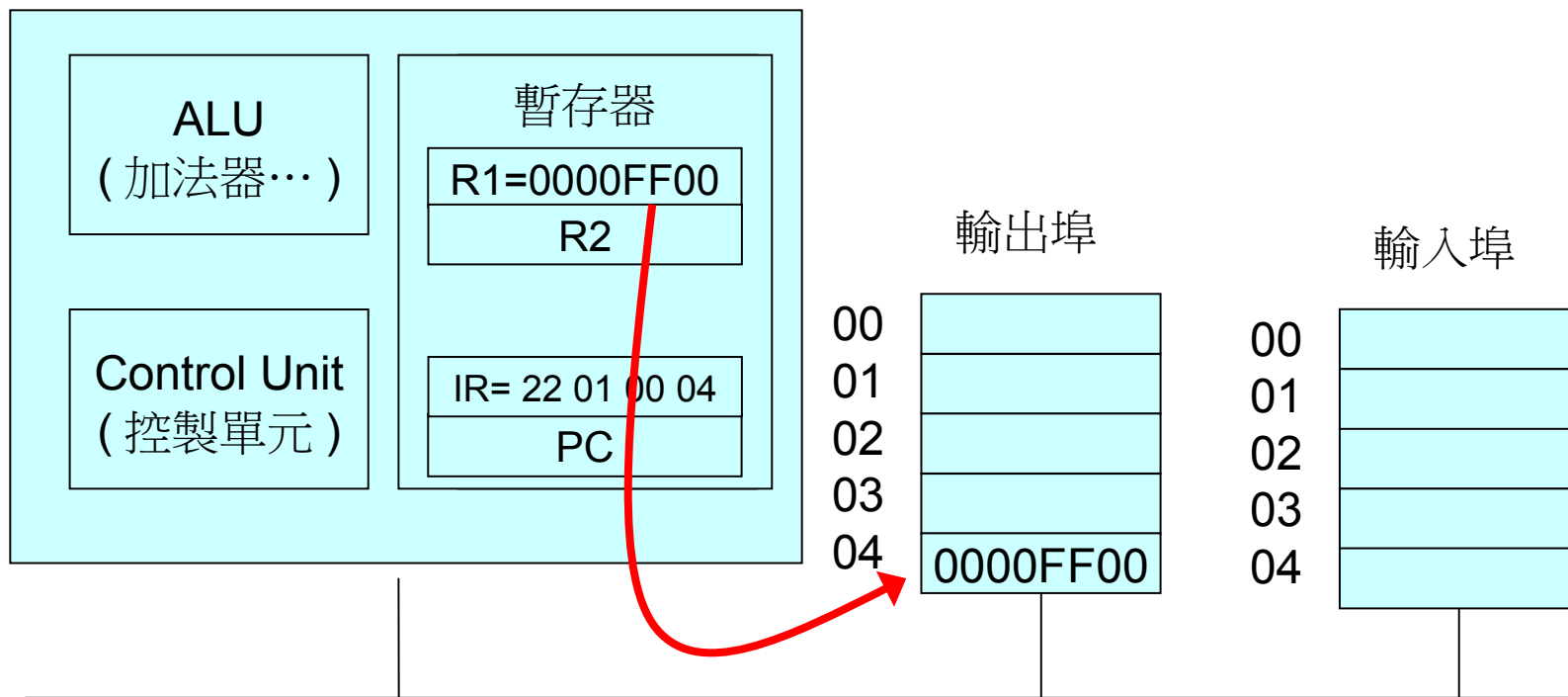
機器指令 WRITE

WRITE R1, Screen = 22 01 00 04

WRITE => 22

R1 => 01

Screen => 00 04



註：00FF00 是 RGB 的綠色

程式 = 指令串
列

機器指令（整合範例 – 組譯）

位址

組合語言

機器語言

00 00
00 04
00 08
00 0C
00 10
00 14
00 18
00 1C
00 20
00 24
00 28
00 2C
00 30

LOOP


EXIT

LOAD R1, sum
LOAD R2, i
COMP R2, TEN
JGT EXIT
INC R2
ADD R1, R2, R1
JMP LOOP
STORE R1, sum
STORE R2, i
RETURN
WORD sum 0
WORD i 0
WORD TEN 10

01 01 00 28
01 02 00 2C
04 02 00 30
0A 00 00 1C
1C 02 00 00
03 01 02 01
09 00 00 08
02 01 00 28
02 02 00 2C
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 0A

從 Java 到 組合語言

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



```
WORD sum    0  
WORD i      0  
  
LOAD  R1, sum  
LOAD  R2, i  
LOOP  COMP R2, 10  
      JGT  EXIT  
      INC  R2  
      ADD  R1, R2, R1  
      JMP  LOOP  
EXIT  STORE sum, R1  
      STORE i, R2
```

機器指令（整合範例 – 組譯）

位址

組合語言

機器語言

00 00
00 04
00 08
00 0C
00 10
00 14
00 18
00 1C
00 20
00 24
00 28
00 2C
00 30

LOOP

EXIT

LOAD R1, sum
LOAD R2, i
COMP R2, TEN
JGT EXIT
INC R2
ADD R1, R2, R1
JMP LOOP
STORE R1, sum
STORE R2, i
RETURN
WORD sum 0
WORD i 0
WORD TEN 10

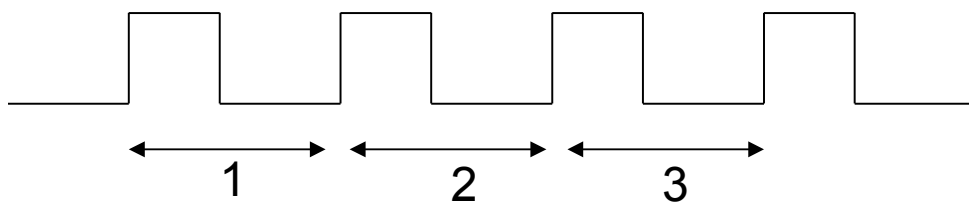
01 01 00 28
01 02 00 2C
04 02 00 30
0A 00 00 1C
1C 02 00 00
03 01 02 01
09 00 00 08
02 01 00 28
02 02 00 2C
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 0A

機器指令（整合範例 – 執行）

位址	組合語言	記憶體	暫存器 PC=0
00 00	LOAD r1, sum	01 01 00 28	R1 = 0
00 04	LOAD r2, i	01 02 00 2C	R2 = 0
00 08	OOP COMP r2, TEN	04 02 00 30	
00 0C	JGT EXIT	0A 00 00 1C	
00 10	INC r2	1C 02 00 00	R2=1 R2 =2 R2 = 3 ...
00 14	ADD r1, r2, r1	03 01 02 01	R1=1 R1 =3 R1 = 6 ...
00 18	JMP LOOP	09 00 00 08	
00 1C	EXIT STORE r1, sum	02 01 00 28	sum = R1 = 55
00 20	STORE r2, i	02 02 00 2C	i = R2 = 10
00 24	RETURN	00 00 00 00	
00 28	WORD sum 0	00 00 00 00	→ 00 00 00 37
00 2C	WORD i 0	00 00 00 00	→ 00 00 00 0A
00 30	WORD TEN 10	00 00 00 0A	

控制單元 (Control Unit)

- 每個指令通常是 3-5 個時脈完成，以 LOAD R1, sum 為例：
 - 時脈 1：（載入指令）將指令載入指令暫存器
 - 時脈 2：（指令解碼）用多工器根據指令的運算碼，進行動作。
 - 假如 運算碼 為 01，則根據將暫存多工器設為 01，記憶體多工器設為 8196，此時資料自動從 mem(8196) 流向暫存器 R1。
 - 時脈 3：將程式計數器加上 4，以便進行下一個指令。

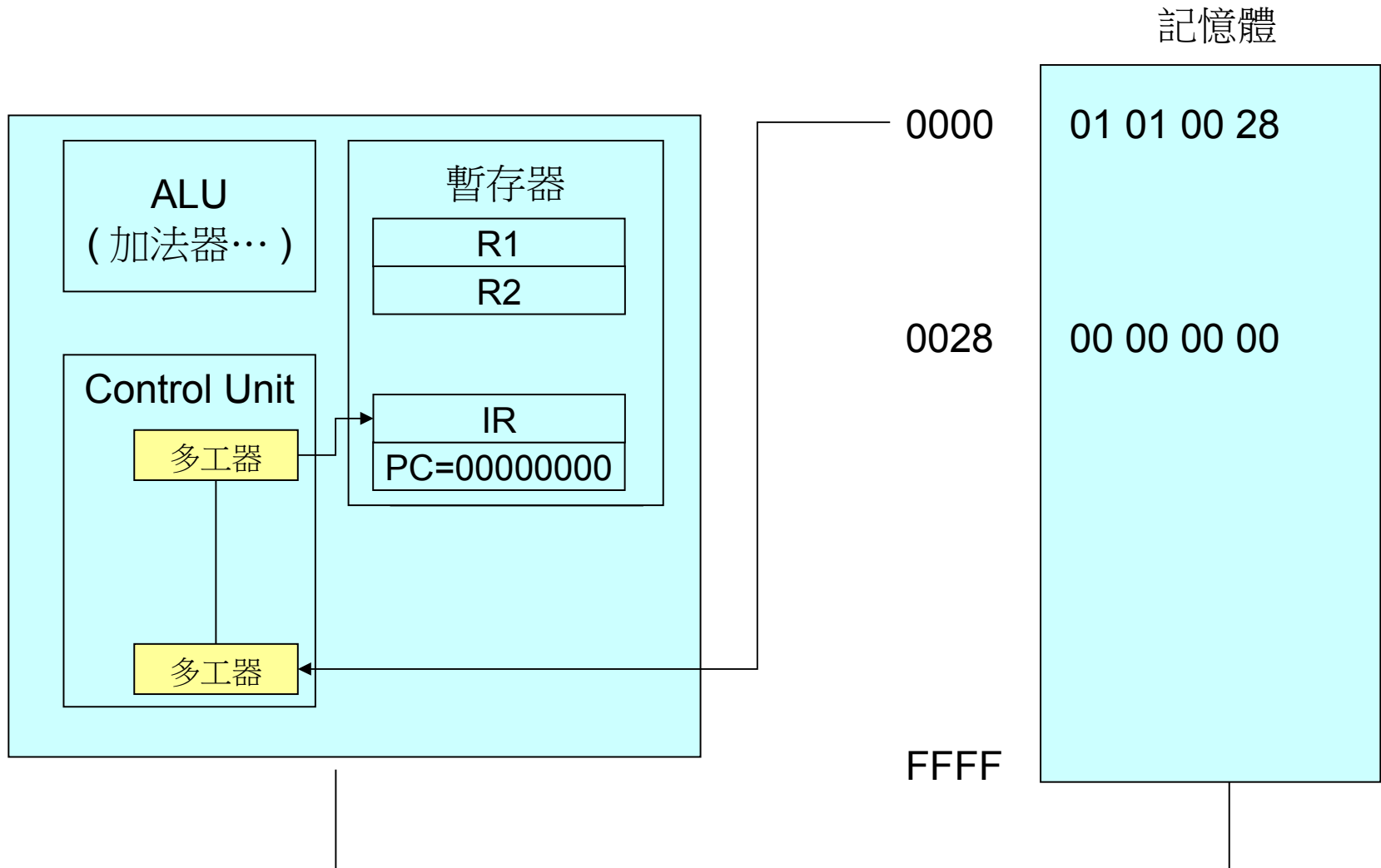


1. IR = 01 01 00 28
2. R1 = mem(0028)
3. PC = PC + 4

時脈 1

LOAD R1, sum = 01 01 00 28

1. **IR = 01 01 00 28**
2. $R1 = \text{mem}(0028)$
3. $PC = PC + 4$



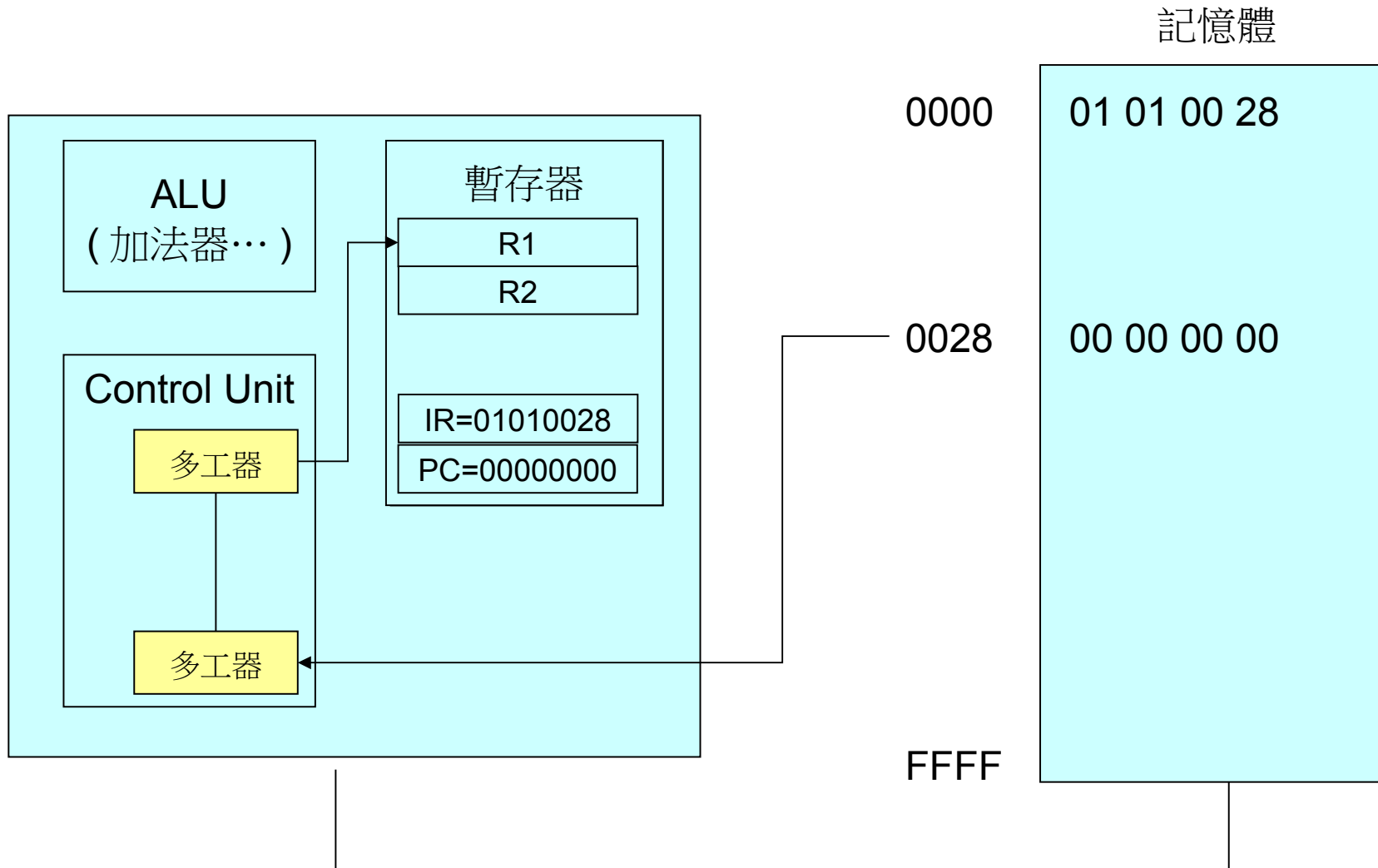
時脈 2

LOAD R1, sum = 01 01 00 28

1. IR = 01 01 00 28

2. R1 = mem(0028)

3. PC = PC + 4



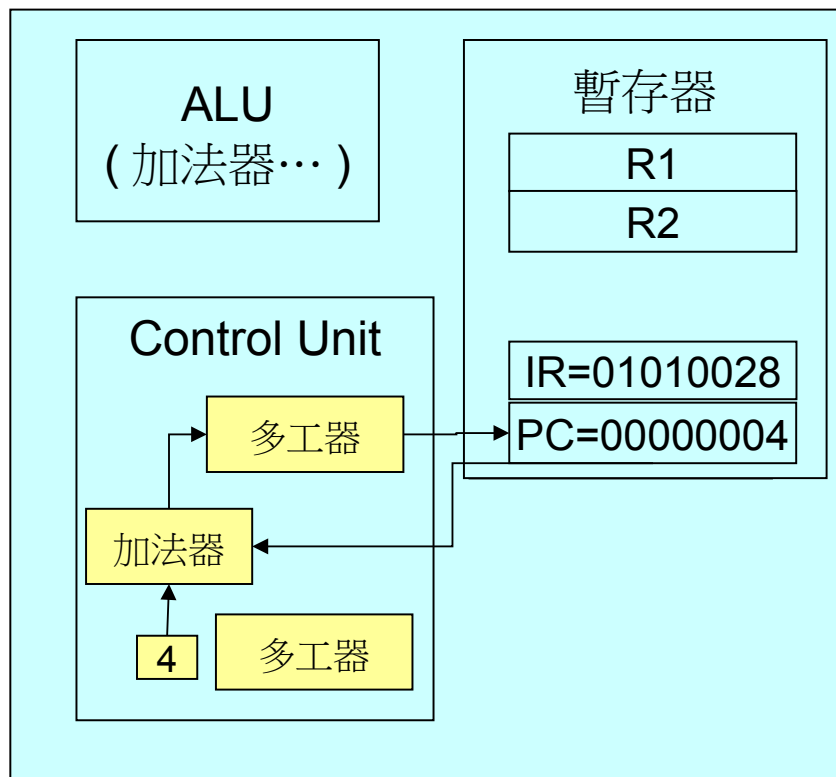
時脈 3

LOAD R1, sum = 01 01 00 28

1. IR = 01 01 00 28

2. R1 = mem(0028)

3. PC = PC + 4



記憶體

0000

01 01 00 28

0028

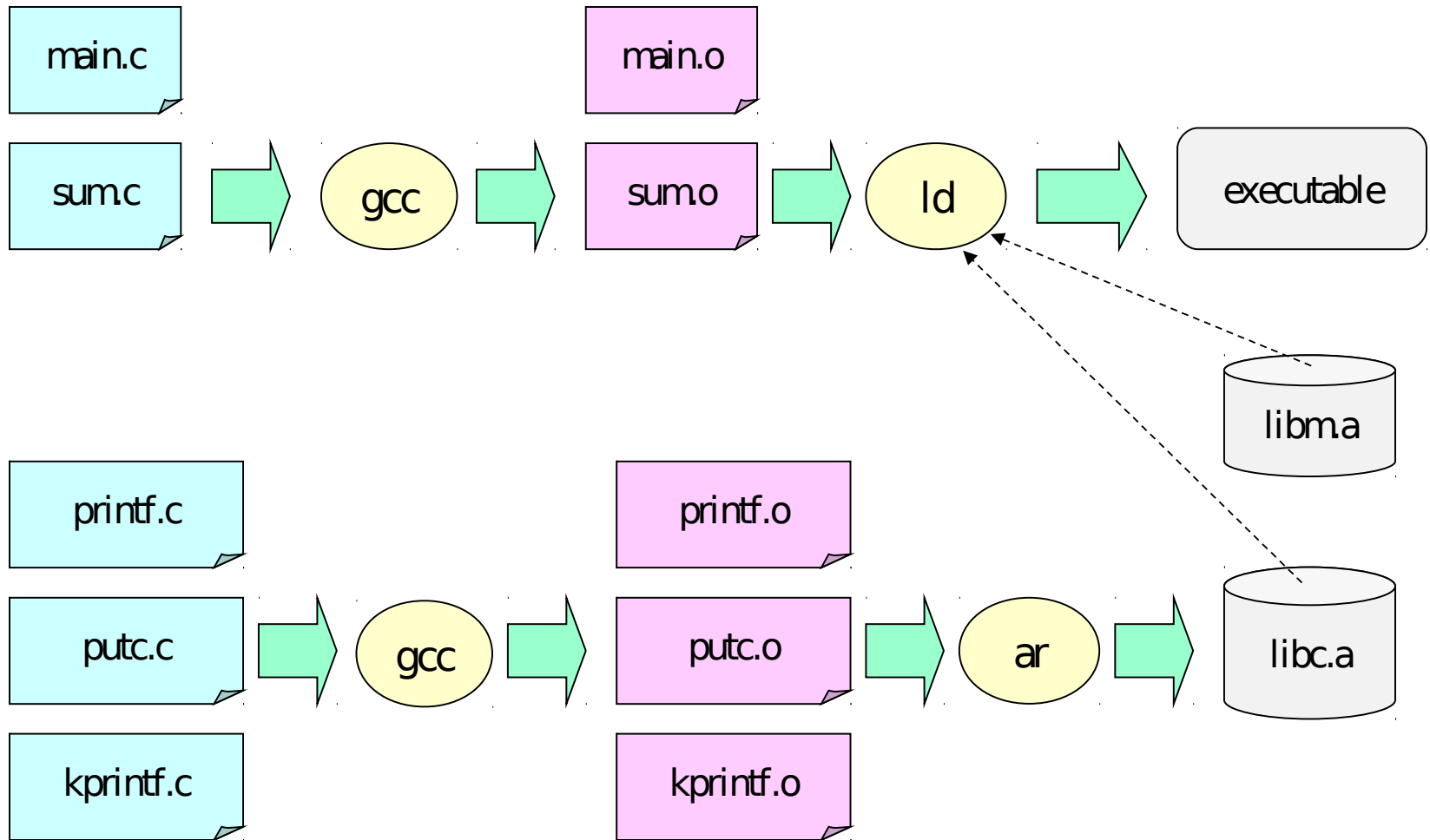
00 00 00 00

FFFF

編譯與組譯－使用 GCC (實務操作)

陳鍾誠 於金門大學

GNU 工具使用的基本流程圖



C 語言程式範例

►範例 1.2 程式 main.c 與 sum.c

C 語言主程式 (main.c)

```
#include <stdio.h>
int main(void) {
    int sum1 = sum(10);
    printf("sum=%d\n", sum1);
    system("pause");
    return 1;
}
```

C 語言函數 (sum.c)

```
int sum(int n) {
    int s=0;
    int i;
    for (i=1; i<=n;i++) {
        s = s + i;
    }
    return s;
}
```

使用 gcc 產生組合語言

- 使用 **-S** 參數可要求 gcc 產生組合語言

```
gcc -S sum.c -o sum.s  
gcc -S main.c -o main.s
```


將 gcc 當成組譯器使用

圖 1.6 將 gcc 當成組譯器使用

```
C:\ch01>gcc -S sum.c -o sum.s
```

```
C:\ch01>gcc -S main.c -o main.s
```

```
C:\ch01>gcc main.s sum.s -o sum2
```

```
C:\ch01>sum2
```

```
sum= 55
```

```
請按任意鍵繼續 . . .
```

同時組譯並連結

圖 1.7 利用 gcc 編譯 C 語言 main.c 同時組譯組合語言 sum.s

```
C:\ch01>gcc main.c sum.s -o sum3
```

```
C:\ch01>sum3
```

```
sum=55
```

```
請按任意鍵繼續 . . .
```

IA32 的組合語言 (實務操作)

陳鍾誠 於金門大學

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

```
C:\ch03>gcc main.c gnu_add.s -o add
```

```
C:\ch03>add
```

```
asmMain()=3
```

說明

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

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

```
C:\ch03>gcc main.c gnu_sum.s -o sum
```

```
C:\ch03>sum
```

```
asmMain()=55
```

說明

編譯 main.c、組譯 gnu_sum.s 並連結為 sum.exe

執行 sum.exe

輸出結果為 55