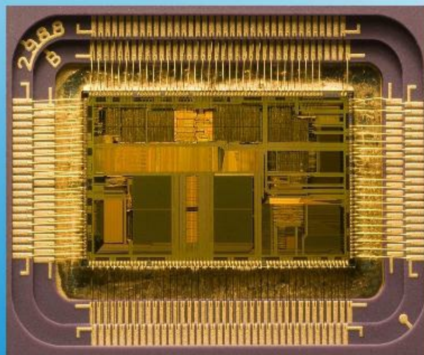
「開放電腦計畫」之

系統程式

使用 JavaScript 實作

- *虛擬機 VM0*
- *組譯器 AS0*
- *編譯器 J0C*
- *中間碼 IR0*



作者：陳鍾誠

– 本書部分圖片與內容來自維基百科

採用「創作共用」的「姓名標示、相同方式分享」之授權

開放電腦計畫 -- 系統程式

2014 年 8 月出版

作者：陳鍾誠衍生自維基百科（創作共用：姓名標示、相同方式分享授權）

開放電腦計畫 -- 系統程式

- 前言
  - 序
  - 授權聲明
- 開放電腦計畫
  - 硬體：計算機結構
  - 軟體：系統程式
  - 參考文獻
- CPU0 處理器
  - CPU0 指令集
  - 實作：CPU0 的指令表
- 虛擬機 - vm0
  - 組譯範例
  - VM0 虛擬機設計
  - 結語
- 組譯器 - as0
  - 組譯範例
  - AS0 組譯器設計
  - 程式說明
  - 結語
- 編譯器
  - 編譯器：高階語言轉中間碼 - j0c
  - 編譯器：中間碼轉組合語言 - ir2as
- 結語

前言

序

本書是「開放電腦計畫的軟體部份」，描述如何設計系統軟體的方法，透過這本書，我們希望讓「系統程式」這門課變成是很容易理解並實作的。

我們相信，透過實作的訓練，您將對理論會有更深刻的體會，而這些體會，將會進一步讓您更瞭解「現代電腦工業的結構」是如何建構出來的。

授權聲明

本書許多資料修改自維基百科，採用 創作共用：姓名標示、相同方式分享 授權，若您想要修改本書產生衍生著作時，至少應該遵守下列授權條件：

1. 標示原作者姓名 (陳鍾誠+維基百科)。
2. 採用 創作共用：姓名標示、相同方式分享 的方式公開衍生著作。

另外、當本書中有文章或素材並非採用 姓名標示、相同方式分享 時，將會在該文章或素材後面標示其授權，此時該文章將以該標示的方式授權釋出，請修改者注意這些授權標示，以避免產生侵權糾紛。

例如有些文章可能不希望被作為「商業性使用」，此時就可能會採用創作共用：姓名標示、非商業性、相同方式分享 的授權，此時您就不應當將該文章用於商業用途上。

最後、若讀者有需要轉貼或修改這些文章使用，請遵守「創作共用」的精神，讓大家都可以在「開放原始碼」的基礎上逐步改進這些作品。

開放電腦計畫

如果您是資工系畢業的學生，必然會上過「計算機結構、編譯器、作業系統、系統程式」等等課程， 這些課程都是設計出一台電腦所必需的基本課程。但是如果有人問您「您是否會設計電腦呢？」，相信大部分人 的回答應該是：「我不會，也沒有設計過」。

光是設計一個作業系統，就得花上十年的工夫，遑論還要自己設計「CPU、匯流排、組譯器、編譯器、作業系統」 等等。因此，我們都曾經有過這樣的夢想，然後在年紀越大，越來越瞭解整個工業結構之後，我們就放棄了這樣 一個夢想，因為我們必須與現實妥協。

但是，身為一個大學教師，我有責任教導學生，告訴他們「電腦是怎麼做出來的」，因此我不自量力的提出了 這樣一個計畫，那就是「開放電腦計畫」，我們將以「千里之行、始於足下」的精神，設計出一台全世界最簡單 且清楚的「電腦」，包含「軟體與硬體」。

從 2007 年我開始寫「系統程式」這本書以來，就有一個想法逐漸在內心發酵， 這個想法就是：「我想從 CPU 設計、組譯器、虛擬機、編譯器到作業系統」，自己打造一台電腦，於是、「開放電腦計畫」就誕生了！

那麼、開放電腦計畫的「產品」會是什麼呢？

應該有些人會認為是一套自行編寫的軟硬體程式，當然、這部份是包含在「開放電腦計畫」當中的。

但是、更重要的事情是，我們希望透過「開放電腦計畫」讓學生能夠學會整個「電腦的軟硬體設計方式」，並且透過這個踏腳石瞭解整個「電腦軟硬體工業」，進而能夠達到「以理論指導實務、以實務驗證理論」的目標。

為了達成這個目標，我們將「開放電腦計畫」分成三個階段，也就是「簡單設計 (程式) => 理論闡述 (書籍) => 開源實作 (工業軟硬體與流程)」，整體的構想說明如下：

1. 簡單設計(程式)： 採用 Verilog + C 設計「CPU、組譯器、編譯器、作業系統」等軟硬體，遵循 KISS (Keep It Simple and Stupid) 原則，不考慮「效能」 與「商業競爭力」等問題，甚至在實用性上進行了不少妥協，一律採用「容易理解」為最高指導原則，目的是清楚的展現整個「軟硬體系統」的架構。
2. 理論闡述(書籍)： 但是、要瞭解像「處理器、系統軟體、編譯器、作業系統」這些領域，只有程式是不夠的。因為程式通常不容易懂，而且對於沒有背景知識的人而言，往往難如天書。所以我們將撰寫一系列書籍，用來說明上述簡單程式的設計原理，然後開始進入「計算機結構、編譯器、作業系統、系統程式」的理論體系中，導引出進一步的設計可能性與工業考量等議題。
3. 開源實作(工業)：一但有了前述的理論與實作基礎之後，我們就會採用「開放原始碼」來進行案例研究。舉例而言、在「計算機結構」上我們會以 ARM 為實務核心、「編譯器」領域則以 gcc, LLVM 為研究標的，「作業系統」上則會對 FreeRTOS、Linux 等進行案例研究，「虛擬機」上則會以 QEMU、V8 等開源案例為研究對象。

軟體：系統程式

開發環境	gcc icarus node.js	arduino	CodeSourcery GNU tool chain		GNU tool chain
虛擬機	vm0	IMAVR	QEMU / v8		
作業系統	os0		FreeRTOS	Linux	
編譯器	cc0	avrg++	llvm/gcc/g++		
組譯器	as0	avrasm	as		

硬體：計算機結構

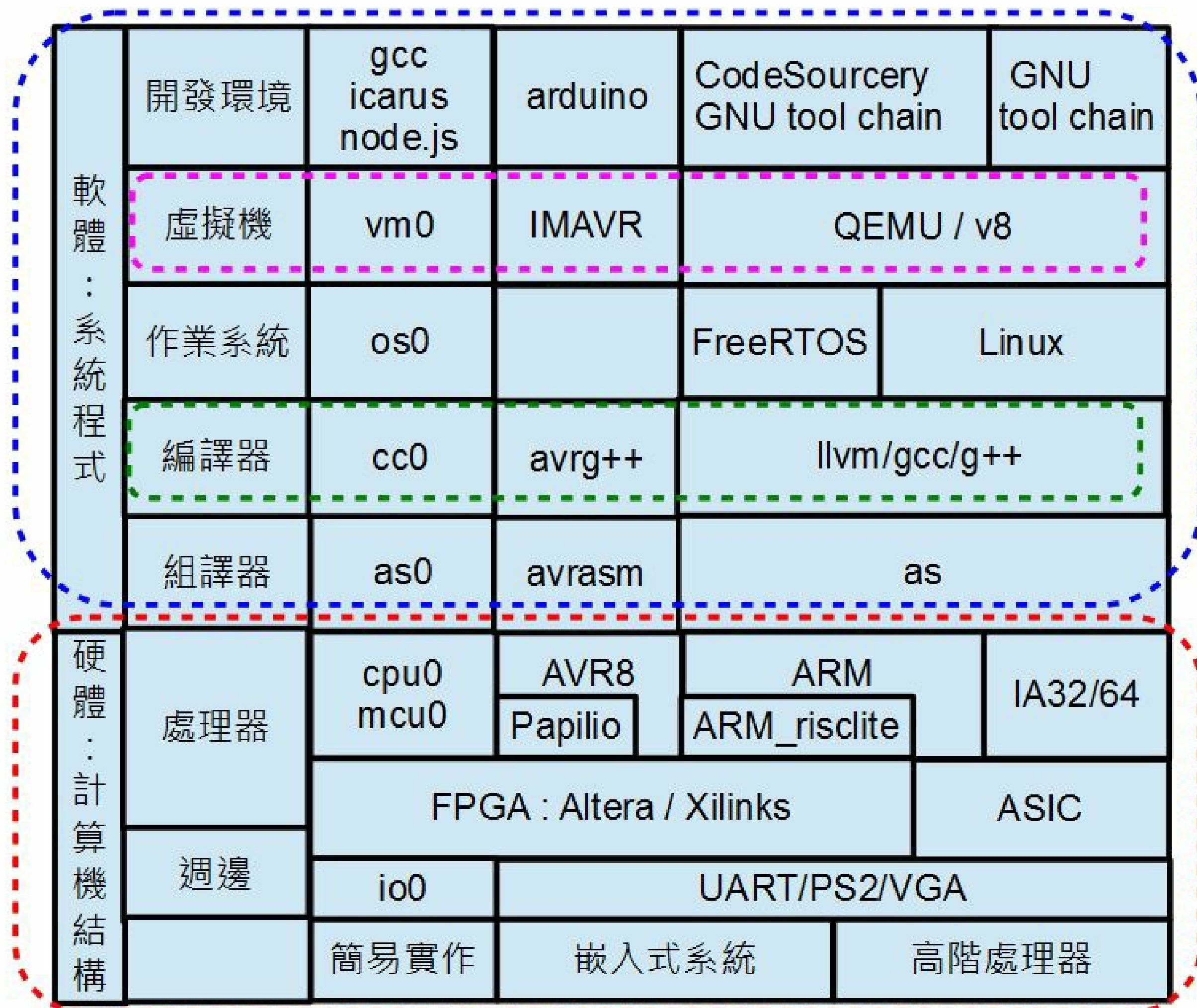
處理器	cpu0 mcu0	AVR8 Papilio	ARM ARM_risclite		IA32/64
	FPGA : Altera / Xilinks			ASIC	
週邊	io0	UART/PS2/VGA			
	簡易實作	嵌入式系統		高階處理器	

圖、開放電腦計畫地圖

根據以上規劃，本書乃為一系列書籍中的一本，完整的書籍架構如下：

開放電腦計畫書籍	簡易程式	工業實作
系統程式	as0, vm0, cc0, os0	gcc/llvm
計算機結構	mcu0, cpu0	ARM/OpenRISC
編譯器 c	0c, j0c g	cc/llvm
作業系統	os0, XINU, MINIX	FreeRTOS, Linux

這些書籍分別描述不同的面向，其涵蓋範圍如下圖所示：



圖、開放電腦計畫書籍圖

硬體：計算機結構

在硬體方面，我們將自行設計兩款處理器，一款是用來展示簡單「微處理器」設計原理的16 位元微控制器 MCU0，而另一款則是用來展示「高階處理器」設計原理的 32 位元處理器 CPU0。

透過 MCU0，我們希望展示一顆「最簡易微處理器」的設計方法，我們將採用「流程式」與「區塊式」的方法分別實作一遍，讓讀者可以分別從「硬體人」與「軟體人」的角度去體會處理器的設計方式。由於「流程式」的方法比較簡單，因此我們會先用此法進行設計，當讀者理解何謂「微處理器」之後，在將同樣的功能改用「區塊式的方法」實作一遍，這樣應該就能逐漸「由易至難、由淺入深」了。

在 MCU0 當中，我們採用「CPU 與記憶體」合一的設計方式，這種方式比較像「系統單晶片」(SOC) 的設計方法，其記憶體容量較小，因此可以直接用 Verilog 陣列宣告放入 FPGA 當中使用，不需考慮外部 DRAM 存取速度較慢的問題，也不用考慮「記憶階層」的速度問題，因此設計起來會相對容易許多。

接著，我們將再度設計一個 32 位元的處理器 -- CPU0。並透過 CPU0 來討論「當 CPU 速度比 DRAM 記憶體快上許多」的時候，如何能透過快取 (cache) 與記憶體管理單元 (MMU) 達到「又快又大」的目的，並且討論如何透過「流水線」架構 (Pipeline) 達到加速的目的，這些都屬於「高階處理器」所需要討論的問題。

軟體：系統程式

有了 MCU0 與 CPU0 等硬體之後，我們就可以建構運作於這些硬體之上的軟體了，這些軟體包含「組譯器、虛擬機、編譯器、作業系統」等等。

我們已經分別用 C 與 JavaSript 建構出簡易的「組譯器、虛擬機、編譯器」工具了，讓我們先說明一下在 CPU0 上這些程式的使用方法，以下示範是採用 node.js+Javascript 實作的工具版本，因此必須安裝 node.js 才能執行。

組合語言 (Assembly Language)

接著、讓我們從組合語言的角度，來看看 CPU0 處理器的設計，以下是一個可以計算 1+2+...+10 的程式，計算完成之後會透過呼叫軟體中斷 SWI 程序 (類似 DOS 時代的 INT 中斷)，在螢幕上印出下列訊息。

```
1+...+10=55
```

以下的檔案 sum.as0 正是完成這樣功能的一個 CPU0 組合語言程式。

檔案：sum.as0

```
        LD      R1, sum         ; R1 = sum = 0
        LD      R2, i           ; R2 = i = 1
        LDI     R3, 10          ; R3 = 10
FOR:    CMP     R2, R3          ; if (R2 > R3)
        JGT     EXIT            ;   goto EXIT
        ADD     R1, R1, R2      ; R1 = R1 + R2 (sum = sum + i)
        ADDI    R2, R2, 1       ; R2 = R2 + 1  ( i  = i + 1)
        JMP     FOR             ; goto FOR
EXIT:   ST      R1, sum         ; sum = R1
        ST      R2, i           ; i = R2
        LD      R9, msgptr      ; R9= pointer(msg) = &msg
        SWI     3               ; SWI 3 : 印出 R9 (=&msg) 中的字串
        MOV     R9, R1          ; R9 = R1 = sum
        SWI     4               ; SWI 4 : 印出 R9 (=R1=sum) 中的整數
        RET                     ; return 返回上一層呼叫函數
i:      RESW    1               ; int i
sum:    WORD    0               ; int sum=0
msg:    BYTE    "1+...+10=", 0    ; char *msg = "sum="
msgptr: WORD    msg             ; char &msgptr = &msg
```

組譯器 (Assembler)

我們可以用以下指令呼叫「組譯器 AS0」對上述檔案進行組譯：

```
node as0 sum.as0 sum.ob0
```

上述的程式經過組譯之後，會輸出組譯報表，如下所示。

sum.as0 的組譯報表

```
0000          LD      R1,sum          L 00 001F003C
0004          LD      R2,i            L 00 002F0034
0008          LDI     R3,10           L 08 0830000A
000C FOR      CMP     R2,R3           A 10 10230000
0010          JGT     EXIT            J 23 2300000C
0014          ADD     R1,R1,R2        A 13 13112000
0018          ADDI    R2,R2,1         A 1B 1B220001
001C          JMP     FOR             J 26 26FFFFEC
0020 EXIT     ST      R1,sum          L 01 011F001C
0024          ST      R2,i            L 01 012F0014
0028          LD      R9,msgptr       L 00 009F0022
002C          SWI     3               J 2A 2A000003
0030          MOV     R9,R1           A 12 12910000
0034          SWI     2               J 2A 2A000002
0038          RET                     J 2C 2C000000
003C i        RESW    1               D F0 00000000
0040 sum      WORD    0               D F2 00000000
0044 msg      BYTE    "1+...+10=",0   D F3 312B2E2E2E2B31303D00
004E msgptr   WORD    msg             D F2 00000044
```

最後「組譯器 AS0」會輸出機器碼到目的檔 sum.ob0 當中，其內容如下所示。

sum.as0 的機器碼 (以 16 進位顯示)

```
001F003C 002F0034 0830000A 10230000
2300000C 13112000 1B220001 26FFFFEC
011F001C 012F0014 009F0022 2A000003
12910000 2A000002 2C000000 00000000
00000000 312B2E2E 2E2B3130 3D000000
0044
```

虛擬機 (Virtual Machine)

如果我們用「虛擬機 VM0」去執行上述的目的檔 sum.ob0，會看到程式的執行結果，是在 螢幕上列印出 1+...+10=55，以下是我們的操作過程。

```
1+...+10=55
```

編譯器 (Compiler)

當然、一個完整的現代電腦應該包含比組譯器更高階的工具，不只支援組合語言，還要支援高階語言。

因此、我們設計了一個稱為 J0 的高階語言，語法有點像 JavaScript，但卻是經過簡化的版本。

然後、我們又設計了一個可以用來編譯 J0 語言的編譯器，稱為 J0C (J0 Compiler)，可以用來將 J0 語言編譯成中間碼， 也可以直接將中間碼轉換為 CPU0 的組合語言。

以下是一個 J0 語言的範例，

檔案：sum.j0

```
s = sum(10);
return s;

function sum(n) {
  s = 0;
  i=1;
  while (i<=10) {
    s = s + i;
    i++;
  }
  return s;
}
```

當我們使用 j0c 編譯器將上述程式編譯之後，會輸出兩個檔案，一個是 sum.ir，是編譯器中間格式 (Intermediate Representation, 虛擬碼 pcode) 的輸出檔， 其內容如下：

```
D:\Dropbox\Public\web\oc\code>node j0c sum
         arg      10
         call     T1       sum
         =        s        T1
         return   s
sum      function
         param    n
         =        s        0
         =        i        1
L1
         <=       T2       i        10
```

```
        if0     T2      L2
        +       T3      s       i
        =       s       T3
        ++      i
        goto    L1
L2
        return  s
        endf
```

另一個是將上述中間格式轉換成轉換成 CPU0 組合語言之後的結果，如下所示：

```
sum
        POP     n
        LDI     R1      0
        ST      R1      s
        LDI     R1      1
        ST      R1      i
L1
        LD      R1      i
        LDI     R2      10
        LDI     R3      0
        CMP     R1      R2
        JLE     else1
        LDI     R3      1
else1
        ST      R3      T1
        LDI     R1      T1
        CMP     R1      0
        JEQ     L2
        LD      R1      s
        LD      R2      i
        ADD     R3      R1      R2
        ST      R3      T2
        LDI     R1      T2
        ST      R1      s
        LD      R1      i
        ADDI    R1      R1      1
        ST      R1      i
```

```
        JMP     L1
L2
        LD      R1      s
        RET
        LDI     R1      10
        PUSH    R1
        CALL    sum
        ST      R1      T3
        LDI     R1      T3
        ST      R1      s

s       WORD    0
i       WORD    0
T1      WORD    0
T2      WORD    0
T3      WORD    0
```

上述由 j0c 所編譯產生的組合語言，感覺相對冗長，是因為這個編譯器是最簡版本，完全沒有做任何優化動作，甚至連暫存器都是 每次重新載入的，所以效率並不會很好。

作業系統 (Operating System)

當然囉！一個完整的電腦還必須要有作業系統，不過如果是嵌入式系統的話，沒有作業系統也沒關係，只要將全部的程式連結在一起， 就可以形成一台電腦了，目前開放電腦計畫的「作業系統」還在研究開發當中，希望很快就能提供大家一個最簡單的作業系統版本。

目前我們已經寫了一個可以進行兩個行程切換 「Task Switching」 範例，接著我們將參考 UNIXv6, L4 等作業系統，以建構更 完整的簡易作業系統。

當然、即使我們從 CPU 硬體一路設計到組譯器、虛擬機、編譯器、作業系統等，未來仍然有更多領域等待我們去探索，例如「網路模組、TCP/IP、 Ethernet、無線 RF 的硬體模組、繪圖卡、OpenGL、.....」等等，希望我們能夠用最簡單的話語，將這些電腦的原理說明清楚，並用簡單的方式 實作得更完整。

參考文獻

- 陳鍾誠的網站/免費電子書：Verilog 電路設計
- 系統程式 陳鍾誠著, 旗標出版社.
- JavaScript (6) – Node.js 命令列程式設計

CPU0 處理器

商用的處理器通常因為強調效能的原因，其設計都會讓指令格式變得難懂且複雜。

為了讓處理器變得更簡單，更容易理解，我們設計了 CPU0 處理器，這個處理計的設計原則是 KISS，也就是 Keep It Simple and Stupid。

CPU0 的設計不求速度，只求清楚易懂，因此在指令級與指令格式上都盡量簡單，編碼都以 4 位元為單位，因此也很容易可以用人腦將組合語言翻成機器碼。

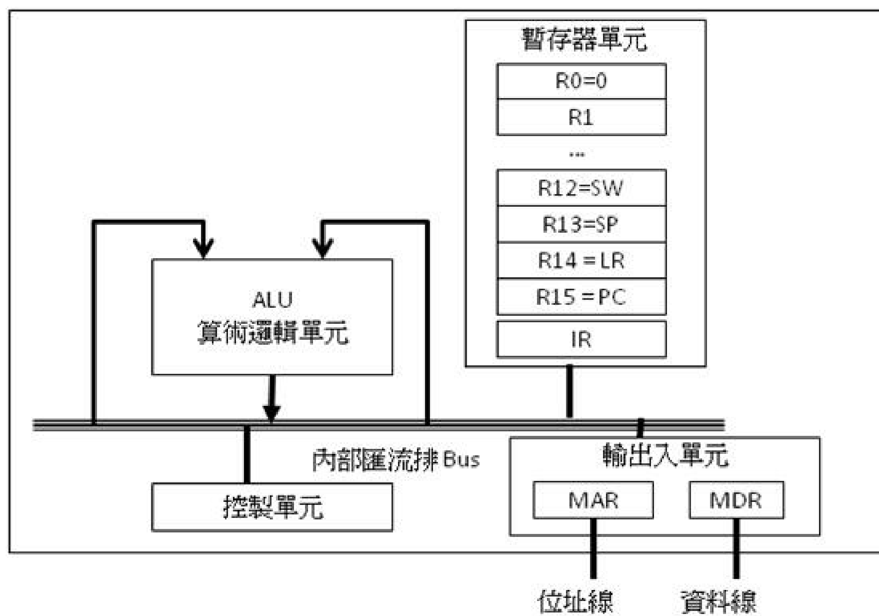
以下是一個 CPU0 的組合語言範例，該程式所作的事情是計算 1+2+...+10 的結果，最後應該會得到總和為 55。

檔案：sum.as0

```
        LD      R1, sum        ; R1 = sum = 0
        LD      R2, i          ; R2 = i = 1
        LDI     R3, 10         ; R3 = 10
FOR:    CMP     R2, R3         ; if (R2 > R3)
        JGT     EXIT           ;   goto EXIT
        ADD     R1, R1, R2     ; R1 = R1 + R2 (sum = sum + i)
        ADDI    R2, R2, 1      ; R2 = R2 + 1  ( i  = i + 1)
        JMP     FOR            ; goto FOR
EXIT:   ST      R1, sum        ; sum = R1
        ST      R2, i          ; i = R2
        LD      R9, msgptr     ; R9= pointer(msg) = &msg
        SWI     3              ; SWI 3 : 印出 R9 (=&msg) 中的字串
        MOV     R9, R1         ; R9 = R1 = sum
        SWI     4              ; SWI 4 : 印出 R9 (=R1=sum) 中的整數
        RET                    ; return 返回上一層呼叫函數
i:      RESW    1              ; int i
sum:    WORD    0              ; int sum=0
msg:    BYTE    "1+...+10=", 0   ; char *msg = "sum="
msgptr: WORD    msg            ; char &msgptr = &msg
```

CPU0 指令集

CPU0 是一個簡易的 32 位元單匯流排處理器，其架構如下圖所示，包含R0..R15, IR, MAR, MDR 等暫存器，其中 IR是指令暫存器， R0 是一個永遠為常數 0 的唯讀暫存器，R15 是程式計數器 (Program Counter : PC)，R14 是連結暫存器 (Link Register : LR)， R13 是堆疊指標暫存器 (Stack Pointer : SP)，而 R12 是狀態暫存器 (Status Word : SW)。



圖、CPU0 的架構圖

CPU0 包含『載入儲存』、『運算指令』、『跳躍指令』、『堆疊指令』等四大類指令，以下表格是 CPU0 的指令編碼表，記載了 CPU0 的指令集與每個指令的編碼。

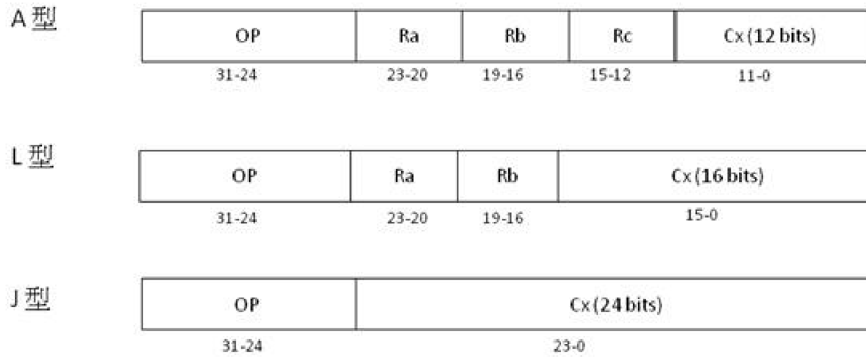
格式	指令	OP	說明	語法	語意
L	LD	00	載入word	LD Ra, [Rb+Cx]	Ra=[Rb+Cx]
L	ST	01	儲存word	ST Ra, [Rb+Cx]	Ra=[Rb+Cx]
L	LDB	02	載入 byte	LDB Ra, [Rb+Cx]	Ra=(byte)[Rb+Cx]
L	STB	03	儲存 byte	STB Ra, [Rb+Cx]	Ra=(byte)[Rb+Cx]
A	LDR	04	LD的暫存器版	LDR Ra, [Rb+Rc]	Ra=[Rb+Rc]
A	STR	05	ST的暫存器版	STR Ra, [Rb+Rc]	Ra=[Rb+Rc]
A	LBR	06	LDB的暫存器版	LBR Ra, [Rb+Rc]	Ra=(byte)[Rb+Rc]
A	SBR	07	STB的暫存器版	SBR Ra, [Rb+Rc]	Ra=(byte)[Rb+Rc]
L	LDI	08	載入常數	LDI Ra, Cx	Ra=Cx
A	CMP	10	比較	CMP Ra, Rb	SW=Ra >=< Rb
A	MOV	12	移動	MOV Ra, Rb	Ra=Rb
A	ADD	13	加法	ADD Ra, Rb, Rc	Ra=Rb+Rc
A	SUB	14	減法	SUB Ra, Rb, Rc	Ra=Rb-Rc
A	MUL	15	乘法	MUL Ra, Rb, Rc	Ra=Rb*Rc

A	DIV	16	除法	DIV Ra, Rb, Rc	Ra=Rb/Rc
A	AND	18	邏輯 AND	AND Ra, Rb, Rc	Ra=Rb and Rc
A	OR	19	邏輯 OR	OR Ra, Rb, Rc	Ra=Rb or Rc
A	XOR	1A	邏輯 XOR	XOR Ra, Rb, Rc	Ra=Rb xor Rc
A	ADDI	1B	常數加法	ADDI Ra, Rb, Cx	Ra=Rb + Cx
A	ROL	1C	向左旋轉	ROL Ra, Rb, Cx	Ra=Rb rol Cx
A	ROR	1D	向右旋轉	ROR Ra, Rb, Cx	Ra=Rb ror Cx
A	SHL	1E	向左移位	SHL Ra, Rb, Cx	Ra=Rb << Cx
A	SHR	1F	向右移位	SHR Ra, Rb, Cx	Ra=Rb >> Cx
J	JEQ	20	跳躍 (相等)	JEQ Cx	if SW(=) PC=PC+Cx
J	JNE	21	跳躍 (不相等)	JNE Cx	if SW(!=) PC=PC+Cx
J	JLT	22	跳躍 (<)	JLT Cx	if SW(<) PC=PC+Cx
J	JGT	23	跳躍 (>)	JGT Cx	if SW(>) PC=PC+Cx
J	JLE	24	跳躍 (<=)	JLE Cx	if SW(<=) PC=PC+Cx
J	JGE	25	跳躍 (>=)	JGE Cx	if SW(>=) PC=PC+Cx
J	JMP	26	跳躍 (無條件)	JMP Cx	PC=PC+Cx
J	SWI	2A	軟體中斷	SWI Cx	LR=PC; PC=Cx; INT=1
J	CALL	2B	跳到副程式	CALL Cx	LR=PC; PC=PC+Cx
J	RET	2C	返回	RET	PC=LR
J	IRET	2D	中斷返回	IRET	PC=LR; INT=0
A	PUSH	30	推入word	PUSH Ra	SP-=4; [SP]=Ra;
A	POP	31	彈出 word	POP Ra	Ra=[SP]; SP+=4;
A	PUSHB	32	推入 byte	PUSHB Ra	SP--; [SP]=Ra; (byte)
A	POPB	33	彈出 byte	POPB Ra	Ra=[SP]; SP++; (byte)

CPU0 指令格式

CPU0 所有指令長度均為 32 位元，這些指令也可根據編碼方式分成三種不同的格式，分別是 A 型、J 型與 L 型。

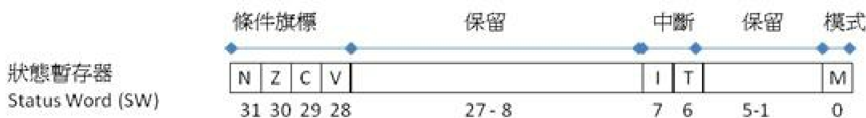
大部分的運算指令屬於A (Arithmatic) 型，而載入儲存指令通常屬於 L (Load & Store) 型，跳躍指令則通常屬於 J (Jump) 型， 這三種型態的指令格式如下圖所示。



圖、CPU0的指令格式

狀態暫存器

R12 狀態暫存器 (Status Word : SW) 是用來儲存 CPU 的狀態值，這些狀態是許多旗標的組合。例如，零旗標 (Zero，簡寫為Z) 代表比較的結果為 0，負旗標 (Negative ，簡寫為N) 代表比較的結果為負值，另外常見的旗標還有進位旗標 (Carry ，簡寫為 C)， 溢位旗標 (Overflow，簡寫為 V) 等等。下圖顯示了 CPU0 的狀態暫存器格式，最前面的四個位元 N、Z、C、V所代表的， 正是上述的幾個旗標值。



圖、CPU0 中狀態暫存器 SW 的結構

條件旗標的 N、Z 旗標值可以用來代表比較結果是大於 (>)、等於 (=) 還是小於 (<)，當執行 CMP Ra, Rb 動作後，會有下列三種可能的情形。

1. 若 Ra > Rb，則 N=0, Z=0。
2. 若 Ra < Rb，則 N=1, Z=0。
3. 若 Ra = Rb，則 N=0, Z=1。

如此，用來進行條件跳躍的 JGT、JGE、JLT、JLE、JEQ、JNE指令，就可以根據 SW 暫存器當中的 N、Z 等旗標決定是否進行跳躍。

SW 中還包含中斷控制旗標 I (Interrupt) 與 T (Trap)，用以控制中斷的啟動與禁止等行為，假如將 I 旗標設定為 0，則CPU0將禁止所有種類的中斷，也就是對任何中斷都不會起反應。但如果只是將 T 旗標設定為0，則只會禁止軟體 中斷指令 SWI (Software Interrupt)，不會禁止由硬體觸發的中斷。

SW 中還儲存有『處理器模式』的欄位，M=0 時為『使用者模式』 (user mode) 與 M=1 時為『特權模式』 (super mode) 等， 這在作業系統的設計上經常被用來製作安全保護功能。在使用者模式當中，任何設定狀態暫存器 R12 的動作都會被視為是非法的， 這是為了進行保護功能的緣故。但是在特權模式中，允許進

行任何動作，包含設定中斷旗標與處理器模式等位元， 通常作業系統會使用特權模式 (M=1)，而一般程式只能處於使用者模式 (M=0)。

位元組順序

CPU0 採用大者優先 (Big Endian) 的位元組順序 (Byte Ordering)，因此代表值越大的位元組會在記憶體的前面 (低位址處)，代表值小者會在高位址處。

由於 CPU0 是 32 位元的電腦，因此，一個字組 (Word) 占用 4 個位元組 (Byte)，因此，像 LD R1, [100] 這樣的指令，其實是將記憶體 100-103 中的字組取出，存入到暫存器 R1 當中。

LDB 與 STB 等指令，其中的 B 是指 Byte，因此，LDB R1, [100] 會將記憶體 100 中的 byte 取出，載入到 R1 當中。但是，由於 R1 的大小是 32 bits，相當於 4個 byte，此時，LDB 與 STB 指令到底是存取四個 byte 當中的哪一個byte呢？這個問題的答案是byte 3，也就是最後的一個 byte。

中斷程序

CPU0 的中斷為不可重入式中斷，其中斷分為軟體中斷 SWI (Trap) 與硬體中斷 HWI (Interrupt) 兩類。

硬體中斷發生時，中段代號 INT_ADDR 會從中段線路傳入，此時執行下列動作：

1. LR=PC; INT=1
2. PC=INT_ADDR

軟體中斷 SWI Cx 發生時，會執行下列動作：

1. LR=PC; INT=1
2. PC=Cx;

中斷最後可以使用 IRET 返回，返回前會設定允許中斷狀態。

1. PC=LR; INT=0

實作：CPU0 的指令表

雖然 CPU0 處理器按理講應該直接以硬體實作，但是我們恐怕不容易直接請「台積電」或「聯電」幫我們燒一顆，因此在實作上我們使用了 FPGA + Verilog + Altera DE2-70 進行 CPU 設計。

但是電腦光是有硬體的話，仍然是不能使用的，否則您可以試試在 PC 上不要安裝作業系統，然後想辦法使用那台電腦，您肯定是會望著電腦興嘆的。

因此、就算 CPU 已經設計好了，我們仍然需要「組譯器、編譯器、作業系統」等系統軟體 (System Software)，才能成為一台真正可以用的電腦。

另外、如果我們能夠設計出「虛擬機」，那麼在這台電腦的硬體還沒有被生產出來之前，我們也能將程式放到「虛擬機」上去執行，因此 我們將會在本書的後半部描述這些「系統軟體」的結構，並且用 JavaScript 與 C 語言各自實作一組軟體系統。

我們將在下兩章中詳細說明組譯器「AS0.js 」與虛擬機「VM0.js」的實作方法，並詳細的列出原始碼。

現在、我們將先列出「虛擬機 AS0」與「組譯器 VM0」都會用到的共同部分，也就是「處理器 CPU0.js」與「指令表 opTable.js」 兩個程式的原始碼，並講解程式內容與執行結果。

在 JavaScript 當中要設計出指令表 opTable.js 非常的簡單，因為 JavaScript 的物件本身就是個符號表， 因此我們只要用 this[op.name] = op 這行指令就能在 opTable 這個建構函數當中，將指令物件插入到表格內。

檔案：opTable.js

```
var c = require("./ccc");

var Op = function(line) {
    var tokens = line.split(/\s+/);
    this.name = tokens[0];
    this.id   = parseInt(tokens[1], 16);
    this.type = tokens[2];
}

var opTable = function(opList) {
  for (i in  opList) {
    var op = new Op(opList[i]);
    this[op.name] = op;
  }
}

opTable.prototype.ID = function(op) {
  return this[op].id;
}

opTable.prototype.dump=function() {
    for (key in this) {
      var op = this[key];
      if (typeof(op)!="function")
        c.log("%s %s %s", c.fill(' ', op.name, 8), c.hex(op.id, 2), op.ty
pe);
    }
}

module.exports = opTable;
```

在上述程式碼中，每個指令包含了「指令名稱 (name), 指令代碼 (id) 與指令型態 (type)」等三個欄位，舉例

而言，當 一個載入指令的字串定義為 "LD 00 L" 時，就會被函數 Op = function(line) 拆解為 { name="LD", id=0x00, type="L" } 這樣的物件，然後新增到指令表當中。

利用上述的 opTable.js，我們可以輕易的建構出 CPU0 處理器的指令表，以下是 cpu0.js 程式的原始碼，該程式建構出了 CPU0 的完整指令表，包含 「LD, ST, ..., PUSHB, POPB」等真實的指令。

另外、以下表格當中還包含了「RESW, RESB, WORD, BYTE」等資料定義假指令，其中 RESW 用來保留 n 個 Word，RESB 用來保留 n 個 BYTE， WORD 則用來定義有初始值的整數變數，BYTE 則用來定義有初始值的位元組變數，像是 8 位元整數或字串等。

檔案：cpu0.js

```
var opTable = require("./optable");
var opList = [ "LD 00 L", "ST  01 L", "LDB 02 L",  "STB 03 L", "LDR 04 L
",
"STR 05 L", "LBR 06 L",  "SBR 07 L",  "LDI 08 L", "CMP 10 A", "MOV 12 A",

"ADD 13 A",  "SUB 14 A",  "MUL 15 A", "DIV 16 A",  "AND 18 A", "OR  19 A"
,  "XOR 1A A",
"ADDI 1B A", "ROL 1C A",  "ROR 1D A", "SHL 1E A", "SHR 1F A",
"JEQ 20 J", "JNE 21 J",  "JLT 22 J", "JGT 23 J", "JLE 24 J",  "JGE 25 J",
 "JMP 26 J",
"SWI 2A J", "JSUB 2B J","RET 2C J", "PUSH 30 J", "POP 31 J",  "PUSHB 32 J
",
"POPB 33 J", "RESW F0 D", "RESB F1 D", "WORD F2 D", "BYTE F3 D"];

var cpu = { "opTable" : new opTable(opList) };

cpu.opTable.dump();

module.exports = cpu;
```

執行結果：

```
D:\Dropbox\Public\oc\code>node cpu0.js
LD        00 L
ST        01 L
LDB       02 L
STB       03 L
LDR       04 L
STR       05 L
```

```
LBR      06 L
SBR      07 L
LDI      08 L
CMP      10 A
MOV      12 A
ADD      13 A
SUB      14 A
MUL      15 A
DIV      16 A
AND      18 A
OR       19 A
XOR      1A A
ADDI     1B A
ROL      1C A
ROR      1D A
SHL      1E A
SHR      1F A
JEQ      20 J
JNE      21 J
JLT      22 J
JGT      23 J
JLE      24 J
JGE      25 J
JMP      26 J
SWI      2A J
JSUB     2B J
RET      2C J
PUSH     30 J
POP      31 J
PUSHB    32 J
POPB     33 J
RESW     F0 D
RESB     F1 D
WORD     F2 D
BYTE     F3 D
```

細心的讀者可能會注意到，我們在 opTable.js 當中引入了 ccc.js 這個函式庫，這個函式庫是「開放電腦計畫」當中的常用函數集合， 其原始碼如下所示。

```
var util = require("util");
var assert = require("assert");
var fs = require("fs");

var c = {};                  // 本模組的傳回物件
c.log = console.log;         // 將 console.log 名稱縮短一點
c.format = util.format;      // 字串格式化
c.assert = assert.ok;        // assert 函數，若不符合條件則程式會停止
c.bits = function(word, from, to) { return word << (31-to) >>> (31-to+fr
om); }       // 取得 from 到 to 之間的位元
c.signbits = function(word, from, to) { return word << (31-to) >> (31-to
+from); }    // 取得 from 到 to 之間的位元
c.nonull = function(o) { if (o == null) return ""; else return o; }
              // 將 null 改為空字串傳回

c.space = "
        "; // 空白字串，dup() 函數中使用到的。
c.dup = function(ch, n) {          // 傳回 ch 重複 n 次的字串；範例：dup
('x', 3) = 'xxx'
  assert.ok(n < c.space.length);
  return c.space.substr(0, n).replace(/ /g, ch);
}

c.fill = function(ch, o, len) {     // 將字串填滿 ch，例如：fill(' ', 35,
 5) = '35   '; fill('0', 35, -5) = '00035';
  var str = o.toString();
  if (len >= 0)
    return str+c.dup(ch, len-str.length);
  else
    return c.dup(ch, -len-str.length)+str;
}

c.base = function(n, b, len) {      // 將數字 n 轉換為以 b 為基底的字串；
例如：base(31, 16, 5) = '0001F';
  var str = n.toString(b);
  return c.dup('0', len-str.length)+str;
}
```

```
c.hex = function(n, len) {          // 將數字 n 轉換 16 進位；例如：hex(31, 5) = '0001F'; hex(-3, 5) = 'FFFFD'
  var str = (n < 0 ? (0xFFFFFFFF + n + 1) : n).toString(16).toUpperCase();
  if (n < 0)
    return c.fill('F', str, -len).substr(-len);
  else
    return c.fill('0', str, -len).substr(-len);
}

c.str2hex = function(str) {          // 將字串轉為 16 進位碼，例如：str2hex('Hello!') = '48656C6C6F21'
  var hex="";
  for (i=0; i<str.length; i++) {
    var code = str.charCodeAt(i);
    hex += c.hex(code, 2);
  }
  return hex;
}

c.error = function(msg, err) {
   c.log(msg);
   c.log("Error : (%s):%s", err.name, err.message);
   c.log(err.stack);
   process.exit(1);
}

c.test = function() {
  c.log("bits(0xF3A4, 4, 7)=%s", c.hex(c.bits(0xF3A4, 4, 7), 4));
  c.log("dup('x', 3)=%s", c.dup('x', 3));
  c.log("fill('0', 35, -5)=%s", c.fill('0', 35, -5));
  c.log("base(100, 16, 5)=%s", c.base(100, 16, 5));
  c.log("hex(-100)=%s", c.hex(-100, 6));
  c.log("str2hex(Hello!)=%s", c.str2hex("Hello!"));
}
```

```
c.test();

module.exports = c;
```

以上程式的單元測試 c.test() 執行結果如下

```
D:\Dropbox\Public\oc\code>node ccc
bits(0xF3A4, 4, 7)=000A
dup('x', 3)=xxx
fill('0', 35, -5)=00035
base(100, 16, 5)=00064
hex(-100)=FFFF9C
str2hex(Hello!)=48656C6C6F21
```

虛擬機 - vm0

在前幾章中，我們介紹了 CPU0 處理器的指令集，以及組譯器的實作方式，文章網址如下：

現在，我們將接焦點轉移到虛擬機 VM0 的實作上，說明一個最簡易的虛擬機是如何設計出來的。

組譯範例

首先、讓讀者回顧一下，在上一篇文章中，我們設計了一個組譯器，可以組譯像以下的組合語言程式。

組合語言：sum.as0

```
        LD      R1, sum         ; R1 = sum = 0
        LD      R2, i           ; R2 = i = 1
        LDI     R3, 10          ; R3 = 10
FOR:    CMP     R2, R3          ; if (R2 > R3)
        JGT     EXIT            ;   goto EXIT
        ADD     R1, R1, R2      ; R1 = R1 + R2 (sum = sum + i)
        ADDI    R2, R2, 1       ; R2 = R2 + 1  ( i  = i + 1)
        JMP     FOR             ; goto FOR
EXIT:   ST      R1, sum         ; sum = R1
        ST      R2, i           ; i = R2
        LD      R9, msgptr      ; R9= pointer(msg) = &msg
        SWI     3               ; SWI 3 : 印出 R9 (=&msg) 中的字串
        MOV     R9, R1          ; R9 = R1 = sum
        SWI     4               ; SWI 2 : 印出 R9 (=R1=sum) 中的整數
        RET                     ; return 返回上一層呼叫函數
i:      RESW    1               ; int i
sum:    WORD    0               ; int sum=0
msg:    BYTE    "1+...+10=", 0    ; char *msg = "sum="
msgptr: WORD    msg             ; char &msgptr = &msg
```

我們可以用 AS0 組譯器對這樣的 CPU0 組合語言進行組譯，以下是組譯過程與結果，會輸出機器碼到目的檔中。

```
D:\Dropbox\Public\oc\code>node as0 sum.as0 sum.ob0
...
...
=================SAVE OBJ FILE================

00 :  001F003C 002F0034 0830000A 10230000
```

```
10 :  2300000C 13112000 1B220001 26FFFFEC
20 :  011F001C 012F0014 009F001D 2A000003
30 :  12910000 2A000002 2C000000 00000000
40 :  00000000 73756D3D 00000000 44
```

接著、我們就可以用虛擬機 VM0 來執行這個目的檔，我們可以選擇用預設不傾印的方式，得到以下的簡要執行結果。

虛擬機執行過程 (不傾印)

```
D:\oc\code>node vm0 sum.ob0
1+...+10=55
```

也可以用加上 -d 參數的方式，傾印每一個指令的執行過程，如下所示：

虛擬機執行過程 (詳細傾印)

```
D:\oc\code>node vm0 sum.ob0 -d


00 :  001F003C 002F0034 0830000A 10230000
10 :  2300000C 13112000 1B220001 26FFFFEC
20 :  011F001C 012F0014 009F0022 2A000003
30 :  12910000 2A000004 2C000000 00000000
40 :  00000000 312B2E2E 2E2B3130 3D000000
50 :  0044
PC=0000 IR=001F003C SW=00000000 R[01]=0x00000000=0
PC=0004 IR=002F0034 SW=00000000 R[02]=0x00000000=0
PC=0008 IR=0830000A SW=00000000 R[03]=0x0000000A=10
PC=000C IR=10230000 SW=80000000 R[0C]=0x80000000=-2147483648
PC=0010 IR=2300000C SW=80000000 R[00]=0x00000000=0
PC=0014 IR=13112000 SW=80000000 R[01]=0x00000000=0
PC=0018 IR=1B220001 SW=80000000 R[02]=0x00000001=1
PC=001C IR=26FFFFEC SW=80000000 R[0F]=0x0000000C=12
PC=000C IR=10230000 SW=80000000 R[0C]=0x80000000=-2147483648
PC=0010 IR=2300000C SW=80000000 R[00]=0x00000000=0
PC=0014 IR=13112000 SW=80000000 R[01]=0x00000001=1
PC=0018 IR=1B220001 SW=80000000 R[02]=0x00000002=2
PC=001C IR=26FFFFEC SW=80000000 R[0F]=0x0000000C=12
PC=000C IR=10230000 SW=80000000 R[0C]=0x80000000=-2147483648
PC=0010 IR=2300000C SW=80000000 R[00]=0x00000000=0
```

```
PC=0014 IR=13112000 SW=80000000 R[01]=0x00000003=3
PC=0018 IR=1B220001 SW=80000000 R[02]=0x00000003=3
PC=001C IR=26FFFFEC SW=80000000 R[0F]=0x0000000C=12
PC=000C IR=10230000 SW=80000000 R[0C]=0x80000000=-2147483648
PC=0010 IR=2300000C SW=80000000 R[00]=0x00000000=0
PC=0014 IR=13112000 SW=80000000 R[01]=0x00000006=6
PC=0018 IR=1B220001 SW=80000000 R[02]=0x00000004=4
PC=001C IR=26FFFFEC SW=80000000 R[0F]=0x0000000C=12
PC=000C IR=10230000 SW=80000000 R[0C]=0x80000000=-2147483648
PC=0010 IR=2300000C SW=80000000 R[00]=0x00000000=0
PC=0014 IR=13112000 SW=80000000 R[01]=0x0000000A=10
PC=0018 IR=1B220001 SW=80000000 R[02]=0x00000005=5
PC=001C IR=26FFFFEC SW=80000000 R[0F]=0x0000000C=12
PC=000C IR=10230000 SW=80000000 R[0C]=0x80000000=-2147483648
PC=0010 IR=2300000C SW=80000000 R[00]=0x00000000=0
PC=0014 IR=13112000 SW=80000000 R[01]=0x0000000F=15
PC=0018 IR=1B220001 SW=80000000 R[02]=0x00000006=6
PC=001C IR=26FFFFEC SW=80000000 R[0F]=0x0000000C=12
PC=000C IR=10230000 SW=80000000 R[0C]=0x80000000=-2147483648
PC=0010 IR=2300000C SW=80000000 R[00]=0x00000000=0
PC=0014 IR=13112000 SW=80000000 R[01]=0x00000015=21
PC=0018 IR=1B220001 SW=80000000 R[02]=0x00000007=7
PC=001C IR=26FFFFEC SW=80000000 R[0F]=0x0000000C=12
PC=000C IR=10230000 SW=80000000 R[0C]=0x80000000=-2147483648
PC=0010 IR=2300000C SW=80000000 R[00]=0x00000000=0
PC=0014 IR=13112000 SW=80000000 R[01]=0x0000001C=28
PC=0018 IR=1B220001 SW=80000000 R[02]=0x00000008=8
PC=001C IR=26FFFFEC SW=80000000 R[0F]=0x0000000C=12
PC=000C IR=10230000 SW=80000000 R[0C]=0x80000000=-2147483648
PC=0010 IR=2300000C SW=80000000 R[00]=0x00000000=0
PC=0014 IR=13112000 SW=80000000 R[01]=0x00000024=36
PC=0018 IR=1B220001 SW=80000000 R[02]=0x00000009=9
PC=001C IR=26FFFFEC SW=80000000 R[0F]=0x0000000C=12
PC=000C IR=10230000 SW=80000000 R[0C]=0x80000000=-2147483648
PC=0010 IR=2300000C SW=80000000 R[00]=0x00000000=0
PC=0014 IR=13112000 SW=80000000 R[01]=0x0000002D=45
PC=0018 IR=1B220001 SW=80000000 R[02]=0x0000000A=10
```

```
PC=001C IR=26FFFFEC SW=80000000 R[0F]=0x0000000C=12
PC=000C IR=10230000 SW=40000000 R[0C]=0x40000000=1073741824
PC=0010 IR=2300000C SW=40000000 R[00]=0x00000000=0
PC=0014 IR=13112000 SW=40000000 R[01]=0x00000037=55
PC=0018 IR=1B220001 SW=40000000 R[02]=0x0000000B=11
PC=001C IR=26FFFFEC SW=40000000 R[0F]=0x0000000C=12
PC=000C IR=10230000 SW=00000000 R[0C]=0x00000000=0
PC=0010 IR=2300000C SW=00000000 R[00]=0x00000000=0
m[0040]=55
PC=0020 IR=011F001C SW=00000000 R[01]=0x00000037=55
m[003C]=11
PC=0024 IR=012F0014 SW=00000000 R[02]=0x0000000B=11
PC=0028 IR=009F0022 SW=00000000 R[09]=0x00000044=68
1+...+10=PC=002C IR=2A000003 SW=00000000 R[00]=0x00000000=0
PC=0030 IR=12910000 SW=00000000 R[09]=0x00000037=55
55PC=0034 IR=2A000004 SW=00000000 R[00]=0x00000000=0
PC=0038 IR=2C000000 SW=00000000 R[00]=0x00000000=0
```

如果您詳細追蹤上述過程，就能更清楚的看出每個指令執行時，所造成的暫存器變化，舉例而言，您可以看到程式 在 PC=000C 到 PC=001C 之間循環了很多次，最後一次的循環印出下列內容。

```
PC=000C IR=10230000 SW=40000000 R[0C]=0x40000000=1073741824
PC=0010 IR=2300000C SW=40000000 R[00]=0x00000000=0
PC=0014 IR=13112000 SW=40000000 R[01]=0x00000037=55
PC=0018 IR=1B220001 SW=40000000 R[02]=0x0000000B=11
PC=001C IR=26FFFFEC SW=40000000 R[0F]=0x0000000C=12
PC=000C IR=10230000 SW=00000000 R[0C]=0x00000000=0
PC=0010 IR=2300000C SW=00000000 R[00]=0x00000000=0
m[0040]=55
```

其中得到 R[01]=0x00000037=55 的計算結果，正是整個程式計算 1+2+...+10=55 的結果。

VM0 虛擬機設計

接著、我們要來看看虛擬機 VM0 是如何設計的，但是在這之前，先讓我們看看虛擬機當中一個重要的資料結構， OpTable 指令表這個物件，其程式碼如下：

檔案：opTable.js

```
var c = require("./ccc");
```

```
var Op = function(line) {
    var tokens = line.split(/\s+/);
    this.name = tokens[0];
    this.id   = parseInt(tokens[1], 16);
    this.type = tokens[2];
}

var opTable = function(opList) {
  for (i in  opList) {
    var op = new Op(opList[i]);
    this[op.name] = op;
  }
}

opTable.prototype.ID = function(op) {
  return this[op].id;
}

opTable.prototype.dump=function() {
    for (key in this) {
      var op = this[key];
      if (typeof(op)!="function")
        c.log("%s %s %s", c.fill(' ', op.name, 8), c.hex(op.id, 2), op.ty
pe);
    }
}

module.exports = opTable;
```

然後、我們利用上述的 OpTable 模組，加入了 CPU0 的指令集之後，建出了 CPU0 這個代表處理器的模組， 程式碼如下。

檔案：cpu0.js

```
var opTable = require("./optable");
var opList = [ "LD 00 L", "ST  01 L", "LDB 02 L",  "STB 03 L", "LDR 04 L
",
"STR 05 L", "LBR 06 L",  "SBR 07 L",  "LDI 08 L", "CMP 10 A", "MOV 12 A",
```

```
"ADD 13 A", "SUB 14 A", "MUL 15 A", "DIV 16 A", "AND 18 A", "OR  19 A"
, "XOR 1A A",
"ADDI 1B A", "ROL 1C A", "ROR 1D A", "SHL 1E A", "SHR 1F A",
"JEQ 20 J", "JNE 21 J", "JLT 22 J", "JGT 23 J", "JLE 24 J", "JGE 25 J",
 "JMP 26 J",
"SWI 2A J", "JSUB 2B J","RET 2C J", "PUSH 30 J", "POP 31 J", "PUSHB 32 J
",
"POPB 33 J", "RESW F0 D", "RESB F1 D", "WORD F2 D", "BYTE F3 D"];

var cpu = { "opTable" : new opTable(opList) };

if (process.argv[2] == "-d")
    cpu.opTable.dump();

module.exports = cpu;
```

有了上述的兩個模組作為基礎，我們就可以開始撰寫虛擬機 VM0 了，以下是其原始程式碼。

檔案：vm0.js

```
var c = require("./ccc");
var cpu1 = require("./cpu0");
var Memory = require("./memory");

var isDump = process.argv[3] == "-d";

var IR = 16, PC = 15, LR = 14, SP = 13, SW = 12;
var ID = function(op) { return cpu1.opTable[op].id; }

var run = function(objFile) {
    R = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 0, 13, -1, 0, 16];
    m = new Memory(1);
    m.load(objFile);
    if (isDump) m.dump();
    var stop = false;
    while (!stop) {                                              // 如果尚未
結束
     var tpc = R[PC];
```

```
        R[0] = 0;                                                   // R[0] 永遠
為 0
        R[IR] = m.geti(R[PC]);                                      // 指令擷取
, IR=[PC..PC+3]
        R[PC] += 4;                                                 // 擷取完將
PC 加 4，指向下一個指令
        var op = c.bits(R[IR], 24, 31);                             // 取得 op
欄位，IR[24..31]
        var ra = c.bits(R[IR], 20, 23);                             // 取得 ra
欄位，IR[20..23]
        var rb = c.bits(R[IR], 16, 19);                             // 取得 rb
欄位，IR[16..19]
        var rc = c.bits(R[IR], 12, 15);                             // 取得 rc
欄位，IR[12..15]
        var c24= c.signbits(R[IR], 0,  23);                         // 取得 24
位元的 cx
        var c16= c.signbits(R[IR], 0,  15);                         // 取得 16
位元的 cx
        var c5 = c.bits(R[IR], 0,    4);                            // 取得 16
位元的 cx
        var addr  = R[rb]+c16;
        var raddr = R[rb]+R[rc];                                    // 取得位址[
Rb+Rc]
        var N  = c.bits(R[SW], 31, 31);
        var Z  = c.bits(R[SW], 30, 30);
//      c.log("IR=%s ra=%d rb=%d rc=%d c24=%s c16=%s addr=%s", c.hex(R[IR
], 8), ra, rb, rc, c.hex(c24, 6), c.hex(c16, 4), c.hex(addr, 8))
        switch (op) {                                               // 根據op執
行動作
          case ID("LD") : R[ra] = m.geti(addr); break;              // 處理 LD
指令
          case ID("ST") :                                           // 處理 ST
指令
            m.seti(addr, R[ra]);
            if (isDump) c.log("m[%s]=%s", c.hex(addr,4), m.geti(addr));
            break;
          case ID("LDB"): R[ra] = m.getb(addr); break;              // 處理 LDB
```

```
指令
        case ID("STB"): m.setb(addr, R[ra]); break;          // 處理 STB
指令
        case ID("LDR"): R[ra] = m.geti(raddr); break;        // 處理 LDR
指令
        case ID("STR"): m.seti(raddr, R[ra]); break;         // 處理 STR
指令
        case ID("LBR"): R[ra] = m.getb(raddr); break;        // 處理 LBR
指令
        case ID("SBR"): m.setb(raddr, R[ra]); break;         // 處理 SBR
指令
        case ID("LDI"): R[ra] = c16; break;                  // 處理 LDI
指令
        case ID("CMP"): {                                    // 處理 CMP
指令，根據比較結果，設定 N,Z 旗標
          if (R[ra] > R[rb]) {                               // > : SW(N=
0, Z=0)
            R[SW] &= 0x3FFFFFFF;                             // N=0, Z=0
          } else if (R[ra] < R[rb]) {                        // < : SW(N
=1, Z=0, ....)
            R[SW] |= 0x80000000;                             // N=1;
            R[SW] &= 0xBFFFFFFF;                             // Z=0;
          } else {                                           // = : SW(N
=0, Z=1)
            R[SW] &= 0x7FFFFFFF;                             // N=0;
            R[SW] |= 0x40000000;                             // Z=1;
          }
          ra = 12;
          break;

        }
        case ID("MOV"): R[ra] = R[rb]; break;                // 處理MOV
指令
        case ID("ADD"): R[ra] = R[rb]+R[rc]; break;          // 處理ADD
指令
        case ID("SUB"): R[ra] = R[rb]-R[rc]; break;          // 處理SUB
指令
```

```
        case ID("MUL"): R[ra] = R[rb]*R[rc]; break;          // 處理MUL指令
        case ID("DIV"): R[ra] = R[rb]/R[rc]; break;          // 處理DIV指令
        case ID("AND"): R[ra] = R[rb]&R[rc]; break;          // 處理AND指令
        case ID("OR") : R[ra] = R[rb]|R[rc]; break;          // 處理OR指令
        case ID("XOR"): R[ra] = R[rb]^R[rc]; break;          // 處理XOR指令
        case ID("SHL"): R[ra] = R[rb]<<c5; break;            // 處理SHL指令
        case ID("SHR"): R[ra] = R[rb]>>c5; break;            // 處理SHR指令
        case ID("ADDI"):R[ra] = R[rb] + c16; break;          // 處理 ADDI 指令
        case ID("JEQ"): if (Z==1) R[PC] += c24; break;       // 處理JEQ指令 Z=1
        case ID("JNE"): if (Z==0) R[PC] += c24; break;       // 處理JNE指令 Z=0
        case ID("JLT"): if (N==1&&Z==0) R[PC] += c24; break; // 處理JLT指令 NZ=10
        case ID("JGT"): if (N==0&&Z==0) R[PC] += c24; break; // 處理JGT指令 NZ=00
        case ID("JLE"): if ((N==1&&Z==0)||(N==0&&Z==1)) R[PC]+=c24; break; // 處理JLE指令 NZ=10 or 01
        case ID("JGE"): if ((N==0&&Z==0)||(N==0&&Z==1)) R[PC]+=c24; break; // 處理JGE指令 NZ=00 or 01
        case ID("JMP"): R[PC]+=c24; break;                   // 處理JMP指令
        case ID("SWI"):                                      // 處理SWI指令
            switch (c24) {
                case 3: c.printf("%s", m.getstr(R[9])); break;
                case 4: c.printf("%d", R[9]); break;
                default:
                    var emsg = c.format("SWI cx=%d not found!", c24);
```

```
                    c.error(emsg, null);
                    break;
            }
            break;
        case ID("JSUB"):R[LR] = R[PC]; R[PC]+=c24; break;    // 處理JSUB
指令
        case ID("RET"): if (R[LR]<0) stop=true; else R[PC]=LR; break; /
/ 處理RET指令
        case ID("PUSH"):R[SP]-=4; R[ra]=m.geti(addr); m.seti(R[SP], R[ra
]); break; // 處理PUSH指令
        case ID("POP"): R[ra] = m.geti(R[SP]); R[SP]+=4; break;      //
處理POP指令
        case ID("PUSHB"):R[SP]--; R[ra]=m.getb(addr); m.setb(R[SP], R[ra
]); break; // 處理PUSH指令
        case ID("POPB"):R[ra] = m.getb(R[SP]); R[SP]++; break;       //
處理POPB指令
        default: c.error("OP not found!", null);
      } // switch
      if (isDump)
        c.log("PC=%s IR=%s SW=%s R[%s]=0x%s=%d", // 印出 PC, IR, R[ra]暫
存器的值，以利觀察
              c.hex(tpc,4), c.hex(R[IR],8), c.hex(R[SW],8), c.hex(ra,2),
c.hex(R[ra], 8), R[ra]);
    } // while
}

run(process.argv[2]);
```

從上面的 VM0 虛擬機當中，您可以看到，假如不考慮執行速度的問題，那麼要撰寫一個虛擬機是非常容易的事情。 我們只要去忠實的模擬每一個指令所應該做的動作，就可以完成虛擬機的設計了。

讓我們稍微解釋一下上述 VM0 虛擬機的程式原理，請讀者將焦點先放在以下的程式片段中。

```
    ...
    m = new Memory(1);
    m.load(objFile);
    var stop = false;
    while (!stop) {                                           // 如果尚未
結束
```

```
    ...
    R[IR] = m.geti(R[PC]);                                    // 指令擷取
，IR=[PC..PC+3]
    R[PC] += 4;                                               // 擷取完將
PC 加 4，指向下一個指令
    var op = c.bits(R[IR], 24, 31);                           // 取得 op
欄位，IR[24..31]
    var ra = c.bits(R[IR], 20, 23);                           // 取得 ra
欄位，IR[20..23]
    var rb = c.bits(R[IR], 16, 19);                           // 取得 rb
欄位，IR[16..19]
    var rc = c.bits(R[IR], 12, 15);                           // 取得 rc
欄位，IR[12..15]
    var c24= c.signbits(R[IR], 0,  23);                       // 取得 24
位元的 cx
    var c16= c.signbits(R[IR], 0,  15);                       // 取得 16
位元的 cx
    var c5 = c.bits(R[IR], 0,   4);                           // 取得 16
位元的 cx
    var addr  = R[rb]+c16;
    var raddr = R[rb]+R[rc];                                  // 取得位址[
Rb+Rc]
    var N  = c.bits(R[SW], 31, 31);
    var Z  = c.bits(R[SW], 30, 30);
    switch (op) {                                             // 根據op執
行動作
      case ID("LD") : R[ra] = m.geti(addr); break;            // 處理 LD
指令
      ...
      case ID("JMP"): R[PC]+=c24; break;                      // 處理JMP指
令
      ...
      case ID("JSUB"):R[LR] = R[PC]; R[PC]+=c24; break;       // 處理JSUB
指令
      ...
      case ID("RET"): if (R[LR]<0) stop=true; else R[PC]=LR; break; /
/ 處理RET指令
```

```
    ...
```

首先我們用 m = new Memory(1) 建立一個空的記憶體，然後再用 m.load(objFile) 載入目的檔到記憶體中， 接著就開始進入 while (!stop) 起頭的指令解譯迴圈了，然後接著用 R[IR] = m.geti(R[PC]) 這個指令取出 程式計數暫存器 PC 所指到的記憶體內容 m[PC]，然後放到指令暫存器 IR 當中，接著就可以取出指令暫存器 IR 當中的欄位，像是指令碼 op、暫存器 ra, rb, rc 與常數部分 c24, c16, c5 等欄位。

然後就能對每個指令所應做的動作進行模擬，例如 LD 指令的功能是將記憶體位址 addr = R[rb]+c16 的內容取出，放到編號 ra 的暫存器當中，因此只要用 R[ra] = m.geti(addr) 這樣一個函數呼叫，就可以完成模擬的動作了。

當然、有些模擬動作很簡單，可以用一兩個指令做完，像是 LD, ST, JMP 等都是如此，但有些動作就比較複雜， 像是 JSUB, RET, PUSH, POP 等就要好幾個指令，最複雜的大概是 CMP 與 SWI 這兩個指令了，CMP 由於牽涉到比較動作 且需要設定 N, Z 等旗標，所以程式碼較長如下：

```
...
        case ID("CMP"): {                                    // 處理 CMP
指令，根據比較結果，設定 N,Z 旗標
          if (R[ra] > R[rb]) {                               // > : SW(N=
0, Z=0)
            R[SW] &= 0x3FFFFFFF;                             // N=0, Z=0
          } else if (R[ra] < R[rb]) {                         // < : SW(N
=1, Z=0, ....)
            R[SW] |= 0x80000000;                             // N=1;
            R[SW] &= 0xBFFFFFFF;                             // Z=0;
          } else {                                            // = : SW(N
=0, Z=1)
            R[SW] &= 0x7FFFFFFF;                             // N=0;
            R[SW] |= 0x40000000;                             // Z=1;
          }
          ra = 12;
          break;

        }
...
```

而 SWI 則是軟體中斷，這個部分也可以不做任何事，不過如果要支援某些中斷函數的話，就可以在這個指令中進行模擬， 目前 SWI 指令處理的原始碼如下：

```
        case ID("SWI"):                                       // 處理SWI指
令
```

```
        switch (c24) {
            case 3: c.printf("%s", m.getstr(R[9])); break;
            case 4: c.printf("%d", R[9]); break;
            default:
                var emsg = c.format("SWI cx=%d not found!", c24);
                c.error(emsg, null);
                break;
        }
        break;
```

目前我們支援兩個中斷處理呼叫，也就是 SWI 3 與 SWI 4。

其中的 SWI 4 會在螢幕上印出一個儲存在暫存器 R[9] 當中的整數，而 SWI 3 會在螢幕上印出一個記憶體當中的字串， 這個字串的起始位址也是儲存在暫存器 R[9] 當中的。

結語

透過 VM0，筆者希望能夠讓讀者清楚的瞭解虛擬機的設計方式，當然、VM0 是一個「跑得很慢」的虛擬機。

如果要讓虛擬機跑得很快，通常要搭配「立即編譯技術」(Just in Time Compiler, JIT) ，像是 Java 虛擬機 JVM 就是利用 JIT 才能夠讓 Java 程式跑得夠快。

另外、像是 VMWare、VirtualBox 等，則是在相同的 x86 架構下去執行的，因此重點變成「如何有效的繞過作業系統 的控管，讓機器碼在 CPU 上執行」的問題了。

在開放原始碼的領域，QEMU 是一個非常重要的虛擬機，其做法可以參考下列 QEMU 原作者 bellard 的論文：

- https://www.usenix.org/legacy/event/usenix05/tech/freenix/full_papers/bellard/bellard.pdf

摘要如下：

> The first step is to split each target CPU instruction into fewer simpler instructions called micro operations. Each micro operation is implemented by a small piece of C code. This small C source code is compiled by GCC to an object file. The micro operations are chosen so that their number is much smaller (typically a few hundreds) than all the combinations of instructions and operands of the target CPU. The translation from target CPU instructions to micro operations is done entirely with hand coded code. The source code is optimized for readability and compactness because the speed of this stage is less critical than in an interpreter.
>
> A compile time tool called dyngen uses the object file containing the micro operations as input to generate a dynamic code generator. This dynamic code generator is invoked at runtime to generate a complete host function which concatenates several micro operations.

筆者先前粗略的看了一下，原本以為「QEMU 則是機器法反編譯為 C 語言基本運算後，再度用 gcc 編譯 為

機器碼，才能達到高速執行的目的」，但是仔細看又不是這樣，想想還是不要自己亂解釋好了，不過有高手 J 兄來信說明如下，原文附上：

> QEMU 採取的技術為 portable JIT，本質上是一種 template-based compilation，事先透過 TCG 做 code generation，使得 C-like template 得以在執行時期可對應到不同平台的 machine code，而執行時期沒有 gcc 的介入，我想這點該澄清。

像 bellard 這種高手寫的虛擬機，果然是又快又好啊！

VM0 與 QEMU 相比，速度上致少慢了幾十倍，不過程式碼絕對是簡單很多就是了。

組譯器 - as0

在前面幾章，我們介紹了開放電腦計畫中的「處理器」 -- 包含 CPU0 的結構、指令集與編碼方式。 在本章中，我們將為 CPU0 設計一個組譯器 AS0，以便能更深入理解 CPU0 的結構，並瞭解 組譯器的設計原理。

組譯範例

讓我們先用範例導向的方式，先看看一個 CPU0 的組合語言程式，如下所示：

組合語言：sum.as0

```
        LD      R1, sum         ; R1 = sum = 0
        LD      R2, i           ; R2 = i = 1
        LDI     R3, 10          ; R3 = 10
FOR:    CMP     R2, R3          ; if (R2 > R3)
        JGT     EXIT            ;   goto EXIT
        ADD     R1, R1, R2      ; R1 = R1 + R2 (sum = sum + i)
        ADDI    R2, R2, 1       ; R2 = R2 + 1  ( i  = i + 1)
        JMP     FOR             ; goto FOR
EXIT:   ST      R1, sum         ; sum = R1
        ST      R2, i           ; i = R2
        LD      R9, msgptr      ; R9= pointer(msg) = &msg
        SWI     3               ; SWI 3 : print string &msg
        MOV     R9, R1          ; R9 = R1 = sum
        SWI     2               ; SWI 2 : print number sum
        RET                     ; return to CALLER
i:      RESW     1              ; int i
sum:    WORD     0              ; int sum=0
msg:    BYTE     "sum=", 0      ; char *msg = "sum="
msgptr: WORD     msg            ; char &msgptr = &msg
```

上述程式是一個可以計算 1+2+....+10 之結果的程式，最後會透過軟體中斷 (SWI, Software Interrupt) 的方式，印出訊息到螢幕畫面上，以下是利用我們寫的組譯器 AS0 對上述程式進行組譯的過程：

```
D:\oc\code>node as0 sum.as0 sum.ob0
Assembler:asmFile=sum.as0 objFile=sum.ob0
===============Assemble=============
[ '        LD      R1, sum         ; R1 = sum = 0',
  '        LD      R2, i           ; R2 = i = 1',
  '        LDI     R3, 10          ; R3 = 10',
```

```
 'FOR:    CMP    R2, R3        ; if (R2 > R3)',
 '        JGT    EXIT          ;   goto EXIT',
 '        ADD    R1, R1, R2    ; R1 = R1 + R2 (sum = sum + i)',
 '        ADDI   R2, R2, 1     ; R2 = R2 + 1  ( i  = i + 1)',
 '        JMP    FOR           ; goto FOR',
 'EXIT:   ST     R1, sum       ; sum = R1',
 '        ST     R2, i         ; i = R2',
 '        LD     R9, msgptr    ; R9= pointer(msg) = &msg',
 '        SWI    3             ; SWI 3 : 印出 R9 (=&msg) 中的字串',
 '        MOV    R9, R1        ; R9 = R1 = sum',
 '        SWI    4             ; SWI 2 : 印出 R9 (=R1=sum) 中的整數',
 '        RET                  ; return 返回上一層呼叫函數',
 'i:      RESW   1             ; int i',
 'sum:    WORD   0             ; int sum=0',
 'msg:    BYTE   "1+...+10=", 0   ; char *msg = "sum="',
 'msgptr: WORD   msg           ; char &msgptr = &msg' ]
=================PASS1================
0000          LD       R1,sum            L 00
0004          LD       R2,i              L 00
0008          LDI      R3,10             L 08
000C FOR      CMP      R2,R3             A 10
0010          JGT      EXIT              J 23
0014          ADD      R1,R1,R2          A 13
0018          ADDI     R2,R2,1           A 1B
001C          JMP      FOR               J 26
0020 EXIT     ST       R1,sum            L 01
0024          ST       R2,i              L 01
0028          LD       R9,msgptr         L 00
002C          SWI      3                 J 2A
0030          MOV      R9,R1             A 12
0034          SWI      4                 J 2A
0038          RET                        J 2C
003C i        RESW     1                 D F0
0040 sum      WORD     0                 D F2
0044 msg      BYTE     "1+...+10=",0     D F3
004E msgptr   WORD     msg               D F2
===============SYMBOL TABLE=========
```

```
FOR       000C
EXIT      0020
i         003C
sum       0040
msg       0044
msgptr    004E
=============PASS2==============
0000          LD       R1,sum           L 00 001F003C
0004          LD       R2,i             L 00 002F0034
0008          LDI      R3,10            L 08 0830000A
000C FOR      CMP      R2,R3            A 10 10230000
0010          JGT      EXIT             J 23 2300000C
0014          ADD      R1,R1,R2         A 13 13112000
0018          ADDI     R2,R2,1          A 1B 1B220001
001C          JMP      FOR              J 26 26FFFFEC
0020 EXIT     ST       R1,sum           L 01 011F001C
0024          ST       R2,i             L 01 012F0014
0028          LD       R9,msgptr        L 00 009F0022
002C          SWI      3                J 2A 2A000003
0030          MOV      R9,R1            A 12 12910000
0034          SWI      4                J 2A 2A000004
0038          RET                       J 2C 2C000000
003C i        RESW     1                D F0 00000000
0040 sum      WORD     0                D F2 00000000
0044 msg      BYTE     "1+...+10=",0    D F3 312B2E2E2E2B31303D00
004E msgptr   WORD     msg              D F2 00000044
=================SAVE OBJ FILE================

00 :  001F003C 002F0034 0830000A 10230000
10 :  2300000C 13112000 1B220001 26FFFFEC
20 :  011F001C 012F0014 009F0022 2A000003
30 :  12910000 2A000004 2C000000 00000000
40 :  00000000 312B2E2E 2E2B3130 3D000000
50 :  0044
```

當您組譯完成之後，就可以利用開放電腦計畫中的虛擬機 VM0 執行 AS0 所輸出的目的檔 sum.ob0， 其執行過程如下：

```
D:\oc\code>node vm0 sum.ob0
1+...+10=55
```

AS0 組譯器設計

組譯器的設計，通常採用兩階段的編碼方式，第一階段 (PASS1) 先計算出每個指令的位址，並記住所有標記符號的位址。 然後在第二階段 (PASS2) 才真正將指令轉換為機器碼輸出，在以下 AS0 組譯器的設計當中，我們就採用了這種兩階段的 處理方式。

為了讓組譯器能夠容易修改與移植，我們將一般組譯器都會有的基礎結構 (抽象的組譯器物件) 放在 as.js 這個程式模組當中， 然後將與 CPU0 有關的部分放在 as0.js 這個實作模組當中，以下先列出 as.js 這個抽象物件模組。

檔案：as.js (抽象組譯器物件)

```
var fs = require("fs"); // 引用檔案函式庫
var c = require("./ccc"); // 引用基本函式庫 ccc.js
var Memory = require("./memory"); // 引用記憶體物件 memory.js

var as = function(opTable) { // 抽象組譯器物件
 this.opTable = opTable; // 取得指令表 opTable

 this.assemble = function(asmFile, objFile) { // 組譯器的主要函數
  this.lines = []; this.codes = []; // 設定程式碼行 (lines)，指令陣列 (co
des)
  this.symTable = {}; // 建立空的符號表 (symTable)
  c.log("Assembler:asmFile=%s objFile=%s", asmFile, objFile); // 輸入組合
語言、輸出目的檔
  c.log("===============Assemble=============");
  var text = fs.readFileSync(asmFile, "utf8"); // 讀取檔案到 text 字串中
  this.lines = text.split(/[\r\n]+/); // 將組合語言分割成一行一行
  c.log(this.lines); // 印出組合語言以便觀察
  this.pass1(); // 第一階段：計算位址
  c.log("===============SYMBOL TABLE=========");
  for (s in this.symTable) { // 印出符號表以便觀察
    c.log("%s %s", c.fill(' ',s,8), c.hex(this.symTable[s].address, 4));
  }
  this.pass2(); // 第二階段：建構目的碼
  this.saveObjFile(objFile); // 輸出目的檔
```

```
}

this.pass1 = function() { // 第一階段的組譯
 var address = 0;  // 程式計數器 PC 的起始位址為 0
 c.log("=================PASS1================");
 for (var i in this.lines) { // 對於每一行
   try {
     var code = this.parse(this.lines[i]); // 剖析並建立 code 物件
     code.address = address; // 設定該行的位址
     if (code.label.length != 0) { // 如果有標記符號
       this.symTable[code.label] = code; // 加入符號表中
     }
     this.codes.push(code); // 將剖析完成的指令放入陣列中
     c.log("%s", code); // 印出指令物件
     address += this.size(code); //  計算下一個指令位址
   } catch (err) { // 語法有錯，印出錯誤的行號與內容
     c.error(c.format("line %d : %s", i, this.lines[i]), err);
   }
 }
}

this.pass2 = function(codes) { // 組譯器的第二階段
 c.log("=============PASS2==============");
 for (var i in this.codes) { // 對每一個指令
   try {
     this.translate(this.codes[i]); // 將組合語言指令翻譯成機器碼
     c.log("%s", this.codes[i]); // 印出指令物件 (含組合語言與機器碼)
   } catch (err) { // 語法有錯，印出錯誤的行號與內容
     c.error(c.format("line %d : %s", i, this.lines[i]), err);
   }
 }
}

this.saveObjFile = function(objFile) { // 儲存目的檔
 c.log("=================SAVE OBJ FILE================");
 var obj = ""; // obj 為目的檔的 16 進位字串，初始化為空字串
 for (var i in this.codes) // 對於每個指令
```

```
    obj += this.codes[i].obj; // 都將目的碼加入 obj 字串中。
  var m = new Memory(1); // Memory 物件，用來將 16 進位目的碼轉為 2 進位儲存。
  m.loadhex(obj); // 將 16 進位目的碼載入記憶體
  m.dump(); // 輸出記憶體內容
  m.save(objFile); // 將記憶體內容除存到目的檔 objFile 中。
 }

 this.size = function(code) { // 計算指令所佔空間大小，在 pass1() 當中會呼叫此函數
    var len = 0, unitSize = 1; // len: 指令大小 , unitSize:每單位大小 (BYTE=1, WORD=4)
    switch (code.op.name) { // 根據運算碼 op
      case "RESW" : return 4 * parseInt(code.args[0]);  // 如果是 RESW, 大小為 4*保留量(參數 0)
      case "RESB" : return 1 * parseInt(code.args[0]);  // 如果是 RESB, 大小為 1*保留量(參數 0)
      case "WORD" : unitSize = 4; // 沒有 break，繼續執行到 BYTE 部分的程式 (共用)
      case "BYTE" : // 如果是BYTE, 大小是 1*參數個數
        for (i in code.args) { // 對於 BYTE 或 WORD 中的每個元素
          if (code.args[i].match(/^\".*?\"$/)) // 如果是字串，像 "Hello!"
            len += (code.args[i].length - 2) * unitSize; // 則大小為 unitSize*字串長度
          else // 否則 大小就是 unitSize (BYTE=1, WORD=4)
            len += unitSize;
        }
        return len;
      case "" : return 0; // 如果只是標記, 大小為 0
      default : return 4; // 其他情形 (指令), 大小為 4
    }
 }
}

module.exports = as; // 匯出「抽象組譯器物件 as 」
```

請注意，as.js 模組缺少 parse(), translate() 等函數，由於這兩個函數是與 CPU0 設計有關的部分，因此定義在後續的 as0.js 當中

註：雖然上述程式中的 size() 函數也可能會與 CPU 的設計有關，但是對於 32 bit 的 CPU 而言，可以通用，因此我們 將此函數放在上層的 as.js 當中，如果要定義非 32 位元 CPU、或者重新定義組合語言的語法時，可以覆寫掉這個 size() 函數。

在上述程式中，我們用到了一個 code 物件，以下是該物件之定義模組 code.js 的原始碼：

檔案：code.js (指令物件)

```
var c = require("./ccc"); // 引用基本函式庫 ccc.js

var code = function(line, opTable) { // 指令物件 code
  this.parseR = function(str) { // 剖析暫存器參數 Ra，例如 parse('R3') = 3
    var rmatch = /R(\d+)/.exec(str); // 比對取出 Ra 中的數字
    if (rmatch == null) // 如果比對失敗，則傳回 NaN
      return NaN;
    return parseInt(rmatch[1]); // 否則傳回暫存器代號 (數字)
  }

  this.toString = function() { // 輸出格式化後的指令
    return c.format("%s %s %s %s %s %s %s", c.hex(this.address, 4),
      c.fill(' ',this.label,8), c.fill(' ',this.op.name, 8),
      c.fill(' ',this.args, 16), this.op.type, c.hex(this.op.id,2), this.obj);
  }

  var labCmd = /^((\w+):)?\s*([^;]*)/; // 指令的語法
  var parts  = labCmd.exec(line); // 分割出標記與命令
  var tokens = parts[3].trim().split(/[ ,\t\r]+/); // 將命令分割成基本單元
  var opName = tokens[0]; // 取出指令名稱

  this.label = c.nonull(parts[2]); // 取出標記 (\w+)
  this.args  = tokens.slice(1); // 取出參數部份
  this.op    = opTable[opName]; // 取得指令表中的 OP 物件
  this.obj   = ""; // 清空目的碼 16 進位字串 obj
}

module.exports = code; // 匯出指令物件 code
```

現在、我們以經完成組譯器抽象架構的設計了，可以開始進入與 CPU0 有關的實作部分，也就是 as0.js 的組譯器實作， 補完 as.js 當中所沒有的 parse(), translate() 等函數了，以下是其原始程式碼。

檔案：as0.js (具體的 CPU0 組譯器 AS0)

```
var c = require("./ccc"); // 引用基本函式庫 ccc.js
var as = require("./as"); // 引用抽象組譯器物件 as.js
var code = require("./code"); // 引用指令物件 code.js
var cpu0 = require("./cpu0"); // 引用處理器物件 cpu0.js

var as0 = new as(cpu0.opTable); // 建立 as0 組譯器物件

as0.parse = function(line) { // 剖析組合語言指令，建立 code 物件
    return new code(line, this.opTable);
}

as0.translate = function(code) { // 指令的編碼函數
  var ra=0, rb=0, rc=0, cx=0;
  var pc = code.address + 4; // 提取後PC為位址+4
  var args = code.args, parseR = code.parseR; // 取得 code 物件的函數
  var labelCode = null; // JMP label 中 label 所對應行的物件，稱為 labelC
ode
  if (code.op == undefined) { // 如果沒有指令碼 (只有標記)，則清空目的碼
    code.obj = "";
    return;
  }
  switch (code.op.type) { // 根據指令型態
    case 'J' : // 處理 J 型指令，編出目的碼 OP Ra+cx
      switch (code.op.name) {
        case "RET": case "IRET" : // 如果式返回或中斷返回，則只要輸出 op
碼
            break;
        case "SWI" : // 如果是軟體中斷指令，則只有 cx 參數有常數值
            cx = parseInt(args[0]);
            break;
        default : // 其他跳躍指令，例如 JMP label, JLE label 等
            labelCode = this.symTable[args[0]]; // 取得 label 符號位址
            cx = labelCode.address - pc; // 計算 cx 欄位
```

```
          break;
        }
      code.obj = c.hex(code.op.id,2)+c.hex(cx, 6); // 編出目的碼 OP Ra+cx
      break;
    case 'L' : // 處理 L 型指令，編出目的碼 OP Ra, Rb, cx
      ra = parseR(args[0]); // 取得 Ra 欄位
      switch (code.op.name) {
        case "LDI" :  // 處理  LDI 指令
          cx = parseInt(args[1]); // 取得 cx 欄位
          break;
        default : // 處理 LD, ST, LDB, STB 指令
          if (args[1].match(/^[a-zA-Z]/)){ // 如果是 LD LABEL 這類情況
            labelCode = this.symTable[args[1]]; // 取得標記的 code 物件
            rb = 15; // R[15] is PC
            cx = labelCode.address - pc; // 計算標記與 PC 之間的差值
          } else { // 否則，若是像 LD Ra, Rb+100 這樣的指令
            rb = parseR(args[2]); // 取得 rb 欄位
            cx = parseInt(args[3]); // 取得 cx 欄位 (例如 100)
          }
          break;
      }
      code.obj = c.hex(code.op.id, 2)+c.hex(ra, 1)+c.hex(rb, 1)+c.hex(cx, 
 4); // 編出目的碼 OP Ra, Rb, cx
      break;
    case 'A' : // 處理 A 型指令，編出目的碼 OP Ra, Rb, Rc, cx
      ra = parseR(args[0]); // 取得 Ra 欄位
      switch (code.op.name) {
        case "LDR": case "LBR": case "STR": case "SBR":  // 處理  LDR, L
BR, STR, SBR 指令，例如 LDR Ra, Rb+Rc
          rb = parseR(args[1]); // 取得 Rb 欄位
          rc = parseR(args[2]); // 取得 Rc 欄位
          break;
        case "CMP": case "MOV" : // 處理 CMP 與 MOV 指令，CMP Ra, Rb; MOV
 Ra, Rb
          rb = parseR(args[1]); // 取得 Rb
          break;
        case "SHL": case "SHR": case "ADDI": // 處理 SHL, SHR, ADDI 指令
```

```
，例如 SHL Ra, Rb, Cx
            rb = parseR(args[1]); // 取得 Rb 欄位
            cx = parseInt(args[2]); // 取得 cx 欄位 (例如 3)
            break;
        case "PUSH": case "POP": case "PUSHB": case "POPB" :  // 處理 PU
SH, POP, PUSHB, POPB
            break; // 例如 PUSH Ra, 只要處理 Ra 就好，A 型一進入就已經處理
 Ra 了。
        default :  // 其他情況，像是 ADD, SUB, MUL, DIV, AND, OR, XOR 等
，例如 ADD Ra, Rb, Rc
            rb = parseR(args[1]);  // 取得 Rb 欄位
            rc = parseR(args[2]);  // 取得 Rc 欄位
            break;
      }
      code.obj = c.hex(code.op.id, 2)+c.hex(ra, 1)+c.hex(rb, 1)+c.hex(rc,
1)+c.hex(cx, 3); // 編出目的碼 OP Ra, Rb, Rc, cx
      break;
    case 'D' : { // 我們將資料宣告  RESW, RESB, WORD, BYTE 也視為一種指令
，其形態為 D
      var unitSize = 1; // 預設的型態為 BYTE，資料大小 = 1
      switch (code.op.name) {
        case "RESW":  case "RESB": // 如果是 RESW 或 RESB，例如 a:RESB 2
          code.obj = c.dup('0', this.size(code)*2); // 1 個 byte 的空間要
用兩個16進位的 00 去填充
          break;                                    // 例如：a RESB 2 會
編為 '0000'
        case "WORD": // 如果是 WORD ，佔 4 個 byte
          unitSize = 4;
        case "BYTE": { // 如果是 BYTE ，佔 1 個 byte
          code.obj = ""; // 一開始目的碼為空的
          for (var i in args) { // 對於每個參數，都要編為目的碼
            if (args[i].match(/^\".*?\"$/)) {  // 該參數為字串，例如： "H
ello!" 轉為 68656C6C6F21
              var str = args[i].substring(1, args[i].length-1); // 取得 "
..." 中間的字串內容
              code.obj += c.str2hex(str); // 將字串內容 (例如 Hello!) 轉
為 16 進位 (例如 68656C6C6F21)
```

```
            } else if (args[i].match(/^\d+$/)) {  // 該參數為常數，例如 2
6
              code.obj += c.hex(parseInt(args[i]), unitSize*2); // 將常數
轉為 16 進位目的碼 (例如 26 轉為 1A)
            } else { // 該參數為標記，將標記轉為記憶體位址，例如 msgptr: W
ORD msg 中的 msg 轉為位址 (例如：00000044)
              labelCode = this.symTable[args[i]]; // 取得符號表內的物件
              code.obj += c.hex(labelCode.address, unitSize*2); // 取得位
址並轉為 16 進位，塞入目的碼中。
            }
          }
          break;
        } // case BYTE:
      } // switch
      break;
    } // case 'D'
  }
}

// 使用範例 node as0 sum.as0 sum.ob0
// 其中 argv[2] 為組合語言檔，argv[3] 為目的檔
as0.assemble(process.argv[2], process.argv[3]);
```

在 as0.js 組譯器中我們還匯入了 cpu0.js 這個模組，雖然此模組已經於上一期當中介紹過了，不過由於上一期有幾個指令型態 設錯了 (LDR, STR, LBR, SBR 應該是 A 格式，上一期當中誤植為 L 格式)，因此我們再度列出 cpu0.js 的更正後內容如下：

```
var opTable = require("./optable"); // 引用指令表 opTable 物件

// 指令陣列
var opList = [ "LD 00 L", "ST  01 L", "LDB 02 L",  "STB 03 L", "LDR 04 A
",
"STR 05 A", "LBR 06 A",  "SBR 07 A",  "LDI 08 L", "CMP 10 A", "MOV 12 A",

"ADD 13 A",  "SUB 14 A",  "MUL 15 A", "DIV 16 A",  "AND 18 A", "OR  19 A"
,  "XOR 1A A",
"ADDI 1B A", "ROL 1C A",  "ROR 1D A", "SHL 1E A", "SHR 1F A",
"JEQ 20 J", "JNE 21 J",  "JLT 22 J", "JGT 23 J", "JLE 24 J",  "JGE 25 J",
```

```
 "JMP 26 J",
"SWI 2A J", "JSUB 2B J","RET 2C J", "PUSH 30 J", "POP 31 J",  "PUSHB 32 J
",
"POPB 33 J", "RESW F0 D", "RESB F1 D", "WORD F2 D", "BYTE F3 D"];

var cpu = { "opTable" : new opTable(opList) }; // cpu0 處理器物件，內含一
個指令表 opTable

if (process.argv[3] == "-t") // 如果使用 node cpu0 -t 可以印出指令表
    cpu.opTable.dump();

module.exports = cpu; // 匯出 cpu0 模組。
```

程式說明

在上述的 as.js 程式中，第一階段 pass1() 的工作是將每個組合語言指令的位址編好，並紀錄下所有符號的位址， 這個過程顯示在組譯報表的 PASS1 部分，您可以看到上述 as0 組譯器的輸出範例中，每個指令的位址都被計算出來了， 如下所示：

```
=================PASS1================
0000           LD        R1,sum           L 00
0004           LD        R2,i             L 00
0008           LDI       R3,10            L 08
000C FOR       CMP       R2,R3            A 10
0010           JGT       EXIT             J 23
0014           ADD       R1,R1,R2         A 13
0018           ADDI      R2,R2,1          A 1B
001C           JMP       FOR              J 26
0020 EXIT      ST        R1,sum           L 01
0024           ST        R2,i             L 01
0028           LD        R9,msgptr        L 00
002C           SWI       3                J 2A
0030           MOV       R9,R1            A 12
0034           SWI       4                J 2A
0038           RET                        J 2C
003C i         RESW      1                D F0
0040 sum       WORD      0                D F2
0044 msg       BYTE      "1+...+10=",0    D F3
```

```
004E msgptr   WORD     msg              D F2
```

而且在 PASS1 完成之後，所有符號的位址都會被記錄在符號表當中，如下所示：

```
===============SYMBOL TABLE=========
FOR       000C
EXIT      0020
i         003C
sum       0040
msg       0044
msgptr    004E
```

接著在 PASS2 當中，我們就可以利用這些符號表中的位址，編制出每個指令中的符號的「定址方式、相對位址」等等，如下表所示：

```
=============PASS2==============
0000          LD       R1,sum           L 00 001F003C
0004          LD       R2,i             L 00 002F0034
0008          LDI      R3,10            L 08 0830000A
000C FOR      CMP      R2,R3            A 10 10230000
0010          JGT      EXIT             J 23 2300000C
0014          ADD      R1,R1,R2         A 13 13112000
0018          ADDI     R2,R2,1          A 1B 1B220001
001C          JMP      FOR              J 26 26FFFFEC
0020 EXIT     ST       R1,sum           L 01 011F001C
0024          ST       R2,i             L 01 012F0014
0028          LD       R9,msgptr        L 00 009F0022
002C          SWI      3                J 2A 2A000003
0030          MOV      R9,R1            A 12 12910000
0034          SWI      4                J 2A 2A000004
0038          RET                       J 2C 2C000000
003C i        RESW     1                D F0 00000000
0040 sum      WORD     0                D F2 00000000
0044 msg      BYTE     "1+...+10=",0    D F3 312B2E2E2E2B31303D00
004E msgptr   WORD     msg              D F2 00000044
```

由於 CPU0 設計得很簡單，因此對於一般的指令而言(像是 ADD, MOV, RET 等)，編制出機器碼是很容易的，例如 RET 指令不包含任何參數，因此其機器碼就是在指令碼 OP=2C 後面補 0，直到填滿 32bit (8 個 16 進位數字) 為止， 而 ADD R1,R1,R2 的編碼也很容易，就是將指令碼 OP=13 補上暫存器代號 1, 1, 2 之後再補 0，形成 13112000 的編碼。

最難處理的是有標記的指令，舉例而言，像是 JGT EXIT 的機器碼 2300000C 與 JMP FOR 的機器碼 26FFFFEC 是怎麼來的呢？

關於這點，我們必須用較長的篇幅解釋一下：

在上述 AS0 程式的設計當中，我們一律用「相對於程式計數器 PC 的定址法」來進行標記的編碼 (cx = label.address-PC)， 例如在 JGT EXIT 這個指令中，由於標記 EXIT 的位址是 0020 ，而 JGT EXIT 指令的位址為 0010，因此兩者之差距為 0010，但是由於 JGT EXIT 指令執行時其程式計數器 PC 已經進到下一個位址 (0014) 了(在指令擷取階段完成後就會進到下一個位址)， 所以 PC 與 FOR 標記之間的位址差距為 (cx = label.address-PC= 0020-0014 = 000C) (請注意這是用 16 進位的減法)， 因此整個 JGT EXIT 指令就被組譯為 JGT EXIT = JGT R15+cx = 23 F 000C。 (其中 R15 是 CPU0 的程式計數器 PC，所以暫存器 Ra 部分編為 15 的十六進位值 F)。

但是、有時候相對定址若是負值，也就是標記在指令的前面，像是 JMP FOR 的情況時，最後 cx = label.address-PC 計算出來會是 負值，此時就必須採用 2 補數表示法，例如 JMP FOR 的情況 (cx = label.address-PC = 000C-0020 = -0014) (請注意這是用 16 進位的減法)， 採用 2 補數之後就會變成 FFFFEC，因此 JMP FOR 被編為 26 F FFFEC。

結語

現在、我們已經完成了組譯器 AS0 的設計，並解析了整個組譯器的原始碼，希望透過這種方式，可以讓讀者瞭解 整個組譯器的設計過程。在後續的文章之中，我們還會介紹開放電腦計畫中「虛擬機、編譯器」的 JavaScript 原始碼， 以及實作 CPU0 處理器的 Verilog 原始碼。然後再進入作業系統的設計部分，希望透過這種方式，可以讓讀者瞭解 如何「自己動手設計一台電腦」，完成「開放電腦計畫」的主要目標。

編譯器

在前面的文章中，我們首先介紹了整體架構，接著設計出了 CPU0 的指令集，然後寫出了 AS0 組譯器與 VM0 虛擬機：

但是、直到目前為止，我們都還沒有為開放電腦計畫打造出「高階語言」，因此本文將設計出一個 名為 J0 的高階語言 (代表JavaScript 的精簡版)，並採用 JavaScript 去實作，然後在 node.js 平台中執行。

有了 J0 語言與 J0C 編譯器之後，我們就可以創建出以下的工具鏈：

J0 語言 (j0c 編譯器) => IR0 中間碼 (ir2as 轉換器) => CPU0 組合語言 (AS0 組譯器) => CPU0 機器碼 (VM0 虛擬機執行或 CPU0 FPGA 執行)

編譯器：高階語言轉中間碼 – j0c

在前面的文章中，我們首先介紹了整體架構，接著設計出了 CPU0 的指令集，然後寫出了 AS0 組譯器與 VM0 虛擬機：

但是、直到目前為止，我們都還沒有為開放電腦計畫打造出「高階語言」，因此本文將設計出一個 名為 J0 的高階語言 (代表JavaScript 的精簡版)，並採用 JavaScript 去實作，然後在 node.js 平台中執行。

有了 J0 語言與 J0C 編譯器之後，我們就可以創建出以下的工具鏈：

J0 語言 (j0c 編譯器) => IR0 中間碼 (ir2as 轉換器) => CPU0 組合語言 (AS0 組譯器) => CPU0 機器碼 (VM0 虛擬機執行或 CPU0 FPGA 執行)

JavaScript 簡化版 -- J0 語言

以下是一個 J0 語言的程式範例，支援了 function, while, if, for 等語句，並且支援了「陣列與字典」等資料結構。

檔案：test.j0

```
s = sum(10);

function sum(n) {
  s = 0;
  i=1;
  while (i<=10) {
    s = s + i;
    i++;
  }
  return s;
```

```
}

m = max(3,5);

function max(a, b) {
  if (a > b)
    return a;
  else
    return b;
}

function total(a) {
  s = 0;
  for (i in a) {
    s = s + a[i];
  }
  return s;
}

a = [ 1, 3, 7, 2, 6];
t = total(a);
word = { e:"dog", c:"狗" };
```

原始碼

接著我們用 node.js + javascript 實作出 j0c 編譯器，該編譯器可以將 J0 語言的程式，編譯成一種平坦化的中間碼格式，我們稱這種格式為 IR0，也就是 Intermediate Representation 0 的意思，以下是 j0c 編譯器的完整程式碼。

檔案：j0c.js

```
// j0c 編譯器，用法範例： node j0c test.j0
var fs = require("fs");
var util = require("util");
var log = console.log;     // 將 console.log 名稱縮短一點
var format = util.format;  // 字串格式化
var tokens = [];
var tokenIdx = 0;
var end = "$END";
```

```
var funcName = "main";
var funcStack = [ funcName ];
var irText = "";
var symTable = {};
symTable[funcName] = { type:"function", name:"main", pcodes:[] };

var scan=function(text) {
  var re = new RegExp(/(\/\*[\s\S]*?\*\/)|(\/\/[^\r\n])|(".*?")|(\d+(\.\d*)?)|([a-zA-Z]\w*)|([>=<!\+\-\*\/&%|]+)|(\s+)|(.)/gm);
  var types = [ "", "COMMENT", "COMMENT", "STRING", "INTEGER", "FLOAT", "ID", "OP2", "SPACE", "CH" ];
  tokens = [];
  tokenIdx = 0;
  var lines = 1, m;
  while((m = re.exec(text)) !== null) {
    var token = m[0], type;
    for (i=1; i<=9; i++) {
      if (m[i] !== undefined)
        type = types[i];
    }
    if (!token.match(/^[\s\r\n]/) && type!="COMMENT") {
      tokens.push({ "token":token, "type":type, "lines":lines });
    }
    lines += token.split(/\n/).length-1;
  }
  tokens.push({ "token": end, "type":end, "lines":lines });
  return tokens;
}

var error=function(expect) {
  var token = tokens[tokenIdx];
  log("Error: line=%d token (%s) do not match expect (%s)!", token.lines, token.token, expect);
  log(new Error().stack);
  process.exit(1);
}
```

```
var skip=function(o) { if (isNext(o)) next(o); }

var next=function(o) {
  if (o==null || isNext(o)) {
    return tokens[tokenIdx++].token;
  }
  error(o);
}

var isNext=function(o) {
  if (tokenIdx >= tokens.length)
    return false;
  var token = tokens[tokenIdx].token;
  if (o instanceof RegExp) {
    return token.match(o);
  } else
    return (token == o);
}

var nextType=function(o) {
  if (o==null || isNextType(o)) {
    return tokens[tokenIdx++].token;
  }
  error(o);
}

var isNextType=function(pattern) {
  var type = tokens[tokenIdx].type;
  return (("|"+pattern+"|").indexOf("|"+type+"|")>=0);
}

var pcode=function(label, op, p, p1, p2) {
  symTable[funcName].pcodes.push({"label":label, "op":op, "p":p, "p1":p1,
 "p2":p2});
  var irCode = format("%s\t%s\t%s\t%s\t%s", label, op, p, p1, p2);
  log(irCode);
  irText += irCode+"\n";
```

```
}

var tempIdx = 1;
var nextTemp=function() {
  var name="T"+tempIdx++;
  symTable[name] = { type:"var", name:name };
  return name;
}

var labelIdx = 1;
var nextLabel=function() { return "L"+labelIdx++; }

var elseIdx = 1;
var nextElse=function() { return "else"+elseIdx++; }

var compile=function(text) {
  scan(text);
  PROG();
}

// PROG = STMTS
var PROG=function() {
  STMTS();
}

// STMTS = STMT*
var STMTS=function() {
  while (!isNext("}") && !isNext(end))
    STMT();
}

// BLOCK = { STMTS }
var BLOCK=function() {
  next("{");
  STMTS();
  next("}");
}
```

```
// STMT = FOR | WHILE | IF | FUNCTION | return EXP ; | ASSIGN ; | BLOCK
var STMT=function() {
  if (isNext("for")) {
    FOR();
  } else if (isNext("while")) {
    WHILE();
  } else if (isNext("if")) {
    IF();
  } else if (isNext("function")) {
    FUNCTION();
  } else if (isNext("return")) {
    next("return");
    var e = EXP();
    pcode("", "return", e, "", "");
    next(";");
  } else if (isNext("{")) {
    BLOCK();
  } else {
    ASSIGN();
    next(";");
  }
}

// FOR = for (ID in EXP) BLOCK
var FOR=function() {
  var startLabel = nextLabel(), exitLabel = nextLabel();
  next("for");
  next("(");
  var id = nextType("ID");
  pcode("", "=", id, "0", "");
  next("in");
  var e=EXP();
  next(")");
  var t = nextTemp();
  pcode(startLabel, "<", t, id, e+".length");
  pcode("", "if0", t, exitLabel, "");
```

```
  BLOCK();
  pcode("", "goto", startLabel, "", "");
  pcode(exitLabel, "", "", "", "");
}

// WHILE = while (EXP) BLOCK
var WHILE=function() {
  var startLabel = nextLabel(), exitLabel=nextLabel();
  pcode(startLabel, "", "", "", "");
  next("while");
  next("(");
  var e = EXP();
  next(")");
  pcode("", "if0", e, exitLabel, "");
  BLOCK();
  pcode("", "goto", startLabel, "", "");
  pcode(exitLabel, "", "", "", "");
}

// IF = if (EXP) STMT (else STMT)?
var IF=function() {
  next("if");
  next("(");
  var e = EXP();
  next(")");
  var elseLabel = nextLabel();
  pcode("", "if0", e, elseLabel, "");
  STMT();
  if (isNext("else")) {
    next("else");
    pcode(elseLabel, "", "", "", "");
    STMT();
  }
}

// ASSIGN = ID[++|--]?(=EXP?)?
var ASSIGN=function() {
```

```
  var id, op, hasNext = false;
  if (isNextType("ID")) {
    id = nextType("ID");
    symTable[id] = { type:"var", name:id };
    if (isNext("++") || isNext("--")) {
      var op = next(null);
      pcode("", op, id, "", "");
    }
    hasNext = true;
  }
  if (isNext("=")) {
    next("=");
    var e = EXP();
    if (id != undefined)
      pcode("", "=", id, e, "");
    hasNext = true;
  }
  if (!hasNext)
    return EXP();
}

// EXP=TERM (OP2 TERM)?
var EXP=function() {
  t1 = TERM();
  if (isNextType("OP2")) {
    var op2 = next(null);
    t2 = TERM();
    var t = nextTemp();
    pcode("", op2, t, t1, t2);
    t1 = t;
  }
  return t1;
}

// TERM=STRING | INTEGER | FLOAT | ARRAY | TABLE | ID (TERMS)? | ID [TERMS]?| ( EXP )
var TERM=function() {
```

```
if (isNextType("STRING|INTEGER|FLOAT")) {
  return next(null);
} else if (isNext("[")) {
  return ARRAY();
} else if (isNext("{")) {
  return TABLE();
} else if (isNextType("ID")) { // function call
  var id = next(null);
  if (isNext("(")) {
    next("(");
    while (!isNext(")")) {
      // TERM();
      var arg = next(null);
      pcode("", "arg", arg, "", "");
      skip(",");
    }
    next(")");
    var ret = nextTemp();
    pcode("", "call", ret, id, "");
    return ret;
  }
  var array = id;
  if (isNext("[")) {
    next("[");
    while (!isNext("]")) {
      var idx = TERM();
      var t = nextTemp();
      pcode("", "[]", t, array, idx);
      skip(",");
      array = t;
    }
    next("]");
    return array;
  }
  return id;
} else if (isNext("(")) {
  next("(");
```

```
    var e = EXP();
    next(")");
    return e;
  } else error();
}

// FUNCTION = function ID(ARGS) BLOCK
var FUNCTION = function() {
  next("function");
  funcName = nextType("ID");
  funcStack.push(funcName);
  symTable[funcName] = { type:"function", name:funcName, pcodes: [] };
  pcode(funcName, "function", "", "", "");
  next("(");
  while (!isNext(")")) {
    var arg=nextType("ID");
    pcode("", "param", arg, "", "");
    skip(",");
  }
  next(")");
  BLOCK();
  pcode("", "endf", "", "", "");
  funcStack.pop();
  funcName = funcStack[funcStack.length-1];
}

// ARRAY = [ TERMS ];
var ARRAY = function() {
  next("[");
  var array = nextTemp();
  pcode(array, "array", "", "", "");
  while (!isNext("]")) {
    var t = TERM();
    pcode("", "push", array, t, "");
    skip(",");
  }
  next("]");
```

```
  return array;
}

// TABLE = { (TERM:TERM)* }
var TABLE = function() {
  next("{");
  var table = nextTemp();
  pcode(table, "table", "", "", "");
  while (!isNext("}")) {
    var key = TERM();
    next(":");
    var value = TERM();
    skip(",");
    pcode("", "map", table, key, value);
  }
  next("}");
  return table;
}

var source = fs.readFileSync(process.argv[2], "utf8");
compile(source);
```

執行結果

然後、我們可以用 node.js 來執行上述程式，並且編譯指定的 J0 程式檔，例如以下指令就用 j0c 編譯器去編譯了 test.j0 這個輸入檔，接著畫面上所輸出的就是 IR0 的中間碼。

```
D:\Dropbox\Public\web\oc\code\js>node j0c test.j0
        arg     10
        call    T1      sum
        =       s       T1
sum     function
        param   n
        =       s       0
        =       i       1
L1
        <=      T2      i       10
        if0     T2      L2
        +       T3      s       i
```

```
        =       s       T3
        ++      i
        goto    L1
L2
        return  s
        endf
        arg     3
        arg     5
        call    T4      max
        =       m       T4
max     function
        param   a
        param   b
        >       T5      a       b
        if0     T5      L3
        return  a
L3
        return  b
        endf
total   function
        param   a
        =       s       0
        =       i       0
L4      <       T6      i       a.length
        if0     T6      L5
        []      T7      a       i
        +       T8      s       T7
        =       s       T8
        goto    L4
L5
        return  s
        endf
T9      array
        push    T9      1
        push    T9      3
        push    T9      7
        push    T9      2
```

```
        push    T9      6
        =       a       T9
        arg     a
        call    T10     total
        =       t       T10
T11     table
        map     T11     e       "dog"
        map     T11     c       "狗"
        =       word    T11
```

結語

在開放電腦計劃中，我們希望透過 J0 語言，以及 j0c 編譯器，用簡易的程式揭露「高階語言與編譯器」的設計原理。

在下期中，我們將撰寫程式去將上述 IR0 中間碼轉換為 CPU0 的組合語言，這樣就可以接上先前所作的組譯器 AS0 與虛擬機 VM0，以形成一套簡易但完整的工具鏈。

透過這樣的工具鏈，我們希望能讓熟悉程式人輕易的學會「電腦軟硬體的設計原理」。

編譯器：中間碼轉組合語言 - ir2as

前言

在上文中我們介紹了 j0c 這個編譯器的設計方式，並且設計了一種稱為 ir0 的中間碼格式，用來做為 j0c 編譯器的輸出格式。

在本文中，我們將介紹一個可以將中間碼 ir0 格式轉換成 CPU0 組合語言 (as0) 的程式，該程式稱為 ir2as0，這樣才能接上先前的 as0 組譯器，成為一套完整的工具鏈。

轉換程式

以下是這個轉換程式的原始碼，該程式會將 ir0 格式的中間碼，轉換成 as0 格式的組合語言。

檔案：ir2as.js

```
// ir2as0 中間碼轉換為組合語言，用法範例： node ir2as0 test.ir0 > test.as0
var fs = require("fs");
var util = require("util");
var format = util.format;  // 字串格式化
var log = console.log;     // 將 console.log 名稱縮短一點

// 讀入中間檔，並分割成一行一行的字串
var lines = fs.readFileSync(process.argv[2], "utf8").split("\n");
```

```
// 輸出組合語言
var asm=function(label, op, p, p1, p2) {
  var asCode = format("%s\t%s\t%s\t%s\t%s", label, op, p, p1, p2);
  log(asCode);
}

var cmpCount = 0; // 比較運算的標記不可重複，故加上一個 counter 以茲區分

// 將一行中間碼 line 轉換為組合語言
function ir2as(line) {
  var tokens = line.split("\t"); // 將中間碼分割成一個一個的欄位
  var label = tokens[0];         // 取出標記 label
  var iop = tokens[1], aop="";   // 取出運算 iop
  var p = tokens.slice(2);       // 取出參數部份
  if (label !== "")              // 若有標記，直接輸出一行只含標記的組合語
言
    asm(label, "", "", "", "");
  switch (iop) {                 // 根據運算 iop 的內容，決定要轉成甚麼組
合語言
    case "=":                    // 範例：= X Y 改為 LD R1, Y; ST R1, X
    asm("", "LD", "R1", p[1], "");
    asm("", "ST", "R1", p[0], "");
    break;
    // 範例： + X A B 改為 LD R1, A;  LD R2, B; ADD R3, R1, R2; ST R3, X;
    case "+": case "-": case "*": case "/": case "<<":
    asm("", "LD", "R1", p[1], "");
    asm("", "LD", "R2", p[2], "");
    aop = {"+":"ADD", "-":"SUB", "*":"MUL", "/":"DIV"}[iop];
    asm("", aop, "R3", "R1", "R2");
    asm("", "ST", "R3", p[0], "");
    break;
    // 範例： ++ X 改為 LDI R1, 1; LD R2, X; ADD R2, R1, R2; ST R2, X;
    case "++": case "--":
    asm("", "LDI", "R1", "1", "");
    asm("", "LD",  "R2", p[0], "");
    aop = {"++":"ADD", "--":"SUB" }[iop];
```

```
    asm("", aop, "R2", "R1", "R2");
    asm("", "ST",  "R2", p[0]);
    break;
    // 範例： < X, A, B 改為 LD R1, A; LD R2, B; CMP R1, R2; JLT CSET0; L
DI R1, 1; JMP EXIT0; CSET0: LDI R1, 0; CEXIT0: ST R1, X
    case "<": case "<=": case ">": case ">=": case "==": case "!=":
    asm("", "LD", "R1", p[1], "");
    asm("", "LD", "R2", p[2], "");
    asm("", "CMP", "R1", "R2", "");
    aop = {"<":"JLT", "<=":"JLE", ">":"JGT", ">=":"JGE", "==":"JEQ", "!="
:"JNE"}[iop];
    asm("", aop, "CSET"+cmpCounter, "", "");
    asm("", "LDI", "R1", "1", "");
    asm("",  "JMP", "CEXIT"+cmpCounter, "", "");
    asm("CSET"+cmpCount, "LDI", "R1", "0", "");
    asm("CEXIT"+cmpCount, "ST", "R1", p[0], "");
    break;
    // 範例： call X, F 改為 CALL F; ST R1, X;
    case "call":
    asm("",  "CALL", p[1], "",   "");
    asm("",  "ST",   "R1", p[0], "");
    break;
    // 範例： arg X 改為 LD R1, X; PUSH R1;
    case "arg":
    asm("",  "LD",  "R1", p[0], "");
    asm("",  "PUSH","R1", "", "");
    break;
    case "function": // 範例： sum function 只生成標記 sum，沒有生成組合語
言指令
    break;
    case "endf": // 函數結束，沒有生成組合語言指令
    break;
    case "param": // 範例： param X 改為 POP R1; ST R1, X;
    asm("",  "POP", "R1", "", "");
    asm("",  "ST",  "R1", p[0], "");
    break;
    case "return":  // 範例： return X 改為 LD R1, X; RET;
```

```
    asm("",  "LD","R1", p[0], "");
    asm("",  "RET","", "", "");
    break;
    case "if0":  // 範例: if0 X Label 改為 CMP R0, X; JEQ Label;
    asm("",  "CMP","R0", p[0], "");
    asm("",  "JEQ",p[1], "", "");
    break;
    case "goto":  // 範例: goto Label 改為 JMP label
    asm("",  "JMP", p[0], "", "");
    break;
    case "array":  // 範例: X array 改為 LD R1, X; CALL ARRAY; (註: X=new array())
    asm("",  "LD", "R1", p[0], "");
    asm("",  "CALL", "ARRAY", "", "");
    break;
    case "[]":  // 範例: [] X A i  改為 LD R1, A; LD R2, i; CALL AGET; ST R1, X (註: X=A[i])
    asm("",  "LD", "R1", p[1], "");
    asm("",  "LD", "R2", p[2], "");
    asm("",  "CALL", "AGET", "", "");
    asm("",  "ST", "R1", p[0], "");
    break;
    case "length": // 範例: length len, A 改為 LD R1, A; CALL ALEN; ST R1, len;
    asm("",  "LD", "R1", p[1], "");
    asm("",  "CALL", "ALEN", "", "");
    asm("",  "ST", "R1", p[0], "");
    break;
    case "apush": // 範例: apush A, X 改為 LD R1,A; LD R2, X; CALL APUSH
    asm("",  "LD", "R1", p[0], "");
    asm("",  "LD", "R2", p[1], "");
    asm("",  "CALL", "APUSH", "", "");
    break;
    case "table": // 範例: table T 改為 LD R1,T; CALL TABLE
    asm("",  "LD", "R1", p[0], "");
    asm("",  "CALL", "TABLE", "", "");
    break;
```

```
    case "map": // 範例： map table field value 改為 LD R1, table; LD R2,
 field; LD R3, value; CALL TMAP
      asm("", "LD", "R1", p[0], "");
      asm("", "LD", "R2", p[1], "");
      asm("", "LD", "R3", p[2], "");
      asm("", "CALL", "TMAP", "", "");
      break;
    case "":
      break;
    default:
      log("Error : %s not found!", iop);
  }
}

// 將所有中間碼都轉換為組合語言
for (var i in lines) {
  if (lines[i].trim().length > 0) {
    log("// %s", lines[i]);
    ir2as(lines[i]);
  }
}
```

執行結果

首先我們要使用 j0c 編譯器將 j0 語言的程式，編譯為 ir0 的中間碼格式。然後再利用 ir2as0 將中間碼轉換成 CPU0 的組合語言，以下是一個將 test.j0 編譯 test.ir0 中間檔，然後再利用 ir2as0 將中間檔轉換為 test.as0 組合語言的過程。

```
C:\Dropbox\Public\web\oc\code\js>node j0c test.j0 > test.ir0

C:\Dropbox\Public\web\oc\code\js>node ir2as0 test.ir0 > test.as0
```

以下是 test.j0 => test.ir0 => test.as0 這個編譯轉換過程當中的檔案內容。

高階語言檔： test.j0

```
s = sum(10);

function sum(n) {
  s = 0;
```

```
  i=1;
  while (i<=10) {
    s = s + i;
    i++;
  }
  return s;
}

m = max(3,5);

function max(a, b) {
  if (a > b)
    return a;
  else
    return b;
}

function total(a) {
  s = 0;
  for (i in a) {
    s = s + a[i];
  }
  return s;
}

a = [ 1, 3, 7, 2, 6];
t = total(a);
word = { e:"dog", c:"狗" };
```

中間碼檔案： test.ir0

```
    arg 10
    call    T1  sum
    =   s   T1
sum function
    param   n
    =   s   0
    =   i   1
```

```
L1
    <=  T2  i   10
    if0 T2  L2
    +   T3  s   i
    =   s   T3
    ++  i
    goto    L1
L2
    return  s
    endf
    arg 3
    arg 5
    call    T4  max
    =   m   T4
max function
    param   a
    param   b
    >   T5  a   b
    if0 T5  L3
    return  a
L3
    return  b
    endf
total   function
    param   a
    =   s   0
    =   i   0
L4  length  T6  a
    <   T7  i   T6
    if0 T7  L5
    []  T8  a   i
    +   T9  s   T8
    =   s   T9
    goto    L4
L5
    return  s
    endf
```

```
    array   T10
    apush   T10 1
    apush   T10 3
    apush   T10 7
    apush   T10 2
    apush   T10 6
    =   a   T10
    arg a
    call    T11 total
    =   t   T11
    table   T12
    map T12 e   "dog"
    map T12 c   "狗"
    =   word    T12
```

組合語言檔： test.as0

```
//  arg 10
    LD  R1  10
    PUSH    R1
//  call    T1  sum
    CALL    sum
    ST  R1  T1
//  =   s   T1
    LD  R1  T1
    ST  R1  s
// sum  function
sum
//  param   n
    POP R1
    ST  R1  n
//  =   s   0
    LD  R1  0
    ST  R1  s
//  =   i   1
    LD  R1  1
    ST  R1  i
// L1
```

```
L1
//  <=  T2  i   10
    LD  R1  i
    LD  R2  10
    CMP R1  R2
    JLE CSET0
    LDI R1  1
    JMP CEXIT0
CSET0   LDI R1  0
CEXIT0  ST  R1  T2
//  if0 T2  L2
    CMP R0  T2
    JEQ L2
//  +   T3  s   i
    LD  R1  s
    LD  R2  i
    ADD R3  R1  R2
    ST  R3  T3
//  =   s   T3
    LD  R1  T3
    ST  R1  s
//  ++  i
    LDI R1  1
    LD  R2  i
    ADD R2  R1  R2
    ST  R2  i   undefined
//  goto    L1
    JMP L1
// L2
L2
//  return  s
    LD  R1  s
    RET
//  endf
//  arg 3
    LD  R1  3
    PUSH    R1
```

```
//  arg 5
    LD  R1  5
    PUSH    R1
//  call    T4  max
    CALL    max
    ST  R1  T4
//  =   m   T4
    LD  R1  T4
    ST  R1  m
// max  function
max
//  param   a
    POP R1
    ST  R1  a
//  param   b
    POP R1
    ST  R1  b
//  >   T5  a   b
    LD  R1  a
    LD  R2  b
    CMP R1  R2
    JGT CSET0
    LDI R1  1
    JMP CEXIT0
CSET0   LDI R1  0
CEXIT0  ST  R1  T5
//  if0 T5  L3
    CMP R0  T5
    JEQ L3
//  return  a
    LD  R1  a
    RET
// L3
L3
//  return  b
    LD  R1  b
    RET
```

```
//  endf
// total    function
total
//  param   a
    POP R1
    ST  R1  a
//  =   s   0
    LD  R1  0
    ST  R1  s
//  =   i   0
    LD  R1  0
    ST  R1  i
// L4   length  T6  a
L4
    LD  R1  a
    CALL    ALEN
    ST  R1  T6
//  <   T7  i   T6
    LD  R1  i
    LD  R2  T6
    CMP R1  R2
    JLT CSET0
    LDI R1  1
    JMP CEXIT0
CSET0   LDI R1  0
CEXIT0  ST  R1  T7
//  if0 T7  L5
    CMP R0  T7
    JEQ L5
//  []  T8  a   i
    LD  R1  a
    LD  R2  i
    CALL    AGET
    ST  R1  T8
//  +   T9  s   T8
    LD  R1  s
    LD  R2  T8
```

```
    ADD R3  R1  R2
    ST  R3  T9
//  =   s   T9
    LD  R1  T9
    ST  R1  s
//  goto    L4
    JMP L4
// L5
L5
//  return  s
    LD  R1  s
    RET
//  endf
//  array   T10
    LD  R1  T10
    CALL    ARRAY
//  apush   T10 1
    LD  R1  T10
    LD  R2  1
    CALL    APUSH
//  apush   T10 3
    LD  R1  T10
    LD  R2  3
    CALL    APUSH
//  apush   T10 7
    LD  R1  T10
    LD  R2  7
    CALL    APUSH
//  apush   T10 2
    LD  R1  T10
    LD  R2  2
    CALL    APUSH
//  apush   T10 6
    LD  R1  T10
    LD  R2  6
    CALL    APUSH
//  =   a   T10
```

```
    LD   R1   T10
    ST   R1   a
//  arg a
    LD   R1   a
    PUSH      R1
//  call      T11 total
    CALL      total
    ST   R1   T11
//  =    t    T11
    LD   R1   T11
    ST   R1   t
//  table     T12
    LD   R1   T12
    CALL      TABLE
//  map T12 e    "dog"
    LD   R1   T12
    LD   R2   e
    LD   R3   "dog"
    CALL      TMAP
//  map T12 c    "狗"
    LD   R1   T12
    LD   R2   c
    LD   R3   "狗"
    CALL      TMAP
//  =    word     T12
    LD   R1   T12
    ST   R1   word
```

結語

截至目前為止，我們已經為開放電腦計畫實作了一組簡單的工具鏈，包含用 node.js + javascript 設計的 j0c 編譯器、ir2as0 中間碼轉換器、 as0 組譯器、vm0 虛擬機、以及用 Verilog 設計的 CPU0, MCU0 處理器等等。

這套工具鏈的設計都是以「簡單易懂」為原則，採用 Keep It Simple and Stupid (KISS) 的原則，希望能透過這樣的方式，揭露電腦的各個設計層面，讓讀者可以透過開放電腦計畫理解電腦從軟體到硬體的設計原理。

不過、我們還沒有完成整個計畫，開放電腦計畫顯然還有些缺憾，像是我們還沒有設計作業系統 (OS)，也沒有用 Verilog 設計開放電腦的週邊裝置電路，另外在 FPGA 實際燒錄也只有很簡單的範例程式，還沒辦法形成一套從軟體到硬體串接的很完整的系統。

因此、我們打算在 2014 年暑假在成大與蘇文鈺老師一起舉辦一個「開放FPGA電腦創世紀黑客松」，我們

已經為這個活動建立了一個 facebook 社團，歡迎對「開放電腦計畫」或 FPGA 有興趣的朋友們，一起來參與這個活動，以下是該社團的網址：

- https://www.facebook.com/groups/OpenFPGAComputerPhone/

歡迎大家一同來參加！

結語

在本書當中，我們用 JavaScript 實作了「虛擬機、組譯器、編譯器」等工具，形成了一套工具鏈，說明了「系統軟體」的設計原理。

但可惜的是、筆者對作業系統設計的理解還不夠深入，因此尚未設計出簡易的作業系統，這個任務只好留待下一版再來完成了。