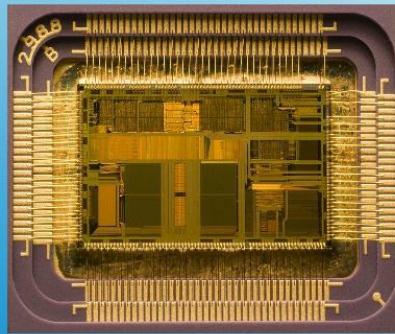


開放電腦計畫系列叢書

計算機結構

使用 Verilog 實作



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 -- 計算機硬體結構

2014 年 7 月出版

作者：陳鍾誠（創作共用：姓名標示、相同方式分享授權）

開放電腦計畫 -- 計算機硬體結構

- 前言
 - 序
 - 授權聲明
- 開放電腦計畫
 - 簡介
 - 硬體：計算機結構
 - 軟體：系統程式
 - 結語
 - 參考文獻
- 電腦硬體架構
 - 電腦的結構
 - CPU0 處理器
 - CPU0 的指令集
 - CPU0 指令格式
 - 狀態暫存器
 - 位元組順序
 - 中斷程序
 - CPU0 的組合語言與機器碼

- 參考文獻
- 硬體描述語言 -- Verilog
 - Verilog 基礎
 - 區塊式設計
 - 流程式設計
 - 結語
 - 參考文獻
- 組合邏輯 (Combinatorial Logic)
 - 簡介
 - 加法器
 - 32 位元加法器
 - 前瞻進位加法器 (Carry Lookahead Adder)
 - 結語
 - 參考文獻
- 算術邏輯單元 ALU 的設計
 - 加減器
 - 採用 CASE 語法設計 ALU 模組
 - 完整的 ALU 設計 (含測試程式)
 - 結語
 - 參考文獻

- 記憶單元 (Memory Unit)
 - 時序邏輯 (Sequential Logic)
 - 正反器 (閂鎖器)
 - SR 正反器
 - 有 enable 的正反器
 - 閘級延遲 (Gate Delay)
 - 利用「閘級延遲」製作脈波變化偵測器 (Pulse Transition Detector, PTD)
 - 使用「脈衝偵測電路」製作「邊緣觸發正反器」
 - 使用「脈衝偵測電路」設計邊緣觸發暫存器
 - 使用「脈衝偵測電路」製作計數電路
 - 暫存器單元
 - 記憶體
 - 結語
- 控制單元
 - 簡介
 - 流程式設計
 - 區塊式設計
- 微處理器 (Micro Processor)
 - MCU0 的迷你版 -- mcu0m
 - MCU0 的區塊式設計 -- MCU0bm.v

- MCU0 完整版
- 輸出入單元 (I/O)
 - 前言
 - BUS (總線, 匯流排)
 - 同步匯流排 (Synchronous BUS)
 - 異步匯流排 (Asynchronous BUS)
 - 匯流排仲裁 (BUS arbitery)
 - 循序與平行輸出入 (Serial vs. Parallel)
 - 常見的輸出入協定
 - MCU0 的輸出入 -- 輪詢篇
- 記憶系統 (Storage)
 - 記憶體階層 (Memory Hierarchy)
 - 快取記憶體 (Cache)
- 高階處理器 (Processor)
 - 哈佛架構 (Harvard Architecture)
 - 流水線架構 (Pipeline)
 - CPU0 迷你版 - CPU0m
 - CPU0 完整版 -- cpu0s
- 速度議題
 - 乘法與除法

- 浮點數運算
 - 繪圖加速功能 (Graphics)
 - 平行處理 (Parallel)
 - 結語
- 附錄
 - 本書內容與相關資源

前言

序

本書是「開放電腦計畫的硬體部份」，描述如何設計電腦硬體的方法，透過這本書，我們希望讓「計算機結構」這門課程，變成是可以動手實作的。

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(包含該文章作者，若有來自維基百科的部份也請一併標示)。
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

簡介

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。

	開發環境	gcc icarus node.js	arduino	CodeSourcery GNU tool chain	GNU tool chain
軟體： 系統 程式	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	
	處理器	cpu0 mcu0	AVR8 Papilio	ARM ARM_risclite	IA32/64
硬體： 計算 機 結構	FPGA : Altera / Xilinx				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：

	開發環境	gcc icarus node.js	arduino	CodeSourcery GNU tool chain	GNU tool chain
軟體 ：系統 程 式	虛擬機	vm0	IMAVR	QEMU / v8	
	作業系統	os0		FreeRTOS	Linux
	編譯器	cc0	avr-g++	llvm/gcc/g++	
	組譯器	as0	avrasm	as	
硬體 ：計算 機結構	處理器	cpu0	AVR8	ARM	IA32/64
		mcu0	Papilio	ARM_risclite	
	FPGA : Altera / Xilinx				ASIC
	週邊	io0	UART/PS2/VGA		
		簡易實作	嵌入式系統	高階處理器	

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 JavaScript 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 $1+2+\dots+10$ 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.asm sum.o
```

上述的程式經過組譯之後，會輸出組譯報表，如下所示。

sum.asm 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 JOC (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。

結語

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。

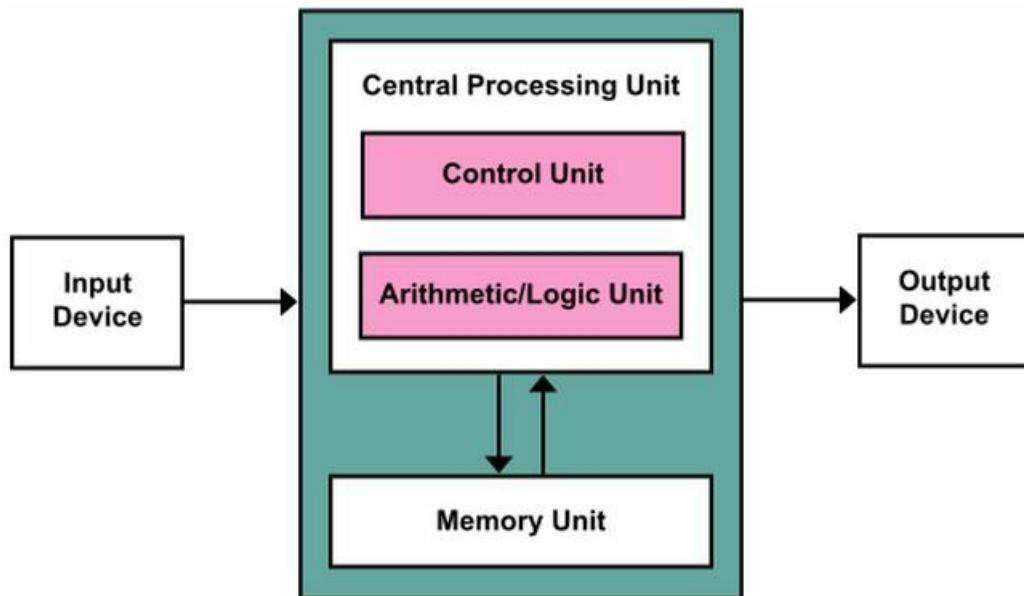
參考文獻

- 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](#)

電腦硬體架構

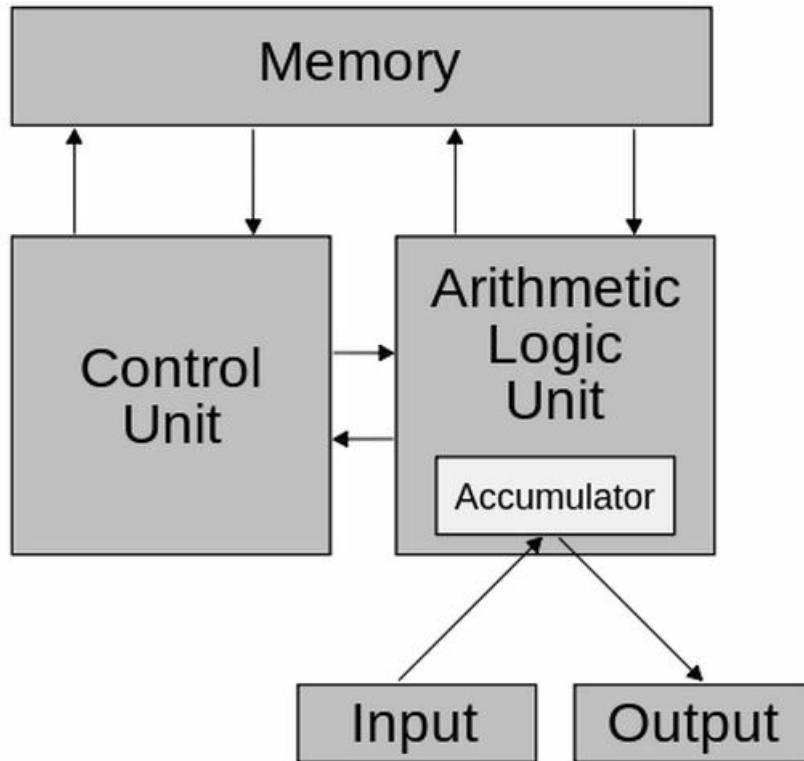
電腦的結構

傳統的電腦架構，最經典的模型是由數學大師「馮紐曼」(Von Neumann)所描述的，因此稱為「馮紐曼架構」，如以下兩個圖片所示：



圖、馮紐曼架構 (1)

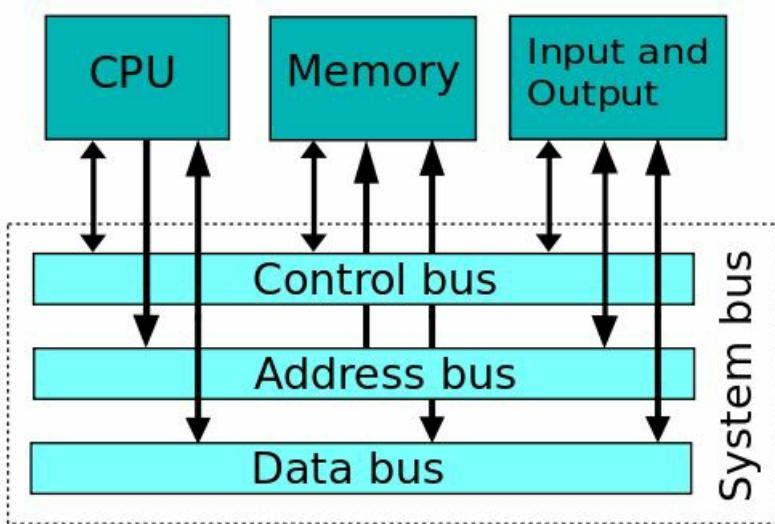
圖片來源：http://en.wikipedia.org/wiki/File:Von_Neumann_Architecture.svg



圖、馮紐曼架構(2)

圖片來源：http://en.wikipedia.org/wiki/File:Von_Neumann_architecture.svg

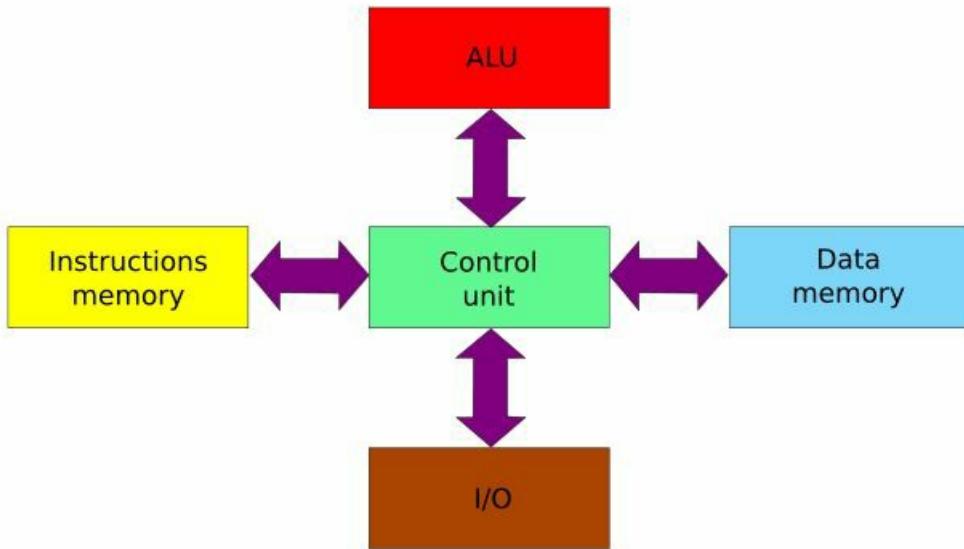
在早期、晶片線路成本還算高的時候，最常見的馮紐曼架構電腦，是採用單匯流排的架構，如下圖所示：



圖、單一匯流排的馮紐曼架構

圖片來源：http://en.wikipedia.org/wiki/File:Computer_system_bus.svg

但是、自從 RISC 精簡指令集電腦出現，由於 pipeline 必須讓指令與資料可以同時被存取，很多電腦改採以下指令與資料分開的哈佛架構 (Harvard Architecture)：



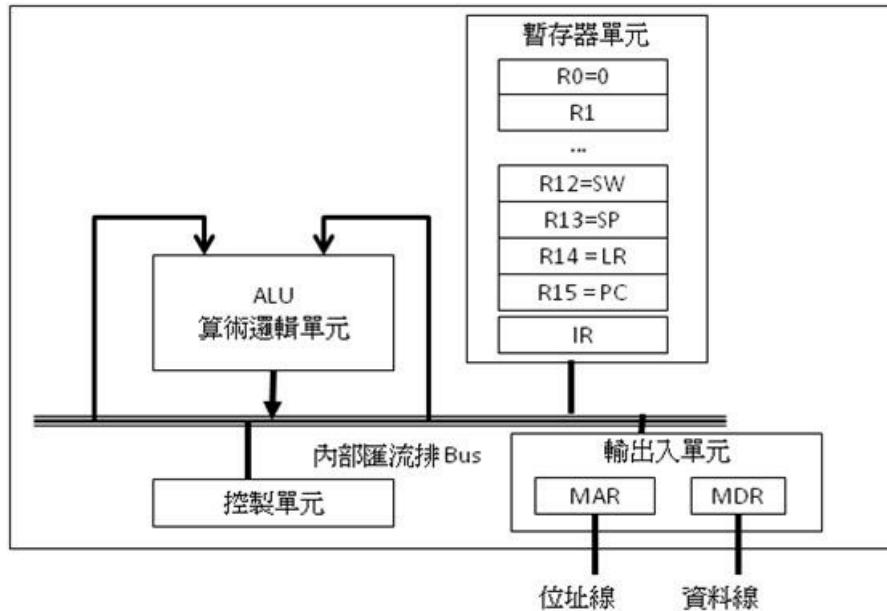
圖、指令與資料匯流排分開的哈佛架構

圖片來源：http://en.wikipedia.org/wiki/File:Harvard_architecture.svg

必須注意的是，雖然哈佛架構當中的指令與資料是完全分開的，但是如果真的將指令與資料個記憶體整個分開的話，會需要兩套記憶體與匯流排，這會讓 CPU 的接腳數大增，也會讓記憶體運用沒效率，因此在實務上，通常是將「快取記憶體」分成兩邊，一邊是指令快取，一邊是資料快取，這樣就只有內部匯流排需要分開成兩套，而 CPU 對外則只要一套匯流排與記憶體，這種架構可以看成哈佛架構的一種變形。

CPU0 處理器

CPU0 是一個簡易的 32 位單匯流排處理器，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 的指令集

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。

A型

OP	Ra	Rb	Rc	Cx{12 bits}
31-24	23-20	19-16	15-12	11-0

L型

OP	Ra	Rb	Cx{16 bits}
31-24	23-20	19-16	15-0

J型

OP	Cx{24 bits}
31-24	23-0

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

CPU0 的組合語言與機器碼

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 $1+2+\dots+10$ 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。

	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
```

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
```

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
```

參考文獻

- 系統程式 (陳鍾誠著, 旗標出版社) -- <http://sp1.wikidot.com/main>
- [JavaScript \(6\) – Node.js 命令列程式設計](#)

硬體描述語言 -- Verilog

Verilog 與 VHDL 都是用來設計數位電路的硬體描述語言，但 VHDL 在1983年被提出後，1987 年被美國國防部和IEEE確定為標準的硬體描述語言。

Verilog 是由 Gateway Design Automation 公司於 1984 年開始發展的， Cadence Design Systems 公司於 1990 年購併了 Gateway 公司， Cadence 隨後將 Verilog 提交到 Open Verilog International 成為開放公用標準，1995 年 Verilog 被 IEEE 認可成為 IEEE 1364-1995 標準，簡稱為 Verilog-95。此一標準於 2001 年更新後成為 Verilog-2001 。

相較於 VHDL 而言， Verilog 的語法較為簡潔，因此經常被專業的數位電路設計者採用，而 VHDL 的使用族群則有較多的初學者。當我們想學習數位電路設計時，經常會難以選擇要用哪一種語言，因為 VHDL 的書籍與教材似乎比 Verilog 多一些，但是 Verilog 的高階設計電路（像是開放原始碼 CPU 等）則比 VHDL 多很多。

筆者是為了要設計 CPU 而學習數位電路設計的，因此決定學習 Verilog 語言，而非 VHDL 語言。雖然筆者也學過 VHDL 語言，但後來發現 Verilog 相當好，相對而言語法簡潔了許多，因此筆者比較偏好 Verilog 語言。

在本文中，我們將介紹 Verilog 的基本語法，並且採用 Icarus 作為主要開發測試工具，以便讓讀者能很快的進入 Verilog 硬體設計的領域。

Verilog 基礎

Verilog 的基本型態

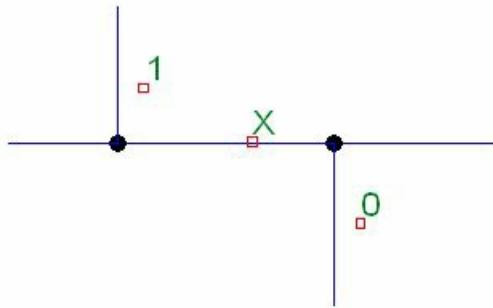
在一般的程式語言當中，資料的最基本型態通常是「位元」(bit)，但是在 Verilog 這種「硬體描述語言」當中，我們必須有「面向硬體」的思考方式，因此最基本的型態從「位元」轉換為「線路」(wire)。

一條線路的可能值，除了 0 與 1 之外，還有可能是未定值 X，以及高阻抗 Z，如下表所示：

值	意義	說明
0	低電位	布林代數中的假值
1	高電位	布林代數中的真值
Z	高阻抗	三態緩衝器的輸出，高阻抗斷線
X	未定值	像是線路未初始化之前，以及有 0,1 兩者衝突的線路值，或者是輸入為 Z 的輸出值

其中的 0 對應到低電位、1 對應到高電位，這是比較容易理解的部分，但是未定值 X 與高阻抗 Z 各代表甚麼意義呢？

對於一條沒有阻抗的線路而言，假如我們在某點對該線路輸出 1, 另一點對該線路輸出 0，那麼這條線路到底應該是高電位還是低電位呢？



圖、造成未定值 X 的情況

對於這種衝突的情況，Verilog 採用 X 來代表該線路的值。

而高阻抗，則基本上是代表斷線，您可以想像該線路如果是「非導體」，例如「塑膠、木頭、開關開路、或者是處於高阻抗 情況的半導體」等，就會使用者種 Z 值來代表。

根據這樣的四種線路狀態，一個原本簡易的 AND 閘，在數位邏輯中只要用 $2*2$ 的真值表就能表示了，但在 Verilog 當中則有 $4*4$ 種可能的情況，以下是 Verilog 中各種運算 (AND, OR, XOR, XNOR) 在這四種型態上的真值表定義：



AND (&)	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

OR ()	0	1	X	Z
0	0	1	X	X
1	1	1	1	1
X	X	1	X	X
Z	X	1	X	X

XOR(^)	0	1	X	Z
0	0	1	X	X
1	1	0	X	X
X	X	X	X	X
Z	1	X	X	X

XNOR(\sim)	0	1	X	Z	
0		1	0	X	X
1		0	1	X	X
X		X	X	X	X
Z		X	X	X	X

在 Verilog 當中，如果我們要宣告一條線路，只要用下列語法就可以了：

```
wire w1;
```

如果我們想一次宣告很多條線路，那麼我們可以用很多個變數描述：

```
wire w, x, y, z;
```

但是如果我們想宣告一整個排線 (例如匯流排)，那我們就可以用下列的陣列語法：

```
wire [31:0] bus;
```

如果想要一次宣告很多組排線，那我們就可以用下列的陣列群語法：

```
wire [31:0] bus [0:3];
```

當然、除了線路之外，Verilog 還有可以穩定儲存位元的型態，稱為 reg (暫存器)，reg 可以用來儲存位元，而非像線路一樣只是「一種連接方式」而已，以下是一些 reg 的宣告方式：

```
reg w;                      // 宣告一位元的暫存器變數 w
```

```
reg x, y, z;                // 宣告三個一位元的暫存器變數 x, y, z
```

```
reg [31:0] r1;           // 宣告 32 位元的暫存器 r1  
reg [31:0] R [0:15];    // 宣告 16 個 32 位元的暫存器群組 R[0..15]
```

在 Verilog 中，wire 與 reg 是比較常用的基本型態，另外還有一些較不常用的基本型態，像是 tri (三態線路)、trireg (三態暫存器)、integer (整數) 等，在此我們先不進行介紹。

Icarus : Verilog 的編譯執行工具

Icarus 是由 Stephen Williams 所設計的 Verilog 開發工具，採用 GPL 授權協議，並且可以在 Linux, BSD, OS X, MS Windows 等環境下執行。

Icarus 支援 Verilog 的 IEEE 1995、IEEE 2001 和 IEEE 2005 三種標準語法，也支援部分的 SystemVerilog 語法，其官方網站網址如下：

- <http://iverilog.icarus.com/>

如果您是 MS Windows 的使用者，可以從以下網址中下載 Icarus 的 MS Windows 版本，其安裝非常容易：

- <http://bleyer.org/icarus/>

範例 1：XOR3 的電路

```
module xor3(input a, b, c, output abc);
wire ab;
xor g1(ab, a, b);
xor g2(abc, c, ab);
endmodule
```

```
module xor3test;
reg a, b, c;
wire abc;

xor3 g(a, b, c, abc);

initial
begin
    a = 0;
    b = 0;
    c = 0;
```

```
end

always #50 begin
    a = a+1;
    $monitor("%4dns monitor: a=%d b=%d c=%d a^b^c=%d", $stime, a, b, c
, abc);
end

always #100 begin
    b = b+1;
end

always #200 begin
    c = c+1;
end

initial #2000 $finish;
```

```
endmodule
```

Icarus 執行結果

```
D:\ccc101\icarus\ccc>iverilog -o xor3test xor3test.v
```

```
D:\ccc101\icarus\ccc>vvp xor3test
```

```
50ns monitor: a=1 b=0 c=0 a^b^c=1
100ns monitor: a=0 b=1 c=0 a^b^c=1
150ns monitor: a=1 b=1 c=0 a^b^c=0
200ns monitor: a=0 b=0 c=1 a^b^c=1
250ns monitor: a=1 b=0 c=1 a^b^c=0
300ns monitor: a=0 b=1 c=1 a^b^c=0
350ns monitor: a=1 b=1 c=1 a^b^c=1
400ns monitor: a=0 b=0 c=0 a^b^c=0
450ns monitor: a=1 b=0 c=0 a^b^c=1
500ns monitor: a=0 b=1 c=0 a^b^c=1
550ns monitor: a=1 b=1 c=0 a^b^c=0
600ns monitor: a=0 b=0 c=1 a^b^c=1
```

650ns monitor: a=1 b=0 c=1 $a \wedge b \wedge c = 0$
700ns monitor: a=0 b=1 c=1 $a \wedge b \wedge c = 0$
750ns monitor: a=1 b=1 c=1 $a \wedge b \wedge c = 1$
800ns monitor: a=0 b=0 c=0 $a \wedge b \wedge c = 0$
850ns monitor: a=1 b=0 c=0 $a \wedge b \wedge c = 1$
900ns monitor: a=0 b=1 c=0 $a \wedge b \wedge c = 1$
950ns monitor: a=1 b=1 c=0 $a \wedge b \wedge c = 0$
1000ns monitor: a=0 b=0 c=1 $a \wedge b \wedge c = 1$
1050ns monitor: a=1 b=0 c=1 $a \wedge b \wedge c = 0$
1100ns monitor: a=0 b=1 c=1 $a \wedge b \wedge c = 0$
1150ns monitor: a=1 b=1 c=1 $a \wedge b \wedge c = 1$
1200ns monitor: a=0 b=0 c=0 $a \wedge b \wedge c = 0$
1250ns monitor: a=1 b=0 c=0 $a \wedge b \wedge c = 1$
1300ns monitor: a=0 b=1 c=0 $a \wedge b \wedge c = 1$
1350ns monitor: a=1 b=1 c=0 $a \wedge b \wedge c = 0$
1400ns monitor: a=0 b=0 c=1 $a \wedge b \wedge c = 1$
1450ns monitor: a=1 b=0 c=1 $a \wedge b \wedge c = 0$
1500ns monitor: a=0 b=1 c=1 $a \wedge b \wedge c = 0$

```
1550ns monitor: a=1 b=1 c=1 a^b^c=1
1600ns monitor: a=0 b=0 c=0 a^b^c=0
1650ns monitor: a=1 b=0 c=0 a^b^c=1
1700ns monitor: a=0 b=1 c=0 a^b^c=1
1750ns monitor: a=1 b=1 c=0 a^b^c=0
1800ns monitor: a=0 b=0 c=1 a^b^c=1
1850ns monitor: a=1 b=0 c=1 a^b^c=0
1900ns monitor: a=0 b=1 c=1 a^b^c=0
1950ns monitor: a=1 b=1 c=1 a^b^c=1
2000ns monitor: a=0 b=0 c=0 a^b^c=0
```

仔細觀察上述輸出結果，您會發現這個結果與真值表的內容完全一致，因此驗證了該設計的正確性！

透過這種方式，您就可以用 Verilog 設計電路的程式，然後用 Icarus 編譯並驗證電路是否正確。

區塊式設計

閘級的線路設計方法

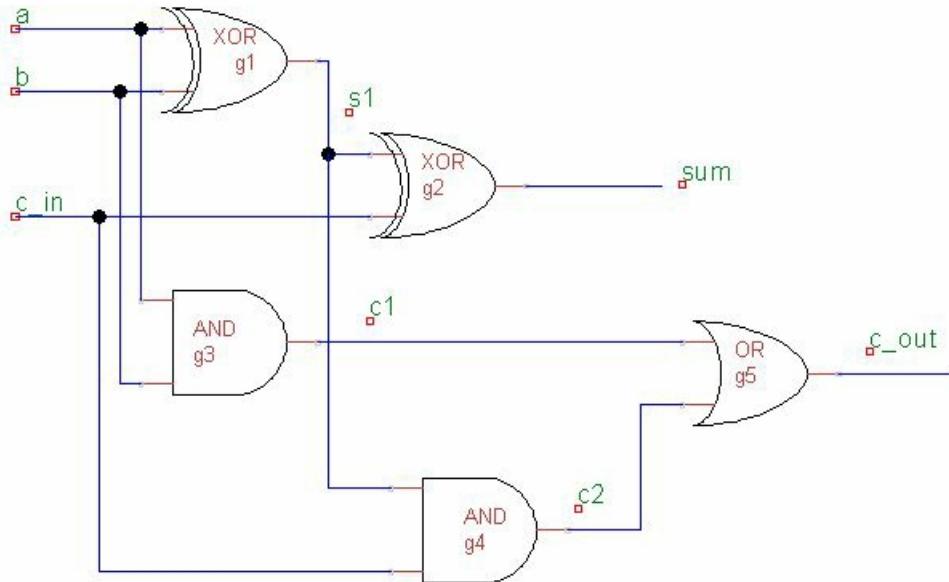
Verilog 既然是硬體描述語言，那當然會有邏輯閘的表示法，Verilog 提供的邏輯閘有 and, nand, or, nor, xor, xnor, not 等元件，因此您可以用下列 Verilog 程式描述一個全加器：

```
module fulladder (input a, b, c_in, output sum, c_out);
wire s1, c1, c2;

xor g1(s1, a, b);
xor g2(sum, s1, c_in);
and g3(c1, a, b);
and g4(c2, s1, c_in) ;
or g5(c_out, c2, c1) ;

endmodule
```

上述程式所對應的電路如下圖所示：



全加器電路圖

這些邏輯閘並不受限於兩個輸入，也可以是多個輸入的，例如以下範例中的 *g* 閘，就一次將三個輸入 *a*, *b*, *c_{in}* 進行 xor 運算，產生輸出 *sum* 的結果。

```
xor g(sum, a, b, c_in);
```

範例 2：全加器的閘級設計

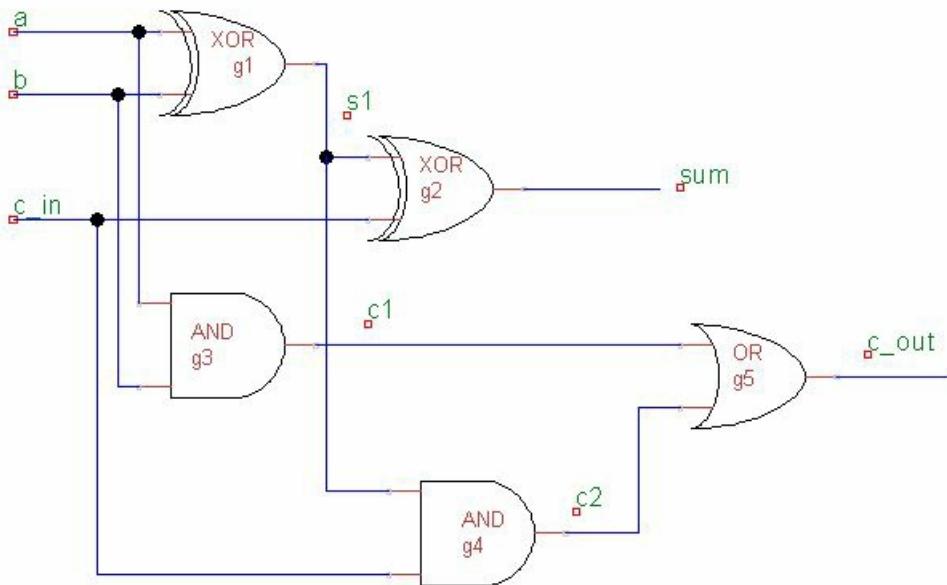
傳統的數位邏輯課程當中，我們通常會用「邏輯閘」的組合方式，來設計出所要的電路，以下我們就用「全加器」當範例，說明如何用「閘級」的語法，在 Verilog 當中設計數位電路。

全加器總共有 3 個輸入 (a, b, c_{in})，兩個輸出值 (sum, c_{out})，其真值表如下所示：

a	b	c_{in}	c_{out}	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

根據這個真值表，我們可以用卡諾圖得到化簡後的電路 (但必須注意的是，卡諾圖化簡出來的電路只有 AND, OR, NOT，沒有 XOR)，然後根據化簡後的算式繪製電路圖。(在此範例中， c_{out} 可以採用卡諾圖化簡出來，但 sum 使用的並非化簡的結果，而是以經驗得到的 XOR 組合式)

當您完成邏輯運算式設計之後，就可以用 TinyCAD 這個軟體，繪製出全加器的電路如下圖所示：



用 TinyCAD 繪製的全加器電路圖

接著我們可以按照以上的線路，根據 Verilog 的語法，設計出對應元件與測試程式如下所示：

程式：fulladder.v

```
// 以下為全加器模組的定義
module fulladder (input a, b, c_in, output sum, c_out);
wire s1, c1, c2;

xor g1(s1, a, b);
xor g2(sum, s1, c_in);
and g3(c1, a, b);
and g4(c2, s1, c_in) ;
or g5(c_out, c2, c1) ;

endmodule

// 以下為測試程式
module main;
```

```
reg a, b, c_in;
wire sum, c_out;

fulladder fa1(a, b, c_in, sum, c_out);

initial begin
    a = 0;  b = 0;  c_in = 0;
    $monitor("%04dns monitor: a=%d b=%d c_in=%d c_out=%d sum=%d", $time, a, b, c_in, c_out, sum);
    #1000 $finish;
end

always #50 c_in = c_in+1;

always #100 b = b+1;

always #200 a = a+1;
```

```
endmodule
```

然後我們就可以利用 Icarus 進行編譯與測試，看看 fulladder.v 的模擬執行結果是否正確。

編譯執行結果：

```
D:\Dropbox\Public\pmag\201306\code>iverilog -o fulladder fulladder.v
```

```
D:\Dropbox\Public\pmag\201306\code>vvp fulladder
```

```
0000ns monitor: a=0 b=0 c_in=0 c_out=0 sum=0
```

```
0050ns monitor: a=0 b=0 c_in=1 c_out=0 sum=1
```

```
0100ns monitor: a=0 b=1 c_in=0 c_out=0 sum=1
```

```
0150ns monitor: a=0 b=1 c_in=1 c_out=1 sum=0
```

```
0200ns monitor: a=1 b=0 c_in=0 c_out=0 sum=1
```

```
0250ns monitor: a=1 b=0 c_in=1 c_out=1 sum=0
```

```
0300ns monitor: a=1 b=1 c_in=0 c_out=1 sum=0
```

```
0350ns monitor: a=1 b=1 c_in=1 c_out=1 sum=1
```

```
0400ns monitor: a=0 b=0 c_in=0 c_out=0 sum=0
```

```
0450ns monitor: a=0 b=0 c_in=1 c_out=0 sum=1
```

```
0500ns monitor: a=0 b=1 c_in=0 c_out=0 sum=1
0550ns monitor: a=0 b=1 c_in=1 c_out=1 sum=0
0600ns monitor: a=1 b=0 c_in=0 c_out=0 sum=1
0650ns monitor: a=1 b=0 c_in=1 c_out=1 sum=0
0700ns monitor: a=1 b=1 c_in=0 c_out=1 sum=0
0750ns monitor: a=1 b=1 c_in=1 c_out=1 sum=1
0800ns monitor: a=0 b=0 c_in=0 c_out=0 sum=0
0850ns monitor: a=0 b=0 c_in=1 c_out=0 sum=1
0900ns monitor: a=0 b=1 c_in=0 c_out=0 sum=1
0950ns monitor: a=0 b=1 c_in=1 c_out=1 sum=0
1000ns monitor: a=1 b=0 c_in=0 c_out=0 sum=1
```

習題 1：請證明 nand 閘是全能的 (提示：只要用 nand 做出 and, or, not 就行了) 習題 2：請用卡諾圖去化簡全加器的 Cout 電路。 習題 3：請寫一個 Verilog 程式用 nand 兜出 or 電路，並測試之。

區塊式設計的注意事項

當您採用區塊式設計時，有一些常見的初學者錯誤必須注意，列舉如下：

1. `assign` 的指定是針對線路，而暫存器的指定必須放在 `always` 或 `initial` 區塊裏。

因此，下列程式中在 assign 指定 reg 型態的變數 o1 = i1 是錯的，必須將 reg 去掉，或者改放到 always 區塊裏。

```
D:\Dropbox\Public\web\co\code>iverilog error1.v -o error1  
error1.v:2: error: reg o1; cannot be driven by primitives or continuous assignment.  
! error(s) during elaboration.  
  
D:\Dropbox\Public\web\co\code>iverilog correct1.v -o correct1  
D:\Dropbox\Public\web\co\code>
```


module not1(input i1, output **reg** o1);
assign o1 = !i1;
endmodule


module not1(input i1, output o1);
assign o1 = !i1;
endmodule

圖、在 assign 裏只能指定 wire 型態的變數，不能指定 reg 型態的

2. 相反的，always 區塊裏的指定，只能針對 reg 型態的變數進行，不能套用在線路 (wire: 含 input, output, inout) 型態的變數上。

因此、下列程式當中若沒有將 o1 加上 reg 型態，則會發生錯誤。

```
D:\Dropbox\Public\web\co\code>iverilog error2.v -o error2  
error2.v:3: error: o1 is not a valid l-value in not1.  
error2.v:1:      : o1 is declared here as wire.  
Elaboration failed  
  
D:\Dropbox\Public\web\co\code>iverilog correct2.v -o correct2  
  
D:\Dropbox\Public\web\co\code>  
  
module not1(input i1, output o1);  
    always @ (i1) begin  
        o1 = !i1;  
    end  
endmodule
```



```
module not1(input i1, output reg o1);  
    always @ (i1) begin  
        o1 = !i1;  
    end  
endmodule
```

圖、在 always 裏才能指定 reg 型態的變數

- 上述的參數 output reg o1 其實是一種將暫存器與線路同時宣告的縮寫，事實上我們可以將該宣告拆開成相同名稱的兩部份 (output o1, reg o1) 或者甚至乾脆使用不同的名稱來宣告暫存器，然後再透過 assign 將線路與暫存器綁在一起，這三種寫法的意義其實都是相同的，請參考下圖：

```
module not1(input i1, output reg o1);
  always @(i1) begin
    o1 = !i1;
  end
endmodule
```

```
module not1(input i1, output o1);
  reg o1;
  always @(i1) begin
    o1 = !i1;
  end
endmodule
```

```
module not1(input i1, output o1);
  reg r;
  always @(i1) begin
    r = !i1;
  end
  assign o1=r;
endmodule
```

圖、同時具備 wire + reg 的宣告方式

4. assign 裏面除了指定敘述 = 與基本運算 (&|!^...) 之外，還可以用 (cond)?case1:case2; 的這種語法。舉例而言，以下是微控制器 mcu0 裏控制單元的範例，您可以看到類似層次性 if else 的語法也可用這種方式在 aluop 這個語句上實現。

```
module control(input [3:0] op, input z, output mw, aw, pcmux, sww,  
output [3:0] aluop);  
  
    assign mw=(op==mcu0.ST);  
    assign aw=(op==mcu0.LD || op==mcu0.ADD);  
    assign sww=(op==mcu0.CMP);  
    assign pcmux=(op==mcu0.JMP || (op==mcu0.JEQ && z));  
    assign aluop=(op==mcu0.LD)?alu0.APASS:  
                (op==mcu0.CMP)?alu0.CMP:  
                (op==mcu0.ADD)?alu0.ADD:alu0.ZERO;  
endmodule
```

流程式設計

所謂 RTL 是 Register Transfer Language 的縮寫，也就是暫存器轉換語言，這種寫法與 C, Java 等高階語言非常相似，因此讓「程式人」也有機會透過 Verilog 設計自己的硬體。

舉例而言，在數位邏輯當中，多工器是一個很有用的電路，假如我們想設計一個二選一的多工器，那麼我們可以很直覺得用以下的 RTL 寫法，去完成這樣的電路設計。

```
module mux(f, a, b, sel);
output f;
input a, b, sel;
reg f; // reg 型態會記住某些值，直到被某個 assign 指定改變為止

always @(a or b or sel) // 當任何變數改變的時候，會執行內部區塊
  if (sel) f = a; // Always 內部的區塊採用 imperative 程式語言的寫法
  .
  else f = b;
endmodule
```

對於上述程式，您還可以進一步的將參數部分化簡，將型態寫入到參數中，成為以下的形式：

```
module mux(output reg f, input a, b, sel);
always @(a or b or sel) // 當任何變數改變的時候，會執行內部區塊
  if (sel) f = a; // Always 內部的區塊採用 imperative 程式語言的寫法
  .
  else f = b;
```

```
endmodule
```

在 verilog 當中，if, case 等陳述一定要放在 always 或 initial 的裡面，always @(cond) 代表在 cond 的條件之下要執行該區塊，例如上述的 always @(a or b or sel) 則是在 a, b, 或 sel 有改變的時後，就必須執行裡面的動作。

有時我們只希望在波型的「正邊緣」或「負邊緣」時，才執行某些動作，這時候就可以用 posedge 或 negedge 這兩個修飾詞，例如以下的程式：

```
always @ (posedge clock) begin // 當 clock 時脈在正邊緣時才執行  
    f = a;  
end
```

而 initial 則通常是在測試程式 test bench 當中使用的，在一開始初始化的時後，可以透過 initial 設定初值，例如以下的程式：

```
initial begin  
    clock = 0  
end
```

Verilog 程式的許多地方，都可以用 #delay 指定時間延遲，例如 #50 就是延遲 50 單位的時間 (通常一

單位時間是一奈秒 ns)。舉例而言，假如我們想要每個 50 奈秒讓 clock 變化一次，那麼我們就可以用下列寫法達到目的：

```
always #50 begin  
    clock = ~clock; // 將 clock 反相 (0 變 1 、1 變 0)  
end
```

以上的延遲也可以寫在裡面，而不是直接寫在 always 後面，例如改用以下寫法，也能得到相同的結果。

```
always begin  
    #50;  
    clock = ~clock; // 將 clock 反相 (0 變 1 、1 變 0)  
end
```

範例 3：計數器的 RTL 設計

接著、讓我們用一個整合的計數器範例，來示範這些語法的實際用途，以下是我們的程式內容。

檔案：counter.v

```
// 定義計數器模組 counter，包含重置 reset，時脈 clock 與暫存器 count
module counter(input reset, clock, output reg [7:0] count);
    always @(reset) // 當 reset 有任何改變時
        if (reset) count = 0; // 如果 reset 是 1，就將 count 重置為 0
    always @ (posedge clock) begin // 在 clock 時脈的正邊緣時
        count = count + 1; // 將 count 加 1
    end
endmodule

module main; // 測試主程式開始
wire [7:0] i; // i: 計數器的輸出值
reg reset, clock; // reset: 重置訊號, clock: 時脈

// 宣告一個 counter 模組 c0、計數器的值透過線路 i 輸出，以便觀察。
counter c0(reset, clock, i);
```

```
initial begin
    $display("%4dns: reset=%d clock=%d i=%d", $stime, reset, clock, i)
; // 印出 0ns: reset=x clock=x i= x
#10 reset = 1; clock=0; // 10ns 之後，將啟動重置訊號，並將 clock 初
值設為 0
    $display("%4dns: reset=%d clock=%d i=%d", $stime, reset, clock, i)
; // 印出 10ns: reset=1 clock=0 i= x
#10 reset = 0; // 又經過 10ns 之後，重置完畢，將 reset 歸零
    $display("%4dns: reset=%d clock=%d i=%d", $stime, reset, clock, i)
; // 印出 20ns: reset=0 clock=0 i= 0
#500 $finish; // 再經過 500ns 之後，結束程式
end
```

```
always #40 begin // 延遲 40ns 之後，開始作下列動作
    clock=~clock; // 將時脈反轉 (0 變 1 、 1 變 0)
#10; // 再延遲 10ns 之後
    $display("%4dns: reset=%d clock=%d i=%d", $stime, reset, clock, i)
; // 印出 reset, clock 與 i 等變數值
```

```
end
```

```
endmodule
```

在上述程式中，\$display() 函數可以用來顯示變數的內容，其作用就像 C 語言的 printf() 一樣。不過、由於 Verilog 設計的是硬體，因此像 \$display() 這樣前面有錢字 \$ 符號的指令，其實是不會被合成為電路的，只是方便除錯時使用而已。

以下是我們用 icarus 軟體編譯並執行上述程式的過程與輸出結果：

```
D:\Dropbox\Public\pmag\201307\code>iverilog -o counter counter.v
```

```
D:\Dropbox\Public\pmag\201307\code>vvp counter
```

```
0ns: reset=x clock=x i= x
```

```
10ns: reset=1 clock=0 i= x
```

```
20ns: reset=0 clock=0 i= 0
```

```
50ns: reset=0 clock=1 i= 1
```

```
100ns: reset=0 clock=0 i= 1
```

```
150ns: reset=0 clock=1 i= 2
```

```
200ns: reset=0 clock=0 i= 2
250ns: reset=0 clock=1 i= 3
300ns: reset=0 clock=0 i= 3
350ns: reset=0 clock=1 i= 4
400ns: reset=0 clock=0 i= 4
450ns: reset=0 clock=1 i= 5
500ns: reset=0 clock=0 i= 5
```

您可以看到，在一開始的時候以下的 initial 區塊會被執行，但由於此時 reset, clock, i 都尚未被賦值，所以第一個 \$display() 印出了代表未定值的 x 符號。

```
initial begin
    $display("%dns: reset=%d clock=%d i=%d", $stime, reset, clock, i)
; // 印出 0ns: reset=x clock=x i= x
#10 reset = 1; clock=0; // 10ns 之後，將啟動重置訊號，並將 clock 初
值設為 0
    $display("%dns: reset=%d clock=%d i=%d", $stime, reset, clock, i)
; // 印出 10ns: reset=1 clock=0 i= x
#10 reset = 0; // 又經過 10ns 之後，重置完畢，將 reset 歸零
```

```

$display("%4dns: reset=%d clock=%d i=%d", $stime, reset, clock, i)
; // 印出 20ns: reset=0 clock=0 i= 0
#500 $finish; // 再經過 500ns 之後，結束程式
end

```

接著 #10 reset = 1; clock=0 指令在延遲 10ns 後，執行 reset=1; clock=0，於是後來的 \$display() 就印出了 10ns: reset=1 clock=0 i= x 的結果。

但是就在 reset 被設為 1 的時候，由於 reset 的值有所改變，因此下列模組中的 always @(reset) 被觸發了，於是開始執行 if (reset) count = 0 這個陳述，將 count 暫存器設定為 0。

```
module counter(input reset, clock, output reg [7:0] count);
    always @(reset)                      // 當 reset 有任何改變時
        if (reset) count = 0;            // 如果 reset 是 1，就將 coun
t 重置為 0
    always @ (posedge clock) begin      // 在 clock 時脈的正邊緣時
        count = count + 1;             // 將 count 加 1
    end
endmodule
```

然後 #10 reset = 0 指令又在延遲 10ns 後執行了 reset = 0，之後再用 \$display() 時，由於 count 已經被設定為 0，所以此時印出的結果為 20ns: reset=0 clock=0 i= 0。

initial 區塊的最後一個陳述，#500 \$finish，會在 520ns 的時候才執行，執行時 \$finish 會將整個測試程式結束。

但在程式結束之前，以下的程式會在延遲 40ns 之後，開始將 clock 反相，然後再等待 10ns 之後用 \$display() 印出變數內容，因此整個區塊每 50ns (=40ns+10ns) 會被執行一次。

```
always #40 begin // 延遲 40ns 之後，開始作下列動作
    clock=~clock; // 將時脈反轉 (0 變 1 、1 變 0)
    #10;           // 再延遲 10ns 之後
    $display("%4dns: reset=%d clock=%d i=%d", $stime, reset, clock, i)
; // 印出 reset, clock 與 i 等變數值
end
```

所以、您才會看到像下面的輸出結果，如果仔細觀察，會發現 clock 每 50ns 變換一次，符合上述的程式邏輯，而且每當 clock 從 0 變成 1 的正邊緣，就會觸發 counter 模組，讓 count 變數加 1，並且透過線路 i 的輸出被我們觀察到。

```
50ns: reset=0 clock=1 i= 1
```

```
100ns: reset=0 clock=0 i= 1
150ns: reset=0 clock=1 i= 2
200ns: reset=0 clock=0 i= 2
250ns: reset=0 clock=1 i= 3
300ns: reset=0 clock=0 i= 3
```

(註：或許您有注意到上期當中我們用 \$monitor() 來觀察全加器的輸出，\$display() 與 \$monitor() 的語法幾乎一模一樣，但是 \$display() 是顯示該時間點的變數內容，而 \$monitor() 則會在受觀察的變數有改變時就列印變數內容，兩者的的功能有明顯的差異)。

阻塞 vs. 非阻塞 (Blocking vs. Nonblocking)

您可能會注意到在 Verilog 當中有兩種指定方式，一種用 = 表示，另一種用 <= 表示，這兩種指定方法看來很類似，但意義上卻有很細緻的差異，一般 Verilog 初學者往往分不清楚，因而造成很多程式上的錯誤。

基本上 = 指令是阻塞式的 (Blocking)，因此程式會按照「循序」的方式，一個指令接著一個指令執行，就像 C 語言裏的 $a=b$, $b=c$ 這樣， $b=c$ 會在 $a=b$ 執行完之後才執行，以下是一個範例。

但是 <= 指令卻是非阻塞式的 (Nonblocking)，所以程式會採用「平行」的方式執行。舉例而言，像是 $a<=b$, $b<=c$ 會同時執行兩者，所以 a 會取得上一輪的 b 值，而 b 則會取得上一輪的 c 值。

Blocking 的語法 (=) 通常用在「組合電路」上，也就是 always @(*) 語句裏面，而 Nonblocking 的語法 (<=) 通常用在採用邊緣觸發的「循序電路」上，也就是 always @(posedge clock) 的語句裏面。

且讓我們用幾個範例來說明 blocking = 與 Nonblocking <= 的差別。

範例 1：

阻塞式 (Blocking =)

```
always @ (a or b or c) begin  
    a=0;  
    b=a;  
    c=b;  
end
```

結果： a=b=c=0;

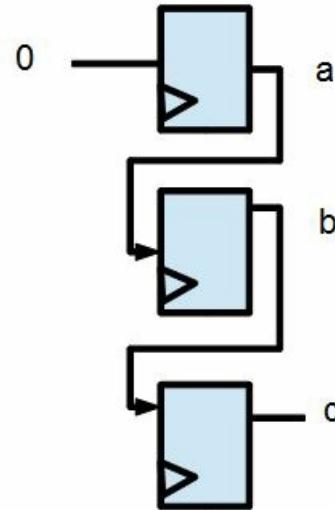
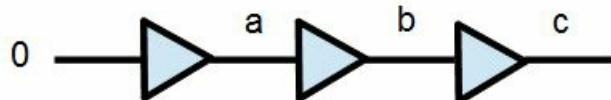
非阻塞式 (Nonblocking <=)

```
always @(posedge clock or reset)  
begin  
    a<=0;  
    b<=a;  
    c<=b;  
end
```

結果： a=0; b=上一輪的 a 值； c=上一輪的 b 值

合成電路：

合成電路：



注意：通常 blocking assignment = 會用在 always @(*) 語句裏面。

注意：通常 nonblocking assignment <= 會用在 always @(posedge clock or reset) 語句裏

結語

有些人說在設計 Verilog 程式的時候，必須先心中有電路，才能夠設計的出來。

但是、從我這樣一個「程式人」的角度看來，並非如此，採用流程式的寫法也可以設計得出 Verilog

程式，不需一定要有電路圖。

當然、採用「流程式」寫法的話，如果是用 blocking 的 = 方式，那麼可能會造成很長的鏈狀結構，這或許會讓電路效能變差。

但是由於「流程式」寫法簡單又清楚，因此程式碼往往比「區塊式寫法」短，而且更容易懂，這是流程式寫法的好處。

當然、如果兩種寫法都會，那是最好的了，我們將在後續的章節當中陸續用完整的案例示範如何用這兩種寫法分別撰寫「開放電腦計畫」中的處理器，以便讓讀者能深入體會兩種寫法的好處與缺點。

參考文獻

- (筆記) dispaly()、strobe()、monitor() 、fwrite()與blocking / nonblocking的關係

組合邏輯 (Combinatorial Logic)

簡介

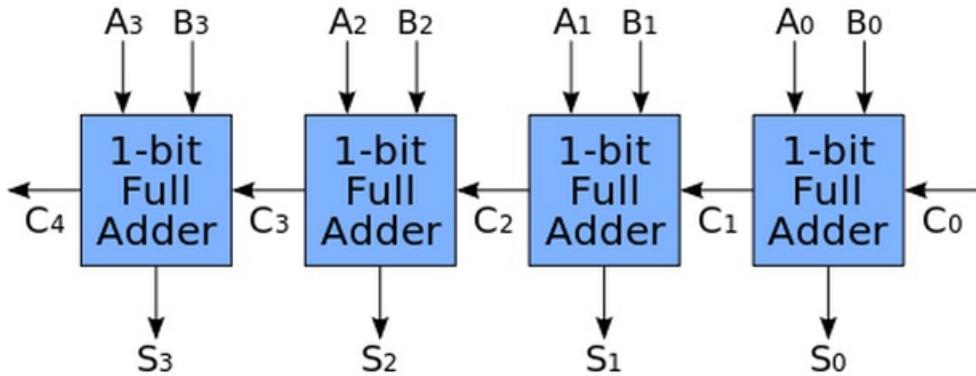
在數位電路當中，邏輯電路通常被分為兩類，一類是沒有「回饋線路」(No feedback) 的組合邏輯電路 (Combinatorial Logic)，另一類是有「回饋線路」的循序邏輯電路 (Sequential Logic)。

組合邏輯的線路只是將輸入訊號轉換成輸出訊號，像是加法器、多工器等都是組合邏輯電路的範例，由於中間不會暫存，因此無法記憶位元。而循序邏輯由於有回饋線路，所以可以製作出像 Flip-Flop，Latch 等記憶單元，可以記憶位元。

在本文中，我們將先專注在組合邏輯上，看看如何用基本的閘級寫法，寫出像多工器、加法器、減法器等組成 CPU 的基礎 電路元件。

加法器

接著、讓我們用先前已經示範過的全加器範例，一個一個連接成四位元的加法器，電路圖如下所示



圖、用 4 個全加器組成 4 位元加法器

上圖寫成 Verilog 就變成以下 adder4 模組的程式內容。

```
module adder4(input signed [3:0] a, input signed [3:0] b, input c_in,
  output signed [3:0] sum, output c_out);
  wire [3:0] c;

  fulladder fa1(a[0], b[0], c_in, sum[0], c[1]) ;
  fulladder fa2(a[1], b[1], c[1], sum[1], c[2]) ;
  fulladder fa3(a[2], b[2], c[2], sum[2], c[3]) ;
  fulladder fa4(a[3], b[3], c[3], sum[3], c_out) ;
endmodule
```

```
fulladder fa4(a[3], b[3], c[3], sum[3], c_out) ;  
  
endmodule
```

以下是完整的 4 位元加法器之 Verilog 程式。

檔案：[adder4.v](#)

```
module fulladder (input a, b, c_in, output sum, c_out);  
wire s1, c1, c2;  
  
xor g1(s1, a, b);  
xor g2(sum, s1, c_in);  
and g3(c1, a, b);  
and g4(c2, s1, c_in) ;  
xor g5(c_out, c2, c1) ;  
  
endmodule
```

```
module adder4(input signed [3:0] a, input signed [3:0] b, input c_in, output signed [3:0] sum, output c_out);
wire [3:0] c;

fulladder fa1(a[0], b[0], c_in, sum[0], c[1]) ;
fulladder fa2(a[1], b[1], c[1], sum[1], c[2]) ;
fulladder fa3(a[2], b[2], c[2], sum[2], c[3]) ;
fulladder fa4(a[3], b[3], c[3], sum[3], c_out) ;

endmodule
```

```
module main;
reg signed [3:0] a;
reg signed [3:0] b;
wire signed [3:0] sum;
wire c_out;

adder4 DUT (a, b, 1'b0, sum, c_out);
```

```
initial
begin
    a = 4'b0101;
    b = 4'b0000;
end

always #50 begin
    b=b+1;
    $monitor("%dns monitor: a=%d b=%d sum=%d", $stime, a, b, sum);
end

initial #2000 $finish;

endmodule
```

執行結果

```
D:\ccc101\icarus\ccc>iverilog -o sadd4 sadd4.v
```

```
D:\ccc101\icarus\ccc>vvp saddr
```

```
50ns monitor: a= 5 b= 1 sum= 6
100ns monitor: a= 5 b= 2 sum= 7
150ns monitor: a= 5 b= 3 sum=-8
200ns monitor: a= 5 b= 4 sum=-7
250ns monitor: a= 5 b= 5 sum=-6
300ns monitor: a= 5 b= 6 sum=-5
350ns monitor: a= 5 b= 7 sum=-4
400ns monitor: a= 5 b=-8 sum=-3
450ns monitor: a= 5 b=-7 sum=-2
500ns monitor: a= 5 b=-6 sum=-1
550ns monitor: a= 5 b=-5 sum= 0
600ns monitor: a= 5 b=-4 sum= 1
650ns monitor: a= 5 b=-3 sum= 2
700ns monitor: a= 5 b=-2 sum= 3
750ns monitor: a= 5 b=-1 sum= 4
800ns monitor: a= 5 b= 0 sum= 5
```

```
850ns monitor: a= 5 b= 1 sum= 6
900ns monitor: a= 5 b= 2 sum= 7
950ns monitor: a= 5 b= 3 sum=-8
1000ns monitor: a= 5 b= 4 sum=-7
1050ns monitor: a= 5 b= 5 sum=-6
1100ns monitor: a= 5 b= 6 sum=-5
1150ns monitor: a= 5 b= 7 sum=-4
1200ns monitor: a= 5 b=-8 sum=-3
1250ns monitor: a= 5 b=-7 sum=-2
1300ns monitor: a= 5 b=-6 sum=-1
1350ns monitor: a= 5 b=-5 sum= 0
1400ns monitor: a= 5 b=-4 sum= 1
1450ns monitor: a= 5 b=-3 sum= 2
1500ns monitor: a= 5 b=-2 sum= 3
1550ns monitor: a= 5 b=-1 sum= 4
1600ns monitor: a= 5 b= 0 sum= 5
1650ns monitor: a= 5 b= 1 sum= 6
```

```
1700ns monitor: a= 5 b= 2 sum= 7
1750ns monitor: a= 5 b= 3 sum=-8
1800ns monitor: a= 5 b= 4 sum=-7
1850ns monitor: a= 5 b= 5 sum=-6
1900ns monitor: a= 5 b= 6 sum=-5
1950ns monitor: a= 5 b= 7 sum=-4
2000ns monitor: a= 5 b=-8 sum=-3
```

在上述執行結果中，您可以看到在沒有溢位的情況下， $sum = a+b$ ，但是一旦加總值超過 7 之後，那就會變成負值，這也正是有號二補數表示法 溢位時會產生的結果。

32 位元加法器

當然、上述的四位元加法器的範圍，只能從 -8 到 +7，這個範圍實在太小了，並不具備任何實用性，但是萬事起頭難，只要您能夠做出四位元加法器，那麼就可以利用這個元件進行串接，將 8 個四位元加法器串接起來，立刻得到了一組 32 位元的加法器，以下是 這個 32 位元加法器的模組定義。

```
module adder32(input signed [31:0] a, input signed [31:0] b, input
c_in, output signed [31:0] sum, output c_out);
```

```

wire [7:0] c;

adder4 a0(a[3:0] , b[3:0],    c_in,  sum[3:0],   c[0]) ;
adder4 a1(a[7:4] , b[7:4],    c[0],   sum[7:4],   c[1]) ;
adder4 a2(a[11:8] , b[11:8],  c[1],   sum[11:8],  c[2]) ;
adder4 a3(a[15:12], b[15:12], c[2],   sum[15:12], c[3]) ;
adder4 a4(a[19:16], b[19:16], c[3],   sum[19:16], c[4]) ;
adder4 a5(a[23:20], b[23:20], c[4],   sum[23:20], c[5]) ;
adder4 a6(a[27:24], b[27:24], c[5],   sum[27:24], c[6]) ;
adder4 a7(a[31:28], b[31:28], c[6],   sum[31:28], c_out);

endmodule

```

有了這個模組之後，您就可以寫出下列完整的程式，以測試驗證該 32 位元加法器是否正確了。

```

module fulladder (input a, b, c_in, output sum, c_out);
wire s1, c1, c2;

```

```
xor g1(s1, a, b) ;  
xor g2(sum, s1, c_in) ;  
and g3(c1, a, b) ;  
and g4(c2, s1, c_in) ;  
xor g5(c_out, c2, c1) ;
```

```
endmodule
```

```
module adder4(input signed [3:0] a, input signed [3:0] b, input c_i  
n, output signed [3:0] sum, output c_out);  
wire [3:0] c;
```

```
fulladder fa1(a[0], b[0], c_in, sum[0], c[1]) ;  
fulladder fa2(a[1], b[1], c[1], sum[1], c[2]) ;  
fulladder fa3(a[2], b[2], c[2], sum[2], c[3]) ;  
fulladder fa4(a[3], b[3], c[3], sum[3], c_out) ;
```

```
endmodule
```

```
module adder32(input signed [31:0] a, input signed [31:0] b, input  
c_in, output signed [31:0] sum, output c_out);  
wire [7:0] c;  
  
adder4 a0(a[3:0] , b[3:0], c_in, sum[3:0], c[0]) ;  
adder4 a1(a[7:4] , b[7:4], c[0], sum[7:4], c[1]) ;  
adder4 a2(a[11:8] , b[11:8], c[1], sum[11:8], c[2]) ;  
adder4 a3(a[15:12], b[15:12], c[2], sum[15:12], c[3]) ;  
adder4 a4(a[19:16], b[19:16], c[3], sum[19:16], c[4]) ;  
adder4 a5(a[23:20], b[23:20], c[4], sum[23:20], c[5]) ;  
adder4 a6(a[27:24], b[27:24], c[5], sum[27:24], c[6]) ;  
adder4 a7(a[31:28], b[31:28], c[6], sum[31:28], c_out);  
  
endmodule
```

```
module main;
```

```
reg signed [31:0] a;
reg signed [31:0] b;
wire signed [31:0] sum;
wire c_out;

adder32 DUT (a, b, 0, sum, c_out);

initial
begin
    a = 60000000;
    b = 3789621;
    $monitor ("%dns monitor: a=%d b=%d sum=%d", $stime, a, b, sum);
end

always #50 begin
    b=b-1000000;
end
```

```
initial #500 $finish;  
  
endmodule
```

然後、我們就可以用 icarus 進行測試，以下是測試結果：

```
D:\Dropbox\Public\web\oc\code>iverilog -o adder32 adder32.v
```

```
D:\Dropbox\Public\web\oc\code>vvp adder32
```

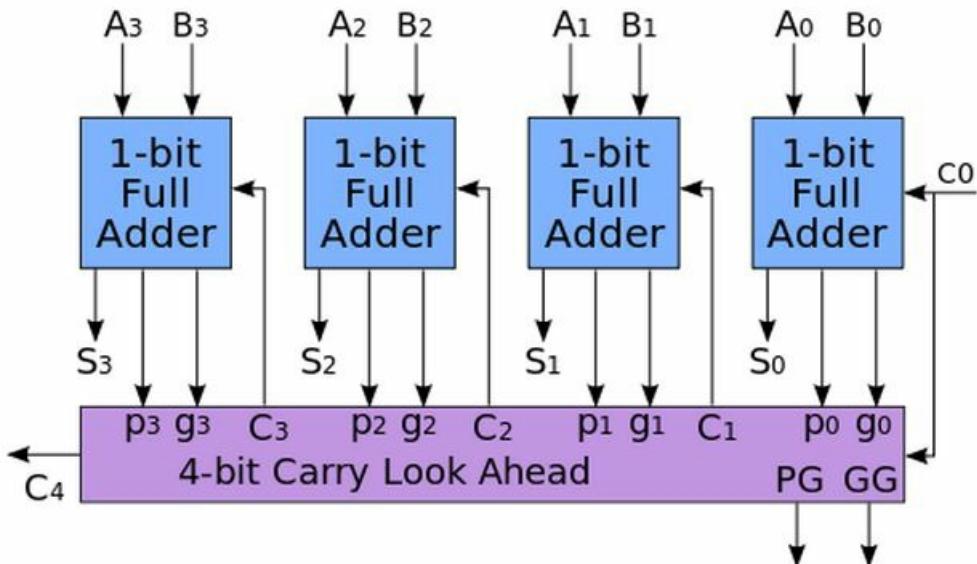
0ns monitor:	a= 60000000	b= 3789621	sum= 63789621
50ns monitor:	a= 60000000	b= 2789621	sum= 62789621
100ns monitor:	a= 60000000	b= 1789621	sum= 61789621
150ns monitor:	a= 60000000	b= 789621	sum= 60789621
200ns monitor:	a= 60000000	b= -210379	sum= 59789621
250ns monitor:	a= 60000000	b= -1210379	sum= 58789621
300ns monitor:	a= 60000000	b= -2210379	sum= 57789621
350ns monitor:	a= 60000000	b= -3210379	sum= 56789621
400ns monitor:	a= 60000000	b= -4210379	sum= 55789621

450ns monitor: a= 60000000 b= -5210379 sum= 54789621
--

500ns monitor: a= 60000000 b= -6210379 sum= 53789621
--

您可以看到 sum 的確是 a+b 的結果，因此這個 32 位元加法器的初步驗證是正確的。

前瞻進位加法器 (Carry Lookahead Adder)



圖、4 位元前瞻進位加法器

圖片來源：http://en.wikipedia.org/wiki/File:4-bit_carry_lookahead_adder.svg

檔案：cladder4.v

```
module cladder4(output [3:0] S, output Cout, PG, GG, input [3:0] A, B,  
input Cin);  
wire [3:0] G, P, C;  
  
assign G = A & B; //Generate  
assign P = A ^ B; //Propagate  
assign C[0] = Cin;  
assign C[1] = G[0] | (P[0]&C[0]);  
assign C[2] = G[1] | (P[1]&G[0]) | (P[1]&P[0]&C[0]);  
assign C[3] = G[2] | (P[2]&G[1]) | (P[2]&P[1]&G[0]) | (P[2]&P[1]&  
P[0]&C[0]);  
assign Cout = G[3] | (P[3]&G[2]) | (P[3]&P[2]&G[1]) | (P[3]&P[2]&  
P[1]&G[0]) | (P[3]&P[2]&P[1]&P[0]&C[0]);  
assign S = P ^ C;
```

```
assign PG = P[3] & P[2] & P[1] & P[0];
assign GG = G[3] | (P[3]&G[2]) | (P[3]&P[2]&G[1]) | (P[3]&P[2]&P[1]&G[0]);
endmodule
```

```
module main;
reg signed [3:0] a;
reg signed [3:0] b;
wire signed [3:0] sum;
wire c_out;

cladder4 DUT (sum, cout, pg, gg, a, b, 0);

initial
begin
  a = 5;
  b = -3;
```

```
$monitor ("%dns monitor: a=%d b=%d sum=%d", $stime, a, b, sum);  
end  
  
endmodule
```

執行結果

```
D:\Dropbox\Public\web\oc\code>iverilog -o cladder4 cladder4.v  
  
D:\Dropbox\Public\web\oc\code>vvp cladder4  
0ns monitor: a= 5 b=-3 sum= 2
```

結語

在本文中，我們大致將 CPU 設計當中最重要組合邏輯電路，也就是「多工器、加法器、減法器」的設計原理說明完畢了，希望透過 Verilog 的實作方式，能讓讀者更瞭解數位電路的設計原理，並且為接下來所要介紹的「處理器設計」進行鋪路的工作。

參考文獻

- LSU EE 3755 -- Spring 2002 -- Computer Organization : Verilog Notes 7 -- Integer Multiply and Divide
- 陳鍾誠的網站 : Verilog 電路設計 -- 多工器
- 陳鍾誠的網站 : Verilog 電路設計 -- 4 位元加法器
- 陳鍾誠的網站 : Verilog 電路設計 -- 加減器
- Wikipedia:Adder
- Wikipedia:Adder–subtractor
- Wikipedia:Multiplexer

【本文由陳鍾誠取材 (主要為圖片) 並修改自維基百科】

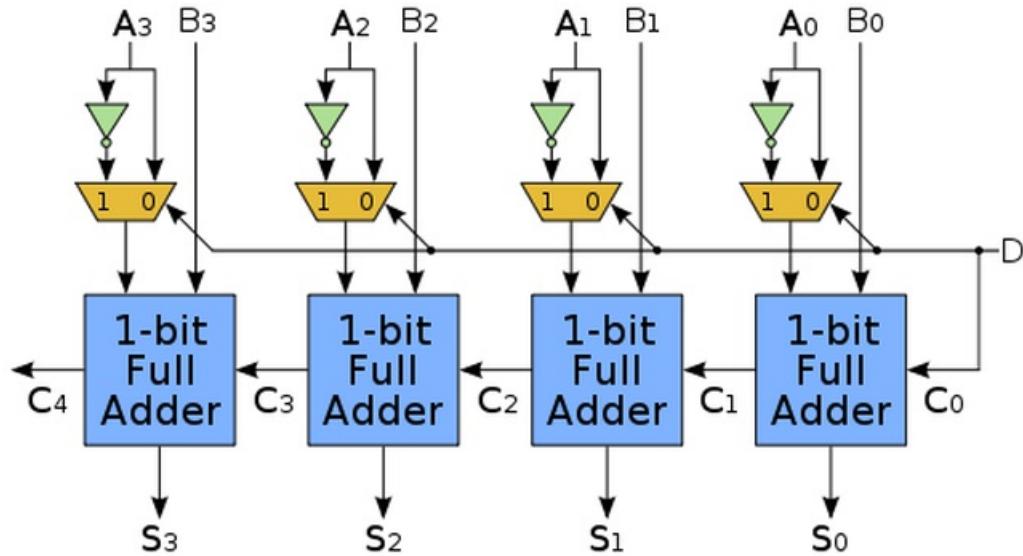
算術邏輯單元 ALU 的設計

在上一章中，我們探討了「組合邏輯電路」的設計方式，採用閘級的拉線方式設計了「多工器」與「加法器」等元件，在本章當中，我們將從加法器再度往上，探討如何設計一個 ALU 單元。

加減器

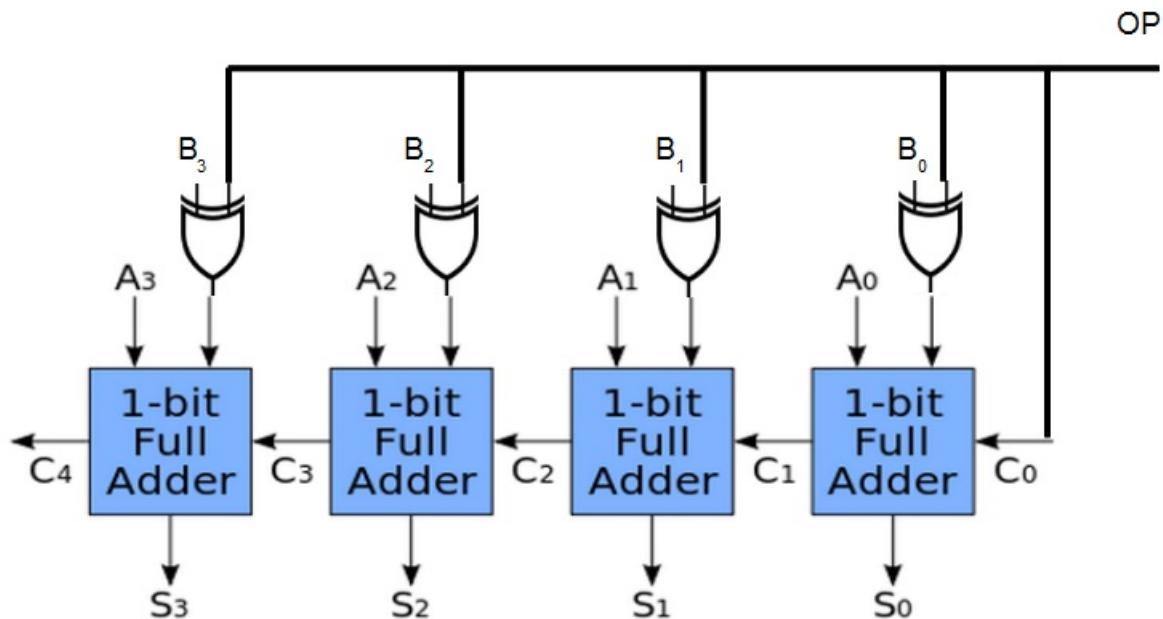
我們只要把加法器，加上一組控制的互斥或閘，並控制輸入進位與否，就可以成為加減器了，這是因為我們採用了二補數的關係。

二補數讓我們可以很容易的延伸加法器電路就能做出減法器。我們可以在運算元 B 之前加上 2 選 1 多工器或 XOR 閘來控制 B 是否應該取補數，並且運用 OP 控制線路來進行控制，以下是採用 2 選 1 多工器的電路做法圖。



圖、採用 2 選 1 多工器控制的加減器電路

另一種更簡單的做法是採用 XOR 閘去控制 B 是否要取補數，如下圖所示：



圖、採用 XOR 控制的加減器電路

清楚了電路圖的布局之後，讓我們來看看如何用 Verilog 實做加減器吧！關鍵部分的程式如下所示，這個模組就對應到上述的「採用 XOR 控制的加減器電路」之圖形。

```
module addSub4(input op, input signed [3:0] a, input signed [3:0] b
```

```
,  
          output signed [3:0] sum, output c_out);  
  
wire [3:0] bop;  
  
xor4 x1(b, {op, op, op, op}, bop);  
adder4 a1(a, bop, op, sum, c_out);  
  
endmodule
```

接著讓我們來看看完整的加減器程式與測試結果。

檔案：[addsub4.v](#)

```
module fulladder (input a, b, c_in, output sum, c_out);  
wire s1, c1, c2;  
  
xor g1(s1, a, b);  
xor g2(sum, s1, c_in);
```

```
and g3(c1, a, b) ;
and g4(c2, s1, c_in) ;
xor g5(c_out, c2, c1) ;

endmodule

module adder4(input signed [3:0] a, input signed [3:0] b, input c_i
n,
                  output signed [3:0] sum, output c_out) ;
wire [3:0] c;

fulladder fa1(a[0], b[0], c_in, sum[0], c[1]) ;
fulladder fa2(a[1], b[1], c[1], sum[1], c[2]) ;
fulladder fa3(a[2], b[2], c[2], sum[2], c[3]) ;
fulladder fa4(a[3], b[3], c[3], sum[3], c_out) ;

endmodule
```

```
module xor4(input [3:0] a, input [3:0] b, output [3:0] y);
    assign y = a ^ b;
endmodule

module addSub4(input op, input signed [3:0] a, input signed [3:0] b
,
    output signed [3:0] sum, output c_out);
    wire [3:0] bop;

    xor4 x1(b, {op, op, op, op}, bop);
    adder4 a1(a, bop, op, sum, c_out);

endmodule

module main;
    reg signed [3:0] a;
    reg signed [3:0] b;
```

```
wire signed [3:0] sum;
reg op;
wire c_out;

addSub4 DUT (op, a, b, sum, c_out);

initial
begin
    a = 4'b0101;
    b = 4'b0000;
    op = 1'b0;
end

always #50 begin
    op=op+1;
    $monitor("%dns monitor: op=%d a=%d b=%d sum=%d", $stime, op, a, b,
    sum);
end
```

```
always #100 begin
    b=b+1;
end

initial #2000 $finish;

endmodule
```

執行結果：

```
D:\ccc101\icarus\ccc>iverilog -o addSub4 addSub4.v
```

```
D:\ccc101\icarus\ccc>vp addSub4
      50ns monitor: op=1 a= 5 b= 0 sum= 5
     100ns monitor: op=0 a= 5 b= 1 sum= 6
     150ns monitor: op=1 a= 5 b= 1 sum= 4
     200ns monitor: op=0 a= 5 b= 2 sum= 7
```

```
250ns monitor: op=1 a= 5 b= 2 sum= 3
300ns monitor: op=0 a= 5 b= 3 sum=-8
350ns monitor: op=1 a= 5 b= 3 sum= 2
400ns monitor: op=0 a= 5 b= 4 sum=-7
450ns monitor: op=1 a= 5 b= 4 sum= 1
500ns monitor: op=0 a= 5 b= 5 sum=-6
550ns monitor: op=1 a= 5 b= 5 sum= 0
600ns monitor: op=0 a= 5 b= 6 sum=-5
650ns monitor: op=1 a= 5 b= 6 sum=-1
700ns monitor: op=0 a= 5 b= 7 sum=-4
750ns monitor: op=1 a= 5 b= 7 sum=-2
800ns monitor: op=0 a= 5 b=-8 sum=-3
850ns monitor: op=1 a= 5 b=-8 sum=-3
900ns monitor: op=0 a= 5 b=-7 sum=-2
950ns monitor: op=1 a= 5 b=-7 sum=-4
1000ns monitor: op=0 a= 5 b=-6 sum=-1
1050ns monitor: op=1 a= 5 b=-6 sum=-5
1100ns monitor: op=0 a= 5 b=-5 sum= 0
```

1150ns monitor: op=1 a= 5 b=-5 sum=-6
1200ns monitor: op=0 a= 5 b=-4 sum= 1
1250ns monitor: op=1 a= 5 b=-4 sum=-7
1300ns monitor: op=0 a= 5 b=-3 sum= 2
1350ns monitor: op=1 a= 5 b=-3 sum=-8
1400ns monitor: op=0 a= 5 b=-2 sum= 3
1450ns monitor: op=1 a= 5 b=-2 sum= 7
1500ns monitor: op=0 a= 5 b=-1 sum= 4
1550ns monitor: op=1 a= 5 b=-1 sum= 6
1600ns monitor: op=0 a= 5 b= 0 sum= 5
1650ns monitor: op=1 a= 5 b= 0 sum= 5
1700ns monitor: op=0 a= 5 b= 1 sum= 6
1750ns monitor: op=1 a= 5 b= 1 sum= 4
1800ns monitor: op=0 a= 5 b= 2 sum= 7
1850ns monitor: op=1 a= 5 b= 2 sum= 3
1900ns monitor: op=0 a= 5 b= 3 sum=-8
1950ns monitor: op=1 a= 5 b= 3 sum= 2
2000ns monitor: op=0 a= 5 b= 4 sum=-7

在上述結果中，您可以看到當 op=0 時，電路所作的是加法運算，例如：200ns monitor: op=0 a= 5 b= 2 sum= 7。而當 op=1 時，電路所做的是減法運算，例如：250ns monitor: op=1 a= 5 b= 2 sum= 3。

採用 CASE 語法設計 ALU 模組

其實，在 Verilog 當中，我們並不需要自行設計加法器，因為 Verilog 提供了高階的「+,-,*,/」等基本運算，可以讓我們直接使用，更方便的是，只要搭配 case 語句，我們就可以很輕易的設計出一個 ALU 單元了。

以下是一個簡易的 ALU 單元之程式碼，

```
// 輸入 a, b 後會執行 op 所指定的運算，然後將結果放在暫存器 y 當中
module alu(input [7:0] a, input [7:0] b, input [2:0] op, output reg [7:0] y);
always@(a or b or op) begin // 當 a, b 或 op 有改變時，就進入此區塊執行。
    case(op)                  // 根據 op 決定要執行何種運算
        3'b000: y = a + b;    // op=000, 執行加法
        3'b001: y = a - b;    // op=001, 執行減法
    endcase
endmodule
```

```
3' b010: y = a * b;      // op=000, 執行乘法
3' b011: y = a / b;      // op=000, 執行除法
3' b100: y = a & b;      // op=000, 執行 AND
3' b101: y = a | b;      // op=000, 執行 OR
3' b110: y = ~a;         // op=000, 執行 NOT
3' b111: y = a ^ b;      // op=000, 執行 XOR
endcase
end
endmodule
```

Verilog 語法的注意事項

上述這種寫法感覺就好像在用高階寫程式一樣，這讓 ALU 的設計變得非常簡單。但是仍然需要注意以下幾點與高階語言不同之處：

注意事項 1. always 語句的用法

case 等陳述句的外面一定要有 always 或 initial 語句，因為這是硬體線路，所以是採用連線 wiring 的方式，always 語句只有在 @(trigger) 中間的 trigger 觸發條件符合時才會被觸發。

當 trigger 中的變數有任何改變的時候，always 語句就會被觸發，像是 always@(a or b or op) 就代表當

(a, b, op) 當中任何一個有改變的時候，該語句就會被觸發。

有時我們可以在 always 語句當中加上 posedge 的條件，指定只有在「正邊緣」(上昇邊緣) 時觸發。或者加上 negedge 的條件，指定只有在「負邊緣」(下降邊緣) 的時候觸發，例如我們可以常常在 Verilog 當中看到下列語句：

```
always @(posedge clock) begin  
    ....  
end
```

上述語句就只有在 clock 這一條線路的電波上昇邊緣會被觸發，如此我們就能更精細的控制觸發的動作，採用正邊緣或負邊緣觸發的方式。

注意事項 2. 指定陳述的左項之限制

在上述程式中，a, b, op 被宣告為 input (輸入線路)，而 y 則宣告為 output reg (輸出暫存器)，在這裏必須注意的是 y 不能只宣告為 output 而不加上 reg，因為只有 reg 型態的變數才能被放在 always 區塊裡的等號左方，進行指定的動作。

事實上，在 Verilog 當中，像 output reg [7:0] y 這樣的宣告，其實也可以用比較繁雜的兩次宣告方式，一次宣告 output，另一次則宣告 reg，如下所示：

```

module alu(input [7:0] a, input [7:0] b, input [2:0] op, output [7:
0] y);
reg y;
always@(a or b or op) begin
....
```

甚至，您也可以將該變數分開為兩個不同名稱，然後再利用 assign 的方式指定，如下所示：

```

// 輸入 a, b 後會執行 op 所指定的運算，然後將結果放在暫存器 y 當中
module alu(input [7:0] a, input [7:0] b, input [2:0] op, output [7:
0] y);
reg ty;
always@(a or b or op) begin // 當 a, b 或 op 有改變時，就進入此區塊執
行。
case(op)                                // 根據 op 決定要執行何種運算
  3'b000: ty = a + b;                  // op=000, 執行加法
  3'b001: ty = a - b;                  // op=000, 執行減法
  3'b010: ty = a * b;                  // op=000, 執行乘法
```

```

3' b011: ty = a / b;      // op=000, 執行除法
3' b100: ty = a & b;      // op=000, 執行 AND
3' b101: ty = a | b;      // op=000, 執行 OR
3' b110: ty = ~a;         // op=000, 執行 NOT
3' b111: ty = a ^ b;      // op=000, 執行 XOR
endcase
$display("base 10 : %dns : op=%d a=%d b=%d y=%d", $stime, op, a, b
, y); // 印出 op, a, b, y 的 10 進位值。
$display("base 2 : %dns : op=%b a=%b b=%b y=%b", $stime, op, a, b
, y); // 印出 op, a, b, y 的 2 進位值。
end
assign y=ty;
endmodule

```

在上述程式中，由於只有 reg 型態的變數可以放在 always 區塊內的等號左邊，因此我們必須用 reg 型態的 ty 去儲存 運算結果。

但是在 assign 指令的等號左邊，則不需要是暫存器型態的變數，也可以是線路型態的變數，因此我們可以用 assign y=ty 這樣一個指令去將 ty 的暫存器內容輸出。

事實上，assign 語句代表的是一種「不需儲存的立即輸出接線」，因此我們才能將 output 型態的變數寫在等號左邊啊！

完整的 ALU 設計（含測試程式）

瞭解了這些 Verilog 語法特性之後，我們就可以搭配測試程式，對這個 ALU 模組進行測試，以下是完整的程式碼：

檔案：[alu.v](#)

```
// 輸入 a, b 後會執行 op 所指定的運算，然後將結果放在暫存器 y 當中
module alu(input [7:0] a, input [7:0] b, input [2:0] op, output reg [7:0] y);
always@(a or b or op) begin // 當 a, b 或 op 有改變時，就進入此區塊執行。
    case(op)                      // 根據 op 決定要執行何種運算
        3'b000: y = a + b;       // op=000, 執行加法
        3'b001: y = a - b;       // op=000, 執行減法
        3'b010: y = a * b;       // op=000, 執行乘法
        3'b011: y = a / b;       // op=000, 執行除法
    endcase
endmodule
```

```

3' b100: y = a & b;           // op=000, 執行 AND
3' b101: y = a | b;           // op=000, 執行 OR
3' b110: y = ~a;              // op=000, 執行 NOT
3' b111: y = a ^ b;           // op=000, 執行 XOR

endcase

$display("base 10 : %dns : op=%d a=%d b=%d y=%d", $stime, op, a, b
, y); // 印出 op, a, b, y 的 10 進位值。
$display("base 2 : %dns : op=%b a=%b b=%b y=%b", $stime, op, a, b
, y); // 印出 op, a, b, y 的 2 進位值。

end

endmodule

```

```

module main;                      // 測試程式開始
reg [7:0] a, b;                  // 宣告 a, b 為 8 位元暫存器
wire [7:0] y;                   // 宣告 y 為 8 位元線路
reg [2:0] op;                    // 宣告 op 為 3 位元暫存器

alu alu1(a, b, op, y);          // 建立一個 alu 單元，名稱為 alu1

```

```
initial begin          // 測試程式的初始化動作
    a = 8' h07;        // 設定 a 為數值 7
    b = 8' h03;        // 設定 b 為數值 3
    op = 3' b000;      // 設定 op 的初始值為 000
end

always #50 begin       // 每個 50 奈秒就作下列動作
    op = op + 1;       // 讓 op 的值加 1
end

initial #1000 $finish; // 時間到 1000 奈秒就結束

endmodule
```

在上述程式中，為了更清楚的印出 ALU 的輸出結果，我們在 ALU 模組的結尾放入以下的兩行 \$display() 指令，以便同時顯示 (op, a, b, y) 等變數的 10 進位與 2 進位結果值，方便讀者觀察。

```
$display("base 10 : %dns : op=%d a=%d b=%d y=%d", $stime, op, a, b  
, y); // 印出 op, a, b, y 的 10 進位值。  
$display("base 2 : %dns : op=%b a=%b b=%b y=%b", $stime, op, a, b  
, y); // 印出 op, a, b, y 的 2 進位值。
```

測試執行結果

上述程式的執行測試結果如下：

```
D:\Dropbox\Public\pmag\201310\code>iverilog -o alu alu.v
```

```
D:\Dropbox\Public\pmag\201310\code>vvp alu
```

```
base 10 :          0ns : op=0 a= 7 b= 3 y= 10
```

```
base 2 :          0ns : op=000 a=00000111 b=00000011 y=00001010
```

```
base 10 :      50ns : op=1 a= 7 b= 3 y= 4
```

```
base 2 :      50ns : op=001 a=00000111 b=00000011 y=00000100
```

```
base 10 : 100ns : op=2 a= 7 b= 3 y= 21
```

```
base 2 : 100ns : op=010 a=00000111 b=00000011 y=00010101
```

base 10 :	150ns : op=3 a= 7 b= 3 y= 2
base 2 :	150ns : op=011 a=00000111 b=00000011 y=00000010
base 10 :	200ns : op=4 a= 7 b= 3 y= 3
base 2 :	200ns : op=100 a=00000111 b=00000011 y=00000011
base 10 :	250ns : op=5 a= 7 b= 3 y= 7
base 2 :	250ns : op=101 a=00000111 b=00000011 y=00000111
base 10 :	300ns : op=6 a= 7 b= 3 y=248
base 2 :	300ns : op=110 a=00000111 b=00000011 y=11111000
base 10 :	350ns : op=7 a= 7 b= 3 y= 4
base 2 :	350ns : op=111 a=00000111 b=00000011 y=00000100
base 10 :	400ns : op=0 a= 7 b= 3 y= 10
base 2 :	400ns : op=000 a=00000111 b=00000011 y=00001010
base 10 :	450ns : op=1 a= 7 b= 3 y= 4
base 2 :	450ns : op=001 a=00000111 b=00000011 y=00000100
base 10 :	500ns : op=2 a= 7 b= 3 y= 21
base 2 :	500ns : op=010 a=00000111 b=00000011 y=00010101
base 10 :	550ns : op=3 a= 7 b= 3 y= 2
base 2 :	550ns : op=011 a=00000111 b=00000011 y=00000010

base 10 :	600ns : op=4 a= 7 b= 3 y= 3
base 2 :	600ns : op=100 a=00000111 b=00000011 y=00000011
base 10 :	650ns : op=5 a= 7 b= 3 y= 7
base 2 :	650ns : op=101 a=00000111 b=00000011 y=00000111
base 10 :	700ns : op=6 a= 7 b= 3 y=248
base 2 :	700ns : op=110 a=00000111 b=00000011 y=11111000
base 10 :	750ns : op=7 a= 7 b= 3 y= 4
base 2 :	750ns : op=111 a=00000111 b=00000011 y=00000100
base 10 :	800ns : op=0 a= 7 b= 3 y= 10
base 2 :	800ns : op=000 a=00000111 b=00000011 y=00001010
base 10 :	850ns : op=1 a= 7 b= 3 y= 4
base 2 :	850ns : op=001 a=00000111 b=00000011 y=00000100
base 10 :	900ns : op=2 a= 7 b= 3 y= 21
base 2 :	900ns : op=010 a=00000111 b=00000011 y=00010101
base 10 :	950ns : op=3 a= 7 b= 3 y= 2
base 2 :	950ns : op=011 a=00000111 b=00000011 y=00000010
base 10 :	1000ns : op=4 a= 7 b= 3 y= 3

執行結果分析

您可以看到一開始 0ns 時，op=0，所以執行加法，得到 $y=a+b=7+3=10$ ，然後 50ns 時 op=1，所以執行減法，以下是整個執行結果的簡化列表：

a	b	op	y
7	3	0 (+)	10
7	3	1 (-)	4
7	3	2 (*)	21
7	3	3 (/)	2
00000111	00000011	4 (AND)	00000011
00000111	00000011	5 (OR)	00000111
00000111	00000011	6 (NOT)	11111000
00000111	00000011	6 (XOR)	00000100

透過上述的測試，我們知道整個 ALU 的設計方式是正確的。

結語

對於沒有學過「硬體描述語言」的人來說，通常會認為要設計一個 ALU 單元，應該是很複雜的。但是從上述的程式當中，您可以看到在 Verilog 當中設計 ALU 其實是很簡單的，只要用 10 行左右的程式碼，甚至不需要自己設計「加法器」就能完成。

這是因為 Verilog 將「+,-,*,/」等運算內建在語言當中了，所以讓整個程式的撰寫只要透過一個 case 語句就能做完了，這種設計方式非常的像「高階語言」，讓硬體的設計變得更加的容易了。

事實上，在使用 Verilog 設計像 CPU 這樣的複雜元件時，ALU 或暫存器等單元都變得非常的容易。真正複雜的其實是控制單元，而這也是 CPU 設計的精髓之所在，我們會在「開放電腦計劃」系列的文章中，完成 CPU 與控制單元的設計。

參考文獻

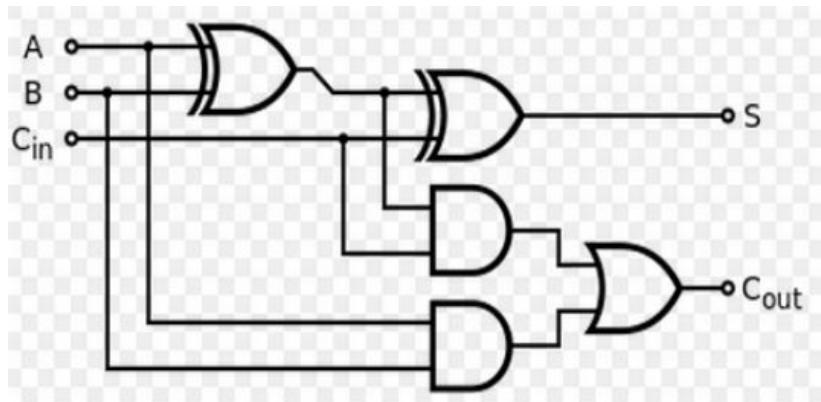
- 陳鍾誠的網站：用 Verilog 設計 ALU

記憶單元 (Memory Unit)

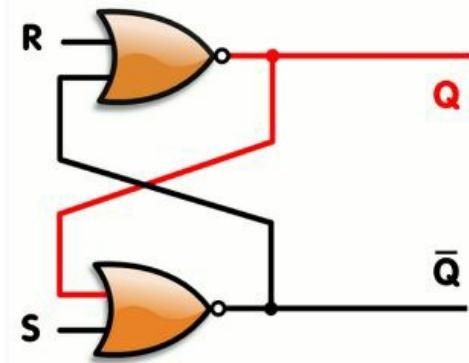
時序邏輯 (Sequential Logic)

組合邏輯 (Combinatorial Logic) 是一種沒有回饋線路的數位電路系統，而循序邏輯 (時序邏輯, Sequential Logic) 則是一種包含回饋線路的系統。

舉例而言，在下圖 (a) 的全加器的電路裏，您可以看到從輸入線路一路輸入接向輸出，這種稱為組合邏輯電路。而在下圖 (b) 的 栓鎖器 (正反器) 線路裏，線路從輸出 Q 又拉回 S 做為輸入，這種有倒勾的線路就稱為循序邏輯電路。



(a) 全加器



(b) 栓鎖器

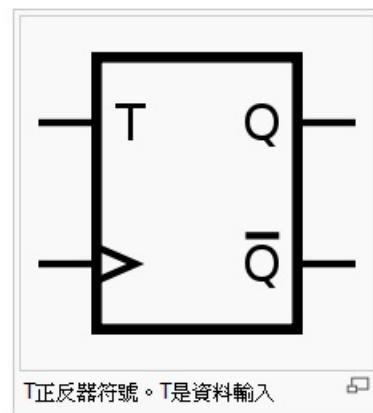
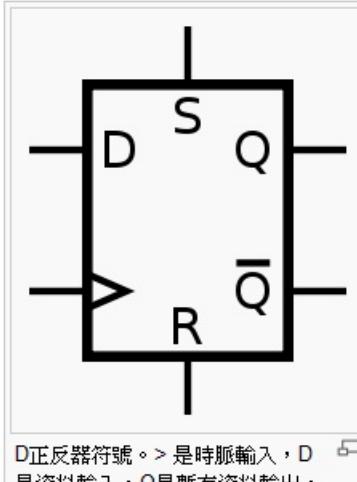
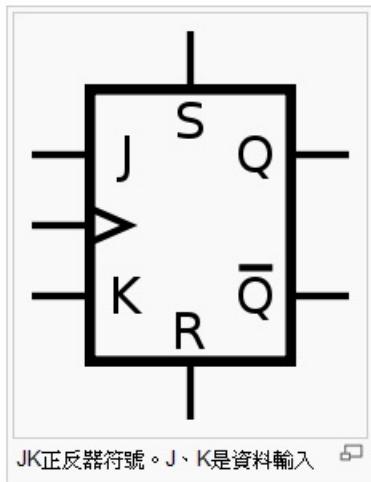
暫存器、靜態記憶體等記憶單元，都是由這種有反饋電路的時序邏輯所構成的，因此要瞭解記憶單元之前，先瞭解時序邏輯的電路結構會是有幫助的。

正反器（門鎖器）

正反器有很多種型式，以下是來自維基百科的一些正反器說明與範例。

- 參考：[http://en.wikipedia.org/wiki/Flip-flop_\(electronics\)](http://en.wikipedia.org/wiki/Flip-flop_(electronics))

摘自維基百科：正反器（英語：Flip-flop, FF，中國大陸譯作觸發器，港澳譯作），學名雙穩態多諧振盪器（Bistable Multivibrator），是一種應用在數位電路上具有記憶功能的循序邏輯元件，可記錄二進位制數位訊號「1」和「0」。正反器是構成序向邏輯電路以及各種複雜數位系統的基本邏輯單元。

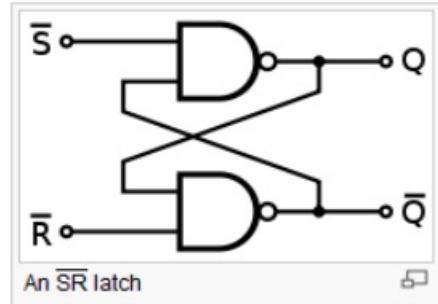
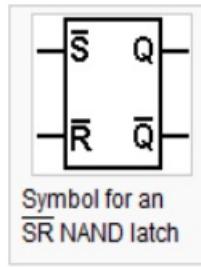


圖、各種正反器

SR 正反器

正反器是可以用來儲存位元，是循序電路的基礎，以下是一個用 NAND 閘構成的正反器。

$\overline{S}\overline{R}$ latch operation		
\overline{S}	\overline{R}	Action
0	0	Restricted combination
0	1	$Q = 1$
1	0	$Q = 0$
1	1	No Change



圖、NAND 閘構成的正反器

我們可以根據上圖實作出對應的 Verilog 程式如下：

檔案：latch.v

```
module latch(input Sbar, Rbar, output Q, Qbar);
    nand LS(Q, Sbar, Qbar);
    nand LR(Qbar, Rbar, Q);
endmodule

module main;
```

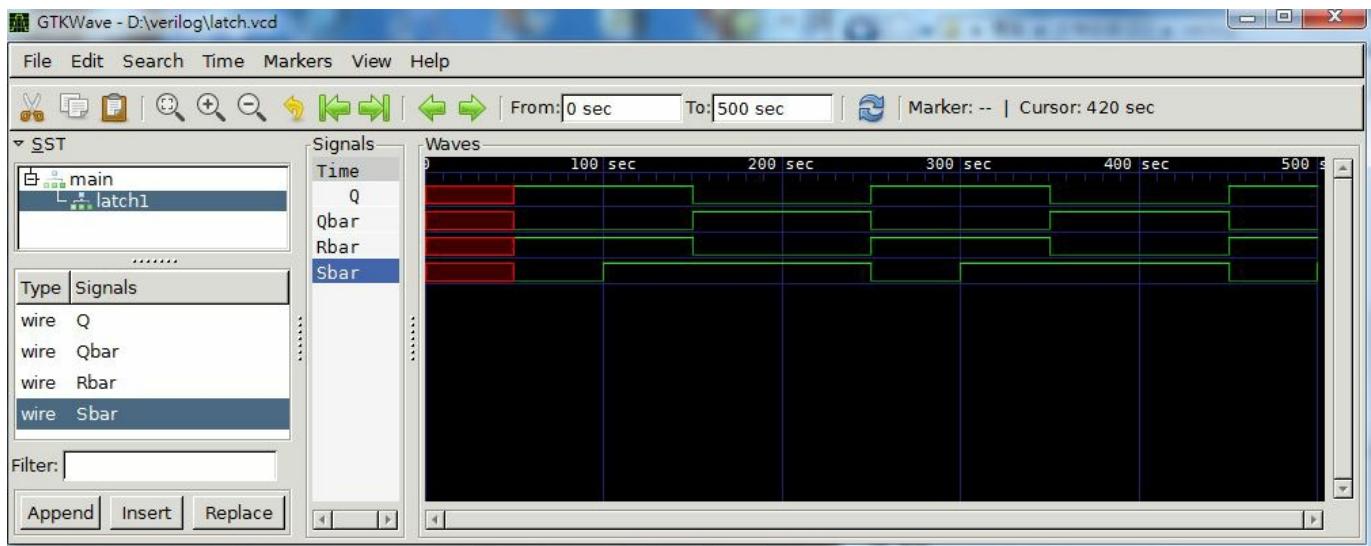
```
reg Sbar, Rbar;  
wire Q, Qbar;  
  
latch latch1(Sbar, Rbar, Q, Qbar);  
  
initial  
begin  
    $monitor("%4dns monitor: Sbar=%d Rbar=%d Q=%d Qbar=%d", $stime, Sbar, Rbar, Q, Qbar);  
    $dumpfile("latch.vcd"); // 輸出給 GTK wave 顯示波型  
    $dumpvars;  
end  
  
always #50 begin  
    Sbar = 0; Rbar = 1;  
    #50;  
    Sbar = 1; Rbar = 1;  
    #50;
```

```
Sbar = 1; Rbar = 0;  
#50;  
end  
  
initial #500 $finish;  
  
endmodule
```

執行結果：

```
D:\verilog>iverilog -o latch latch.v  
  
D:\verilog>vvp latch  
VCD info: dumpfile latch.vcd opened for output.  
0ns monitor: Sbar=x Rbar=x Q=x Qbar=x  
50ns monitor: Sbar=0 Rbar=1 Q=1 Qbar=0  
100ns monitor: Sbar=1 Rbar=1 Q=1 Qbar=0  
150ns monitor: Sbar=1 Rbar=0 Q=0 Qbar=1  
250ns monitor: Sbar=0 Rbar=1 Q=1 Qbar=0
```

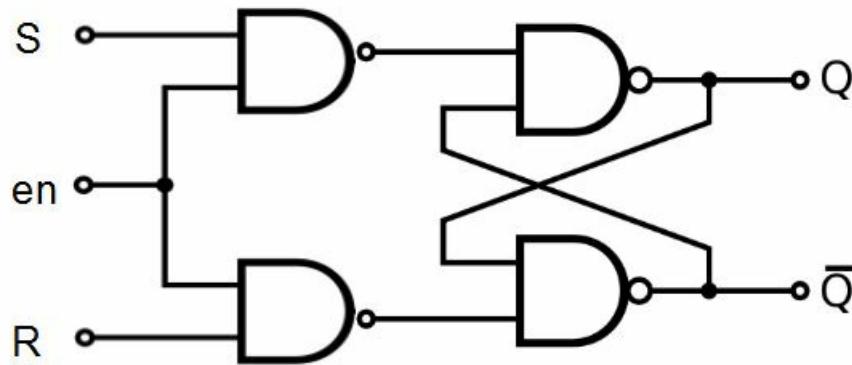
```
300ns monitor: Sbar=1 Rbar=1 Q=1 Qbar=0  
350ns monitor: Sbar=1 Rbar=0 Q=0 Qbar=1  
450ns monitor: Sbar=0 Rbar=1 Q=1 Qbar=0  
500ns monitor: Sbar=1 Rbar=1 Q=1 Qbar=0
```



圖、latch.vcd 的顯示圖形

有 enable 的正反器

如果我們在上述正反器前面再加上兩個 NAND 閘進行控制，就可以形成一組有 enable 的正反器，以下是該正反器的圖形。



圖、有 enable 的正反器

根據上述圖形我們可以設計出以下的 Verilog 程式。

檔案：enLatch.v

```
module latch(input Sbar, Rbar, output Q, Qbar);
    nand LS(Q, Sbar, Qbar);
```

```
nand LR(Qbar, Rbar, Q);  
endmodule  
  
module enLatch(input en, S, R, output Q, Qbar);  
    nand ES(Senbar, en, S);  
    nand ER(Renbar, en, R);  
    latch L1(Senbar, Renbar, Q, Qbar);  
endmodule  
  
module main;  
reg S, en, R;  
wire Q, Qbar;  
  
enLatch enLatch1(en, S, R, Q, Qbar);  
  
initial  
begin  
$monitor("%4dns monitor: en=%d S=%d R=%d Q=%d Qbar=%d", $stime, en
```

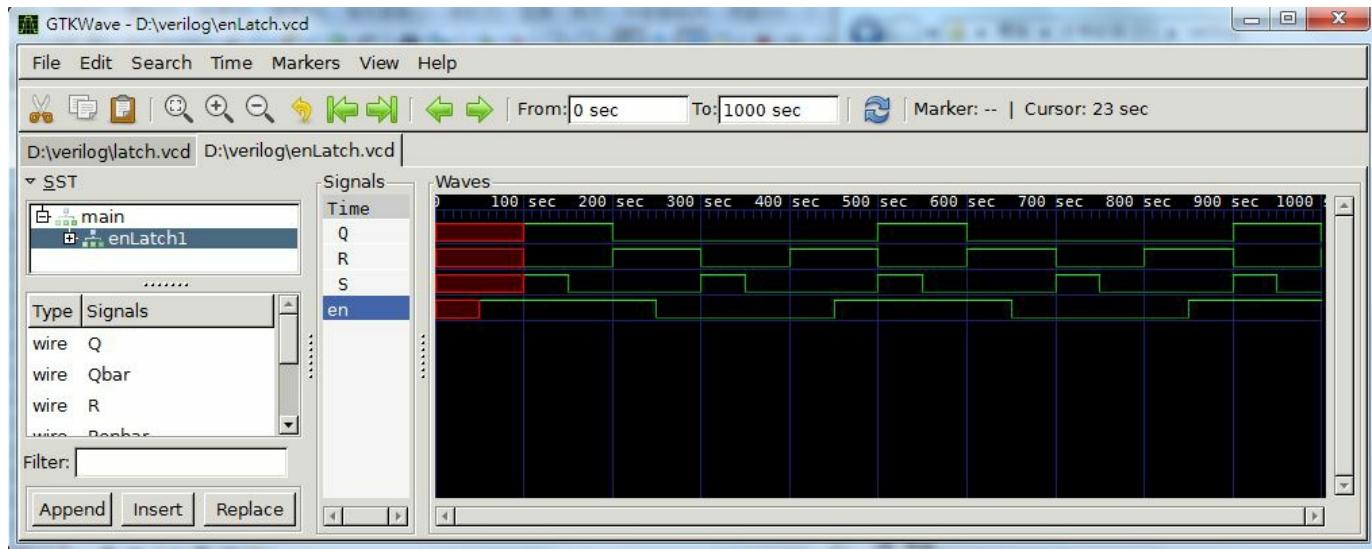
```
, S, R, Q, Qbar);  
$dumpfile("enLatch.vcd"); // 輸出給 GTK wave 顯示波型  
$dumpvars;  
end  
  
always #50 begin  
    en = 1;  
    #50;  
    S = 1; R = 0;  
    #50;  
    S = 0; R = 0;  
    #50;  
    S = 0; R = 1;  
    #50  
    en = 0;  
    #50;  
    S = 1; R = 0;  
    #50;
```

```
S = 0; R = 0;  
#50;  
S = 0; R = 1;  
end  
  
initial #1000 $finish;  
  
endmodule
```

執行結果

```
D:\verilog>iverilog -o enLatch enLatch.v  
  
D:\verilog>vvp enLatch  
VCD info: dumpfile enLatch.vcd opened for output.  
 0ns monitor: en=x Sbar=x Rbar=x Q=x Qbar=x  
 50ns monitor: en=1 Sbar=x Rbar=x Q=x Qbar=x  
100ns monitor: en=1 Sbar=1 Rbar=0 Q=1 Qbar=0  
150ns monitor: en=1 Sbar=0 Rbar=0 Q=1 Qbar=0
```

200ns monitor: en=1 Sbar=0 Rbar=1 Q=0 Qbar=1
250ns monitor: en=0 Sbar=0 Rbar=1 Q=0 Qbar=1
300ns monitor: en=0 Sbar=1 Rbar=0 Q=0 Qbar=1
350ns monitor: en=0 Sbar=0 Rbar=0 Q=0 Qbar=1
400ns monitor: en=0 Sbar=0 Rbar=1 Q=0 Qbar=1
450ns monitor: en=1 Sbar=0 Rbar=1 Q=0 Qbar=1
500ns monitor: en=1 Sbar=1 Rbar=0 Q=1 Qbar=0
550ns monitor: en=1 Sbar=0 Rbar=0 Q=1 Qbar=0
600ns monitor: en=1 Sbar=0 Rbar=1 Q=0 Qbar=1
650ns monitor: en=0 Sbar=0 Rbar=1 Q=0 Qbar=1
700ns monitor: en=0 Sbar=1 Rbar=0 Q=0 Qbar=1
750ns monitor: en=0 Sbar=0 Rbar=0 Q=0 Qbar=1
800ns monitor: en=0 Sbar=0 Rbar=1 Q=0 Qbar=1
850ns monitor: en=1 Sbar=0 Rbar=1 Q=0 Qbar=1
900ns monitor: en=1 Sbar=1 Rbar=0 Q=1 Qbar=0
950ns monitor: en=1 Sbar=0 Rbar=0 Q=1 Qbar=0
1000ns monitor: en=1 Sbar=0 Rbar=1 Q=0 Qbar=1



圖、enLatch.vcd 的顯示圖形

閘級延遲 (Gate Delay)

在 Verilog 模型下，邏輯閘預設是沒有任何延遲的，因此呈現出來永遠是即時的結果，但現實世界的電路總是有少許延遲的，每經過一個閘就會延遲一點點的時間，經過的閘數越多，延遲也就會越久。

- 參考：<http://www.asic-world.com/verilog/gate3.html>

為了模擬這種延遲，Verilog 允許你在閘上面附加延遲時間的語法，您可以分別指定最小延遲 min，典型延遲 typical 與最大延遲 max。

舉例而言，以下的語法宣告了一個 not 閘，其中的 #(1:3:5) 語法指定了最小延遲 min=1，典型延遲 typical=3，最大延遲 max=5。

```
not #(1:3:5) n2(nclk2, clk);
```

假如您不想分別指定這三種延遲，也可以只指定一個延遲參數，這樣 min, typical, max 三者都會設定為該數值，舉例而言，以下是一個宣告延遲固定為 2 的 not 閘。

```
not #2 n1(nclk1, clk);
```

為了說明這種延遲狀況，我們寫了一個範例程式 delay.v 來示範設定了閘級延遲的效果，請參考下列程式：

```
module main;
    reg clk;

    not #2 n1(nclk1, clk);
```

```
not #(1:3:5) n2(nclk2, clk);  
  
initial begin  
    clk = 0;  
    $monitor("%dns monitor: clk=%b nclk1=%d nclk2=%d", $stime, clk, nclk1, nclk2);  
    $dumpfile("delay.vcd"); // 輸出給 GTK wave 顯示波型  
    $dumpvars;  
end  
  
always #10 begin  
    clk = clk + 1;  
end  
  
initial #100 $finish;  
  
endmodule
```

執行結果：

```
D:\Dropbox\Public\web\oc\code>iverilog -o delay delay.v
delay.v:5: warning: choosing typ expression.

D:\Dropbox\Public\web\oc\code>vvp delay
VCD info: dumpfile delay.vcd opened for output.

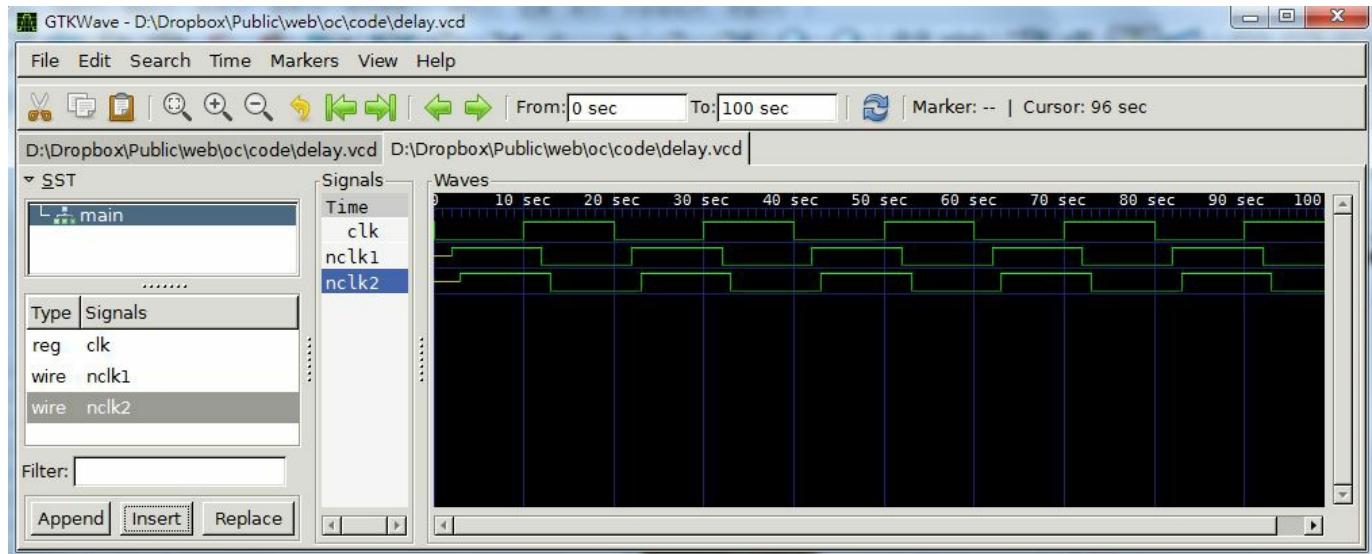
    0ns monitor: clk=0 nclk1=z nclk2=z
    2ns monitor: clk=0 nclk1=1 nclk2=z
    3ns monitor: clk=0 nclk1=1 nclk2=1
   10ns monitor: clk=1 nclk1=1 nclk2=1
   12ns monitor: clk=1 nclk1=0 nclk2=1
   13ns monitor: clk=1 nclk1=0 nclk2=0
   20ns monitor: clk=0 nclk1=0 nclk2=0
   22ns monitor: clk=0 nclk1=1 nclk2=0
   23ns monitor: clk=0 nclk1=1 nclk2=1
   30ns monitor: clk=1 nclk1=1 nclk2=1
   32ns monitor: clk=1 nclk1=0 nclk2=1
```

```
33ns monitor: clk=1 nclk1=0 nclk2=0
40ns monitor: clk=0 nclk1=0 nclk2=0
42ns monitor: clk=0 nclk1=1 nclk2=0
43ns monitor: clk=0 nclk1=1 nclk2=1
50ns monitor: clk=1 nclk1=1 nclk2=1
52ns monitor: clk=1 nclk1=0 nclk2=1
53ns monitor: clk=1 nclk1=0 nclk2=0
60ns monitor: clk=0 nclk1=0 nclk2=0
62ns monitor: clk=0 nclk1=1 nclk2=0
63ns monitor: clk=0 nclk1=1 nclk2=1
70ns monitor: clk=1 nclk1=1 nclk2=1
72ns monitor: clk=1 nclk1=0 nclk2=1
73ns monitor: clk=1 nclk1=0 nclk2=0
80ns monitor: clk=0 nclk1=0 nclk2=0
82ns monitor: clk=0 nclk1=1 nclk2=0
83ns monitor: clk=0 nclk1=1 nclk2=1
90ns monitor: clk=1 nclk1=1 nclk2=1
92ns monitor: clk=1 nclk1=0 nclk2=1
```

93ns monitor: clk=1 nclk1=0 nclk2=0

100ns monitor: clk=0 nclk1=0 nclk2=0

以上輸出的波形如下，您可以看到 nclk1 的延遲固定為 2 ，而 nclk2 的延遲則介於 1 到 5 之間。



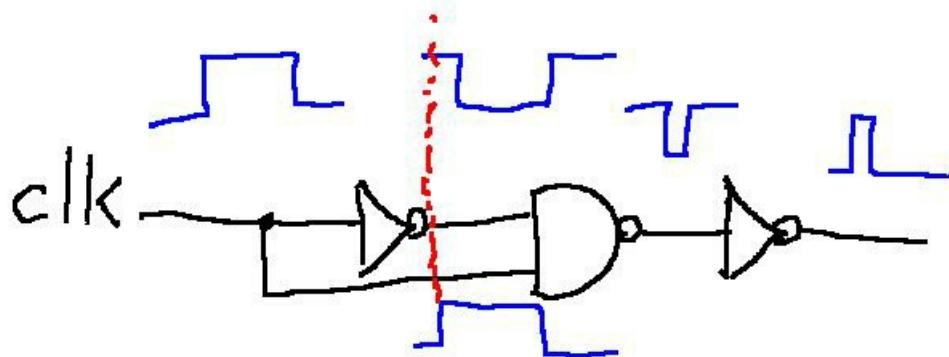
圖、delay.vcd 的顯示波型

利用「閘級延遲」製作脈波變化偵測器（Pulse Transition

Detector, PTD)

雖然延遲現象看起來像是個缺陷，但事實上如果好好的利用這種現象，有時反而可以達到很好的效果，「脈波變化偵測器」電路就是利用這種現象所設計的一種電路，可以用來偵測「脈波的上升邊緣或下降邊緣」。

以下是「脈波變化偵測器」的圖形，其中的關鍵是在左邊的 not 閘身上，由於每個閘都會造成延遲，因此多了 not 閘的那條路徑所造成的延遲較多，這讓輸出部份會因為延遲而形成一個脈衝波形。



圖、脈波變化偵測器

以下是這個電路以 Verilog 實作的結果。

檔案：ptd.v

```
module ptd(input clk, output ppulse);
    not #2 P1(nclkd, clk);
    nand #2 P2(npulse, nclkd, clk);
    not #2 P3(ppulse, npulse);
endmodule

module main;
    reg clk;
    wire p;

    ptd ptd1(clk, p);

    initial begin
        clk = 0;
        $monitor("%dns monitor: clk=%b p=%d", $stime, clk, p);
        $dumpfile("ptd.vcd"); // 輸出給 GTK wave 顯示波型
    end
endmodule
```

```
$dumpvars;  
end  
  
always #50 begin  
    clk = clk + 1;  
end  
  
initial #500 $finish;  
  
endmodule
```

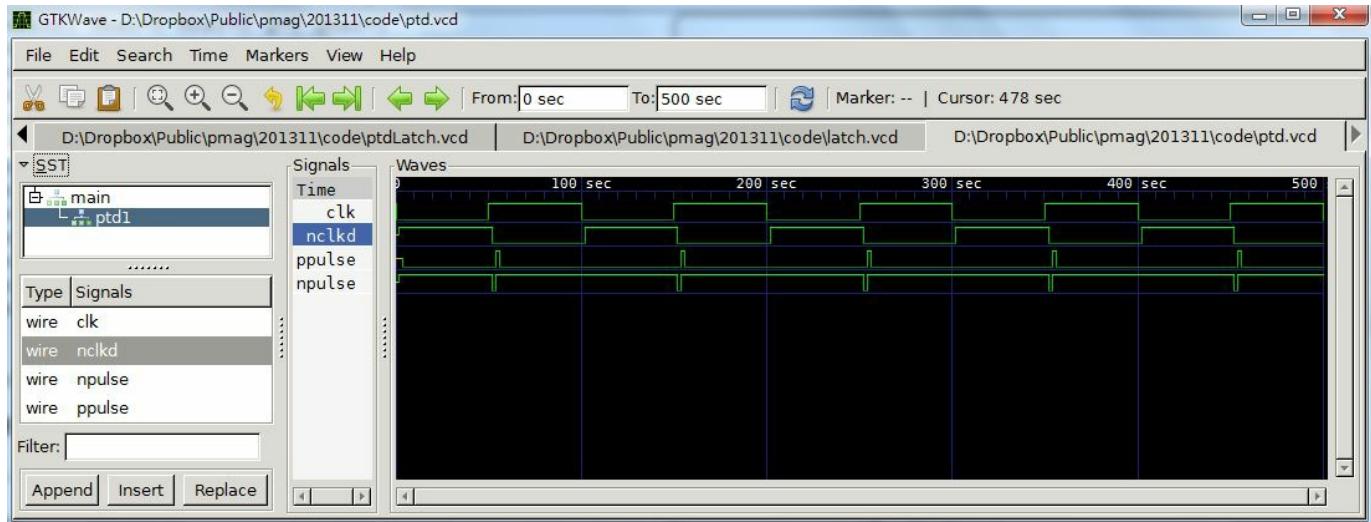
執行結果

```
D:\Dropbox\Public\pmag\201311\code>iverilog -o ptd ptd.v  
  
D:\Dropbox\Public\pmag\201311\code>vvp ptd  
VCD info: dumpfile ptd.vcd opened for output.  
    0ns monitor: clk=0 p=z  
    4ns monitor: clk=0 p=0
```

```
50ns monitor: clk=1 p=0
54ns monitor: clk=1 p=1
56ns monitor: clk=1 p=0
100ns monitor: clk=0 p=0
150ns monitor: clk=1 p=0
154ns monitor: clk=1 p=1
156ns monitor: clk=1 p=0
200ns monitor: clk=0 p=0
250ns monitor: clk=1 p=0
254ns monitor: clk=1 p=1
256ns monitor: clk=1 p=0
300ns monitor: clk=0 p=0
350ns monitor: clk=1 p=0
354ns monitor: clk=1 p=1
356ns monitor: clk=1 p=0
400ns monitor: clk=0 p=0
450ns monitor: clk=1 p=0
454ns monitor: clk=1 p=1
```

456ns monitor: clk=1 p=0

500ns monitor: clk=0 p=0

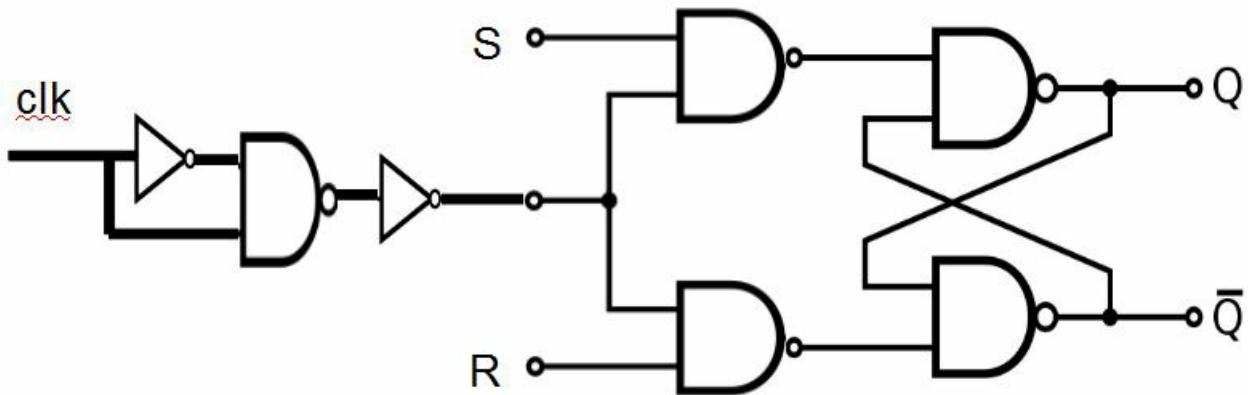


圖、ptd.vcd 的顯示圖形

使用「脈衝偵測電路」製作「邊緣觸發正反器」

有了「正反器」與「脈波變化偵測電路」之後，我們就可以組合出「邊緣觸發正反器」了，以下是

其電路圖。



圖、邊緣觸發的正反器

事實上，上述電路圖只是將「有 enable 的正反器」前面加上一個「脈波變化偵測電路」而已，其實做的 Verilog 程式如下。

檔案：ptdLatch.v

```
module latch(input Sbar, Rbar, output Q, Qbar);  
    nand LS(Q, Sbar, Qbar);
```

```
nand LR(Qbar, Rbar, Q) ;  
endmodule  
  
module enLatch(input en, S, R, output Q, Qbar) ;  
    nand ES(Senbar, en, S) ;  
    nand ER(Renbar, en, R) ;  
    latch L1(Senbar, Renbar, Q, Qbar) ;  
endmodule  
  
module ptd(input clk, output ppulse) ;  
    not #2 P1(nclkd, clk) ;  
    nand #2 P2(npulse, nclkd, clk) ;  
    not #2 P3(ppulse, npulse) ;  
endmodule  
  
module ptdLatch(input clk, S, R, output Q, Qbar) ;  
    ptd PTD(clk, ppulse) ;  
    enLatch EL(ppulse, S, R, Q, Qbar) ;
```

```
endmodule

module main;
reg S, clk, R;
wire Q, Qbar;

ptdLatch ptdLatch1(clk, S, R, Q, Qbar);

initial
begin
    clk = 0;
    $monitor("%4dns monitor: clk=%d ppulse=%d S=%d R=%d Q=%d Qbar=%d",
    $stime, clk, ptdLatch1.ppulse, S, R, Q, Qbar);
    $dumpfile("ptdLatch.vcd"); // 輸出給 GTK wave 顯示波型
    $dumpvars;
end
```

```
always #20 begin  
    clk = ~clk;  
end
```

```
always #50 begin  
    S = 1; R = 0;  
    #50;  
    S = 0; R = 0;  
    #50;  
    S = 0; R = 1;  
    #50;  
end
```

```
initial #500 $finish;  
  
endmodule
```

執行結果

```
D:\verilog>iverilog -o ptdLatch ptdLatch.v
```

```
D:\verilog>vvp ptdLatch
```

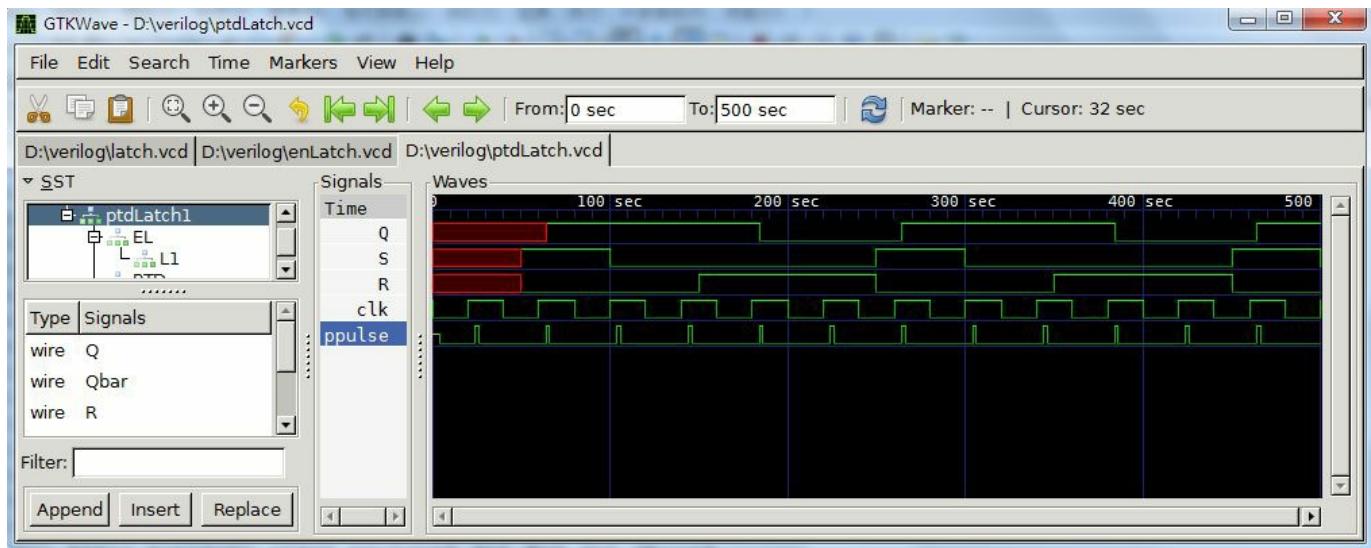
```
VCD info: dumpfile ptdLatch.vcd opened for output.
```

```
0ns monitor: clk=0 ppulse=z S=x R=x Q=x Qbar=x  
4ns monitor: clk=0 ppulse=0 S=x R=x Q=x Qbar=x  
20ns monitor: clk=1 ppulse=0 S=x R=x Q=x Qbar=x  
24ns monitor: clk=1 ppulse=1 S=x R=x Q=x Qbar=x  
26ns monitor: clk=1 ppulse=0 S=x R=x Q=x Qbar=x  
40ns monitor: clk=0 ppulse=0 S=x R=x Q=x Qbar=x  
50ns monitor: clk=0 ppulse=0 S=1 R=0 Q=x Qbar=x  
60ns monitor: clk=1 ppulse=0 S=1 R=0 Q=x Qbar=x  
64ns monitor: clk=1 ppulse=1 S=1 R=0 Q=1 Qbar=0  
66ns monitor: clk=1 ppulse=0 S=1 R=0 Q=1 Qbar=0  
80ns monitor: clk=0 ppulse=0 S=1 R=0 Q=1 Qbar=0  
100ns monitor: clk=1 ppulse=0 S=0 R=0 Q=1 Qbar=0  
104ns monitor: clk=1 ppulse=1 S=0 R=0 Q=1 Qbar=0  
106ns monitor: clk=1 ppulse=0 S=0 R=0 Q=1 Qbar=0
```

120ns monitor: clk=0 ppulse=0 S=0 R=0 Q=1 Qbar=0
140ns monitor: clk=1 ppulse=0 S=0 R=0 Q=1 Qbar=0
144ns monitor: clk=1 ppulse=1 S=0 R=0 Q=1 Qbar=0
146ns monitor: clk=1 ppulse=0 S=0 R=0 Q=1 Qbar=0
150ns monitor: clk=1 ppulse=0 S=0 R=1 Q=1 Qbar=0
160ns monitor: clk=0 ppulse=0 S=0 R=1 Q=1 Qbar=0
180ns monitor: clk=1 ppulse=0 S=0 R=1 Q=1 Qbar=0
184ns monitor: clk=1 ppulse=1 S=0 R=1 Q=0 Qbar=1
186ns monitor: clk=1 ppulse=0 S=0 R=1 Q=0 Qbar=1
200ns monitor: clk=0 ppulse=0 S=0 R=1 Q=0 Qbar=1
220ns monitor: clk=1 ppulse=0 S=0 R=1 Q=0 Qbar=1
224ns monitor: clk=1 ppulse=1 S=0 R=1 Q=0 Qbar=1
226ns monitor: clk=1 ppulse=0 S=0 R=1 Q=0 Qbar=1
240ns monitor: clk=0 ppulse=0 S=0 R=1 Q=0 Qbar=1
250ns monitor: clk=0 ppulse=0 S=1 R=0 Q=0 Qbar=1
260ns monitor: clk=1 ppulse=0 S=1 R=0 Q=0 Qbar=1
264ns monitor: clk=1 ppulse=1 S=1 R=0 Q=1 Qbar=0
266ns monitor: clk=1 ppulse=0 S=1 R=0 Q=1 Qbar=0

280ns monitor: clk=0 ppulse=0 S=1 R=0 Q=1 Qbar=0
300ns monitor: clk=1 ppulse=0 S=0 R=0 Q=1 Qbar=0
304ns monitor: clk=1 ppulse=1 S=0 R=0 Q=1 Qbar=0
306ns monitor: clk=1 ppulse=0 S=0 R=0 Q=1 Qbar=0
320ns monitor: clk=0 ppulse=0 S=0 R=0 Q=1 Qbar=0
340ns monitor: clk=1 ppulse=0 S=0 R=0 Q=1 Qbar=0
344ns monitor: clk=1 ppulse=1 S=0 R=0 Q=1 Qbar=0
346ns monitor: clk=1 ppulse=0 S=0 R=0 Q=1 Qbar=0
350ns monitor: clk=1 ppulse=0 S=0 R=1 Q=1 Qbar=0
360ns monitor: clk=0 ppulse=0 S=0 R=1 Q=1 Qbar=0
380ns monitor: clk=1 ppulse=0 S=0 R=1 Q=1 Qbar=0
384ns monitor: clk=1 ppulse=1 S=0 R=1 Q=0 Qbar=1
386ns monitor: clk=1 ppulse=0 S=0 R=1 Q=0 Qbar=1
400ns monitor: clk=0 ppulse=0 S=0 R=1 Q=0 Qbar=1
420ns monitor: clk=1 ppulse=0 S=0 R=1 Q=0 Qbar=1
424ns monitor: clk=1 ppulse=1 S=0 R=1 Q=0 Qbar=1
426ns monitor: clk=1 ppulse=0 S=0 R=1 Q=0 Qbar=1

```
440ns monitor: clk=0 ppulse=0 S=0 R=1 Q=0 Qbar=1
450ns monitor: clk=0 ppulse=0 S=1 R=0 Q=0 Qbar=1
460ns monitor: clk=1 ppulse=0 S=1 R=0 Q=0 Qbar=1
464ns monitor: clk=1 ppulse=1 S=1 R=0 Q=1 Qbar=0
466ns monitor: clk=1 ppulse=0 S=1 R=0 Q=1 Qbar=0
480ns monitor: clk=0 ppulse=0 S=1 R=0 Q=1 Qbar=0
500ns monitor: clk=1 ppulse=0 S=0 R=0 Q=1 Qbar=0
```



圖、ptdLatch.vcd 的顯示圖形

使用「脈衝偵測電路」設計邊緣觸發暫存器

有了「脈波變化偵測電路」，只要與任何需要偵測脈波變化的元件串接起來，就可以達到「邊緣觸發」的功能。

其實、像是 Verilog 當中的以下程式，其實都是利用類似的「脈波變化偵測電路」所完成的。

```
always @(posedge clock) begin  
    ...  
end
```

如果我們真的不想使用 posedge clock 這種語法，我們也可以用前述的「脈波變化偵測電路」(PTD)來製作這類的邊緣觸發功能，以下是我們用這種方式設計的一個邊緣觸發暫存器。

檔案：ptdRegister.v

```
module ptd(input clk, output ppulse);
```

```
not #2 g1(nclkd, clk);
nand #2 g2(npulse, nclkd, clk);
not #2 g3(ppulse, npulse);

endmodule

module register(input en, input [31:0] d, output reg [31:0] r);
always @(en) begin
  if (en)
    r <= d;
end
endmodule

module main;
reg [31:0] d;
wire [31:0] r;
reg clk;
wire en;
```

```
ptd ptd1(clk, en);
register register1(en, d, r);

initial begin
    clk = 0;
    d = 3;
    $monitor("%4dns monitor: clk=%d en=%d d=%d r=%d", $stime, clk, en,
d, r);
end

always #10 begin
    clk = clk + 1;
end

always #20 begin
    d = d + 1;
end
```

```
initial #100 $finish;  
  
endmodule
```

執行結果

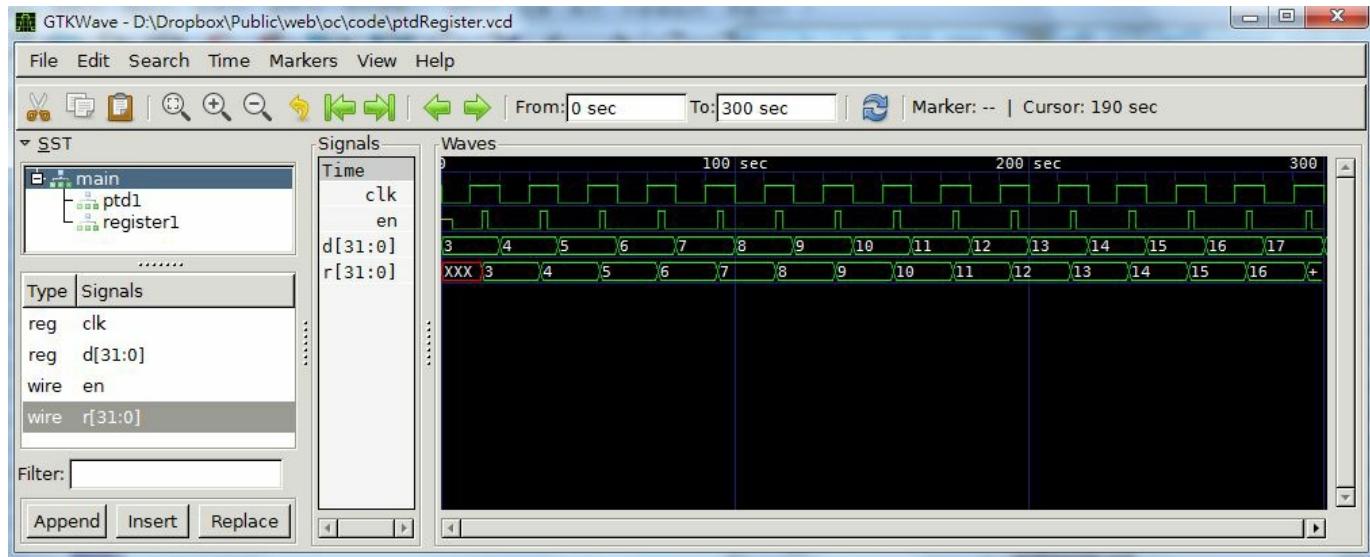
```
D:\Dropbox\Public\web\oc\code>iverilog -o ptdRegister ptdRegister.v
```

```
D:\Dropbox\Public\web\oc\code>vvp ptdRegister
```

0ns monitor: clk=0 en=z d=	3	r=	x
4ns monitor: clk=0 en=0 d=	3	r=	x
10ns monitor: clk=1 en=0 d=	3	r=	x
14ns monitor: clk=1 en=1 d=	3	r=	3
16ns monitor: clk=1 en=0 d=	3	r=	3
20ns monitor: clk=0 en=0 d=	4	r=	3
30ns monitor: clk=1 en=0 d=	4	r=	3
34ns monitor: clk=1 en=1 d=	4	r=	4
36ns monitor: clk=1 en=0 d=	4	r=	4

40ns monitor: clk=0 en=0 d=	5 r=	4
50ns monitor: clk=1 en=0 d=	5 r=	4
54ns monitor: clk=1 en=1 d=	5 r=	5
56ns monitor: clk=1 en=0 d=	5 r=	5
60ns monitor: clk=0 en=0 d=	6 r=	5
70ns monitor: clk=1 en=0 d=	6 r=	5
74ns monitor: clk=1 en=1 d=	6 r=	6
76ns monitor: clk=1 en=0 d=	6 r=	6
80ns monitor: clk=0 en=0 d=	7 r=	6
90ns monitor: clk=1 en=0 d=	7 r=	6
94ns monitor: clk=1 en=1 d=	7 r=	7
96ns monitor: clk=1 en=0 d=	7 r=	7
100ns monitor: clk=0 en=0 d=	8 r=	7

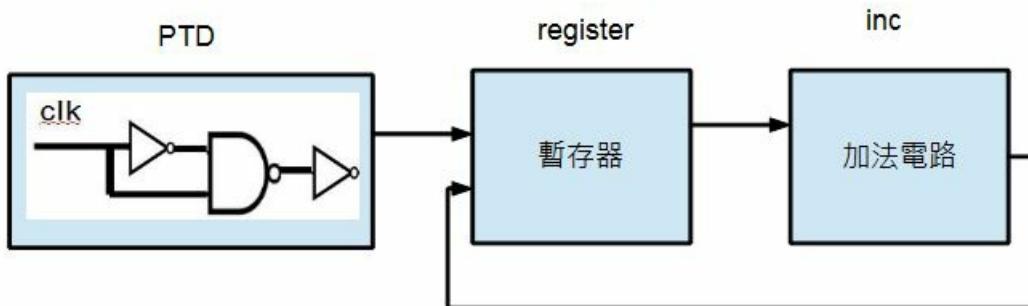
其輸出的波型檔如下圖所示：



圖、在 GTKWave 中顯示的 ptdRegister.vcd 波型檔

使用「脈衝偵測電路」製作計數電路

如果我們將暫存器的輸出接到一個加法電路上，進行回饋性的累加的動作，如下圖所示，那麼整個電路就會變成一個邊緣觸發的計數電路



圖、邊緣觸發的計數電路

以上這種電路可以做為「採用區塊方法設計 CPU 的基礎」，因為 CPU 當中的「程式計數器」(Program Counter) 通常會採用 這種邊緣觸發的設計方式。

以下是上述電路的設計與實作測試結果。

檔案：ptdCounter.v

```

module register(input en, input [31:0] d, output reg [31:0] r);
always @(en) begin
if (en)

```

```
r <= d;  
end  
endmodule  
  
module ptd(input clk, output ppulse);  
    not #2 P1(nclkd, clk);  
    nand #2 P2(npulse, nclkd, clk);  
    not #2 P3(ppulse, npulse);  
endmodule  
  
module inc(input [31:0] i, output [31:0] o);  
    assign o = i + 4;  
endmodule  
  
module main;  
    wire [31:0] r, ro;  
    reg clk;  
    wire en;
```

```
ptd ptd1(clk, en);
register r1(en, ro, r);
inc i1(r, ro);

initial begin
    clk = 0;
    r1.r = 0;
    $monitor("%4dns monitor: clk=%d en=%d r=%d", $stime, clk, en, r);
    $dumpfile("ptdCounter.vcd"); // 輸出給 GTK wave 顯示波型
    $dumpvars;
end

always #10 begin
    clk = clk + 1;
end

initial #100 $finish;
```

```
endmodule
```

執行結果

```
D:\Dropbox\Public\web\oc\code>iverilog -o ptdCounter ptdCounter.v
```

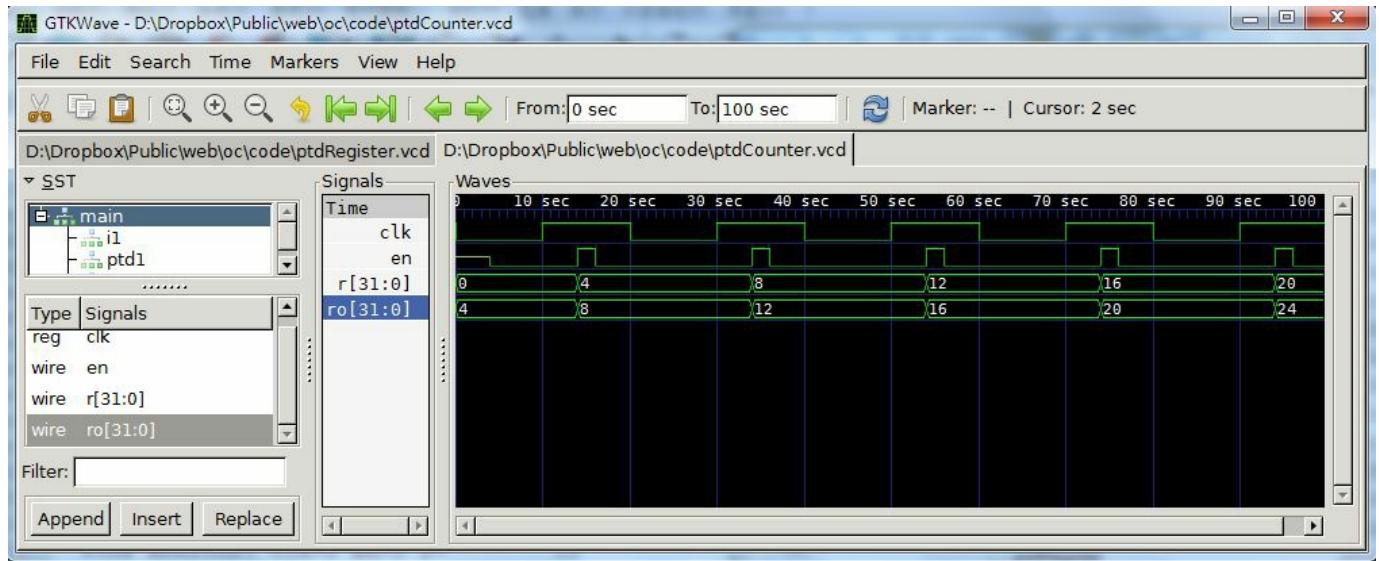
```
D:\Dropbox\Public\web\oc\code>vvp ptdCounter
```

```
VCD info: dumpfile ptdCounter.vcd opened for output.
```

0ns monitor:	clk=0	en=z	r=	0
4ns monitor:	clk=0	en=0	r=	0
10ns monitor:	clk=1	en=0	r=	0
14ns monitor:	clk=1	en=1	r=	4
16ns monitor:	clk=1	en=0	r=	4
20ns monitor:	clk=0	en=0	r=	4
30ns monitor:	clk=1	en=0	r=	4
34ns monitor:	clk=1	en=1	r=	8
36ns monitor:	clk=1	en=0	r=	8
40ns monitor:	clk=0	en=0	r=	8

50ns monitor: clk=1 en=0 r=	8
54ns monitor: clk=1 en=1 r=	12
56ns monitor: clk=1 en=0 r=	12
60ns monitor: clk=0 en=0 r=	12
70ns monitor: clk=1 en=0 r=	12
74ns monitor: clk=1 en=1 r=	16
76ns monitor: clk=1 en=0 r=	16
80ns monitor: clk=0 en=0 r=	16
90ns monitor: clk=1 en=0 r=	16
94ns monitor: clk=1 en=1 r=	20
96ns monitor: clk=1 en=0 r=	20
100ns monitor: clk=0 en=0 r=	20

其輸出的波型檔如下圖所示：



圖、在 GTKWave 中顯示的 ptdCounter.vcd 波型檔

暫存器單元

檔案：regbank.v

```
module regbank(input [3:0] ra1, output [31:0] rd1,  
                input [3:0] ra2, output [31:0] rd2,
```

```
    input clk, input w_en,
          input [3:0] wa, input [31:0] wd);
reg [31:0] r[15:0]; // 宣告 16 個 32 位元的暫存器
assign rd1 = r[ra1]; // 讀取索引值為 ra1 的暫存器
assign rd2 = r[ra2]; // 讀取索引值為 ra2 的暫存器
always @(posedge clk)
begin
  if (w_en) // w_en=1 時寫入到暫存器
    r[wa] <= wd; // 將 wd 寫入到索引值為 wa 的暫存器
end
endmodule
```

```
module main;
reg [3:0] ra1, ra2, wa;
reg clk, w_en;
wire [31:0] rd1, rd2;
reg [31:0] wd;
```

```
regbank rb0(ra1, rd1, ra2, rd2, clk, w_en, wa, wd);

initial
begin
    wa = 0;
    ra1 = 0;
    ra2 = 0;
    wd = 0;
    clk = 0;
    w_en = 1;
end

initial #200 ra1 = 0;

always #50 begin
    clk = clk + 1;
    $monitor("%4dns monitor: ra1=%d rd1=%d ra2=%d rd2=%d wa=%d wd=%d",

```

```
$stime, ra1, rd1, ra2, rd2, wa, wd) ;  
end  
  
always #100 begin  
    wa = wa + 1;  
    wd = wd + 2;  
    ra1 = ra1 + 1;  
    ra2 = ra2 - 1;  
end  
  
initial #1000 $finish;  
  
endmodule
```

執行結果：

```
D:\Dropbox\Public\web\oc\code\verilog>iverilog -o regbank regbank.v
```

D:\Dropbox\Public\web\oc\code\verilog>vvp regbank

50ns monitor: ra1= 0 rd1=	0 ra2= 0 rd2=	0 wa= 0 wd
= 0		
100ns monitor: ra1= 1 rd1=	x ra2=15 rd2=	x wa= 1 wd
= 2		
150ns monitor: ra1= 1 rd1=	2 ra2=15 rd2=	x wa= 1 wd
= 2		
200ns monitor: ra1= 1 rd1=	2 ra2=14 rd2=	x wa= 2 wd
= 4		
250ns monitor: ra1= 1 rd1=	2 ra2=14 rd2=	x wa= 2 wd
= 4		
300ns monitor: ra1= 2 rd1=	4 ra2=13 rd2=	x wa= 3 wd
= 6		
350ns monitor: ra1= 2 rd1=	4 ra2=13 rd2=	x wa= 3 wd
= 6		
400ns monitor: ra1= 3 rd1=	6 ra2=12 rd2=	x wa= 4 wd
= 8		
450ns monitor: ra1= 3 rd1=	6 ra2=12 rd2=	x wa= 4 wd

=	8		
500ns monitor:	ra1= 4 rd1=	8 ra2=11 rd2=	x wa= 5 wd
=	10		
550ns monitor:	ra1= 4 rd1=	8 ra2=11 rd2=	x wa= 5 wd
=	10		
600ns monitor:	ra1= 5 rd1=	10 ra2=10 rd2=	x wa= 6 wd
=	12		
650ns monitor:	ra1= 5 rd1=	10 ra2=10 rd2=	x wa= 6 wd
=	12		
700ns monitor:	ra1= 6 rd1=	12 ra2= 9 rd2=	x wa= 7 wd
=	14		
750ns monitor:	ra1= 6 rd1=	12 ra2= 9 rd2=	x wa= 7 wd
=	14		
800ns monitor:	ra1= 7 rd1=	14 ra2= 8 rd2=	x wa= 8 wd
=	16		
850ns monitor:	ra1= 7 rd1=	14 ra2= 8 rd2=	16 wa= 8 wd
=	16		
900ns monitor:	ra1= 8 rd1=	16 ra2= 7 rd2=	14 wa= 9 wd

```

=           18
950ns monitor: ra1= 8 rd1=          16 ra2= 7 rd2=          14 wa= 9 wd
=
=           18
1000ns monitor: ra1= 9 rd1=          18 ra2= 6 rd2=          12 wa=10 wd
=
=           20

```

記憶體

```

module memory(input clock, reset, en, rw,
               input [31:0] abus, input [31:0] dbus_in, output [31:
0] dbus_out);
    reg [7:0] m [0:128];
    reg [31:0] data;

    always @(clock or reset or abus or en or rw or dbus_in)
begin
    if (reset == 1) begin
        {m[0], m[1], m[2], m[3]}      = 32'h002F000C; // 0000

```

```
LD    R2, K0
      {m[4], m[5], m[6], m[7]} = 32' h001F000C; // 0004
LD    R1, K1
      {m[8], m[9], m[10], m[11]} = 32' h13221000; // 0008 LOOP:
ADD   R2, R2, R1
      {m[12], m[13], m[14], m[15]} = 32' h26FFFFF8; // 000C
JMP   LOOP
      {m[16], m[17], m[18], m[19]} = 32' h00000000; // 0010 K0:
WORD  0
      {m[20], m[21], m[22], m[23]} = 32' h00000001; // 0014 K1:
WORD  1
      data = 32' hZZZZZZZZ;
end else if (abus >=0 && abus < 128) begin
      if (en == 1 && rw == 0) // r_w==0:write
begin
      data = dbus_in;
      {m[abus], m[abus+1], m[abus+2], m[abus+3]} = dbus_in
```

```
;  
    end  
    else if (en == 1 && rw == 1) // r_w==1:read  
        data = {m[abus], m[abus+1], m[abus+2], m[abus+3]} ;  
    else  
        data = 32' hZZZZZZZZ ;  
end else  
    data = 32' hZZZZZZZZ ;  
end  
assign dbus_out = data;  
endmodule
```

```
module main;  
reg clock, reset, en, rw;  
reg [31:0] addr;  
reg [31:0] data_in;  
wire [31:0] data_out;
```

```
memory DUT (.clock(clock), .reset(reset), .en(en), .rw(rw),
    .abus(addr), .dbus_in(data_in), .dbus_out(data_out));

initial // reset: 設定 memory 內容為 0, 1, 2, ..., 127
begin
    clock = 0;
    reset = 1;
    en = 0;
    rw = 1; // rw=1:讀取模式
    #75;
    en = 1;
    reset = 0;
    addr = 0;
    #500;
    addr = 4;
    rw = 0; // 寫入模式
    data_in = 8'h3A;
    #100;
```

```
addr = 0;
rw = 1; // 讀取模式
data_in = 0;
end

always #50 begin
    clock = clock + 1;
    $monitor("%dns monitor: clk=%d en=%d rw=%d, addr=%8h din=%8h dout=%8h",
             $stime, clock, en, rw, addr, data_in, data_out);
end

always #200
begin
    addr=addr+1;
end

initial #2000 $finish;
```

```
endmodule
```

執行結果：

```
D:\Dropbox\Public\web\oc\code\verilog>iverilog -o memory32 memory32.v
```

```
D:\Dropbox\Public\web\oc\code\verilog>vvp memory32
50ns monitor: clk=1 en=0 rw=1, addr=xxxxxxxx din=xxxxxxxx dout=zzz
zzzzz
75ns monitor: clk=1 en=1 rw=1, addr=00000000 din=xxxxxxxx dout=002
f000c
100ns monitor: clk=0 en=1 rw=1, addr=00000000 din=xxxxxxxx dout=002
f000c
150ns monitor: clk=1 en=1 rw=1, addr=00000000 din=xxxxxxxx dout=002
f000c
200ns monitor: clk=0 en=1 rw=1, addr=00000001 din=xxxxxxxx dout=2f0
```

00c00

250ns monitor: clk=1 en=1 rw=1, addr=00000001 din=xxxxxxxx dout=2f0

00c00

300ns monitor: clk=0 en=1 rw=1, addr=00000001 din=xxxxxxxx dout=2f0

00c00

350ns monitor: clk=1 en=1 rw=1, addr=00000001 din=xxxxxxxx dout=2f0

00c00

400ns monitor: clk=0 en=1 rw=1, addr=00000002 din=xxxxxxxx dout=000

c001f

450ns monitor: clk=1 en=1 rw=1, addr=00000002 din=xxxxxxxx dout=000

c001f

500ns monitor: clk=0 en=1 rw=1, addr=00000002 din=xxxxxxxx dout=000

c001f

550ns monitor: clk=1 en=1 rw=1, addr=00000002 din=xxxxxxxx dout=000

c001f

575ns monitor: clk=1 en=1 rw=0, addr=00000004 din=0000003a dout=000

0003a

600ns monitor: clk=0 en=1 rw=0, addr=00000005 din=0000003a dout=000

0003a

650ns monitor: clk=1 en=1 rw=0, addr=00000005 din=0000003a dout=000

0003a

675ns monitor: clk=1 en=1 rw=1, addr=00000000 din=00000000 dout=002
f000c

700ns monitor: clk=0 en=1 rw=1, addr=00000000 din=00000000 dout=002
f000c

750ns monitor: clk=1 en=1 rw=1, addr=00000000 din=00000000 dout=002
f000c

800ns monitor: clk=0 en=1 rw=1, addr=00000001 din=00000000 dout=2f0
00c00

850ns monitor: clk=1 en=1 rw=1, addr=00000001 din=00000000 dout=2f0
00c00

900ns monitor: clk=0 en=1 rw=1, addr=00000001 din=00000000 dout=2f0
00c00

950ns monitor: clk=1 en=1 rw=1, addr=00000001 din=00000000 dout=2f0
00c00

1000ns monitor: clk=0 en=1 rw=1, addr=00000002 din=00000000 dout=000

c0000
1050ns monitor: clk=1 en=1 rw=1, addr=00000002 din=00000000 dout=000
c0000
1100ns monitor: clk=0 en=1 rw=1, addr=00000002 din=00000000 dout=000
c0000
1150ns monitor: clk=1 en=1 rw=1, addr=00000002 din=00000000 dout=000
c0000
1200ns monitor: clk=0 en=1 rw=1, addr=00000003 din=00000000 dout=0c0
00000
1250ns monitor: clk=1 en=1 rw=1, addr=00000003 din=00000000 dout=0c0
00000
1300ns monitor: clk=0 en=1 rw=1, addr=00000003 din=00000000 dout=0c0
00000
1350ns monitor: clk=1 en=1 rw=1, addr=00000003 din=00000000 dout=0c0
00000
1400ns monitor: clk=0 en=1 rw=1, addr=00000004 din=00000000 dout=000
00000
1450ns monitor: clk=1 en=1 rw=1, addr=00000004 din=00000000 dout=000

00000
1500ns monitor: clk=0 en=1 rw=1, addr=00000004 din=00000000 dout=000
00000
1550ns monitor: clk=1 en=1 rw=1, addr=00000004 din=00000000 dout=000
00000
1600ns monitor: clk=0 en=1 rw=1, addr=00000005 din=00000000 dout=000
0003a
1650ns monitor: clk=1 en=1 rw=1, addr=00000005 din=00000000 dout=000
0003a
1700ns monitor: clk=0 en=1 rw=1, addr=00000005 din=00000000 dout=000
0003a
1750ns monitor: clk=1 en=1 rw=1, addr=00000005 din=00000000 dout=000
0003a
1800ns monitor: clk=0 en=1 rw=1, addr=00000006 din=00000000 dout=000
03a22
1850ns monitor: clk=1 en=1 rw=1, addr=00000006 din=00000000 dout=000
03a22
1900ns monitor: clk=0 en=1 rw=1, addr=00000006 din=00000000 dout=000

```
03a22
```

```
1950ns monitor: clk=1 en=1 rw=1, addr=00000006 din=00000000 dout=000
```

```
03a22
```

```
2000ns monitor: clk=0 en=1 rw=1, addr=00000007 din=00000000 dout=003
```

```
a2210
```

結語

在本章中，我們介紹了正反器 (Flip-Flop, Latch, 栓鎖器) 等循序電路的概念，並且利用閘級延遲的現象，設計出了「脈衝偵測電路」(Pause Transition Detector, PTD)，於是我們可以利用「脈衝偵測電路」設計出像「邊緣觸發型」的電路，像是「邊緣觸發型」的正反器、暫存器、計數器等等電路，這些電路是構成電腦當中的記憶線路的基礎。

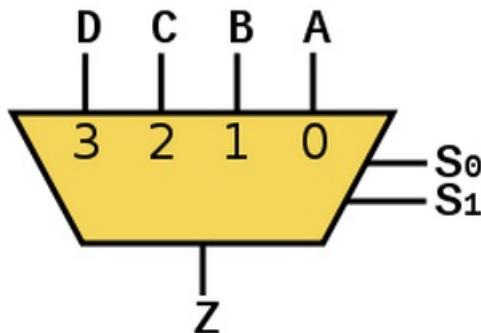
一但理解這些基礎原理之後，我們就可以用 Verilog 的高階語法直接宣告「暫存器、一群暫存器與一整塊記憶體」，這種高階寫法已經非常接近高階語言的寫法，只是由於是設計硬體，所以這些高階指令最後都會被轉換為線路，燒錄到 FPGA 或內建於 ASIC 裡面而已，如此我們就不需要用一條一條的線路去兜出暫存器或記憶體，可以輕鬆的透過 Verilog 設計出記憶單元了。

控制單元

簡介

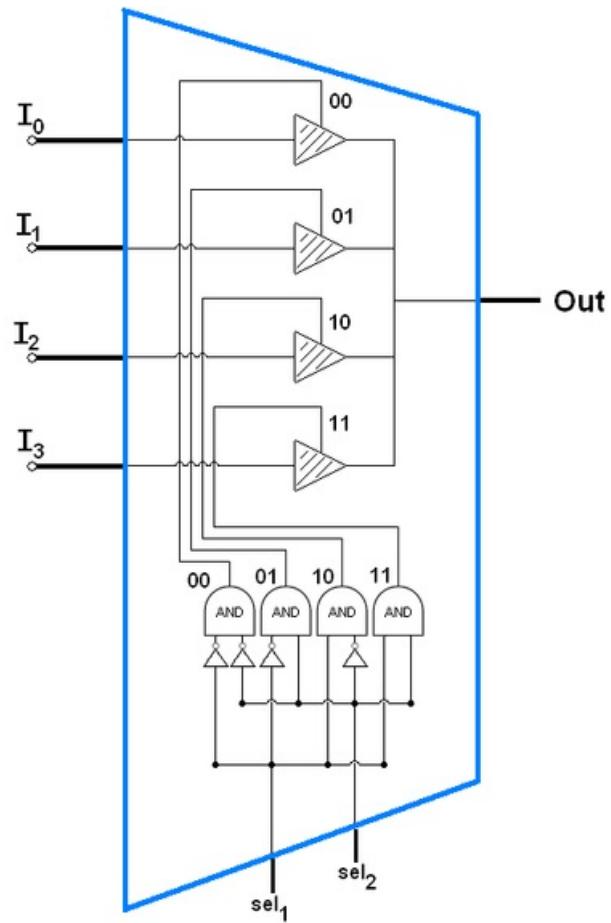
如果您曾經用硬接線的方式設計過 CPU，那就會發現「控制單元」主要就是一堆「開關」與「多工器」的接線。

開關可以用來控制某些資料是否要流過，而多工器則可以從很多組輸入資料中選擇一組輸出，以下是一個四選一多工器的方塊圖。



圖、4 選 1 多工器

4 選 1 多工器的內部電路結構如下：



圖、4 選 1 多工器的內部電路

接著、就讓我們來看一個完整的 Verilog 的 4 選 1 的多工器程式，由於 Verilog 支援像 Case 這樣的高階語法，因此在實作時可以不需要採用細部的接線方式，只要使用 case 語句就可以輕易完成多工器的設計。

檔案：[mux4.v](#)

```
module mux4(input[1:0] select, input[3:0] d, output reg q );
always @( select or d )
begin
  case( select )
    0 : q = d[0];
    1 : q = d[1];
    2 : q = d[2];
    3 : q = d[3];
  endcase
end
endmodule
```

```
module main;
reg [3:0] d;
reg [1:0] s;
wire q;

mux4 DUT (s, d, q);

initial
begin
    s = 0;
    d = 4'b0110;
end

always #50 begin
    s=s+1;
    $monitor("%4dns monitor: s=%d d=%d q=%d", $stime, s, d, q);
end
```

```
initial #1000 $finish;  
  
endmodule
```

執行結果

```
D:\ccc101\icarus>iverilog mux4.v -o mux4
```

```
D:\ccc101\icarus>vvp mux4
```

```
50ns monitor: s=1 d= 6 q=1  
100ns monitor: s=2 d= 6 q=1  
150ns monitor: s=3 d= 6 q=0  
200ns monitor: s=0 d= 6 q=0  
250ns monitor: s=1 d= 6 q=1  
300ns monitor: s=2 d= 6 q=1  
350ns monitor: s=3 d= 6 q=0  
400ns monitor: s=0 d= 6 q=0  
450ns monitor: s=1 d= 6 q=1  
500ns monitor: s=2 d= 6 q=1
```

```
550ns monitor: s=3 d= 6 q=0
600ns monitor: s=0 d= 6 q=0
650ns monitor: s=1 d= 6 q=1
700ns monitor: s=2 d= 6 q=1
750ns monitor: s=3 d= 6 q=0
800ns monitor: s=0 d= 6 q=0
850ns monitor: s=1 d= 6 q=1
900ns monitor: s=2 d= 6 q=1
950ns monitor: s=3 d= 6 q=0
1000ns monitor: s=0 d= 6 q=0
```

您可以看到在上述範例中，輸入資料 6 的二進位是 0110，如下所示：

位置	s	3	2	1	0
位元	d	0	1	1	0

因此當 $s=0$ 時會輸出 0, $s=1$ 時會輸出 1, $s=2$ 時會輸出 1, $s=3$ 時會輸出 0，這就是上述輸出結果的意義。

但是、這種採用多工器硬的接線方式，必須搭配區塊式的設計，才能建構出 CPU，但是這種方式較

為困難，因此我們留待後續章節再來介紹。為了簡單起見，我們會先採用流程式的設計方法。

流程式設計

傳統上、當您設計出「ALU、暫存器」等基本元件之後，就可以設計控制單元，去控制這些「基本元件」，形成一顆 CPU。

但是、在 Verilog 當中，「 $+, -, *, /$, 暫存器」等都是基本語法，因此整個 CPU 的設計其實就是一個控制單元的設計而已，我們只要在適當的時候呼叫「 $+, -, *, /$ 」運算與「暫存器讀取寫入」功能，就能設計完一顆 CPU 了。

換句話說，只要在 Verilog 中設計出控制單元，基本上就已經設計完成整顆 CPU 了，因為「 $+, -, *, /$, 暫存器」等元件都已經內建了。

以下是我們用流程法設計 mcu0 微處理器的重要程式片段，您可以看到在這種作法上，整個處理器就僅僅是一個「控制單元」，而這個「控制單元」的責任就是根據「擷取、解碼、執行」的流程，操控暫存器的流向與運算。

```
module cpu(input clock);
  ...
  always @(posedge clock) begin // 在 clock 時脈的正邊緣時觸發
```

IR = {m[PC], m[PC+1]} ; // 指令擷取階段: IR=m[PC], 2 個 Byte 的
記憶體

```
pc0= PC;           // 儲存舊的 PC 值在 pc0 中。
PC = PC+2;         // 擷取完成, PC 前進到下一個指令位址
case (`OP)          // 解碼、根據 OP 執行動作
  LD: A = `M;      // LD C
  ST: `M = A;       // ST C
  CMP: begin `N=(A < `M); `Z=(A==`M); end // CMP C
  ADD: A = A + `M;   // ADD C
  JMP: PC = `C;       // JMP C
  JEQ: if (`Z) PC=`C; // JEQ C
  ...
endcase
// 印出 PC, IR, SW, A 等暫存器值以供觀察
$display ("%4dns PC=%x IR=%x, SW=%x, A=%d", $stime, pc0, IR, SW
, A);
end
endmodule
```

區塊式設計

```
module mcu(input clock);
    ...
    register#.W(12) PC(clock, 1, pci, pco);
    adder#.W(12) adder0(2, pco, pcnext);
    memory mem(mw, `C, ao, pco, ir, `C, mo);
    register#.W(16) A(~clock, aw, aluout, ao);
    register#.W(16) SW(~clock, sww, aluout, swo);
    alu alu0(aluop, mo, ao, aluout);
    mux#.W(12) muxpc(pcmux, pcnext, `C, pci);
    control cu(`OP, `Z, mw, aw, pcmux, sww, aluop);
    ...
endmodule
```

以流程法設計控制單元

然而、當指令愈來愈多，系統愈來愈複雜時，區塊式的設計方法就會愈來愈困難，此時有兩種解決方式，一種是採用流程式的設計法來撰寫控制單元，操控各種「開關與多工器」。這種設計方法混合了「區塊式與流程式」的設計方法，算是一種折衷性的方法。

```
module control(input [3:0] op, input z, output mw, aw, pcmux, sww,  
output [3:0] aluop);  
  
reg mw, aw, pcmux, sww;  
reg [3:0] aluop;  
  
always @(op) begin  
  
mw = 0;  
aw = 0;  
sww = 0;  
pcmux = 0;  
aluop = alu0.ZERO;  
  
case (op)  
    mcu0.LD: begin aw=1; aluop=alu0.APASS; end      // LD C  
    mcu0.ST: mw=1;                                // ST C  
    mcu0.JMP: pcmux=1;                            // JMP C
```

```

mcu0.JEQ: if (^Z) pcmux=1; // JEQ C
mcu0.CMP: begin sww=1; aluop = alu0.CMP; end // CMP C
mcu0.ADD: begin aw=1; aluop=alu0.ADD; end // ADD C
endcase
end
endmodule

```

以區塊法設計控制單元

當然、我們也可以將上述的控制訊號硬是用 and, or, not 等方式寫下來，這樣就能將整個設計完全「區塊化」，而去掉「流程式」的寫法了。

```

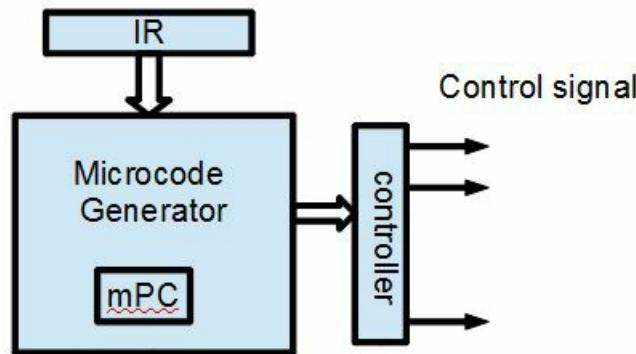
module control(input [3:0] op, input z, output mw, aw, pcmux, sww,
output [3:0] aluop);
assign mw=(op==mcu0.ST);
assign aw=(op==mcu0.LD || op==mcu0.ADD);
assign sww=(op==mcu0.CMP);
assign pcmux=(op==mcu0.JMP || (op==mcu0.JEQ && z));
assign aluop=(op==mcu0.LD)?alu0.APASS:(op==mcu0.CMP)?alu0.CMP:(op

```

```
==mcu0.ADD)?alu0.ADD:alu0.ZERO;  
endmodule
```

以微指令設計控制單元

另一種可以克服區塊式複雜度問題的方法，是採用「微指令」(microcode or microprogram)的設計方法，這種方法將指令的「擷取 (fetch)、解碼 (decode)、執行 (execute) 與寫回 (write-back)」等分成 T1, T2, T3, T4, ... 等子步驟，然後用一個微型計數器 mPC 控制這些子步驟的執行，如下圖所示。



由於 T1, T2, T3, T4 代表「擷取 (fetch)、解碼 (decode)、執行 (execute) 與寫回 (write-back)」，每個步驟各佔用一個 Clock，於是一個指令需要四個 Clock 才能完成，因此我們可以用以下方法將各個「開關與多工器」的控制編為一個表格。

C1 C2 C3 Ck

T1

T2

T3

T4

這樣就可以很有系統的運用「區塊建構法」將控制單元也區塊化了，於是我們就不需要採用「流程式的寫法」，也能透過「按表操課」的方法完成「處理器的區塊式建構」了。

微處理器 (Micro Processor)

現代的處理器通常會被分為「微處理器」與「高階處理器」兩類。

微處理器指的是能力較弱，通常指令長度較短，以 8 位於為主，少數為 16 位元，像是 8051、AVR8 或 PIC 等處理器，這些處理器通常不會搭配外部記憶體，因此常用在「單晶片」或者「嵌入式系統」當中。

而高階處理器則是能力較強，通常指令寬度較大 (32 位元以上)，像是 Intel 的 x86 等等，高階的 ARM11、ARM Cortex 處理器等等，這些處理器的定址空間很大，可以搭配容量很大的記憶體 (ex: 4GB)，而且速度很快，因此需要內建「記憶體管理單元」(Memory Management Unit, MMU) 與多層快取機制 (cache)，以便能夠充分發揮處理器的效能。

在本章當中，我們將透過 mcu0 這個簡易的架構，說明「微處理器」的設計方法。

當然、「微處理器」的設計方法有很多種，在本章中我們將展示兩種設計方法，一種是「流程式」的設計法，這種方法比較符合「軟體程式人」的設計習慣，寫起來有點類似 C 語言的寫法。

另一種是「區塊式」的設計法，這種方法是傳統「硬體人」的設計方法，採用像積木一樣的方式由下往上拼接出來，然後再透過線路連接各個區塊形成一個更大的區塊。

MCU0 的迷你版 -- mcu0m

MCU0 處理器

MCU0 為一個 16 位元處理器，包含指令暫存器 IR，程式計數器 PC，狀態佔存器 SW 與累積器 A 等四個暫存器。

指令表

OP	name	格式	意 義
0	LD	LD C	$A = [C]$
1	ADD	ADD C	$A = A + [C]$
2	JMP	JMP C	$PC = C$
3	ST	ST C	$[C] = A$
4	CMP	CMP C	$SW = A \text{ CMP } [C]$
5	JEQ	JEQ C	if $SW[30]=Z=1$ then $PC = C$

組合語言與機器碼

00	LOOP:	LD I	0016
02		CMP K10	401A
04		JEQ EXIT	5012
06		ADD K1	1018
08		ST I	3016
0A		LD SUM	0014
0C		ADD I	1016
0E		ST SUM	3014
10		JMP LOOP	2000
12	EXIT:	JMP EXIT	2012
14	SUM:	WORD 0	0000
16	I:	WORD 0	0000
18	K1:	WORD 1	0001
1A	K10:	WORD 10	000A

轉成 Hex 輸入檔格式：

檔案：mcu0m.hex

00 16 // 00	LOOP:	LD	I
40 1A // 02		CMP	K10
50 12 // 04		JEQ	EXIT實作
10 18 // 06		ADD	K1
30 16 // 08		ST	I
00 14 // 0A		LD	SUM
10 16 // 0C		ADD	I
30 14 // 0E		ST	SUM
20 00 // 10		JMP	LOOP
20 12 // 12	EXIT:	JMP	EXIT
00 00 // 14	SUM:	WORD	0
00 00 // 16	I:	WORD	0
00 01 // 18	K1:	WORD	1
00 0A // 1A	K10:	WORD	10

Verilog 程式實作

檔案：mcu0m.v

```
`define N      SW[15] // 負號旗標
`define Z      SW[14] // 零旗標
`define OP     IR[15:12] // 運算碼
`define C      IR[11:0]  // 常數欄位
`define M      {m[`C], m[`C+1]}

module cpu(input clock); // CPU0-Mini 的快取版: cpu0mc 模組
  parameter [3:0] LD=4'h0, ADD=4'h1, JMP=4'h2, ST=4'h3, CMP=4'h4, JEQ=4'h5;
  reg signed [15:0] A;    // 宣告暫存器 R[0..15] 等 16 個 32 位元暫存器
  reg [15:0] IR;    // 指令暫存器 IR
  reg [15:0] SW;    // 指令暫存器 IR
  reg [15:0] PC;    // 程式計數器
  reg [15:0] pc0;
  reg [7:0]  m [0:32]; // 內部的快取記憶體
  integer i;
```

```
initial // 初始化
begin
    PC = 0; // 將 PC 設為起動位址 0
    SW = 0;
    $readmemh("mcu0m.hex", m);
    for (i=0; i < 32; i=i+2) begin
        $display("%8x: %8x", i, {m[i], m[i+1]});
    end
end

always @(posedge clock) begin // 在 clock 時脈的正邊緣時觸發
    IR = {m[PC], m[PC+1]}; // 指令擷取階段: IR=m[PC], 2 個 Byte 的
記憶體
    pc0= PC; // 儲存舊的 PC 值在 pc0 中。
    PC = PC+2; // 擷取完成, PC 前進到下一個指令位址
    case (^OP) // 解碼、根據 OP 執行動作
        LD: A = `M; // LD C
        ST: `M = A; // ST C
```

```
CMP: begin `N=(A < `M); `Z=(A==`M); end // CMP C
ADD: A = A + `M;           // ADD C
JMP: PC = `C;             // JMP C
JEQ: if (`Z) PC=`C;      // JEQ C
endcase
// 印出 PC, IR, SW, A 等暫存器值以供觀察
$display ("%4dns PC=%x IR=%x, SW=%x, A=%d", $stime, pc0, IR, SW
, A);
end
endmodule

module main;                  // 測試程式開始
reg clock;                   // 時脈 clock 變數

cpu cpux(clock);            // 宣告 cpu0mc 處理器

initial clock = 0;           // 一開始 clock 設定為 0
always #10 clock=~clock;     // 每隔 10ns 反相，時脈週期為 20ns
```

```
initial #2000 $finish;           // 在 2000 奈秒的時候停止測試。  
endmodule
```

執行結果

```
D:\Dropbox\Public\web\oc\code>iiverilog -o mcu0m mcu0m.v
```

```
D:\Dropbox\Public\web\oc\code>vvp cpu16m
```

```
WARNING: cpu16m.v:20: $readmemh(cpu16m.hex): Not enough words in the  
file for th  
e requested range [0:32].
```

```
00000000:    0016
```

```
00000002:    401a
```

```
00000004:    5012
```

```
00000006:    1018
```

```
00000008:    3016
```

```
0000000a:    0014
```

```
0000000c:    1016
```

0000000e:	3014	
00000010:	2000	
00000012:	2012	
00000014:	0000	
00000016:	0000	
00000018:	0001	
0000001a:	000a	
0000001c:	xxxx	
0000001e:	xxxx	
10ns	PC=0000 IR=0016, SW=0000, A=	0
30ns	PC=0002 IR=401a, SW=8000, A=	0
50ns	PC=0004 IR=5012, SW=8000, A=	0
70ns	PC=0006 IR=1018, SW=8000, A=	1
90ns	PC=0008 IR=3016, SW=8000, A=	1
110ns	PC=000a IR=0014, SW=8000, A=	0
130ns	PC=000c IR=1016, SW=8000, A=	1
150ns	PC=000e IR=3014, SW=8000, A=	1
170ns	PC=0010 IR=2000, SW=8000, A=	1

190ns	PC=0000	IR=0016,	SW=8000,	A=	1
210ns	PC=0002	IR=401a,	SW=8000,	A=	1
230ns	PC=0004	IR=5012,	SW=8000,	A=	1
250ns	PC=0006	IR=1018,	SW=8000,	A=	2
270ns	PC=0008	IR=3016,	SW=8000,	A=	2
290ns	PC=000a	IR=0014,	SW=8000,	A=	1
310ns	PC=000c	IR=1016,	SW=8000,	A=	3
330ns	PC=000e	IR=3014,	SW=8000,	A=	3
350ns	PC=0010	IR=2000,	SW=8000,	A=	3
370ns	PC=0000	IR=0016,	SW=8000,	A=	2
390ns	PC=0002	IR=401a,	SW=8000,	A=	2
410ns	PC=0004	IR=5012,	SW=8000,	A=	2
430ns	PC=0006	IR=1018,	SW=8000,	A=	3
450ns	PC=0008	IR=3016,	SW=8000,	A=	3
470ns	PC=000a	IR=0014,	SW=8000,	A=	3
490ns	PC=000c	IR=1016,	SW=8000,	A=	6
510ns	PC=000e	IR=3014,	SW=8000,	A=	6
530ns	PC=0010	IR=2000,	SW=8000,	A=	6

550ns	PC=0000	IR=0016,	SW=8000,	A=	3
570ns	PC=0002	IR=401a,	SW=8000,	A=	3
590ns	PC=0004	IR=5012,	SW=8000,	A=	3
610ns	PC=0006	IR=1018,	SW=8000,	A=	4
630ns	PC=0008	IR=3016,	SW=8000,	A=	4
650ns	PC=000a	IR=0014,	SW=8000,	A=	6
670ns	PC=000c	IR=1016,	SW=8000,	A=	10
690ns	PC=000e	IR=3014,	SW=8000,	A=	10
710ns	PC=0010	IR=2000,	SW=8000,	A=	10
730ns	PC=0000	IR=0016,	SW=8000,	A=	4
750ns	PC=0002	IR=401a,	SW=8000,	A=	4
770ns	PC=0004	IR=5012,	SW=8000,	A=	4
790ns	PC=0006	IR=1018,	SW=8000,	A=	5
810ns	PC=0008	IR=3016,	SW=8000,	A=	5
830ns	PC=000a	IR=0014,	SW=8000,	A=	10
850ns	PC=000c	IR=1016,	SW=8000,	A=	15
870ns	PC=000e	IR=3014,	SW=8000,	A=	15
890ns	PC=0010	IR=2000,	SW=8000,	A=	15

910ns	PC=0000	IR=0016,	SW=8000,	A=	5
930ns	PC=0002	IR=401a,	SW=8000,	A=	5
950ns	PC=0004	IR=5012,	SW=8000,	A=	5
970ns	PC=0006	IR=1018,	SW=8000,	A=	6
990ns	PC=0008	IR=3016,	SW=8000,	A=	6
1010ns	PC=000a	IR=0014,	SW=8000,	A=	15
1030ns	PC=000c	IR=1016,	SW=8000,	A=	21
1050ns	PC=000e	IR=3014,	SW=8000,	A=	21
1070ns	PC=0010	IR=2000,	SW=8000,	A=	21
1090ns	PC=0000	IR=0016,	SW=8000,	A=	6
1110ns	PC=0002	IR=401a,	SW=8000,	A=	6
1130ns	PC=0004	IR=5012,	SW=8000,	A=	6
1150ns	PC=0006	IR=1018,	SW=8000,	A=	7
1170ns	PC=0008	IR=3016,	SW=8000,	A=	7
1190ns	PC=000a	IR=0014,	SW=8000,	A=	21
1210ns	PC=000c	IR=1016,	SW=8000,	A=	28
1230ns	PC=000e	IR=3014,	SW=8000,	A=	28
1250ns	PC=0010	IR=2000,	SW=8000,	A=	28

1270ns	PC=0000	IR=0016,	SW=8000,	A=	7
1290ns	PC=0002	IR=401a,	SW=8000,	A=	7
1310ns	PC=0004	IR=5012,	SW=8000,	A=	7
1330ns	PC=0006	IR=1018,	SW=8000,	A=	8
1350ns	PC=0008	IR=3016,	SW=8000,	A=	8
1370ns	PC=000a	IR=0014,	SW=8000,	A=	28
1390ns	PC=000c	IR=1016,	SW=8000,	A=	36
1410ns	PC=000e	IR=3014,	SW=8000,	A=	36
1430ns	PC=0010	IR=2000,	SW=8000,	A=	36
1450ns	PC=0000	IR=0016,	SW=8000,	A=	8
1470ns	PC=0002	IR=401a,	SW=8000,	A=	8
1490ns	PC=0004	IR=5012,	SW=8000,	A=	8
1510ns	PC=0006	IR=1018,	SW=8000,	A=	9
1530ns	PC=0008	IR=3016,	SW=8000,	A=	9
1550ns	PC=000a	IR=0014,	SW=8000,	A=	36
1570ns	PC=000c	IR=1016,	SW=8000,	A=	45
1590ns	PC=000e	IR=3014,	SW=8000,	A=	45
1610ns	PC=0010	IR=2000,	SW=8000,	A=	45

1630ns	PC=0000	IR=0016,	SW=8000,	A=	9
1650ns	PC=0002	IR=401a,	SW=8000,	A=	9
1670ns	PC=0004	IR=5012,	SW=8000,	A=	9
1690ns	PC=0006	IR=1018,	SW=8000,	A=	10
1710ns	PC=0008	IR=3016,	SW=8000,	A=	10
1730ns	PC=000a	IR=0014,	SW=8000,	A=	45
1750ns	PC=000c	IR=1016,	SW=8000,	A=	55
1770ns	PC=000e	IR=3014,	SW=8000,	A=	55
1790ns	PC=0010	IR=2000,	SW=8000,	A=	55
1810ns	PC=0000	IR=0016,	SW=8000,	A=	10
1830ns	PC=0002	IR=401a,	SW=4000,	A=	10
1850ns	PC=0004	IR=5012,	SW=4000,	A=	10
1870ns	PC=0012	IR=2012,	SW=4000,	A=	10
1890ns	PC=0012	IR=2012,	SW=4000,	A=	10
1910ns	PC=0012	IR=2012,	SW=4000,	A=	10
1930ns	PC=0012	IR=2012,	SW=4000,	A=	10
1950ns	PC=0012	IR=2012,	SW=4000,	A=	10
1970ns	PC=0012	IR=2012,	SW=4000,	A=	10

MCU0 的區塊式設計 -- MCU0bm.v

前言

我們曾經在下列文章中設計出了 MCU0 迷你版這個只有六個指令的微控制器，整個實作只有 51 行。

- ### • 開放電腦計畫(6) – 一顆只有 51 行 Verilog 程式碼的 16 位元處理器 MCU0

但是、上述程式雖然簡單，但卻是採用流程式的寫法。雖然、筆者不覺得流程式的寫法有甚麼特別的缺陷，但是對那些習慣採用硬體角度設計 Verilog 程式的人而言，似乎採用「區塊式的設計方式」才是正統，所以、筆者將於本文中採用「區塊式的方式重新設計」MCU0 迷你版，以便能學習「硬體設計者」的思考方式。

MCU0 迷你版的指令表

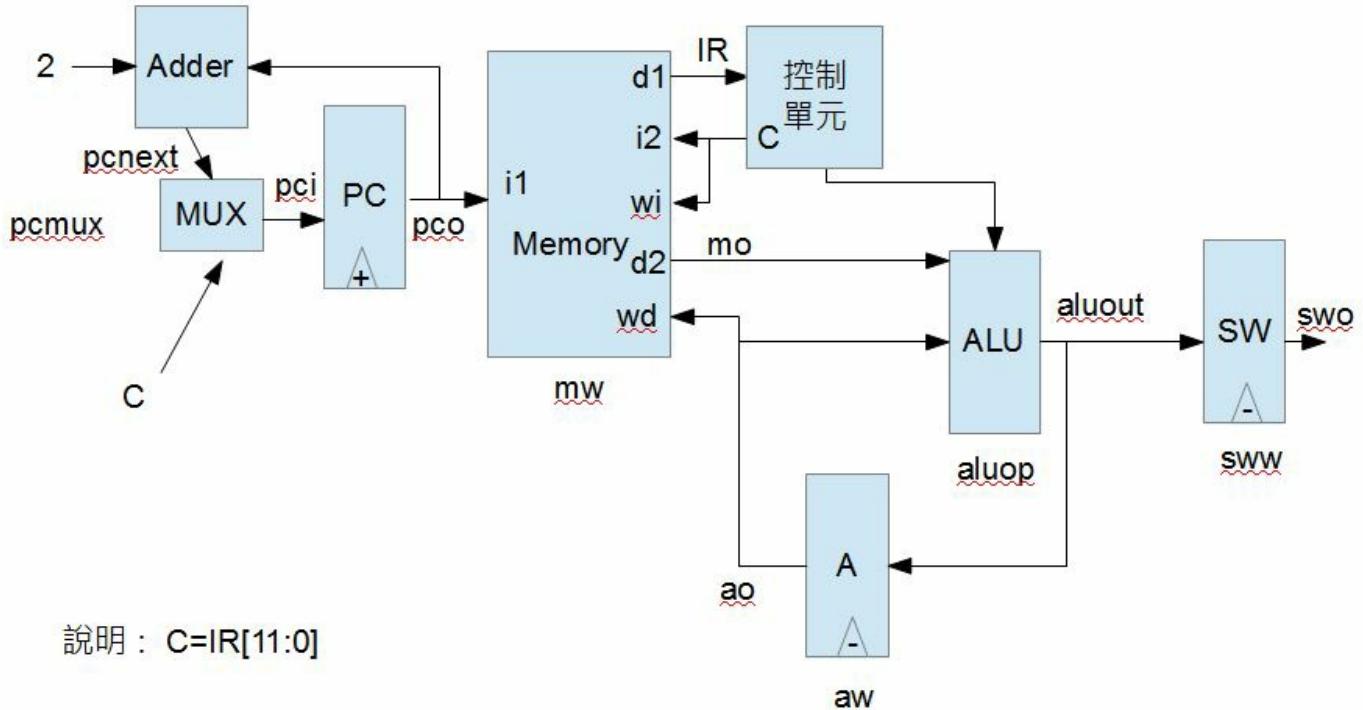
為了方便讀者閱讀，不需要查閱前文，我們再次列出了 MCU0 迷你版的指令表如下：

OP name 格式 意 義

0	LD	LD C	$A = [C]$
1	ADD	ADD C	$A = A + [C]$
2	JMP	JMP C	$PC = C$
3	ST	ST C	$[C] = A$
4	CMP	CMP C	$SW = A \text{ CMP } [C]$
5	JEQ	JEQ C	if $SW[30]=Z=1$ then $PC = C$

MCU0 迷你版的區塊設計圖

在MCU0 迷你版裏，總共有三個暫存器，分別是 A, PC 與 SW，一個具有兩組讀取 ($i1/d1, i2/d2$) 與一組寫入的記憶體 (wi/wd)，還有一個算術邏輯單元 ALU，這個電路的設計圖如下。



圖、MCU0bm 的區塊設計圖

由於筆者不熟悉數位電路設計的繪圖軟體，因此就簡單的用 LibreOffice 的 Impress 繪製了上圖，純粹採用區塊表達法，並沒有使用標準的數位電路設計圖示。

原始碼

根據上圖，我們設計出了下列 Verilog 程式，您應該可以很清楚的找出程式與圖形之間的對應關係。

```
module memory(input w, input [11:0] wi, input [15:0] wd, input [11:0] i1, output [15:0] d1, input [11:0] i2, output [15:0] d2);
    integer i;
    reg [7:0] m[0:2**12-1];
    initial begin
        $readmemh("mcu0m.hex", m);
        for (i=0; i < 32; i=i+2) begin
            $display("%x: %x", i, {m[i], m[i+1]});
        end
    end
    assign d1 = {m[i1], m[i1+1]};
    assign d2 = {m[i2], m[i2+1]};
    always @(w) begin
        if (w) {m[wi], m[wi+1]} = wd;
    end
endmodule
```

```
    end  
endmodule  
  
module adder#(parameter W=16) (input [W-1:0] a, input [W-1:0] b, output [W-1:0] c);  
    assign c = a + b;  
endmodule  
  
module register#(parameter W=16) (input clock, w, input [W-1:0] ri,  
output [W-1:0] ro);  
reg [W-1:0] r;  
always @(posedge clock) begin  
    if (w) r = ri;  
end  
assign ro=r;  
endmodule  
  
module alu(input [3:0] op, input [15:0] a, input [15:0] b, output r
```

```
eg [15:0] c;
parameter [3:0] ZERO=4' h0, ADD=4' h1, CMP=4' he, APASS=4' hf;
always @(*) begin
  case (op)
    ADD: c = a+b;
    CMP: begin c[15]=(a < b); c[14]=(a==b); c[13:0]=14' h0; end
    APASS: c = a;
    default: c = 0;
  endcase
end
endmodule
```

```
module mux#(parameter W=16)(input sel, input [W-1:0] i0, i1, output
[W-1:0] o);
  assign o=(sel)?i1:i0;
endmodule
```

```
`define OP ir[15:12]
```

```

`define C  ir[11:0]
`define N  SW.r[15]
`define Z  SW.r[14]

module control(input [3:0] op, input z, output mw, aw, pcmux, sww,
output [3:0] aluop);
    assign mw=(op==mcu0. ST) ;
    assign aw=(op==mcu0. LD || op==mcu0. ADD) ;
    assign sww=(op==mcu0. CMP) ;
    assign pcmux=(op==mcu0. JMP || (op==mcu0. JEQ && z)) ;
    assign aluop=(op==mcu0. LD)?alu0. APASS:(op==mcu0. CMP)?alu0. CMP:(op
==mcu0. ADD)?alu0. ADD:alu0. ZERO;
endmodule

module mcu(input clock);
    parameter [3:0] LD=4' h0, ADD=4' h1, JMP=4' h2, ST=4' h3, CMP=4' h4, JEQ=4' h
5;
    wire mw, aw, pcmux, sww;

```

```
wire [3:0] aluop;
wire [11:0] pco, pci, pcnext;
wire [15:0] aluout, ao, swo, ir, mo;

register#.W(12) PC(clock, 1, pci, pco);
adder#.W(12) adder0(2, pco, pcnext);
memory mem(mw, `C, ao, pco, ir, `C, mo);
register#.W(16) A(~clock, aw, aluout, ao);
register#.W(16) SW(~clock, sww, aluout, swo);
alu alu0(aluop, mo, ao, aluout);
mux#.W(12) muxpc(pcmux, pcnext, `C, pci);
control cu(`OP, `Z, mw, aw, pcmux, sww, aluop);

initial begin
    PC.r = 0; SW.r = 0;
end
endmodule
```

```
module main;          // 測試程式開始
reg clock;           // 時脈 clock 變數

mcu mcu0(clock);

initial begin
    clock = 0;
    $monitor("%4dns pc=%x ir=%x mo=%x sw=%x a=%d mw=%b aluout=%x", $time, mcu0.PC.r, mcu0.ir, mcu0.mo, mcu0.SW.r, mcu0.A.r, mcu0.mw, mcu0.aluout);
    #1000 $finish;
end
always #5 begin
    clock=~clock;    // 每隔 5ns 反相，時脈週期為 10ns
end
endmodule
```

輸入的機器碼 mcu0m.hex

為了測試上述程式，我們同樣採用了計算 $SUM=1+2+\dots+10$ 的這個程式作為輸入，以下是機器碼與對應的組合語言程式。

00 16 // 00	LOOP:	LD	I
40 1A // 02		CMP	N
50 12 // 04		JEQ	EXIT
10 18 // 06		ADD	K1
30 16 // 08		ST	I
00 14 // 0A		LD	SUM
10 16 // 0C		ADD	I
30 14 // 0E		ST	SUM
20 00 // 10		JMP	LOOP
20 12 // 12	EXIT:	JMP	EXIT
00 00 // 14	SUM:	WORD	0
00 00 // 16	I:	WORD	0
00 01 // 18	K1:	WORD	1
00 0A // 1A	N:	WORD	10

執行結果

編寫完成之後，我們就可以測試整個 mcu0bm.v 程式了，其執行結果如下所示。

```
D:\Dropbox\Public\web\co\code\mcu0>iverilog mcu0bm.v -o mcu0bm
```

```
D:\Dropbox\Public\web\co\code\mcu0>vvp mcu0bm
```

```
WARNING: mcu0bm.v:5: $readmemh(mcu0m.hex): Not enough words in the file for the requested range [0:4095].
```

```
00000000: 0016
```

```
00000002: 401a
```

```
00000004: 5012
```

```
00000006: 1018
```

```
00000008: 3016
```

```
0000000a: 0014
```

```
0000000c: 1016
```

```
0000000e: 3014
```

00000010:	2000							
00000012:	2012							
00000014:	0000							
00000016:	0000							
00000018:	0001							
0000001a:	000a							
0000001c:	xxxx							
0000001e:	xxxx							
0ns	pc=000	ir=0016	mo=0000	sw=0000	a=	0	mw=0	aluout=0000
5ns	pc=002	ir=401a	mo=000a	sw=0000	a=	0	mw=0	aluout=0000
15ns	pc=004	ir=5012	mo=2012	sw=0000	a=	0	mw=0	aluout=0000
25ns	pc=006	ir=1018	mo=0001	sw=0000	a=	0	mw=0	aluout=0001
30ns	pc=006	ir=1018	mo=0001	sw=0000	a=	1	mw=0	aluout=0002
35ns	pc=008	ir=3016	mo=0001	sw=0000	a=	1	mw=1	aluout=0000
45ns	pc=00a	ir=0014	mo=0000	sw=0000	a=	1	mw=0	aluout=0000
50ns	pc=00a	ir=0014	mo=0000	sw=0000	a=	0	mw=0	aluout=0000
55ns	pc=00c	ir=1016	mo=0001	sw=0000	a=	0	mw=0	aluout=0001
60ns	pc=00c	ir=1016	mo=0001	sw=0000	a=	1	mw=0	aluout=0002

65ns	pc=00e	ir=3014	mo=0001	sw=0000	a=	1	mw=1	aluout=0000
75ns	pc=010	ir=2000	mo=0016	sw=0000	a=	1	mw=0	aluout=0000
85ns	pc=000	ir=0016	mo=0001	sw=0000	a=	1	mw=0	aluout=0001
95ns	pc=002	ir=401a	mo=000a	sw=0000	a=	1	mw=0	aluout=0000
105ns	pc=004	ir=5012	mo=2012	sw=0000	a=	1	mw=0	aluout=0000
115ns	pc=006	ir=1018	mo=0001	sw=0000	a=	1	mw=0	aluout=0002
120ns	pc=006	ir=1018	mo=0001	sw=0000	a=	2	mw=0	aluout=0003
125ns	pc=008	ir=3016	mo=0002	sw=0000	a=	2	mw=1	aluout=0000
135ns	pc=00a	ir=0014	mo=0001	sw=0000	a=	2	mw=0	aluout=0001
140ns	pc=00a	ir=0014	mo=0001	sw=0000	a=	1	mw=0	aluout=0001
145ns	pc=00c	ir=1016	mo=0002	sw=0000	a=	1	mw=0	aluout=0003
150ns	pc=00c	ir=1016	mo=0002	sw=0000	a=	3	mw=0	aluout=0005
155ns	pc=00e	ir=3014	mo=0003	sw=0000	a=	3	mw=1	aluout=0000
165ns	pc=010	ir=2000	mo=0016	sw=0000	a=	3	mw=0	aluout=0000
175ns	pc=000	ir=0016	mo=0002	sw=0000	a=	3	mw=0	aluout=0002
180ns	pc=000	ir=0016	mo=0002	sw=0000	a=	2	mw=0	aluout=0002
185ns	pc=002	ir=401a	mo=000a	sw=0000	a=	2	mw=0	aluout=0000
195ns	pc=004	ir=5012	mo=2012	sw=0000	a=	2	mw=0	aluout=0000

205ns	pc=006	ir=1018	mo=0001	sw=0000	a=	2	mw=0	aluout=0003
210ns	pc=006	ir=1018	mo=0001	sw=0000	a=	3	mw=0	aluout=0004
215ns	pc=008	ir=3016	mo=0003	sw=0000	a=	3	mw=1	aluout=0000
225ns	pc=00a	ir=0014	mo=0003	sw=0000	a=	3	mw=0	aluout=0003
235ns	pc=00c	ir=1016	mo=0003	sw=0000	a=	3	mw=0	aluout=0006
240ns	pc=00c	ir=1016	mo=0003	sw=0000	a=	6	mw=0	aluout=0009
245ns	pc=00e	ir=3014	mo=0006	sw=0000	a=	6	mw=1	aluout=0000
255ns	pc=010	ir=2000	mo=0016	sw=0000	a=	6	mw=0	aluout=0000
265ns	pc=000	ir=0016	mo=0003	sw=0000	a=	6	mw=0	aluout=0003
270ns	pc=000	ir=0016	mo=0003	sw=0000	a=	3	mw=0	aluout=0003
275ns	pc=002	ir=401a	mo=000a	sw=0000	a=	3	mw=0	aluout=0000
285ns	pc=004	ir=5012	mo=2012	sw=0000	a=	3	mw=0	aluout=0000
295ns	pc=006	ir=1018	mo=0001	sw=0000	a=	3	mw=0	aluout=0004
300ns	pc=006	ir=1018	mo=0001	sw=0000	a=	4	mw=0	aluout=0005
305ns	pc=008	ir=3016	mo=0004	sw=0000	a=	4	mw=1	aluout=0000
315ns	pc=00a	ir=0014	mo=0006	sw=0000	a=	4	mw=0	aluout=0006
320ns	pc=00a	ir=0014	mo=0006	sw=0000	a=	6	mw=0	aluout=0006
325ns	pc=00c	ir=1016	mo=0004	sw=0000	a=	6	mw=0	aluout=000a

330ns	pc=00c	ir=1016	mo=0004	sw=0000	a=	10	mw=0	aluout=000e
335ns	pc=00e	ir=3014	mo=000a	sw=0000	a=	10	mw=1	aluout=0000
345ns	pc=010	ir=2000	mo=0016	sw=0000	a=	10	mw=0	aluout=0000
355ns	pc=000	ir=0016	mo=0004	sw=0000	a=	10	mw=0	aluout=0004
360ns	pc=000	ir=0016	mo=0004	sw=0000	a=	4	mw=0	aluout=0004
365ns	pc=002	ir=401a	mo=000a	sw=0000	a=	4	mw=0	aluout=0000
375ns	pc=004	ir=5012	mo=2012	sw=0000	a=	4	mw=0	aluout=0000
385ns	pc=006	ir=1018	mo=0001	sw=0000	a=	4	mw=0	aluout=0005
390ns	pc=006	ir=1018	mo=0001	sw=0000	a=	5	mw=0	aluout=0006
395ns	pc=008	ir=3016	mo=0005	sw=0000	a=	5	mw=1	aluout=0000
405ns	pc=00a	ir=0014	mo=000a	sw=0000	a=	5	mw=0	aluout=000a
410ns	pc=00a	ir=0014	mo=000a	sw=0000	a=	10	mw=0	aluout=000a
415ns	pc=00c	ir=1016	mo=0005	sw=0000	a=	10	mw=0	aluout=000f
420ns	pc=00c	ir=1016	mo=0005	sw=0000	a=	15	mw=0	aluout=0014
425ns	pc=00e	ir=3014	mo=000f	sw=0000	a=	15	mw=1	aluout=0000
435ns	pc=010	ir=2000	mo=0016	sw=0000	a=	15	mw=0	aluout=0000
445ns	pc=000	ir=0016	mo=0005	sw=0000	a=	15	mw=0	aluout=0005
450ns	pc=000	ir=0016	mo=0005	sw=0000	a=	5	mw=0	aluout=0005

455ns	pc=002	ir=401a	mo=000a	sw=0000	a=	5	mw=0	aluout=0000
465ns	pc=004	ir=5012	mo=2012	sw=0000	a=	5	mw=0	aluout=0000
475ns	pc=006	ir=1018	mo=0001	sw=0000	a=	5	mw=0	aluout=0006
480ns	pc=006	ir=1018	mo=0001	sw=0000	a=	6	mw=0	aluout=0007
485ns	pc=008	ir=3016	mo=0006	sw=0000	a=	6	mw=1	aluout=0000
495ns	pc=00a	ir=0014	mo=000f	sw=0000	a=	6	mw=0	aluout=000f
500ns	pc=00a	ir=0014	mo=000f	sw=0000	a=	15	mw=0	aluout=000f
505ns	pc=00c	ir=1016	mo=0006	sw=0000	a=	15	mw=0	aluout=0015
510ns	pc=00c	ir=1016	mo=0006	sw=0000	a=	21	mw=0	aluout=001b
515ns	pc=00e	ir=3014	mo=0015	sw=0000	a=	21	mw=1	aluout=0000
525ns	pc=010	ir=2000	mo=0016	sw=0000	a=	21	mw=0	aluout=0000
535ns	pc=000	ir=0016	mo=0006	sw=0000	a=	21	mw=0	aluout=0006
540ns	pc=000	ir=0016	mo=0006	sw=0000	a=	6	mw=0	aluout=0006
545ns	pc=002	ir=401a	mo=000a	sw=0000	a=	6	mw=0	aluout=0000
555ns	pc=004	ir=5012	mo=2012	sw=0000	a=	6	mw=0	aluout=0000
565ns	pc=006	ir=1018	mo=0001	sw=0000	a=	6	mw=0	aluout=0007
570ns	pc=006	ir=1018	mo=0001	sw=0000	a=	7	mw=0	aluout=0008

575ns	pc=008	ir=3016	mo=0007	sw=0000	a=	7	mw=1	aluout=0000
585ns	pc=00a	ir=0014	mo=0015	sw=0000	a=	7	mw=0	aluout=0015
590ns	pc=00a	ir=0014	mo=0015	sw=0000	a=	21	mw=0	aluout=0015
595ns	pc=00c	ir=1016	mo=0007	sw=0000	a=	21	mw=0	aluout=001c
600ns	pc=00c	ir=1016	mo=0007	sw=0000	a=	28	mw=0	aluout=0023
605ns	pc=00e	ir=3014	mo=001c	sw=0000	a=	28	mw=1	aluout=0000
615ns	pc=010	ir=2000	mo=0016	sw=0000	a=	28	mw=0	aluout=0000
625ns	pc=000	ir=0016	mo=0007	sw=0000	a=	28	mw=0	aluout=0007
630ns	pc=000	ir=0016	mo=0007	sw=0000	a=	7	mw=0	aluout=0007
635ns	pc=002	ir=401a	mo=000a	sw=0000	a=	7	mw=0	aluout=0000
645ns	pc=004	ir=5012	mo=2012	sw=0000	a=	7	mw=0	aluout=0000
655ns	pc=006	ir=1018	mo=0001	sw=0000	a=	7	mw=0	aluout=0008
660ns	pc=006	ir=1018	mo=0001	sw=0000	a=	8	mw=0	aluout=0009
665ns	pc=008	ir=3016	mo=0008	sw=0000	a=	8	mw=1	aluout=0000
675ns	pc=00a	ir=0014	mo=001c	sw=0000	a=	8	mw=0	aluout=001c
680ns	pc=00a	ir=0014	mo=001c	sw=0000	a=	28	mw=0	aluout=001c
685ns	pc=00c	ir=1016	mo=0008	sw=0000	a=	28	mw=0	aluout=0024
690ns	pc=00c	ir=1016	mo=0008	sw=0000	a=	36	mw=0	aluout=002c

695ns	pc=00e	ir=3014	mo=0024	sw=0000	a=	36	mw=1	aluout=0000
705ns	pc=010	ir=2000	mo=0016	sw=0000	a=	36	mw=0	aluout=0000
715ns	pc=000	ir=0016	mo=0008	sw=0000	a=	36	mw=0	aluout=0008
720ns	pc=000	ir=0016	mo=0008	sw=0000	a=	8	mw=0	aluout=0008
725ns	pc=002	ir=401a	mo=000a	sw=0000	a=	8	mw=0	aluout=0000
735ns	pc=004	ir=5012	mo=2012	sw=0000	a=	8	mw=0	aluout=0000
745ns	pc=006	ir=1018	mo=0001	sw=0000	a=	8	mw=0	aluout=0009
750ns	pc=006	ir=1018	mo=0001	sw=0000	a=	9	mw=0	aluout=000a
755ns	pc=008	ir=3016	mo=0009	sw=0000	a=	9	mw=1	aluout=0000
765ns	pc=00a	ir=0014	mo=0024	sw=0000	a=	9	mw=0	aluout=0024
770ns	pc=00a	ir=0014	mo=0024	sw=0000	a=	36	mw=0	aluout=0024
775ns	pc=00c	ir=1016	mo=0009	sw=0000	a=	36	mw=0	aluout=002d
780ns	pc=00c	ir=1016	mo=0009	sw=0000	a=	45	mw=0	aluout=0036
785ns	pc=00e	ir=3014	mo=002d	sw=0000	a=	45	mw=1	aluout=0000
795ns	pc=010	ir=2000	mo=0016	sw=0000	a=	45	mw=0	aluout=0000
805ns	pc=000	ir=0016	mo=0009	sw=0000	a=	45	mw=0	aluout=0009
810ns	pc=000	ir=0016	mo=0009	sw=0000	a=	9	mw=0	aluout=0009
815ns	pc=002	ir=401a	mo=000a	sw=0000	a=	9	mw=0	aluout=0000

825ns	pc=004	ir=5012	mo=2012	sw=0000	a=	9	mw=0	aluout=0000
835ns	pc=006	ir=1018	mo=0001	sw=0000	a=	9	mw=0	aluout=000a
840ns	pc=006	ir=1018	mo=0001	sw=0000	a=	10	mw=0	aluout=000b
845ns	pc=008	ir=3016	mo=000a	sw=0000	a=	10	mw=1	aluout=0000
855ns	pc=00a	ir=0014	mo=002d	sw=0000	a=	10	mw=0	aluout=002d
860ns	pc=00a	ir=0014	mo=002d	sw=0000	a=	45	mw=0	aluout=002d
865ns	pc=00c	ir=1016	mo=000a	sw=0000	a=	45	mw=0	aluout=0037
870ns	pc=00c	ir=1016	mo=000a	sw=0000	a=	55	mw=0	aluout=0041
875ns	pc=00e	ir=3014	mo=0037	sw=0000	a=	55	mw=1	aluout=0000
885ns	pc=010	ir=2000	mo=0016	sw=0000	a=	55	mw=0	aluout=0000
895ns	pc=000	ir=0016	mo=000a	sw=0000	a=	55	mw=0	aluout=000a
900ns	pc=000	ir=0016	mo=000a	sw=0000	a=	10	mw=0	aluout=000a
905ns	pc=002	ir=401a	mo=000a	sw=0000	a=	10	mw=0	aluout=4000
910ns	pc=002	ir=401a	mo=000a	sw=4000	a=	10	mw=0	aluout=4000
915ns	pc=004	ir=5012	mo=2012	sw=4000	a=	10	mw=0	aluout=0000
925ns	pc=012	ir=2012	mo=2012	sw=4000	a=	10	mw=0	aluout=0000

您可以清楚的看到，該程式在 870ns 時計算出了總合 SUM=55 的結果，這代表 mcu0bm.v 的設計完成

了計算 $1+...+10$ 的功能。

結語

在上述實作中，採用區塊式設計的 mcu0bm.v 總共有 98 行，比起同樣功能的流程式設計 mcu0m.v 的 51 行多了將近一倍，而且程式的設計難度感覺高了不少，但是我們可以很清楚的掌握到整個設計的硬體結構，這是採用流程式設計所難以確定的。

當然、由於筆者是「程式人員」，並非硬體設計人員，因此比較喜歡採用流程式的設計方式。不過採用了區塊式設計法設計出 mcu0bm.v 之後，也逐漸開始能理解這種「硬體導向」的設計方式，這大概是我在撰寫本程式時最大的收穫了。

MCU0 完整版

MCU0 的架構

MCU0 是一顆 16 位元的 CPU，所有暫存器都是 16 位元的，總共有 (IR, SP, LR, SW, PC, A) 等暫存器，如下所示：

```
`define A      R[0]      // 積累器
`define LR    R[1]      // 狀態暫存器
`define SW    R[2]      // 狀態暫存器
```

```
`define SP    R[3]      // 堆疊暫存器
`define PC    R[4]      // 程式計數器
```

這些暫存器的功能與說明如下：

暫存器 名稱	功能	說明
IR	指令 暫存 器	用來儲存從記憶體載入的機器碼指令
A=R[0]	累積 器	用來儲存計算的結果，像是加減法的結果。
LR=R[1]	連結 暫存 器	用來儲存函數呼叫的返回位址
SW=R[2]	狀態 暫存	用來儲存 CMP 比較指令的結果旗標，像是負旗標 N 與零旗標 Z 等。作為條件跳躍 JEQ 等指令是否跳躍的判斷依據。

	器	
SP=R[3]	堆疊 暫存 器	堆疊指標，PUSH, POP 指令會用到。
PC=R[4]	程式 計數 器	用來儲存指令的位址 (也就是目前執行到哪個指令的記憶體位址)

MCU0 的指令表

指令暫存器 IR 的前 4 個位元是指令代碼 OP，由於 4 位元只能表達 16 種指令，這數量太少不敷使用，因此當 OP=0xF 時，我們繼續用後面的位元作為延伸代碼，以便有更多的指令可以使用，以下是 MCU0 微控制器的完整指令表。

代 碼	名稱	格式	說明	語意
0	LD	LD C	載入	$A = [C]$
1	ST	ST C	儲存	$[C] = A$

2	ADD	ADD C	加法	$A = A + [C]$
3	SUB	SUB C	減法	$A = A - [C]$
4	MUL	MUL C	乘法	$A = A * [C]$
5	DIV	DIV C	除法	$A = A / [C]$
6	AND	AND C	位元 AND 運算	$A = A \& [C]$
7	OR	OR C	位元 OR 運算	$A = A [C]$
8	XOR	XOR C	位元 XOR 運算	$A = A ^ [C]$
9	CMP	CMP C	比較	$SW = A \text{ CMP } [C] ; N = (A < [C]), Z = (A == [C])$

A	JMP	JMP C	跳躍	PC = C
B	JEQ	JEQ C	相等時跳躍	if Z then PC = C
C	JLT	JLT C	小於時跳躍	if N then PC = C
D	JLE	JLE C	小於或等於時跳躍	if Z or N then PC = C
E	CALL	CALL C	呼叫副程式	LR=PC; PC = C
F	OP8		OP為8位元的運算	
F0	LDI	LDI Ra,C4	載入常數	Ra=C4
F2	MOV	MOV Ra,Rb	暫存器移動	Ra=Rb

F3	PUSH	PUSH Ra	堆疊推入	SP--; [SP] = Ra
F4	POP	POP Ra	堆疊取出	Ra=[SP]; SP++;
F5	SHL	SHL Ra,C4	左移	Ra = Ra << C4
F6	SHR	SHL Ra,C4	右移	Ra = Ra >> C4
F7	ADDI	ADDI Ra,C4	常數加法	Ra = Ra + C4
F8	SUBI	SUBI Ra,C4	常數減法	Ra = Ra - C4
F9	NEG	NEG Ra	反相	Ra = ~Ra
FA	SWI	SWI C	軟體中斷	BIOS 中斷呼叫

FD	NSW	NSW	狀態反相	$N=\sim N, Z=\sim Z$; 由於沒有 JGE, JGT, JNE，因此可用此指令將 SW 反相，再用 JLE, JLT, JEQ 完成跳躍動作
FE	RET	RET	返回	$PC = LR$
FF	IRET	IRET	從中斷返回	$PC = LR; I=0;$

mcu0 程式碼

檔案：mcu0s.v

```
`define OP    IR[15:12] // 運算碼
`define C     IR[11:0]  // 常數欄位
`define SC8   $signed(IR[7:0]) // 常數欄位
`define C4    IR[3:0]   // 常數欄位
`define Ra   IR[7:4]   // Ra
`define Rb   IR[3:0]   // Rb
`define A    R[0]      // 累積器
`define LR   R[1]      // 狀態暫存器
`define SW   R[2]      // 狀態暫存器
```

```
`define SP    R[3]      // 堆疊暫存器
`define PC    R[4]      // 程式計數器
`define N     `SW[15]    // 負號旗標
`define Z     `SW[14]    // 零旗標
`define I     `SW[3]     // 是否中斷中
`define M     m[`C]     // 存取記憶體
```

```
module cpu(input clock); // CPU0-Mini 的快取版: cpu0mc 模組
  parameter [3:0] LD=4' h0, ST=4' h1, ADD=4' h2, SUB=4' h3, MUL=4' h4, DIV=4' h5,
    AND=4' h6, OR=4' h7, XOR=4' h8, CMP=4' h9, JMP=4' hA, JEQ=4' hB, JLT=4' hC, JLE=4' hD,
    JSUB=4' hE, OP8=4' hF;
  parameter [3:0] LDI=4' h0, MOV=4' h2, PUSH=4' h3, POP=4' h4, SHL=4' h5,
    SHR=4' h6, ADDI=4' h7, SUBI=4' h8, NEG=4' h9, SWI=4' hA, NSW=4' hD, RET=4' hE,
    IRET=4' hF;
  reg [15:0] IR;      // 指令暫存器
  reg signed [15:0] R[0:4];
  reg signed [15:0] pc0;
```

```

reg signed [15:0] m [0:4096]; // 內部的快取記憶體
integer i;
initial // 初始化
begin
`PC = 0; // 將 PC 設為起動位址 0
`SW = 0;
$readmemh("mcu0s.hex", m);
end

always @(posedge clock) begin // 在 clock 時脈的正邊緣時觸發
    IR = m[`PC]; // 指令擷取階段: IR=m[PC], 2 個 Byte
    的記憶體
    pc0= `PC; // 儲存舊的 PC 值在 pc0 中。
    `PC = `PC+1; // 擷取完成, PC 前進到下一個指令位址
    case (`OP) // 解碼、根據 OP 執行動作
        LD: `A = `M; // LD C
        ST: `M = `A; // ST C
        ADD: `A = `A + `M; // ADD C

```

```
SUB: `A = `A - `M;           // SUB C
MUL: `A = `A * `M;           // MUL C
DIV: `A = `A / `M;           // DIV C
AND: `A = `A & `M;           // AND C
OR : `A = `A | `M;           // OR  C
XOR: `A = `A ^ `M;           // XOR C
CMP: begin `N=(`A < `M); `Z=(`A==`M); end // CMP C
JMP: `PC = `C; // JSUB C
JEQ: if (`Z) `PC=`C;         // JEQ C
JLT: if (`N) `PC=`C;         // JLT C
JLE: if (`N || `Z) `PC=`C; // JLE C
JSUB:begin `LR = `PC; `PC = `C; end // JSUB C
OP8: case (IR[11:8])        // OP8: 加長運算碼
      LDI: R[`Ra] = `C4;           // LDI C
      ADDI: R[`Ra] = R[`Ra] + `C4; // ADDI C
      SUBI: R[`Ra] = R[`Ra] - `C4; // ADDI C
      MOV: R[`Ra] = R[`Rb];        // MOV Ra, Rb
      PUSH: begin `SP=`SP-1; m[`SP] = R[`Ra]; end // PUSH Ra
```

```

POP: begin R[`Ra] = m[`SP]; `SP=`SP+1; end // POP Ra
SHL: R[`Ra] = R[`Ra] << `C4; // SHL C
SHR: R[`Ra] = R[`Ra] >> `C4; // SHR C
SWI: $display("SWI C8=%d A=%d", `SC8, `A); // SWI C
NEG: R[`Ra] = ~R[`Ra]; // NEG Ra
NSW: begin `N=~`N; `Z=~`Z; end // NSW (negate
N, Z)

RET: `PC = `LR; // RET
IRET: begin `PC = `LR; `I = 0; end // IRET
default: $display("op8=%d , not defined!", IR[11:8]);
endcase
endcase
// 印出 PC, IR, SW, A 等暫存器值以供觀察
$display("%4dns PC=%x IR=%x, SW=%x, A=%d SP=%x LR=%x", $stime, p
c0, IR, `SW, `A, `SP, `LR);
end
endmodule

```

```

module main;                      // 測試程式開始
reg clock;                       // 時脈 clock 變數

cpu mcu0(clock);                 // 宣告 mcu0 處理器

initial clock = 0;                // 一開始 clock 設定為 0
always #10 clock=~clock;          // 每隔 10ns 反相，時脈週期為 20ns
initial #1000 $finish;           // 停止測試。

endmodule

```

組合語言

檔案：mcu0s.hex

0020 // 00	RESET:	LD	X
2021 // 01		ADD	Y
3021 // 02		SUB	Y
4021 // 03		MUL	Y

5021	// 04	DIV	Y
7021	// 05	OR	Y
6021	// 06	AND	Y
8021	// 07	XOR	Y
0020	// 08	LD	X
F503	// 09	SHL	A, 3
F603	// 0A	SHR	A, 3
F701	// 0B	ADDI	1
0023	// 0C	LD	STACKEND
F230	// 0D	MOV	SP, A
E011	// 0E	JSUB	MIN
0022	// 0F	LD	Z
A010	// 10	HALT:	JMP HALT
F301	// 11	MIN:	PUSH LR
0020	// 12	LD	X
9021	// 13	CMP	Y
FD00	// 14	NSW	
C018	// 15	JLT	ELSE

1022	// 16		ST	Z
A019	// 17		JMP	NEXT
0021	// 18	ELSE:	LD	Y
1022	// 19	NEXT:	ST	Z
F401	// 1A		POP	LR
FE00	// 1B		RET	
0000	// 1C			
0000	// 1D			
0000	// 1E			
0000	// 1F			
0003	// 20	X:	WORD	3
0005	// 21	Y:	WORD	5
0000	// 22	Z:	WORD	0
007F	// 23	STACKEND:	WORD	127

執行結果

```
D:\Dropbox\Public\web\oc\code\mcu0>iverilog -o mcu0s mcu0s.v
```

D:\Dropbox\Public\web\oc\code\mcu0>vvp mcu0s

WARNING: mcu0s.v:29: \$readmemh(mcu0s.hex): Not enough words in the file for the requested range [0:4096].

10ns	PC=0000	IR=0020,	SW=0000,	A=	3	SP=xxxx	LR=xxxx
30ns	PC=0001	IR=2021,	SW=0000,	A=	8	SP=xxxx	LR=xxxx
50ns	PC=0002	IR=3021,	SW=0000,	A=	3	SP=xxxx	LR=xxxx
70ns	PC=0003	IR=4021,	SW=0000,	A=	15	SP=xxxx	LR=xxxx
90ns	PC=0004	IR=5021,	SW=0000,	A=	3	SP=xxxx	LR=xxxx
110ns	PC=0005	IR=7021,	SW=0000,	A=	7	SP=xxxx	LR=xxxx
130ns	PC=0006	IR=6021,	SW=0000,	A=	5	SP=xxxx	LR=xxxx
150ns	PC=0007	IR=8021,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
170ns	PC=0008	IR=0020,	SW=0000,	A=	3	SP=xxxx	LR=xxxx
190ns	PC=0009	IR=f503,	SW=0000,	A=	24	SP=xxxx	LR=xxxx
210ns	PC=000a	IR=f603,	SW=0000,	A=	3	SP=xxxx	LR=xxxx
230ns	PC=000b	IR=f701,	SW=0000,	A=	4	SP=xxxx	LR=xxxx
250ns	PC=000c	IR=0023,	SW=0000,	A=	127	SP=xxxx	LR=xxxx

270ns	PC=000d	IR=f230,	SW=0000,	A=	127	SP=007f	LR=xxxx
290ns	PC=000e	IR=e011,	SW=0000,	A=	127	SP=007f	LR=000f
310ns	PC=0011	IR=f301,	SW=0000,	A=	127	SP=007e	LR=000f
330ns	PC=0012	IR=0020,	SW=0000,	A=	3	SP=007e	LR=000f
350ns	PC=0013	IR=9021,	SW=8000,	A=	3	SP=007e	LR=000f
370ns	PC=0014	IR=fd00,	SW=4000,	A=	3	SP=007e	LR=000f
390ns	PC=0015	IR=c018,	SW=4000,	A=	3	SP=007e	LR=000f
410ns	PC=0016	IR=1022,	SW=4000,	A=	3	SP=007e	LR=000f
430ns	PC=0017	IR=a019,	SW=4000,	A=	3	SP=007e	LR=000f
450ns	PC=0019	IR=1022,	SW=4000,	A=	3	SP=007e	LR=000f
470ns	PC=001a	IR=f401,	SW=4000,	A=	127	SP=007f	LR=000f
490ns	PC=001b	IR=fe00,	SW=4000,	A=	127	SP=007f	LR=000f
510ns	PC=000f	IR=0022,	SW=4000,	A=	3	SP=007f	LR=000f
530ns	PC=0010	IR=a010,	SW=4000,	A=	3	SP=007f	LR=000f
550ns	PC=0010	IR=a010,	SW=4000,	A=	3	SP=007f	LR=000f
570ns	PC=0010	IR=a010,	SW=4000,	A=	3	SP=007f	LR=000f
590ns	PC=0010	IR=a010,	SW=4000,	A=	3	SP=007f	LR=000f
610ns	PC=0010	IR=a010,	SW=4000,	A=	3	SP=007f	LR=000f

630ns	PC=0010	IR=a010,	SW=4000,	A=	3	SP=007f	LR=000f
650ns	PC=0010	IR=a010,	SW=4000,	A=	3	SP=007f	LR=000f
670ns	PC=0010	IR=a010,	SW=4000,	A=	3	SP=007f	LR=000f
690ns	PC=0010	IR=a010,	SW=4000,	A=	3	SP=007f	LR=000f
710ns	PC=0010	IR=a010,	SW=4000,	A=	3	SP=007f	LR=000f
730ns	PC=0010	IR=a010,	SW=4000,	A=	3	SP=007f	LR=000f
750ns	PC=0010	IR=a010,	SW=4000,	A=	3	SP=007f	LR=000f
770ns	PC=0010	IR=a010,	SW=4000,	A=	3	SP=007f	LR=000f
790ns	PC=0010	IR=a010,	SW=4000,	A=	3	SP=007f	LR=000f
810ns	PC=0010	IR=a010,	SW=4000,	A=	3	SP=007f	LR=000f
830ns	PC=0010	IR=a010,	SW=4000,	A=	3	SP=007f	LR=000f
850ns	PC=0010	IR=a010,	SW=4000,	A=	3	SP=007f	LR=000f
870ns	PC=0010	IR=a010,	SW=4000,	A=	3	SP=007f	LR=000f
890ns	PC=0010	IR=a010,	SW=4000,	A=	3	SP=007f	LR=000f
910ns	PC=0010	IR=a010,	SW=4000,	A=	3	SP=007f	LR=000f
930ns	PC=0010	IR=a010,	SW=4000,	A=	3	SP=007f	LR=000f
950ns	PC=0010	IR=a010,	SW=4000,	A=	3	SP=007f	LR=000f
970ns	PC=0010	IR=a010,	SW=4000,	A=	3	SP=007f	LR=000f

990ns PC=0010 IR=a010, SW=4000, A= 3 SP=007f LR=000f

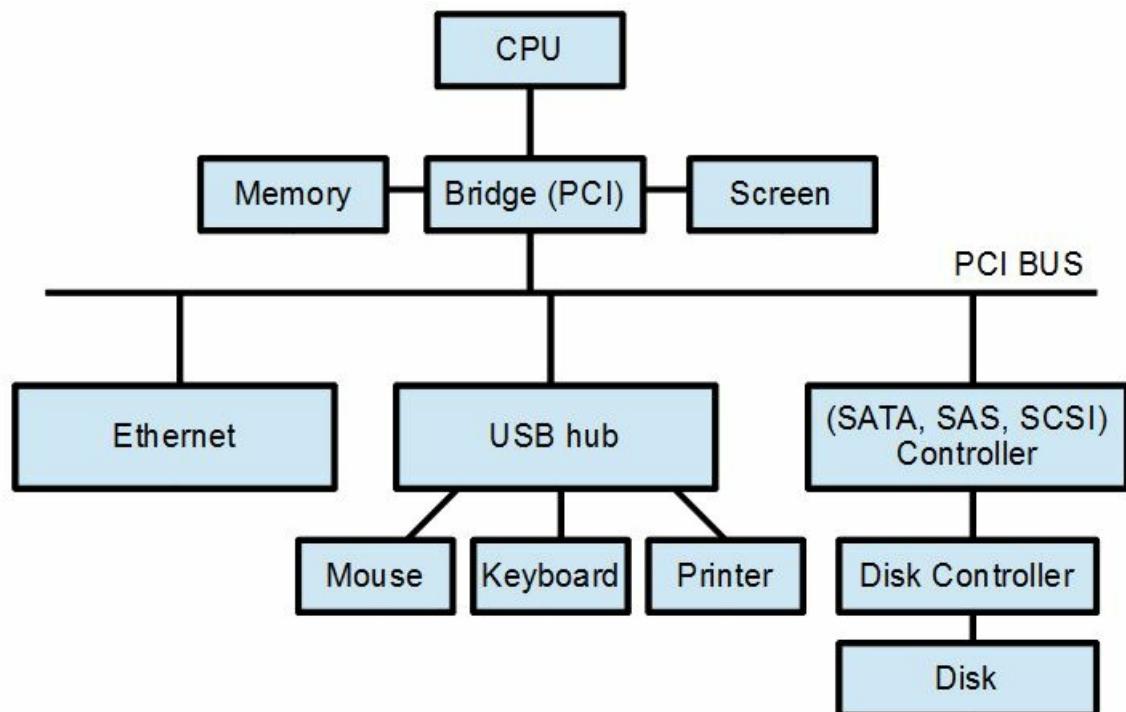
結語

由於 16 位元處理器的指令長度很短，因此空間必須有效利用，所以我們將一些不包含記憶體位址的指令，編到最後的 0xF 的 OP 代碼當中，這樣就可以再度延伸出一大群指令空間，於是讓指令數可以不受限於 4 位元 OP 碼的 16 個指令，而能延伸為 30 個左右的指令。

在使用 Verilog 這種硬體描述語言設計處理器時，位元數越少，往往處理器的指令長度越少，這時處理器不見得會更好設計，往往反而會更難設計，指令集的編碼相對會困難一些。

輸出入單元 (I/O)

前言



圖、PC 汇流排的連接結構

BUS (總線, 匯流排)

由於線路多的話會很混亂，而且成本很高。舉例而言，假如有 n 個節點(裝置)，所有節點之間都要互相直接相連，那麼就需要 $n*(n-1)/2$ 這個多組線路，這將會是個密密麻麻的災難。

如果、我們讓所有的節點都連接到一組共用的線路，這套線路就稱為 BUS。只要大家都遵循一套固定的傳送規則，我們就可以用 BUS 作為所有人的通訊橋梁。

以下是用 Verilog 宣告 BUS 線路的三種方法，分別是 wire, wand 與 wor。

```
wire [n-1:0] BUS;  
wand [n-1:0] andBUS;  
wor  [n-1:0] orBUS;
```

下列是採用 wire 方式宣告 BUS 的一個範例，當某個 seli 選擇線為 1 時，就會將對應的來源資料 sourcei 放到 BUS 上。而那些沒被選到的來源，由於是放高阻抗 Z，所以會處於斷線的狀態。

```
wire [n-1:0] BUS;  
parameter [n-1:0] disable = n' bZ;
```

```
assign BUS = sel1?source1:disable;  
assign BUS = sel2?source2:disable;  
...  
assign BUS = selk?sourcek:disable;
```

另外兩種宣告 BUS 的方法，也就是 wand 與 wor，與 wire 實在很像，差別只在於 wand 的 disable 是用 n'b1，而 wor 的 disable 是用 n'b0。

同步匯流排 (Synchronous BUS)

```
module master(input clock, w, output [15:0] address, inout [15:0] data);  
    reg [15:0] ar, dr;  
    assign address = ar;  
    assign data = (w)?dr : 16' hzzzz;  
  
    always @(*) begin  
        if (!w)
```

```
    dr=#1 data;  
  end  
endmodule  
  
module rdevice(input clock, w, input [15:0] address, output [15:0]  
data);  
  reg [15:0] dr;  
  
  assign data=(!w)?dr:16' hZZZZ;  
  
  always @(clock or w or address) begin  
    if (!w && address == 16' hFFF0)  
      dr = #1 16' he3e3;  
  end  
endmodule  
  
module wdevice(input clock, w, input [15:0] address, input [15:0] d  
ata);
```

```
reg [15:0] dr;

always @(clock or w or address) begin
    if (w && address == 16' hFFF8)
        dr = #1 data;
    end
endmodule

module main;
reg clock, w;
wire [15:0] abus, dbus;

master m(clock, w, abus, dbus);
rdevice rd(clock, w, abus, dbus);
wdevice wd(clock, w, abus, dbus);
initial begin
    $monitor("%4dns abus=%x dbus=%x w=%x m.ar=%x m.dr=%x rd.%dr=%x wd.%d
```

```

r=%x", $stime, abus, dbus, w, m.ar, m.dr, rd.dr, wd.dr);
clock = 0;
#10; m.ar=16' h0000; w=0;
#50; m.ar=16' hFFF0;
#50; m.ar=16' hFFF8; m.dr=16' h71F0; w=1;
#300; $finish;
end

always #5 clock=~clock; // 每隔 5ns 反相，時脈週期為 10ns
endmodule

```

執行結果

```

D:\Dropbox\Public\web\co\code>iverilog syncbus.v -o syncbus

D:\Dropbox\Public\web\co\code>vvp syncbus
 0ns abus=xxxx dbus=xxxx w=x m.ar=xxxx m.dr=xxxx rd.dr=xxxx wd.dr=
xxxx
 10ns abus=0000 dbus=xxxx w=0 m.ar=0000 m.dr=xxxx rd.dr=xxxx wd.dr=

```

xxxx

60ns abus=ffff0 dbus=xxxx w=0 m. ar=ffff0 m. dr=xxxx rd. dr=xxxx wd. dr=xxxx

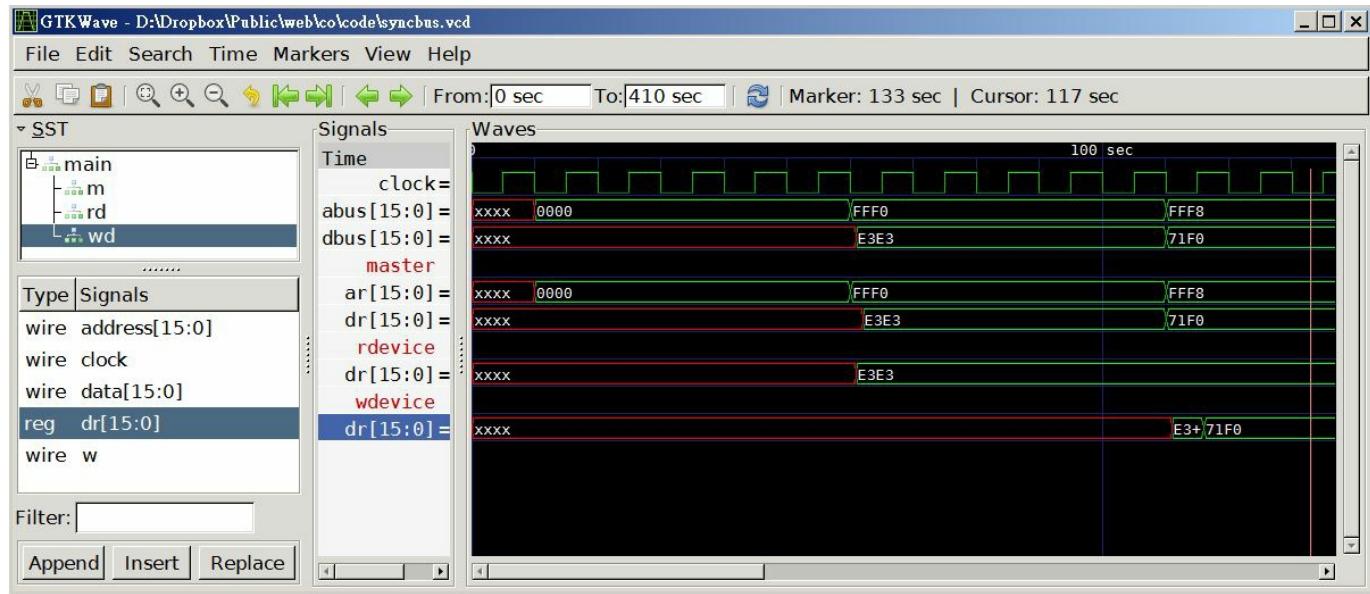
61ns abus=ffff0 dbus=e3e3 w=0 m. ar=ffff0 m. dr=xxxx rd. dr=e3e3 wd. dr=xxxx

62ns abus=ffff0 dbus=e3e3 w=0 m. ar=ffff0 m. dr=e3e3 rd. dr=e3e3 wd. dr=xxxx

110ns abus=ffff8 dbus=71f0 w=1 m. ar=ffff8 m. dr=71f0 rd. dr=e3e3 wd. dr=xxxx

111ns abus=ffff8 dbus=71f0 w=1 m. ar=ffff8 m. dr=71f0 rd. dr=e3e3 wd. dr=e3e3

116ns abus=ffff8 dbus=71f0 w=1 m. ar=ffff8 m. dr=71f0 rd. dr=e3e3 wd. dr=71f0



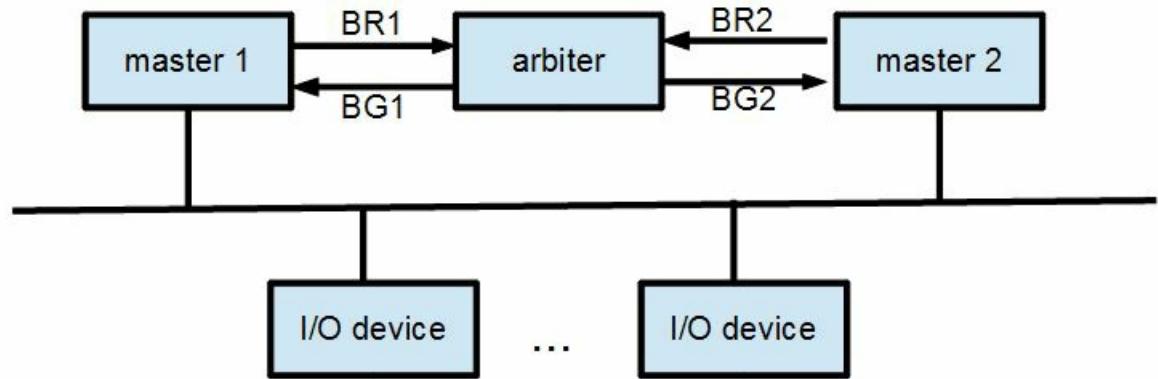
圖、上述同步 BUS 的波形圖

異步匯流排 (Asynchronous BUS)

異步匯流排是指沒有共同 Clock 訊號的匯流排，因此無法依賴 Clock 進行同步，所以必須依靠「主控就緒」(Master Ready)，「從動就緒」(Slave Ready) 等訊號，來進行握手 (Handshaking) 的協調程序。

匯流排仲裁 (BUS arbitery)

當有很多個主控裝置都有可能請求使用 BUS 的時候，就必須要加入一個仲裁機制，通常是由一個仲裁者 (arbiter) 進行仲裁。



循序與平行輸出入 (Serial vs. Parallel)

常見的輸出入協定

MCU0 的輸出入 -- 輪詢篇

在本文中，我們利用輪詢的方式實作了 MCU0 的鍵盤與文字輸出的函數。

MCU0 的中斷位元

在電腦中，進行輸出入所採用的方式，在指令上可分為「專用輸出入指令」與「記憶體映射輸出入」兩種，在本文中我們將用「記憶體映射輸出入」進行輸出入。

另外、進行輸出入的驅動方式，可分為「輪詢」與「中斷」兩種方式，本文將採用「輪詢」的方式實作。

MCU0 的輸出入實作方式

```
`define OP    IR[15:12] // 運算碼
`define C     IR[11:0]  // 常數欄位
`define SC8   $signed(IR[7:0]) // 常數欄位
`define C4    IR[3:0]   // 常數欄位
`define Ra   IR[7:4]   // Ra
`define Rb   IR[3:0]   // Rb
`define A    R[0]      // 累積器
`define LR   R[1]      // 狀態暫存器
`define SW   R[2]      // 狀態暫存器
```

```
`define SP    R[3]      // 堆疊暫存器
`define PC    R[4]      // 程式計數器
`define N     `SW[15]    // 負號旗標
`define Z     `SW[14]    // 零旗標
`define I     `SW[3]     // 是否中斷中
`define M     m[`C]     // 存取記憶體
```

```
module mcu(input clock, input interrupt, input[2:0] irq);
  parameter [3:0] LD=4' h0, ST=4' h1, ADD=4' h2, SUB=4' h3, MUL=4' h4, DIV=4' h5,
  AND=4' h6, OR=4' h7, XOR=4' h8, CMP=4' h9, JMP=4' hA, JEQ=4' hB, JLT=4' hC, JLE=4' hD,
  CALL=4' hE, OP8=4' hF;
  parameter [3:0] LDI=4' h0, MOV=4' h2, PUSH=4' h3, POP=4' h4, SHL=4' h5,
  SHR=4' h6, ADDI=4' h7, SUBI=4' h8, NEG=4' h9, SWI=4' hA, NSW=4' hD, RET=4' hE,
  IRET=4' hF;
  reg [15:0] IR;      // 指令暫存器
  reg signed [15:0] R[0:4];
  reg signed [15:0] pc0;
  reg signed [15:0] m [0:4095]; // 內部的快取記憶體
```

```
integer i;  
initial // 初始化  
begin  
    `PC = 0; // 將 PC 設為起動位址 0  
    `SW = 0;  
    $readmemh("mcu0io.hex", m);  
    for (i=0; i < 32; i=i+1) begin  
        $display("%x %x", i, m[i]);  
    end  
end  
  
always @(posedge clock) begin // 在 clock 時脈的正邊緣時觸發  
    IR = m[`PC]; // 指令擷取階段: IR=m[PC], 2 個 Byte  
的記憶體  
    pc0= `PC; // 儲存舊的 PC 值在 pc0 中。  
    `PC = `PC+1; // 擷取完成, PC 前進到下一個指令位址  
    case (`OP) // 解碼、根據 OP 執行動作
```

```
LD: `A = `M; // LD C
ST: `M = `A; // ST C
ADD: `A = `A + `M; // ADD C
SUB: `A = `A - `M; // SUB C
MUL: `A = `A * `M; // MUL C
DIV: `A = `A / `M; // DIV C
AND: `A = `A & `M; // AND C
OR : `A = `A | `M; // OR C
XOR: `A = `A ^ `M; // XOR C
CMP: begin `N=(`A < `M) ; `Z=(`A==`M) ; end // CMP C
JMP: `PC = `C; // JSUB C
JEQ: if (`Z) `PC=`C; // JEQ C
JLT: if (`N) `PC=`C; // JLT C
JLE: if (`N || `Z) `PC=`C; // JLE C
CALL:begin `LR = `PC; `PC = `C; end // CALL C
OP8: case (IR[11:8]) // OP8: 加長運算碼
      LDI: R[`Ra] = `C4; // LDI C
      ADDI: R[`Ra] = R[`Ra] + `C4; // ADDI C
```

```

SUBI: R[`Ra] = R[`Ra] - `C4;                                // ADDI C
MOV:  R[`Ra] = R[`Rb];                                     // MOV Ra, Rb
PUSH: begin `SP= `SP-1; m[`SP] = R[`Ra]; end // PUSH Ra
POP:  begin R[`Ra] = m[`SP]; `SP= `SP+1; end // POP Ra
SHL:  R[`Ra] = R[`Ra] << `C4;                                // SHL C
SHR:  R[`Ra] = R[`Ra] >> `C4;                                // SHR C
SWI:   $display("SWI C8=%d A=%d", `SC8, `A); // SWI C
NEG:   R[`Ra] = ~R[`Ra];                                    // NEG Ra
NSW:  begin `N=~`N; `Z=~`Z; end                           // NSW (negate
N, Z)

RET:   `PC = `LR;                                         // RET
IRET: begin `PC = `LR; `I = 0; end // IRET
default: $display("op8=%d , not defined!", IR[11:8]);
endcase
endcase
// 印出 PC, IR, SW, A 等暫存器值以供觀察
$c0, IR, `SW, `A, `SP, `LR);

```

```
if (!`I && interrupt) begin
    `I = 1;
    `LR = `PC;
    `PC = irq;
end
end
endmodule
```

```
module keyboard;
reg [7:0] ch[0:20];
reg [7:0] i;
initial begin
    i=0;
    {ch[0], ch[1], ch[2], ch[3], ch[4], ch[5], ch[6], ch[7], ch[8], ch[9], ch[10],
     ], ch[11], ch[12], ch[13]} = "hello verilog!";
    main.mcu0.m[16'h07F0] = 0;
    main.mcu0.m[16'h07F1] = 0;
end
```

```
always #20 begin
    if (main.mcu0.m[16' h07F0] == 0) begin
        main.mcu0.m[16' h07F1] = {8' h0, ch[i]};
        main.mcu0.m[16' h07F0] = 1;
        $display("key = %c", ch[i]);
        i = i+1;
    end
end

endmodule
```

```
module screen;
reg [7:0] ch;
initial begin
    main.mcu0.m[16' h07F2] = 0;
    main.mcu0.m[16' h07F3] = 0;
end
```

```
always #10 begin
    if (main.mcu0.m[16' h07F2] == 1) begin
        ch = main.mcu0.m[16' h07F3][7:0];
        $display("screen %c", ch);
        main.mcu0.m[16' h07F2] = 0;
    end
end
endmodule

module main;                      // 測試程式開始
reg clock;                       // 時脈 clock 變數
reg interrupt;
reg [2:0] irq;

mcu mcu0(clock, interrupt, irq); // 告知 cpu0mc 處理器
keyboard kb0();
screen sc0();
```

```

initial begin
    clock = 0;           // 一開始 clock 設定為 0
    interrupt = 0;
    irq = 2;
end
always #10 clock=~clock; // 每隔 10ns 反相，時脈週期為 20ns

initial #4000 $finish; // 停止測試。

endmodule

```

輸入機器碼與組合語言

07F0 // 00	WAITK:	LD	0x7F0	; wait keyboard
9010 // 01		CMP	K0	
B000 // 02		JEQ	WAIT	
07F1 // 03		LD	0x7F1	; read key
1011 // 04		ST	KEY	

0010	// 05	LD	K0
17F0	// 06	ST	0x7F0 ; release keyboard
07F2	// 07	WAITS:	LD 0x7F2 ; wait screen
0010	// 08	CMP	K0
B007	// 09	JEQ	WAITS
0011	// 0A	LD	KEY ; print key
17F3	// 0B	ST	0x7F3
F001	// 0C	LDI	1
17F2	// 0D	ST	0x7F2 ; eanble screen
A000	// 0E	JMP	WAIT
0000	// 0F		
0000	// 10	K0:	WORD 0
0000	// 11	KEY:	WORD 0

執行結果

```
D:\Dropbox\Public\web\oc\code\mcu0>iverilog -o mcu0io mcu0io.v
```

D:\Dropbox\Public\web\oc\code\mcu0>vvp mcu0io

WARNING: mcu0io.v:29: \$readmemh(mcu0io.hex): Not enough words in the
file for th
e requested range [0:4095].

00000000 07f0

00000001 9010

00000002 b000

00000003 07f1

00000004 1011

00000005 0010

00000006 17f0

00000007 07f2

00000008 0010

00000009 b007

0000000a 0011

0000000b 17f3

0000000c f001

0000000d 17f2

0000000e	a000
0000000f	0000
00000010	0000
00000011	0000
00000012	xxxx
00000013	xxxx
00000014	xxxx
00000015	xxxx
00000016	xxxx
00000017	xxxx
00000018	xxxx
00000019	xxxx
0000001a	xxxx
0000001b	xxxx
0000001c	xxxx
0000001d	xxxx
0000001e	xxxx
0000001f	xxxx

10ns PC=0000 IR=07f0, SW=0000, A= 0 SP=xxxx LR=xxxx

key = h

30ns PC=0001 IR=9010, SW=4000, A= 0 SP=xxxx LR=xxxx

50ns PC=0002 IR=b000, SW=4000, A= 0 SP=xxxx LR=xxxx

70ns PC=0000 IR=07f0, SW=4000, A= 1 SP=xxxx LR=xxxx

90ns PC=0001 IR=9010, SW=0000, A= 1 SP=xxxx LR=xxxx

110ns PC=0002 IR=b000, SW=0000, A= 1 SP=xxxx LR=xxxx

130ns PC=0003 IR=07f1, SW=0000, A= 104 SP=xxxx LR=xxxx

150ns PC=0004 IR=1011, SW=0000, A= 104 SP=xxxx LR=xxxx

170ns PC=0005 IR=0010, SW=0000, A= 0 SP=xxxx LR=xxxx

190ns PC=0006 IR=17f0, SW=0000, A= 0 SP=xxxx LR=xxxx

key = e

210ns PC=0007 IR=07f2, SW=0000, A= 0 SP=xxxx LR=xxxx

230ns PC=0008 IR=0010, SW=0000, A= 0 SP=xxxx LR=xxxx

250ns PC=0009 IR=b007, SW=0000, A= 0 SP=xxxx LR=xxxx

270ns PC=000a IR=0011, SW=0000, A= 104 SP=xxxx LR=xxxx

290ns PC=000b IR=17f3, SW=0000, A= 104 SP=xxxx LR=xxxx

310ns PC=000c IR=f001, SW=0000, A= 1 SP=xxxx LR=xxxx

330ns PC=000d IR=17f2, SW=0000, A=	1 SP=xxxx LR=xxxx
screen h	
350ns PC=000e IR=a000, SW=0000, A=	1 SP=xxxx LR=xxxx
370ns PC=0000 IR=07f0, SW=0000, A=	1 SP=xxxx LR=xxxx
390ns PC=0001 IR=9010, SW=0000, A=	1 SP=xxxx LR=xxxx
410ns PC=0002 IR=b000, SW=0000, A=	1 SP=xxxx LR=xxxx
430ns PC=0003 IR=07f1, SW=0000, A=	101 SP=xxxx LR=xxxx
450ns PC=0004 IR=1011, SW=0000, A=	101 SP=xxxx LR=xxxx
470ns PC=0005 IR=0010, SW=0000, A=	0 SP=xxxx LR=xxxx
490ns PC=0006 IR=17f0, SW=0000, A=	0 SP=xxxx LR=xxxx
key = 1	
510ns PC=0007 IR=07f2, SW=0000, A=	0 SP=xxxx LR=xxxx
530ns PC=0008 IR=0010, SW=0000, A=	0 SP=xxxx LR=xxxx
550ns PC=0009 IR=b007, SW=0000, A=	0 SP=xxxx LR=xxxx
570ns PC=000a IR=0011, SW=0000, A=	101 SP=xxxx LR=xxxx
590ns PC=000b IR=17f3, SW=0000, A=	101 SP=xxxx LR=xxxx
610ns PC=000c IR=f001, SW=0000, A=	1 SP=xxxx LR=xxxx
630ns PC=000d IR=17f2, SW=0000, A=	1 SP=xxxx LR=xxxx

screen e

650ns PC=000e IR=a000, SW=0000, A= 1 SP=xxxx LR=xxxx

670ns PC=0000 IR=07f0, SW=0000, A= 1 SP=xxxx LR=xxxx

690ns PC=0001 IR=9010, SW=0000, A= 1 SP=xxxx LR=xxxx

710ns PC=0002 IR=b000, SW=0000, A= 1 SP=xxxx LR=xxxx

730ns PC=0003 IR=07f1, SW=0000, A= 108 SP=xxxx LR=xxxx

750ns PC=0004 IR=1011, SW=0000, A= 108 SP=xxxx LR=xxxx

770ns PC=0005 IR=0010, SW=0000, A= 0 SP=xxxx LR=xxxx

790ns PC=0006 IR=17f0, SW=0000, A= 0 SP=xxxx LR=xxxx

key = 1

810ns PC=0007 IR=07f2, SW=0000, A= 0 SP=xxxx LR=xxxx

830ns PC=0008 IR=0010, SW=0000, A= 0 SP=xxxx LR=xxxx

850ns PC=0009 IR=b007, SW=0000, A= 0 SP=xxxx LR=xxxx

870ns PC=000a IR=0011, SW=0000, A= 108 SP=xxxx LR=xxxx

890ns PC=000b IR=17f3, SW=0000, A= 108 SP=xxxx LR=xxxx

910ns PC=000c IR=f001, SW=0000, A= 1 SP=xxxx LR=xxxx

930ns PC=000d IR=17f2, SW=0000, A= 1 SP=xxxx LR=xxxx

screen 1

950ns	PC=000e	IR=a000,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
970ns	PC=0000	IR=07f0,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
990ns	PC=0001	IR=9010,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
1010ns	PC=0002	IR=b000,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
1030ns	PC=0003	IR=07f1,	SW=0000,	A=	108	SP=xxxx	LR=xxxx
1050ns	PC=0004	IR=1011,	SW=0000,	A=	108	SP=xxxx	LR=xxxx
1070ns	PC=0005	IR=0010,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
1090ns	PC=0006	IR=17f0,	SW=0000,	A=	0	SP=xxxx	LR=xxxx

key = o

1110ns	PC=0007	IR=07f2,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
1130ns	PC=0008	IR=0010,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
1150ns	PC=0009	IR=b007,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
1170ns	PC=000a	IR=0011,	SW=0000,	A=	108	SP=xxxx	LR=xxxx
1190ns	PC=000b	IR=17f3,	SW=0000,	A=	108	SP=xxxx	LR=xxxx
1210ns	PC=000c	IR=f001,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
1230ns	PC=000d	IR=17f2,	SW=0000,	A=	1	SP=xxxx	LR=xxxx

screen 1

1250ns	PC=000e	IR=a000,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
--------	---------	----------	----------	----	---	---------	---------

1270ns	PC=0000	IR=07f0,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
1290ns	PC=0001	IR=9010,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
1310ns	PC=0002	IR=b000,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
1330ns	PC=0003	IR=07f1,	SW=0000,	A=	111	SP=xxxx	LR=xxxx
1350ns	PC=0004	IR=1011,	SW=0000,	A=	111	SP=xxxx	LR=xxxx
1370ns	PC=0005	IR=0010,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
1390ns	PC=0006	IR=17f0,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
key =							
1410ns	PC=0007	IR=07f2,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
1430ns	PC=0008	IR=0010,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
1450ns	PC=0009	IR=b007,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
1470ns	PC=000a	IR=0011,	SW=0000,	A=	111	SP=xxxx	LR=xxxx
1490ns	PC=000b	IR=17f3,	SW=0000,	A=	111	SP=xxxx	LR=xxxx
1510ns	PC=000c	IR=f001,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
1530ns	PC=000d	IR=17f2,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
screen o							
1550ns	PC=000e	IR=a000,	SW=0000,	A=	1	SP=xxxx	LR=xxxx

1570ns	PC=0000	IR=07f0,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
1590ns	PC=0001	IR=9010,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
1610ns	PC=0002	IR=b000,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
1630ns	PC=0003	IR=07f1,	SW=0000,	A=	32	SP=xxxx	LR=xxxx
1650ns	PC=0004	IR=1011,	SW=0000,	A=	32	SP=xxxx	LR=xxxx
1670ns	PC=0005	IR=0010,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
1690ns	PC=0006	IR=17f0,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
key = v							
1710ns	PC=0007	IR=07f2,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
1730ns	PC=0008	IR=0010,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
1750ns	PC=0009	IR=b007,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
1770ns	PC=000a	IR=0011,	SW=0000,	A=	32	SP=xxxx	LR=xxxx
1790ns	PC=000b	IR=17f3,	SW=0000,	A=	32	SP=xxxx	LR=xxxx
1810ns	PC=000c	IR=f001,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
1830ns	PC=000d	IR=17f2,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
screen							
1850ns	PC=000e	IR=a000,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
1870ns	PC=0000	IR=07f0,	SW=0000,	A=	1	SP=xxxx	LR=xxxx

1890ns	PC=0001	IR=9010,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
1910ns	PC=0002	IR=b000,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
1930ns	PC=0003	IR=07f1,	SW=0000,	A=	118	SP=xxxx	LR=xxxx
1950ns	PC=0004	IR=1011,	SW=0000,	A=	118	SP=xxxx	LR=xxxx
1970ns	PC=0005	IR=0010,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
1990ns	PC=0006	IR=17f0,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
key = e							
2010ns	PC=0007	IR=07f2,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
2030ns	PC=0008	IR=0010,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
2050ns	PC=0009	IR=b007,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
2070ns	PC=000a	IR=0011,	SW=0000,	A=	118	SP=xxxx	LR=xxxx
2090ns	PC=000b	IR=17f3,	SW=0000,	A=	118	SP=xxxx	LR=xxxx
2110ns	PC=000c	IR=f001,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
2130ns	PC=000d	IR=17f2,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
screen v							
2150ns	PC=000e	IR=a000,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
2170ns	PC=0000	IR=07f0,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
2190ns	PC=0001	IR=9010,	SW=0000,	A=	1	SP=xxxx	LR=xxxx

2210ns	PC=0002	IR=b000,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
2230ns	PC=0003	IR=07f1,	SW=0000,	A=	101	SP=xxxx	LR=xxxx
2250ns	PC=0004	IR=1011,	SW=0000,	A=	101	SP=xxxx	LR=xxxx
2270ns	PC=0005	IR=0010,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
2290ns	PC=0006	IR=17f0,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
key = r							
2310ns	PC=0007	IR=07f2,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
2330ns	PC=0008	IR=0010,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
2350ns	PC=0009	IR=b007,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
2370ns	PC=000a	IR=0011,	SW=0000,	A=	101	SP=xxxx	LR=xxxx
2390ns	PC=000b	IR=17f3,	SW=0000,	A=	101	SP=xxxx	LR=xxxx
2410ns	PC=000c	IR=f001,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
2430ns	PC=000d	IR=17f2,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
screen e							
2450ns	PC=000e	IR=a000,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
2470ns	PC=0000	IR=07f0,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
2490ns	PC=0001	IR=9010,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
2510ns	PC=0002	IR=b000,	SW=0000,	A=	1	SP=xxxx	LR=xxxx

2530ns	PC=0003	IR=07f1,	SW=0000,	A=	114	SP=xxxx	LR=xxxx
2550ns	PC=0004	IR=1011,	SW=0000,	A=	114	SP=xxxx	LR=xxxx
2570ns	PC=0005	IR=0010,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
2590ns	PC=0006	IR=17f0,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
key = i							
2610ns	PC=0007	IR=07f2,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
2630ns	PC=0008	IR=0010,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
2650ns	PC=0009	IR=b007,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
2670ns	PC=000a	IR=0011,	SW=0000,	A=	114	SP=xxxx	LR=xxxx
2690ns	PC=000b	IR=17f3,	SW=0000,	A=	114	SP=xxxx	LR=xxxx
2710ns	PC=000c	IR=f001,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
2730ns	PC=000d	IR=17f2,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
screen r							
2750ns	PC=000e	IR=a000,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
2770ns	PC=0000	IR=07f0,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
2790ns	PC=0001	IR=9010,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
2810ns	PC=0002	IR=b000,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
2830ns	PC=0003	IR=07f1,	SW=0000,	A=	105	SP=xxxx	LR=xxxx

2850ns	PC=0004	IR=1011,	SW=0000,	A=	105	SP=xxxx	LR=xxxx
2870ns	PC=0005	IR=0010,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
2890ns	PC=0006	IR=17f0,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
key = 1							
2910ns	PC=0007	IR=07f2,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
2930ns	PC=0008	IR=0010,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
2950ns	PC=0009	IR=b007,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
2970ns	PC=000a	IR=0011,	SW=0000,	A=	105	SP=xxxx	LR=xxxx
2990ns	PC=000b	IR=17f3,	SW=0000,	A=	105	SP=xxxx	LR=xxxx
3010ns	PC=000c	IR=f001,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
3030ns	PC=000d	IR=17f2,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
screen i							
3050ns	PC=000e	IR=a000,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
3070ns	PC=0000	IR=07f0,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
3090ns	PC=0001	IR=9010,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
3110ns	PC=0002	IR=b000,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
3130ns	PC=0003	IR=07f1,	SW=0000,	A=	108	SP=xxxx	LR=xxxx
3150ns	PC=0004	IR=1011,	SW=0000,	A=	108	SP=xxxx	LR=xxxx

3170ns	PC=0005	IR=0010,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
3190ns	PC=0006	IR=17f0,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
key = o							
3210ns	PC=0007	IR=07f2,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
3230ns	PC=0008	IR=0010,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
3250ns	PC=0009	IR=b007,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
3270ns	PC=000a	IR=0011,	SW=0000,	A=	108	SP=xxxx	LR=xxxx
3290ns	PC=000b	IR=17f3,	SW=0000,	A=	108	SP=xxxx	LR=xxxx
3310ns	PC=000c	IR=f001,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
3330ns	PC=000d	IR=17f2,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
screen 1							
3350ns	PC=000e	IR=a000,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
3370ns	PC=0000	IR=07f0,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
3390ns	PC=0001	IR=9010,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
3410ns	PC=0002	IR=b000,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
3430ns	PC=0003	IR=07f1,	SW=0000,	A=	111	SP=xxxx	LR=xxxx
3450ns	PC=0004	IR=1011,	SW=0000,	A=	111	SP=xxxx	LR=xxxx
3470ns	PC=0005	IR=0010,	SW=0000,	A=	0	SP=xxxx	LR=xxxx

3490ns	PC=0006	IR=17f0,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
key = g							
3510ns	PC=0007	IR=07f2,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
3530ns	PC=0008	IR=0010,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
3550ns	PC=0009	IR=b007,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
3570ns	PC=000a	IR=0011,	SW=0000,	A=	111	SP=xxxx	LR=xxxx
3590ns	PC=000b	IR=17f3,	SW=0000,	A=	111	SP=xxxx	LR=xxxx
3610ns	PC=000c	IR=f001,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
3630ns	PC=000d	IR=17f2,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
screen o							
3650ns	PC=000e	IR=a000,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
3670ns	PC=0000	IR=07f0,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
3690ns	PC=0001	IR=9010,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
3710ns	PC=0002	IR=b000,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
3730ns	PC=0003	IR=07f1,	SW=0000,	A=	103	SP=xxxx	LR=xxxx
3750ns	PC=0004	IR=1011,	SW=0000,	A=	103	SP=xxxx	LR=xxxx
3770ns	PC=0005	IR=0010,	SW=0000,	A=	0	SP=xxxx	LR=xxxx

3790ns	PC=0006	IR=17f0,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
key = !							
3810ns	PC=0007	IR=07f2,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
3830ns	PC=0008	IR=0010,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
3850ns	PC=0009	IR=b007,	SW=0000,	A=	0	SP=xxxx	LR=xxxx
3870ns	PC=000a	IR=0011,	SW=0000,	A=	103	SP=xxxx	LR=xxxx
3890ns	PC=000b	IR=17f3,	SW=0000,	A=	103	SP=xxxx	LR=xxxx
3910ns	PC=000c	IR=f001,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
3930ns	PC=000d	IR=17f2,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
screen g							
3950ns	PC=000e	IR=a000,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
3970ns	PC=0000	IR=07f0,	SW=0000,	A=	1	SP=xxxx	LR=xxxx
3990ns	PC=0001	IR=9010,	SW=0000,	A=	1	SP=xxxx	LR=xxxx

結語

以上的輸出入方式，並非典型的設計，而是屬於「系統單晶片」(SOC)的設計方式，因此直接將「鍵盤」與「螢幕」的輸出入暫存器直接內建在 MCU0 的記憶體之內，這樣的設計會比將「輸出入控制卡」與「CPU」分開的方式更容易一些，但是由於這種 ASIC 的量產費用昂貴，所以目前還

很少有這種設計方式。

不過、就簡單性而言，這樣的設計確實非常簡單，因此符合「開放電腦計畫」的 Keep it Simple and Stupid (KISS) 原則，所以我們先介紹這樣一個簡易的輸出入設計方式，以便讓讀者能從最簡單的架構入手。

記憶系統 (Storage)

除了記憶體之外，電腦裏還有隨身碟、記憶卡、硬碟、光碟、磁帶等儲存裝置，這些裝置構成一整個儲存體系，只有充分考慮這些儲存體的速度問題，才能讓電腦得到最好的效能。

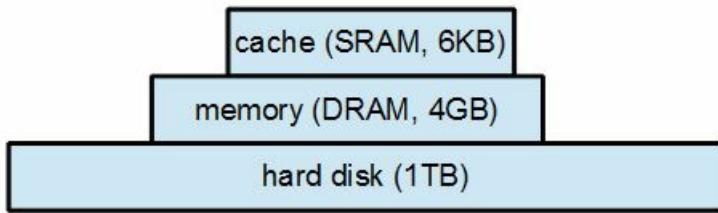
事實上、記憶體也有很多種，而且速度不一，像是靜態記憶體 SRAM、動態記憶體 DRAM、唯讀記憶體 ROM 等等，而且靜態記憶體的速度又可以分為很多種等級，因此整個儲存體會形成一個多層次不同速度的記憶體階層。

「高階處理器」與「微處理器」之間最大的差別，是「高階處理器」會利用各種記憶單元的速度差異，有效的安排並平衡「既快又大」的這種速度與大小的考量。

因此、「高階處理器」通常有「快取」(cache)、「記憶體管理單元」(Memory Management Unit, MMU) 等機制，以便能讓電腦能夠「容量又大、速度又快」。

要能夠讓「高階處理器」充分利用這種「記憶體階層」特性，首先讓我們來看看一個經典的「三階層」情況，那就是 (cache/memory/disk)，如下圖所示。

記憶體階層 (Memory Hierarchy)



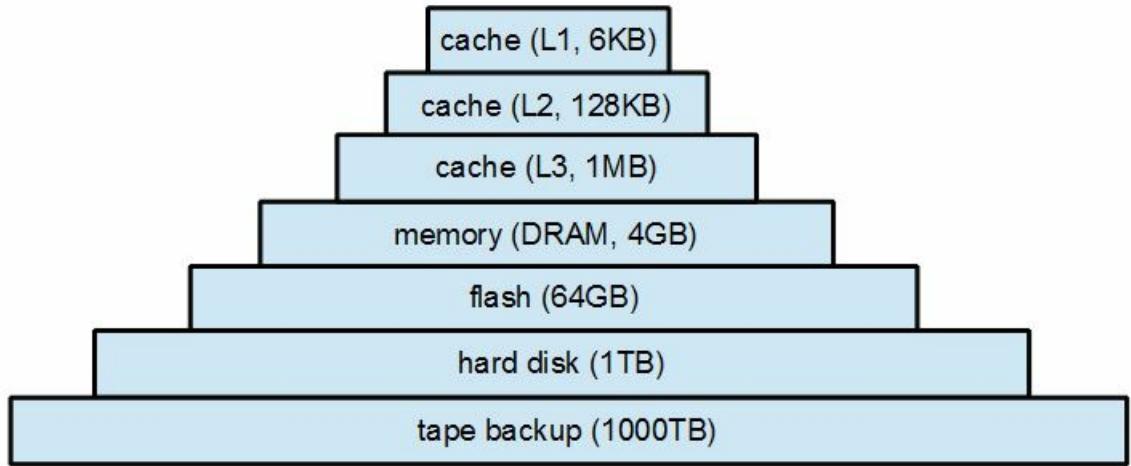
圖、常見的三層式記憶體階層

在上圖中，cache 是直接封裝在 CPU 內部的靜態記憶體，其運作速度與 CPU 的內部電路一樣快，因此可以在 1 個 Clock Cycle 之內完成存取。

而身為主記憶體的 DRAM，速度比靜態記憶體慢上數十倍，因此必須耗費幾十個 Cycle 才能完成存取。

最下層的「硬碟」(hard disk) 速度更慢，由於依賴讀寫頭與硬盤轉動的機械性動作，因此又比 DRAM 慢上數百倍 (雖然轉到了之後讀取還算快速，但是仍然相對緩慢，而且每次必須讀一大塊，否則轉了好久才讀 1 個 byte 將會慢如蝸牛)。

當然、有些電腦包含更多種類的記憶單元，這些記憶裝置的速度與容量不一，以下是一個更多層次的記憶階層範例。



圖、更多層次的記憶體階層

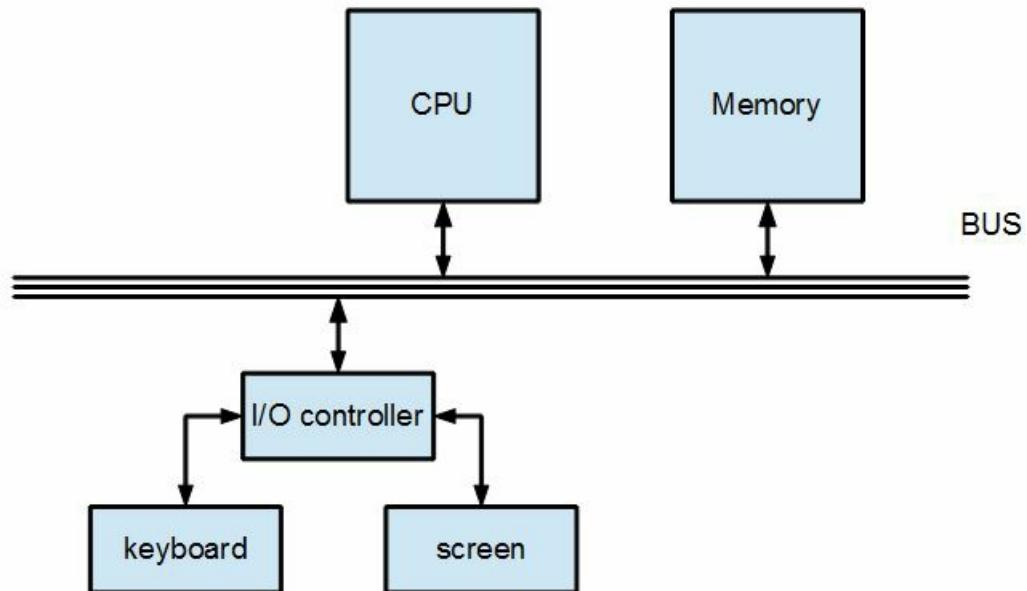
為了要讓電腦能夠又快又大，「高階處理器」通常採用了「快取」(cache) 與「記憶體管理單元」(MMU) 等兩個技術，其中的 cache 位於 CPU 內部，用來儲存常用的「指令與資料」，而 MMU 則是利用「分段或分頁」等機制，讓記憶體管理更有效率，甚至可以用「虛擬記憶體」技術把「硬碟」拿來當「備援記憶體」使用，讓容量可以進一步提升。

當然、要使用這些「快取」與「暫存」技術，都必須付出相對應的代價，那就是讓你的 CPU 設計更加複雜，而且有時也會「快不起來」。

(套句俗話說，出來混的，總有一天是要還的，CPU 的設計也是如此)

快取記憶體 (Cache)

傳統的處理器通常採用「單一匯流排馮紐曼架構」設計，這種架構非常簡單且易懂，如下圖所示：

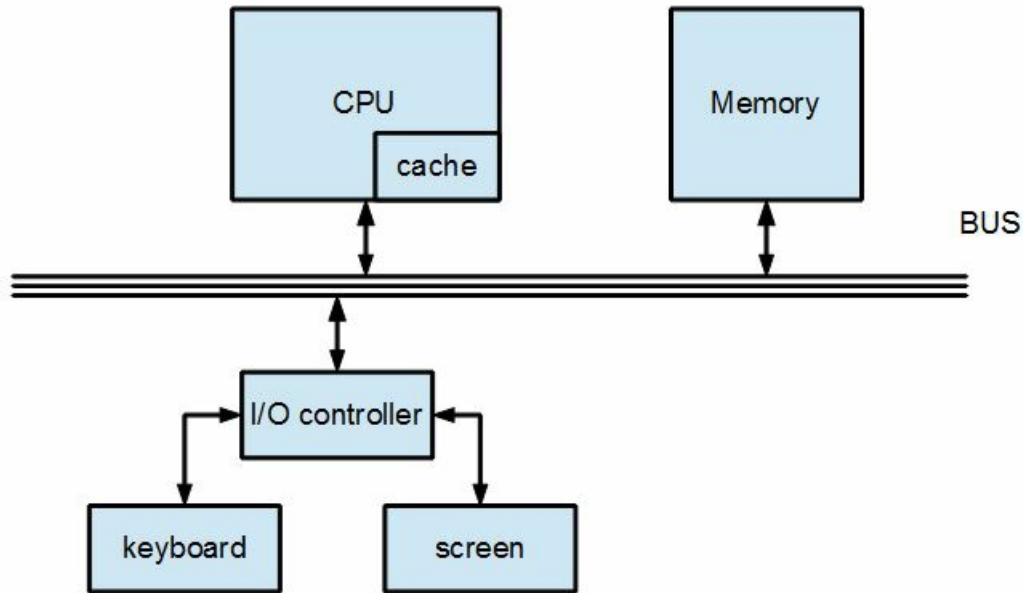


圖、單一匯流排的馮紐曼架構

但是、以當今的技術，上圖中的記憶體通常採用 DRAM 為主。

這是因為快速的靜態記憶體 (SRAM) 仍然相當昂貴，而速度慢上數十倍的動態記憶體 (DRAM) 則相對便宜，因此電腦的主記憶體通常仍然採用 DRAM 為主。

如果要讓 CPU 的執行速度更快，就不能讓 CPU 遷就於 DRAM 的速度跟著變慢，此時我們可以在 CPU 內部加入一個快取記憶體 (cache)，讓位於 DRAM 中常用到的指令與資料放入快取當中，如下圖所示：



圖、單一匯流排的馮紐曼架構

當 CPU 想要執行一個指令時，如果該指令已經在 cache 當中，就不需要讀取 DRAM，因此「指令擷取」階段就可以快上數十倍。

當然、如果該指令需要存取資料，而該資料已經在 cache 當中了，那麼 CPU 就不需要從 DRAM 中讀取資料，因此「資料讀取」階段就可以快上數十倍。

當需要寫入資料時，如果暫時先寫到 cache 當中，而不是直接寫回 DRAM，那麼「資料寫入」的動作就可以快上數十倍。

當然、上述的美好情況不會永遠都成立，假如想存取的指令或資料不存在 cache 當中，那麼就必須從 DRAM 讀取資料，這時 CPU 的速度就會被打回原形，變成與 DRAM 的速度一樣。

而且、當資料被放入 cache 之後，由於 DRAM 當中還有一份同樣的資料，要又如何才能快速的查出資料是否在 cache 當中，就是一個不容易的問題了。這個問題的解決，必須依靠某種「記憶體位址的標籤」(tag)，透過這種標籤我們可以查出某個位址的內容到底是在 cache 當中的那一格。

而標籤的設計方法，大致可分為「直接映射」(direct-mapping)，「關連映射」(associate-mapping)與「組關連映射」(set associative-mapping)等幾種方法。其中最常採用的是後者。

對這些 cache 作法的細節有興趣的讀者，可以參考下列文件。

- [維基百科：CPU快取](#)

高階處理器 (Processor)

在前一章當中，我們說明了「微處理器」的設計方式，在本章中、我們將說明「高階處理器」的設計方式。

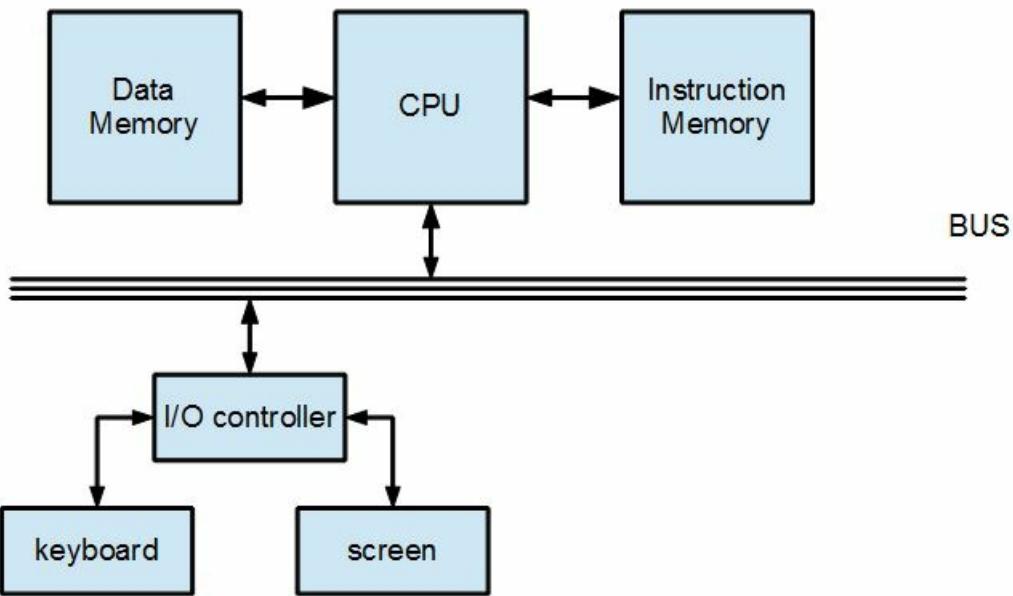
與「微處理器」比起來，「高階處理器」除了指令寬度較大之外，能定址的記憶體空間通常也很大，而且會內建「記憶體管理單元」(Memory Management Unit, MMU) 與多層快取機制 (cache) 等等，這些都是為了充分發揮處理器的效能的設計，這也正是為何稱為「高階」處理器的原因。

在本章當中，我們將透過 `cpu0` 這個架構，給出一個「處理器」的簡易範例之後，就開始針對現今的高階處理器之結構進行探討，以便讓讀者在能清楚的理解現代高階處理器的設計原理與特性。

哈佛架構 (Harvard Architecture)

在「單一匯流排馮紐曼架構」之下，由於指令與資料放在同一個記憶體當中，而且只有一套匯流排，因此指令與資料勢必無法同時存取。

如果、我們希望同時進行指令與資料的存取，那麼就可以將指令與資料分別放在兩塊不同的記憶體當中，並且各用一套內部匯流排連接到 CPU，這樣就有可能同時存取「指令和資料」，也就有可能將「指令擷取」(instruction fetch) 與「資料存取」(data access) 階段重疊執行了。

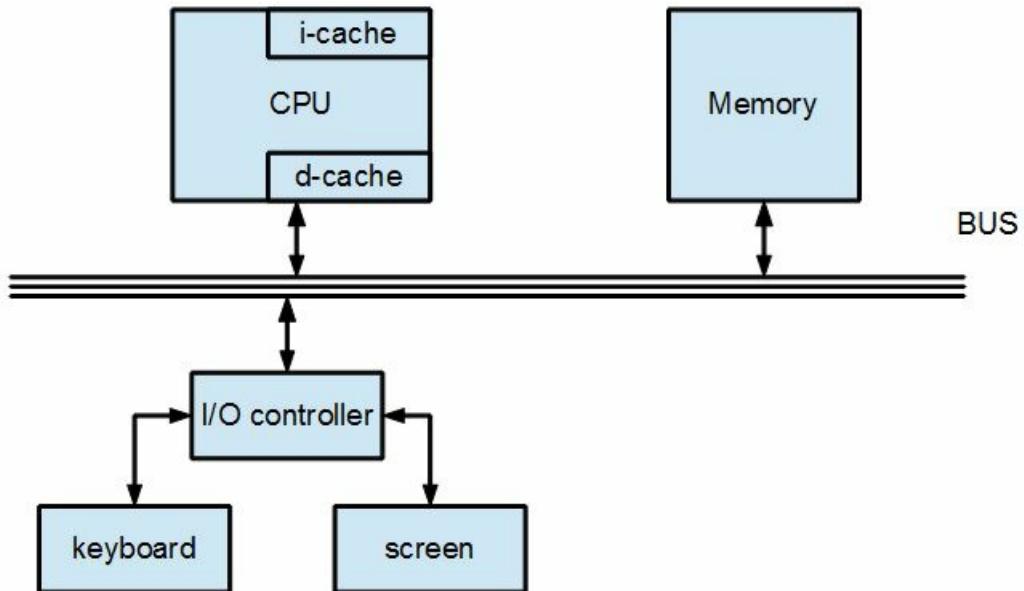


圖、指令與資料分開為兩套記憶體的哈佛架構

採用「指令與資料分開」的兩套記憶體模式，如果都是 DRAM 的話，那麼該處理器仍然會式非常緩慢的，因為受限於 DRAM 的速度限制，因此這種方式根本就是「花了兩倍力氣卻得不到太多好處」，可以說是一種很爛的設計。

但是、假如我們不是將主記憶體分為兩套，而是將 cache 分為「指令快取」(i-cache) 與「資料快

取」(d-cache) 的「變種哈佛架構」話，那麼速度就真的是會變快了。



圖、採用兩套快取的單一匯流排哈佛架構

不過、其實還可以變得更快，只要我們能夠用「流水線的模式」，就能讓 CPU 快上五倍，這必須讓指令的每個階段都能重疊起來才行 (包含「指令擷取」與「資料存取」階段的重疊，這也正是為何要討論「哈佛架構」的原因)。

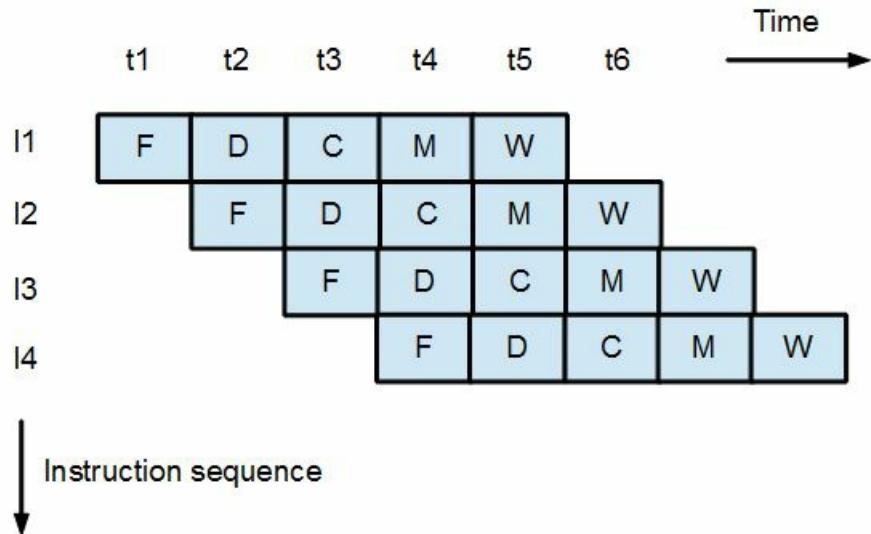
流水線架構 (Pipeline)

Pipeline 是一種讓指令分成幾個步驟，然後兩個指令的不同步驟可以「瀑布式重疊」的方法。

舉例而言、假如我們將一個指令的執行分為五個步驟如下：

1. 擷取 F (Fetch)
2. 解碼 D (Decode)
3. 計算 C (Compute)
4. 存取 M (Memory Access)
5. 寫回 W (Write Back)

那麼在理想的情況下，我們就能夠讓這些指令充分的重疊執行，如下圖所示。



圖、理想的 Pipeline 執行情況

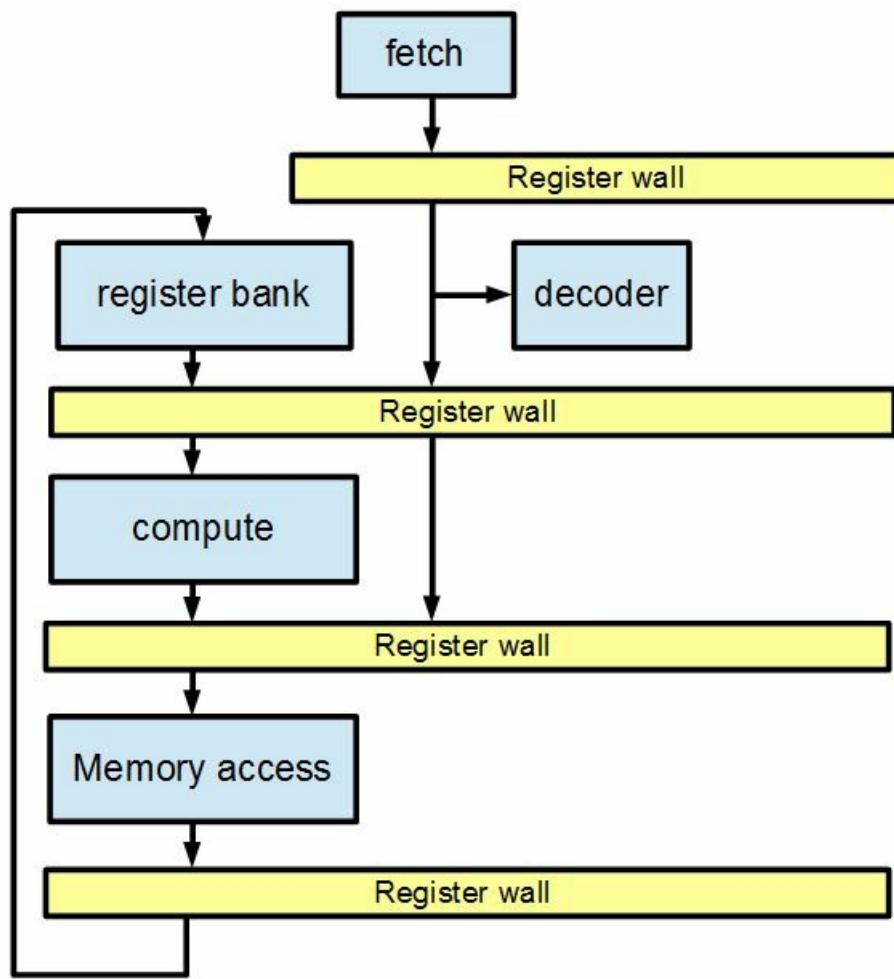
如此、只要讓每個步驟所需的時間幾乎一樣，然後將 Clock 調快五倍，就能讓速度整整加快五倍。

這就是所謂的 Pipeline 流水線架構的原理。

然而、流水線架構的處理器設計比較困難，因為上圖的理想情況畢竟只是理想，現實生活中有很多因素會干擾流水線的運行，造成流水線某個階段受到阻礙無法完成，必須暫停等後某個步驟完成，這種暫停會形成流水線當中的氣泡 (bubble)，這種情況統稱為「指令圍障」 (instruction hazard)。

Hazard 有好幾種，像是資料圍障 (data hazard)、快取失敗 (cache miss)、分支延遲 (branch penalty) 等等。

流水線架構的設計，通常必須用暫存器圍牆 (Register wall) 將每個步驟之間相互隔開，以避免線路之間的干擾，如下所示：



圖、以暫存器圍牆分隔流水線架構的每個階段

Pipeline 處理器最困難的地方，就是要盡可能的減少這些 hazard。然而、有些 hazard 是可以用硬體方式消除的，但是很多 hazard 的消除卻是需要編譯器的配合才能夠做得到。

透過編譯器的協助減少 hazard 的方法，包括「指令重排、插入 NOP 指令、分支預測」等等。

想要進一步瞭解流水線架構細節的讀者，可以參考下列文件。

- 維基百科：流水線
- 維基百科：指令管線化
- 維基百科：分支預測器

CPU0 迷你版 – CPU0m

在本章中，我們將透過設計一顆只支援 4 個指令的超微小處理器 CPU0m (CPU0-Mini) 開始，來解說處理器的設計方式。

只有 4 個指令的處理器 – CPU0m

在此、我們將用最簡單的方式，在完全不考慮成本與實用性的情況之下，設計一個將記憶體嵌入處理器內部的 CPU，也就是整個記憶體都用 Verilog 內嵌在 CPU 理面。

我們從 CPU0 的指令集當中，挑出了以下的四個指令，以便寫出一個可以計算 $1+2+\dots+n+\dots$ 的組合語言程式(喔！不、應該說是機器語言程式)，然後用 Verilog 實作一個可以執行這些指令的 CPU0m 處理器。

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]
A	ADD	13	加法	ADD Ra, Rb, Rc	Ra=Rb+Rc
J	JMP	26	跳躍(無條件)	JMP Cx	PC=PC+Cx

然後，我們就可以用這幾個指令寫出以下的程式：

位址	機器碼	標記	組合語言	對照的 C 語言
0000	001F0018		LD R1, K1	R1 = K1
0004	002F0010		LD R2, K0	R2 = K0
0008	003F0014		LD R3, SUM	R3 = SUM

000C	13221000	LOOP:	ADD R2, R2, R1	R2 = R2 + R1
0010	13332000		ADD R3, R3, R2	R3 = R3 + R2
0014	26FFFFFF4		JMP LOOP	goto LOOP
0018	00000000	K0:	WORD 0	int K0=0
001C	00000001	K1:	WORD 1	int K1=1
0020	00000000	SUM:	WORD 0	int SUM=0

這個程式的行為模式，是會讓暫存器 R3 (對應到 SUM) 從 0, 1, 1+2, 1+2+3, ... 一路向上跑，而且是永無止境的無窮迴圈。因此我們會看到 R3 的內容會是 0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55 ... ，的情況。

透過 CPU0m，讀者將可以開始瞭解一顆 CPU 的結構與設計方式，並且更清楚的理解 CPU 的控制單元之功能。

指令提取、解碼與執行

CPU0 在執行一個指令時，必須經過提取、解碼與執行等三大階段，以下是這三個階段的詳細步驟。

階段 (a)：提取階段

動作1、提取指令 : $IR = m[PC]$

動作2、更新計數器 : $PC = PC + 4$

階段 (b)：解碼階段

動作3、解碼 : 將 IR 分解為 (運算代碼 op, 暫存器代號 ra, rb, rc, 常數 c5, c16, c24 等欄位)。

階段 (c)：執行階段

動作4、執行 : 根據運算代碼 op，執行對應的動作。

執行階段的動作根據指令的類型而有所不同。舉例而言，假如執行的指令是 ADD R1, R2, R1，那麼執行階段所進行的動作如下：

格式	指令	OP	說明	語法	執行動作
L	LD	00	載入word	LD Ra, [Rb+Cx]	$R[ra] = m[rb+c16]$ (4byte)
L	ST	01	儲存word	ST Ra, [Rb+Cx]	$m[rb+c16] = R[ra]$ (4byte)
A	ADD	13	加法	ADD Ra, Rb, Rc	$R[ra] = R[rb] + R[rc]$
J	JMP	26	跳躍 (無條件)	JMP Cx	$PC = PC + c24$

CPU0m 模組

以下就是我們所設計的 CPU0m 模組，以及測試的主程式，我們在程式中寫了詳細的說明，請讀者對照閱讀。

檔案：[CPU0m](#)

```
`define PC R[15] // 程式計數器 PC 其實是 R[15] 的別名

module CPU(input clock); // CPU0-Mini 的快取版: cpu0m 模組
    parameter [7:0] LD = 8'h00, ST=8'h01, ADD=8'h13, JMP=8'h26; // 支
    援 4 個指令
    reg signed [31:0] R [0:15]; // 宣告暫存器 R[0..15] 等 16 個 32 位
    元暫存器
    reg signed [31:0] IR;           // 指令暫存器 IR
    reg [7:0] m [0:128];          // 內部的快取記憶體
    reg [7:0] op;                 // 變數: 運算代碼 op
    reg [3:0] ra, rb, rc;         // 變數: 暫存器代號 ra, rb, rc
```

```

reg signed [11:0] cx12;           // 變數: 12 位元常數 cx12
reg signed [15:0] cx16;           // 變數: 16 位元常數 cx16
reg signed [23:0] cx24;           // 變數: 24 位元常數 cx24
reg signed [31:0] addr;           // 變數: 暫存記憶體位址

initial // 初始化
begin
    `PC = 0;                      // 將 PC 設為起動位址 0
    R[0] = 0;                      // 將 R[0] 暫存器強制設定為 0
    {m[0], m[1], m[2], m[3]} = 32' h001F0018; // 0000 LD R1,
K1
    {m[4], m[5], m[6], m[7]} = 32' h002F0010; // 0004 LD R2,
K0
    {m[8], m[9], m[10], m[11]} = 32' h003F0014; // 0008 LD R3,
SUM
    {m[12], m[13], m[14], m[15]} = 32' h13221000; // 000C LOOP: ADD R2,
R2, R1
    {m[16], m[17], m[18], m[19]} = 32' h13332000; // 0010 ADD R3,

```

R3, R2

```
{m[20], m[21], m[22], m[23]}= 32' h26FFFFF4; // 0014      JMP LOOP
{m[24], m[25], m[26], m[27]}= 32' h00000000; // 0018 K0: WORD 0
{m[28], m[29], m[30], m[31]}= 32' h00000001; // 001C K1: WORD 1
{m[32], m[33], m[34], m[35]}= 32' h00000000; // 0020 SUM: WORD 0
```

end

always @(posedge clock) begin // 在 clock 時脈的正邊緣時觸發
 IR = {m[`PC], m[`PC+1], m[`PC+2], m[`PC+3]}; // 指令擷取階段
 : IR=m[PC], 4 個 Byte 的記憶體
 `PC = `PC+4; // 擷取完成, PC

前進到下一個指令位址

{op, ra, rb, rc, cx12} = IR; // 解碼階段: 將
IR 解為 {op, ra, rb, rc, cx12}

cx24 = IR[23:0]; // 解
出 IR[23:0] 放入 cx24

cx16 = IR[15:0]; // 解
出 IR[15:0] 放入 cx16

```
addr = R[rb]+cx16;                                // 記憶體存取
位址 = PC+cx16

case (op) // 根據 OP 執行對應的動作
    LD: begin // 載入指令: R[ra] = m[addr]
        R[ra] = {m[addr], m[addr+1], m[addr+2], m[addr+3]};
        $write("%4dns %8x : LD %x,%x,%-4x", $stime, `PC, ra, rb,
cx16);
        end
    ST: begin // 儲存指令: m[addr] = R[ra]
        {m[addr], m[addr+1], m[addr+2], m[addr+3]} = R[ra];
        $write("%4dns %8x : ST %x,%x,%-4x", $stime, `PC, ra, rb,
cx16);
        end
    ADD: begin // 加法指令: R[ra] = R[rb]+R[rc]
        R[ra] = R[rb]+R[rc];
        $write("%4dns %8x : ADD %x,%x,%-4x", $stime, `PC, ra, rb,
rc);
        end
```

```
JMP:begin // 跳躍指令: PC = PC + cx24
    `PC = `PC + cx24; // 跳躍目標位址=PC+cx
    $write("%4dns %8x : JMP %-8x", $stime, `PC, cx24);
    end
endcase
$display(" R[%2d]=%4d", ra, R[ra]); // 顯示目標暫存器的值
end
endmodule

module main; // 測試程式開始
reg clock; // 時脈 clock 變數

cpu cpu0m(clock); // 告訴 cpu0m 處理器

initial clock = 0; // 一開始 clock 設定為 0
always #10 clock=~clock; // 每隔 10 奈秒將 clock 反相，產生週期為
// 20 奈秒的時脈
initial #640 $finish; // 在 640 奈秒的時候停止測試。(因為這時的
```

```
R[1] 恰好是 1+2+...+10=55 的結果)
```

```
endmodule
```

測試結果

上述程式使用 icarus 測試與執行的結果如下所示。

```
D:\Dropbox\Public\web\oc\code>iverilog -o cpu0m cpu0m.v
```

```
D:\Dropbox\Public\web\oc\code>vvp cpu0m
```

```
10ns 00000004 : LD 1, f, 0018 R[ 1]= 1
30ns 00000008 : LD 2, f, 0010 R[ 2]= 0
50ns 0000000c : LD 3, f, 0014 R[ 3]= 0
70ns 00000010 : ADD 2, 2, 1 R[ 2]= 1
90ns 00000014 : ADD 3, 3, 2 R[ 3]= 1
110ns 0000000c : JMP ffffff4 R[15]= 12
130ns 00000010 : ADD 2, 2, 1 R[ 2]= 2
150ns 00000014 : ADD 3, 3, 2 R[ 3]= 3
170ns 0000000c : JMP ffffff4 R[15]= 12
```

190ns	00000010	:	ADD	2, 2, 1	R[2]=	3
210ns	00000014	:	ADD	3, 3, 2	R[3]=	6
230ns	0000000c	:	JMP	fffff4	R[15]=	12
250ns	00000010	:	ADD	2, 2, 1	R[2]=	4
270ns	00000014	:	ADD	3, 3, 2	R[3]=	10
290ns	0000000c	:	JMP	fffff4	R[15]=	12
310ns	00000010	:	ADD	2, 2, 1	R[2]=	5
330ns	00000014	:	ADD	3, 3, 2	R[3]=	15
350ns	0000000c	:	JMP	fffff4	R[15]=	12
370ns	00000010	:	ADD	2, 2, 1	R[2]=	6
390ns	00000014	:	ADD	3, 3, 2	R[3]=	21
410ns	0000000c	:	JMP	fffff4	R[15]=	12
430ns	00000010	:	ADD	2, 2, 1	R[2]=	7
450ns	00000014	:	ADD	3, 3, 2	R[3]=	28
470ns	0000000c	:	JMP	fffff4	R[15]=	12
490ns	00000010	:	ADD	2, 2, 1	R[2]=	8
510ns	00000014	:	ADD	3, 3, 2	R[3]=	36
530ns	0000000c	:	JMP	fffff4	R[15]=	12

550ns	00000010	:	ADD	2, 2, 1	R[2]=	9
570ns	00000014	:	ADD	3, 3, 2	R[3]=	45
590ns	0000000c	:	JMP	ffffff4	R[15]=	12
610ns	00000010	:	ADD	2, 2, 1	R[2]=	10
630ns	00000014	:	ADD	3, 3, 2	R[3]=	55

從上述輸出訊息當中，您可以看到程式的執行是正確的，其中 R[2] 從 0, 1, 2, 一路上數，而 R[3] 則從 0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55 一路累加上來，完成了我們想要的程式功能。

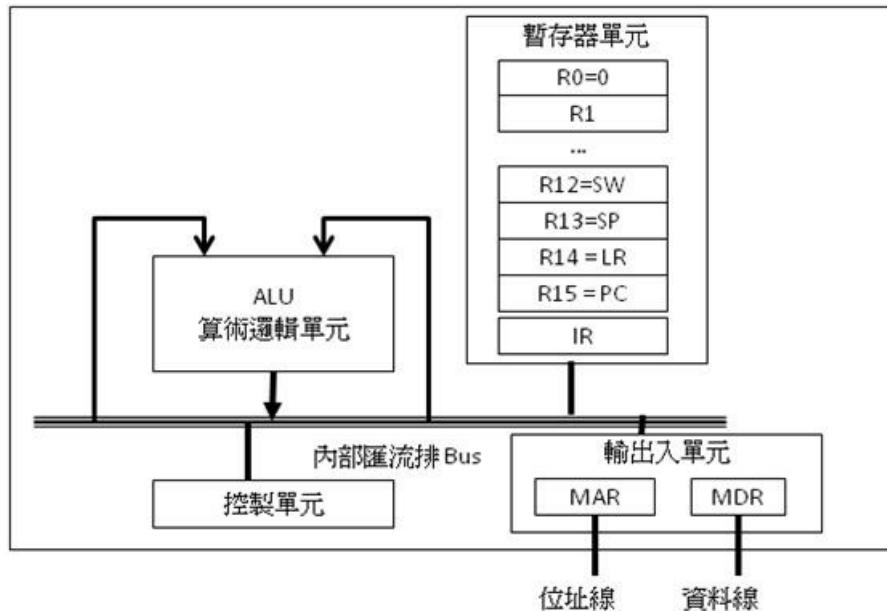
結語

其實、CPU0m 這樣的設計應該還不能稱之為快取，而是在程式不大的情況之下，將 SRAM 直接包入在 CPU 當中的一種作法，這種作法的好處是記憶體存取速度很快，但相對的記憶體成本也很貴，因為這些記憶體是直接用靜態記憶體的方式內建在 CPU 當中的。

這種方式比較像 SOC 系統單晶片的做法，在程式很小的情況之下，直接將記憶體包入 SOC 當中，會得到比較高速的電路，可惜的是這種做法不像目前的電腦架構一樣，是採用外掛 DRAM 的方式，可以大幅降低記憶體的成本，增大記憶體的容量 就是了。

CPU0 完整版 -- cpu0s

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 的指令集

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。

A型

OP	Ra	Rb	Rc	Cx{12 bits}
31-24	23-20	19-16	15-12	11-0

L型

OP	Ra	Rb	Cx{16 bits}
31-24	23-20	19-16	15-0

J型

OP	Cx{24 bits}
31-24	23-0

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)。

CPU0 處理器的 Verilog 實作 -- CPU0sc.v

檔案：cpu0sc.v

```
`define PC    R[15]    // 程式計數器
`define LR    R[14]    // 連結暫存器
`define SP    R[13]    // 堆疊暫存器
`define SW    R[12]    // 狀態暫存器

// 狀態暫存器旗標位元
`define N    `SW[31]  // 負號旗標
`define Z    `SW[30]  // 零旗標
`define C    `SW[29]  // 進位旗標
`define V    `SW[28]  // 溢位旗標
`define I    `SW[7]   // 硬體中斷許可
`define T    `SW[6]   // 軟體中斷許可
`define M    `SW[0]   // 模式位元
```

```
module cpu0c(input clock); // CPU0-Mini 的快取版: cpu0mc 模組
    parameter [7:0] LD=8' h00, ST=8' h01, LDB=8' h02, STB=8' h03, LDR=8' h04, ST
    R=8' h05,
        LBR=8' h06, SBR=8' h07, ADDI=8' h08, CMP=8' h10, MOV=8' h12, ADD=8' h13, SUB
    =8' h14,
        MUL=8' h15, DIV=8' h16, AND=8' h18, OR=8' h19, XOR=8' h1A, ROL=8' h1C, ROR=8
    ' h1D,
        SHL=8' h1E, SHR=8' h1F, JEQ=8' h20, JNE=8' h21, JLT=8' h22, JGT=8' h23, JLE=
    8' h24,
        JGE=8' h25, JMP=8' h26, SWI=8' h2A, CALL=8' h2B, RET=8' h2C, IRET=8' h2D,
        PUSH=8' h30, POP=8' h31, PUSHB=8' h32, POPB=8' h33;
    reg signed [31:0] R [0:15]; // 宣告暫存器 R[0..15] 等 16 個 32 位
    元暫存器
    reg signed [31:0] IR;           // 指令暫存器 IR
    reg [7:0] m [0:256];          // 內部的快取記憶體
    reg [7:0] op;                 // 變數: 運算代碼 op
```

```
reg [3:0] ra, rb, rc;           // 變數: 暫存器代號 ra, rb, rc
reg [4:0] c5;                  // 變數: 5 位元常數 c5
reg signed [11:0] c12;          // 變數: 12 位元常數 c12
reg signed [15:0] c16;          // 變數: 16 位元常數 c16
reg signed [23:0] c24;          // 變數: 24 位元常數 c24
reg signed [31:0] sp, jaddr, laddr, raddr;
reg signed [31:0] temp;
reg signed [31:0] pc;
```

```
integer i;
initial // 初始化
begin
    `PC = 0;                      // 將 PC 設為起動位址 0
    `SW = 0;
    R[0] = 0;                     // 將 R[0] 暫存器強制設定為 0
    $readmemh("cpu0s.hex", m);
    for (i=0; i < 255; i=i+4) begin
        $display("%8x: %8x", i, {m[i], m[i+1], m[i+2], m[i+3]});
```

```
end
end

always @ (posedge clock) begin // 在 clock 時脈的正邊緣時觸發
    pc = `PC;
    IR = {m[`PC], m[`PC+1], m[`PC+2], m[`PC+3]}; // 指令擷取階段
: IR=m[PC], 4 個 Byte 的記憶體
    `PC = `PC+4; // 擷取完成, PC
前進到下一個指令位址
    {op, ra, rb, rc, c12} = IR; // 解碼階段: 將 I
R 解為 {op, ra, rb, rc, c12}
    c5 = IR[4:0];
    c24 = IR[23:0];
    c16 = IR[15:0];
    jaddr = `PC+c16;
    laddr = R[rb]+c16;
    raddr = R[rb]+R[rc];
    case (op) // 根據 OP 執行對應的動作
```

```
LD: begin // 載入指令: R[ra] = m[addr]
    R[ra] = {m[laddr], m[laddr+1], m[laddr+2], m[laddr+3]};
    $display("%4dns %8x : LD      R%-d R%-d 0x%x ; R%-2d=0x%x=%
-d", $stime, pc, ra, rb, c16, ra, R[ra], R[ra]);
end

ST: begin // 儲存指令: m[addr] = R[ra]
    {m[laddr], m[laddr+1], m[laddr+2], m[laddr+3]} = R[ra];
    $display("%4dns %8x : ST      R%-d R%-d 0x%x ; R%-2d=0x%x=%
-d", $stime, pc, ra, rb, c16, ra, R[ra], R[ra]);
end

LDB:begin // 載入byte;      LDB Ra, [Rb+ Cx];      Ra<=(byte) [
Rb+ Cx]
    R[ra] = { 24' b0, m[laddr] } ;
    $display("%4dns %8x : LDB      R%-d R%-d 0x%x ; R%-2d=0x%x=%
-d", $stime, pc, ra, rb, c16, ra, R[ra], R[ra]);
end

STB:begin // 儲存byte;      STB Ra, [Rb+ Cx];      Ra=>(byte) [
Rb+ Cx]
```

```

m[laddr] = R[ra][7:0];
$display("%4dns %8x : STB    R%-d R%-d 0x%x ; R%-2d=0x%x=%
-d", $stime, pc, ra, rb, c16, ra, R[ra], R[ra]);
end

LDR:begin // LD 的 Rc 版; LDR Ra, [Rb+Rc]; Ra<=[Rb+ Rc
]
    R[ra] = {m[raddr], m[raddr+1], m[raddr+2], m[raddr+3]};
    $display("%4dns %8x : LDR    R%-d R%-d R%-d      ; R%-2d=0x%8
x=%-d", $stime, pc, ra, rb, rc, ra, R[ra], R[ra]);
end

STR:begin // ST 的 Rc 版; STR Ra, [Rb+Rc]; Ra=>[Rb+ Rc
]
    {m[raddr], m[raddr+1], m[raddr+2], m[raddr+3]} = R[ra];
    $display("%4dns %8x : STR    R%-d R%-d R%-d      ; R%-2d=0x%8
x=%-d", $stime, pc, ra, rb, rc, ra, R[ra], R[ra]);
end

LBR:begin // LDB 的 Rc 版; LBR Ra, [Rb+Rc]; Ra<=(byte) [
Rb+ Rc]

```

```

        R[ra] = { 24' b0, m[raddr] } ;
        $display("%4dns %8x : LBR    R%-d R%-d R%-d      ; R%-2d=0x%8
x=%-d", $stime, pc, ra, rb, rc, ra, R[ra], R[ra]);
        end

SBR:begin // STB 的 Rc 版; SBR Ra, [Rb+Rc];      Ra=>(byte) [
Rb+ Rc]
        m[raddr] = R[ra][7:0];
        $display("%4dns %8x : SBR    R%-d R%-d R%-d      ; R%-2d=0x%8
x=%-d", $stime, pc, ra, rb, rc, ra, R[ra], R[ra]);
        end

MOV:begin // 移動;           MOV Ra, Rb;      Ra<=Rb
        R[ra] = R[rb];
        $display("%4dns %8x : MOV    R%-d R%-d      ; R%-2d=0x%8x
=%-d", $stime, pc, ra, rb, ra, R[ra], R[ra]);
        end

CMP:begin // 比較;           CMP Ra, Rb;      SW=(Ra >=< R
b)
        temp = R[ra]-R[rb];

```

```

`N=(temp<0); `Z=(temp==0);

$display("%4dns %8x : CMP    R%-d R%-d      ; SW=0x%x", $stime, pc, ra, rb, `SW);

end

ADDI:begin // R[a] = Rb+c16; // 立即值加法; LDI Ra, Rb+C
x; Ra<=Rb + Cx

    R[ra] = R[rb]+c16;

    $display("%4dns %8x : ADDI   R%-d R%-d %-d ; R%-2d=0x%8x=%-d",
$stime, pc, ra, rb, c16, ra, R[ra], R[ra]);

end

ADD: begin // 加法指令: R[ra] = R[rb]+R[rc]
R[ra] = R[rb]+R[rc];

$display("%4dns %8x : ADD    R%-d R%-d R%-d      ; R%-2d=0x%8
x=%-d", $stime, pc, ra, rb, rc, ra, R[ra], R[ra]);

end

SUB:begin // 減法;           SUB Ra, Rb, Rc;       Ra<=Rb-Rc
R[ra] = R[rb]-R[rc];

$display("%4dns %8x : SUB    R%-d R%-d R%-d      ; R%-2d=0x%8

```

```

x=%-d", $stime, pc, ra, rb, rc, ra, R[ra], R[ra]);
    end
    MUL:begin // 乘法;           MUL Ra, Rb, Rc;      Ra<=Rb*Rc
        R[ra] = R[rb]*R[rc];
        $display("%4dns %8x : MUL    R%-d R%-d R%-d    ; R%-2d=0x%8
x=%-d", $stime, pc, ra, rb, rc, ra, R[ra], R[ra]);
    end
    DIV:begin // 除法;           DIV Ra, Rb, Rc;      Ra<=Rb/Rc
        R[ra] = R[rb]/R[rc];
        $display("%4dns %8x : DIV    R%-d R%-d R%-d    ; R%-2d=0x%8
x=%-d", $stime, pc, ra, rb, rc, ra, R[ra], R[ra]);
    end
    AND:begin // 位元 AND;     AND Ra, Rb, Rc;      Ra<=Rb and R
c
        R[ra] = R[rb]&R[rc];
        $display("%4dns %8x : AND    R%-d R%-d R%-d    ; R%-2d=0x%8
x=%-d", $stime, pc, ra, rb, rc, ra, R[ra], R[ra]);

```

```
    end

    OR:begin // 位元 OR;      OR Ra, Rb, Rc;      Ra<=Rb or
        Rc

        R[ra] = R[rb] | R[rc];
        $display("%4dns %8x : OR      R%-d R%-d R%-d      ; R%-2d=0x%8
x=%-d", $stime, pc, ra, rb, rc, ra, R[ra], R[ra]);
    end

    XOR:begin // 位元 XOR;      XOR Ra, Rb, Rc;      Ra<=Rb xor R
        C
        R[ra] = R[rb] ^ R[rc];
        $display("%4dns %8x : XOR      R%-d R%-d R%-d      ; R%-2d=0x%8
x=%-d", $stime, pc, ra, rb, rc, ra, R[ra], R[ra]);
    end

    SHL:begin // 向左移位;      SHL Ra, Rb, Cx;      Ra<=Rb << Cx
        R[ra] = R[rb]<<c5;
        $display("%4dns %8x : SHL      R%-d R%-d %-d      ; R%-2d=0x%8
x=%-d", $stime, pc, ra, rb, c5, ra, R[ra], R[ra]);
    end
```

SHR:begin // 向右移位; SHR Ra, Rb, Cx; Ra<=Rb >

> Cx

R[ra] = R[rb]>>c5;

\$display("%4dns %8x : SHR R%-d R%-d %-d ; R%-2d=0x%x
x=%-d", \$stime, pc, ra, rb, c5, ra, R[ra], R[ra]);
end

JMP:begin // 跳跃指令: PC = PC + cx24

`PC = `PC + c24;

\$display("%4dns %8x : JMP 0x%x ; PC=0x%x", \$stime,
pc, c24, `PC);
end

JEQ:begin // 跳跃 (相等); JEQ Cx; if SW(=) P

C PC+Cx

if (^Z) `PC=`PC+c24;

\$display("%4dns %8x : JEQ 0x%08x ; PC=0x%x", \$stime,
pc, c24, `PC);
end

JNE:begin // 跳跃 (不相等); JNE Cx; if SW(!=) PC P

C+Cx

```
if (!`Z) `PC=`PC+c24;
$display("%4dns %8x : JNE 0x%08x      ; PC=0x%x", $stime,
pc, c24, `PC);
end
JLT:begin // 跳躍 ( < );
          JLT Cx;      if SW(<) PC P
```

C+Cx

```
if (`N) `PC=`PC+c24;
$display("%4dns %8x : JLT 0x%08x      ; PC=0x%x", $stime,
pc, c24, `PC);
end
JGT:begin // 跳躍 ( > );
          JGT Cx;      if SW(>) PC P
```

C+Cx

```
if (!`N&&!`Z) `PC=`PC+c24;
$display("%4dns %8x : JGT 0x%08x      ; PC=0x%x", $stime,
pc, c24, `PC);
end
JLE:begin // 跳躍 ( <= );
          JLE Cx;      if SW(<=) PC
```

PC+Cx

```
    if (^N || ^Z) `PC=`PC+c24;
        $display("%4dns %8x : JLE 0x%08x      ; PC=0x%x", $stime,
pc, c24, `PC);
    end
```

```
JGE:begin // 跳躍 ( >= );
JGE Cx;      if SW(>=) PC
```

PC+Cx

```
    if (!^N || ^Z) `PC=`PC+c24;
        $display("%4dns %8x : JGE 0x%08x      ; PC=0x%x", $stime,
pc, c24, `PC);
    end
```

```
SWI:begin // 軟中斷;      SWI Cx;      LR <= PC; PC <= Cx
```

; INT<=1

```
    `LR=`PC; `PC= c24; `I = 1'b1;
        $display("%4dns %8x : SWI 0x%08x      ; PC=0x%x", $stime,
pc, c24, `PC);
    end
```

```
CALL:begin // 跳到副程式;      CALL Cx;      LR<=PC; PC<=PC+Cx
```

```

`LR=`PC;`PC=`PC + c24;
$display("%4dns %8x : CALL 0x%08x      ; PC=0x%x", $stime,
pc, c24, `PC);
end

RET:begin // 返回;           RET;           PC <= LR
`PC=`LR;
$display("%4dns %8x : RET      ; PC=0x%x", $stime,
pc, `PC);
if (`PC<0) $finish;
end

IRET:begin // 中斷返回;       IRET;          PC <= LR; INT<
=0
`PC=`LR;`I = 1'b0;
$display("%4dns %8x : IRET      ; PC=0x%x", $stime,
pc, `PC);
end

PUSH:begin // 推入 word;    PUSH Ra;        SP-=4; [SP]<=Ra;
sp = `SP-4; `SP = sp; {m[sp], m[sp+1], m[sp+2], m[sp+3]} =

```

```
R[ra];  
    $display("%4dns %8x : PUSH R%-d          ; R%-2d=0x%8x,  
SP=0x%x", $stime, pc, ra, R[ra], `SP);  
    end  
POP:begin // 弹出 word;      POP Ra;      Ra=[SP];SP+=4;  
    sp = `SP; R[ra]={m[sp], m[sp+1], m[sp+2], m[sp+3]}; `SP =  
sp+4;  
    $display("%4dns %8x : POP   R%-d          ; R%-2d=0x%8x,  
SP=0x%x", $stime, pc, ra, R[ra], `SP);  
    end  
PUSHB:begin // 推入 byte;    PUSHB Ra;    SP--;[SP]<=Ra; (byte  
)  
    sp = `SP-1; `SP = sp; m[sp] = R[ra];  
    $display("%4dns %8x : PUSHB R%-d          ; R[%-d]=0x%8x  
, SP=0x%x", $stime, pc, ra, R[ra], `SP);  
    end  
POPB:begin // 弹出 byte;    POPB Ra;    Ra<=[SP];SP++; (byte)  
    sp = `SP+1; `SP = sp; R[ra]=m[sp];
```

```
$display("%4dns %8x : POPB R%-d ; R[%-d]=0x%8x
, SP=0x%x", $stime, pc, ra, R[ra], `SP);
end
endcase
end
endmodule

module main; // 測試程式開始
reg clock; // 時脈 clock 變數

cpu0c cpu(clock); // 宣告 cpu0mc 處理器

initial clock = 0; // 一開始 clock 設定為 0
always #10 clock=~clock; // 每隔 10 奈秒將 clock 反相，產生週期為
20 奈秒的時脈
initial #2000 $finish; // 在 640 奈秒的時候停止測試。(因為這時的
R[1] 恰好是  $1+2+\dots+10=55$  的結果)
```

```
endmodule
```

程式碼解析與執行

在上一章的 CPU0mc.v 當中，我們直接使用下列程式將「機器碼」塞入到記憶體當中，但是這樣做顯然彈性不太夠。

```
{m[0], m[1], m[2], m[3]} = 32' h001F0018; // 0000 LD R1,  
K1  
{m[4], m[5], m[6], m[7]} = 32' h002F0010; // 0004 LD R2,  
K0  
{m[8], m[9], m[10], m[11]} = 32' h003F0014; // 0008 LD R3,  
SUM  
{m[12], m[13], m[14], m[15]}= 32' h13221000; // 000C LOOP: ADD R2,  
R2, R1  
{m[16], m[17], m[18], m[19]}= 32' h13332000; // 0010 ADD R3,  
R3, R2  
{m[20], m[21], m[22], m[23]}= 32' h26FFFFF4; // 0014 JMP LOOP  
{m[24], m[25], m[26], m[27]}= 32' h00000000; // 0018 K0: WORD 0
```

```
{m[28], m[29], m[30], m[31]}= 32' h00000001; // 001C K1: WORD 1  
{m[32], m[33], m[34], m[35]}= 32' h00000000; // 0020 SUM: WORD 0
```

因此，在本章的 CPU0sc.v 這個程式中，我們採用讀取外部檔案的方式，將機器碼寫在「cpu0s.hex」這個檔案中，然後再用下列指令將該 16 進位的機器碼檔案讀入。

```
$readmemh("cpu0s.hex", m);
```

其中的 readmemh 是一個可以讀取 16 進位的文字檔的函數，上述指令會將 cpu0s.hex 這個檔案內的 16 進位字串 讀入到「記憶體變數」 m 當中。

以下是 cpu0s.hex 的完整內容。

輸入檔：cpu0s.hex

00 DF 00 B6 // 0	LD R13, StackEnd
08 40 00 04 // 4	ADDI R4, 4
08 50 00 08 // 8	ADDI R5, 8
05 4D 50 00 // c	STR R4, [R13+R5]
04 6D 50 00 // 10	LDR R6, [R13+R5]
07 5D 40 00 // 14	SBR R5, [R13+R4]

06 6D 40 00	// 18	LBR R6, [R13+R4]
08 E0 FF FF	// 1C	ADDI R14, R0, -1
30 E0 00 00	// 20	PUSH R14
13 85 40 00	// 24	ADD R8, R5, R4
14 85 40 00	// 28	SUB R8, R5, R4
15 85 40 00	// 2c	MUL R8, R5, R4
16 85 40 00	// 30	DIV R8, R5, R4
18 85 40 00	// 34	AND R8, R5, R4
19 85 40 00	// 38	OR R8, R5, R4
1A 85 40 00	// 3c	XOR R8, R5, R4
1E 85 00 03	// 40	SHL R8, R5, 3
1F 85 00 02	// 44	SHR R8, R5, 2
10 45 00 00	// 48	CMP R4, R5
20 00 00 18	// 4c	JEQ L1
23 00 00 14	// 50	JGT L1
25 00 00 10	// 54	JGE L1
22 00 00 0C	// 58	JLT L1
24 00 00 08	// 5c	JLE L1

21 00 00 04 // 60	JNE L1
26 00 00 00 // 64	JMP L1
08 10 00 0A // 68	L1: ADDI R1, R0, 10
2B 00 00 08 // 6c	CALL SUM
31 E0 00 00 // 70	POP R14
2C 00 00 00 // 74	RET
30 E0 00 00 // 78	SUM: PUSH R14
12 30 00 00 // 7c	MOV R3, R0 // R3 = i = 0
02 4F 00 24 // 80	LDB R4, k1 // R4 = 1
08 20 00 00 // 84	ADDI R2, 0 // SUM = R2 = 0
13 22 30 00 // 88	LOOP: ADD R2, R2, R3 // SUM = SUM + i
13 33 40 00 // 8c	ADD R3, R3, R4 // i = i + 1
10 31 00 00 // 90	CMP R3, R1 // if (i < R1)
24 FF FF F0 // 94	JLE LOOP // goto LOOP
01 2F 00 0D // 98	ST R2, s
03 3F 00 0D // 9c	STB R3, i
31 E0 00 00 // a0	POP R14
2C 00 00 00 // a4	RET // return

01	// a8	k1:	BYTE 1	// char K1=1
00 00 00 00	// a9	s:	WORD 0	// int s
00	// ad	i:	BYTE 0	// char i=1
00 01 02 03	// ae	Stack:	BYTE 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 , 10	
, 11				
04 05 06 07	// b2			
08 09 0A 0B	// b6			
00 00 00 BA	// ba	StackEnd:	WORD StackEnd	
01 02 03 04	// be	Data:	BYTE 0, 1, 2, 3, 4, 5, 6, 7, 8	
05 06 07 08	// c2			

上述程式的內容，大致是先準備好堆疊，然後就開始測試 ADDI, STR, LDR, ADD, SUB, ... 等指令。接著在呼叫 CMP R4, R5 之後進行跳躍測試動作，由於 R4=4, R5=8，所以 CMP 的結果會是「小於」，因此在後面的 JEQ, JGT, JGE 等指令都不會真的跳躍，直到執行 JLT L1 時就會真的跳到 L1 去。

接著用 ADDI R1, R0, 10 將 R1 設為 10，然後就用 CALL SUM 這個指令呼叫 SUM 這個副程式，於是跳到位於 0x78 的 SUM: PUSH R14 這一行，並開始執行副程式，該副程式會計算 $1+2+\dots+R1$ 的結果，放在 R2 當中，並在最後用 STB R2, s 這個指令存入變數 s 當中，然後在執行完 RET 指令後返回上一層，也就是 0x70 行的 POP R14 指令，接著在執行 RET 指令時，由於此時 R14 為 -1，因此

Verilog 程式就在完成 RET 指令時發現 PC 已經小於 0 了，因此執行\$finish` 指令停止整個程式。

```
RET:begin // 返回;           RET;          PC <= LR  
  `PC=`LR;  
  $display("%dns %8x : RET  
ime, pc, `PC);  
  if (`PC<0) $finish;  
end
```

執行結果

有了上述的程式 cpu0sc.v 與輸入的機器碼 cpu0s.hex 檔案之後，我們就可以用下列指令進行編譯與執行，以下是該程式編譯與執行的結果。

```
D:\verilog>iverilog -o cpu0sc cpu0sc.v
```

```
D:\verilog>vvp cpu0sc
```

```
WARNING: cpu0sc.v:40: $readmemh(cpu0s.hex): Not enough words in the  
file for the
```

requested range [0:256].

00000000: 00df00b6

00000004: 08400004

00000008: 08500008

0000000c: 054d5000

00000010: 046d5000

00000014: 075d4000

00000018: 066d4000

0000001c: 08e0ffff

00000020: 30e00000

00000024: 13854000

00000028: 14854000

0000002c: 15854000

00000030: 16854000

00000034: 18854000

00000038: 19854000

0000003c: 1a854000

00000040: 1e850003

00000044:	1f850002
00000048:	10450000
0000004c:	20000018
00000050:	23000014
00000054:	25000010
00000058:	2200000c
0000005c:	24000008
00000060:	21000004
00000064:	26000000
00000068:	0810000a
0000006c:	2b000008
00000070:	31e00000
00000074:	2c000000
00000078:	30e00000
0000007c:	12300000
00000080:	024f0024
00000084:	08200000
00000088:	13223000

0000008c:	13334000
00000090:	10310000
00000094:	24fffff0
00000098:	012f000d
0000009c:	033f000d
000000a0:	31e00000
000000a4:	2c000000
000000a8:	01000000
000000ac:	00000001
000000b0:	02030405
000000b4:	06070809
000000b8:	0a0b0000
000000bc:	00ba0102
000000c0:	03040506
000000c4:	0708xxxx
000000c8:	xxxxxxxx
000000cc:	xxxxxxxx
000000d0:	xxxxxxxx

000000d4: xxxxxxxx
000000d8: xxxxxxxx
000000dc: xxxxxxxx
000000e0: xxxxxxxx
000000e4: xxxxxxxx
000000e8: xxxxxxxx
000000ec: xxxxxxxx
000000f0: xxxxxxxx
000000f4: xxxxxxxx
000000f8: xxxxxxxx
000000fc: xxxxxxxx

10ns 00000000 : LD R13 R15 0x00b6 ; R13=0x000000ba=186
30ns 00000004 : ADDI R4 R0 4 ; R4 =0x00000004=4
50ns 00000008 : ADDI R5 R0 8 ; R5 =0x00000008=8
70ns 0000000c : STR R4 R13 R5 ; R4 =0x00000004=4
90ns 00000010 : LDR R6 R13 R5 ; R6 =0x00000004=4
110ns 00000014 : SBR R5 R13 R4 ; R5 =0x00000008=8
130ns 00000018 : LBR R6 R13 R4 ; R6 =0x00000008=8

150ns	0000001c	:	ADDI	R14	R0	-1	;	R14=0xffffffff=-1
170ns	00000020	:	PUSH	R14			;	R14=0xffffffff, SP=0x000000

b6

190ns	00000024	:	ADD	R8	R5	R4	;	R8 =0x0000000c=12
210ns	00000028	:	SUB	R8	R5	R4	;	R8 =0x00000004=4
230ns	0000002c	:	MUL	R8	R5	R4	;	R8 =0x00000020=32
250ns	00000030	:	DIV	R8	R5	R4	;	R8 =0x00000002=2
270ns	00000034	:	AND	R8	R5	R4	;	R8 =0x00000000=0
290ns	00000038	:	OR	R8	R5	R4	;	R8 =0x0000000c=12
310ns	0000003c	:	XOR	R8	R5	R4	;	R8 =0x0000000c=12
330ns	00000040	:	SHL	R8	R5	3	;	R8 =0x00000040=64
350ns	00000044	:	SHR	R8	R5	2	;	R8 =0x00000002=2
370ns	00000048	:	CMP	R4	R5		;	SW=0x80000000
390ns	0000004c	:	JEQ	0x00000018			;	PC=0x00000050
410ns	00000050	:	JGT	0x00000014			;	PC=0x00000054
430ns	00000054	:	JGE	0x00000010			;	PC=0x00000058
450ns	00000058	:	JLT	0x0000000c			;	PC=0x00000068
470ns	00000068	:	ADDI	R1	R0	10	;	R1 =0x0000000a=10

490ns	0000006c	:	CALL	0x00000008	;	PC=0x00000078
510ns	00000078	:	PUSH	R14	;	R14=0x00000070, SP=0x0000000

b2

530ns	0000007c	:	MOV	R3 R0	;	R3 =0x00000000=0
550ns	00000080	:	LDB	R4 R15 0x0024	;	R4 =0x00000001=1
570ns	00000084	:	ADDI	R2 R0 0	;	R2 =0x00000000=0
590ns	00000088	:	ADD	R2 R2 R3	;	R2 =0x00000000=0
610ns	0000008c	:	ADD	R3 R3 R4	;	R3 =0x00000001=1
630ns	00000090	:	CMP	R3 R1	;	SW=0x80000000
650ns	00000094	:	JLE	0x00fffff0	;	PC=0x00000088
670ns	00000088	:	ADD	R2 R2 R3	;	R2 =0x00000001=1
690ns	0000008c	:	ADD	R3 R3 R4	;	R3 =0x00000002=2
710ns	00000090	:	CMP	R3 R1	;	SW=0x80000000
730ns	00000094	:	JLE	0x00fffff0	;	PC=0x00000088
750ns	00000088	:	ADD	R2 R2 R3	;	R2 =0x00000003=3
770ns	0000008c	:	ADD	R3 R3 R4	;	R3 =0x00000003=3
790ns	00000090	:	CMP	R3 R1	;	SW=0x80000000
810ns	00000094	:	JLE	0x00fffff0	;	PC=0x00000088

830ns	00000088	:	ADD	R2	R2	R3		;	R2 =0x00000006=6
850ns	0000008c	:	ADD	R3	R3	R4		;	R3 =0x00000004=4
870ns	00000090	:	CMP	R3	R1			;	SW=0x80000000
890ns	00000094	:	JLE	0x00fffff0				;	PC=0x00000088
910ns	00000088	:	ADD	R2	R2	R3		;	R2 =0x0000000a=10
930ns	0000008c	:	ADD	R3	R3	R4		;	R3 =0x00000005=5
950ns	00000090	:	CMP	R3	R1			;	SW=0x80000000
970ns	00000094	:	JLE	0x00fffff0				;	PC=0x00000088
990ns	00000088	:	ADD	R2	R2	R3		;	R2 =0x0000000f=15
1010ns	0000008c	:	ADD	R3	R3	R4		;	R3 =0x00000006=6
1030ns	00000090	:	CMP	R3	R1			;	SW=0x80000000
1050ns	00000094	:	JLE	0x00fffff0				;	PC=0x00000088
1070ns	00000088	:	ADD	R2	R2	R3		;	R2 =0x00000015=21
1090ns	0000008c	:	ADD	R3	R3	R4		;	R3 =0x00000007=7
1110ns	00000090	:	CMP	R3	R1			;	SW=0x80000000
1130ns	00000094	:	JLE	0x00fffff0				;	PC=0x00000088
1150ns	00000088	:	ADD	R2	R2	R3		;	R2 =0x0000001c=28

1170ns	0000008c	:	ADD	R3	R3	R4		;	R3 =0x00000008=8
1190ns	00000090	:	CMP	R3	R1			;	SW=0x80000000
1210ns	00000094	:	JLE	0x00fffff0				;	PC=0x00000088
1230ns	00000088	:	ADD	R2	R2	R3		;	R2 =0x00000024=36
1250ns	0000008c	:	ADD	R3	R3	R4		;	R3 =0x00000009=9
1270ns	00000090	:	CMP	R3	R1			;	SW=0x80000000
1290ns	00000094	:	JLE	0x00fffff0				;	PC=0x00000088
1310ns	00000088	:	ADD	R2	R2	R3		;	R2 =0x0000002d=45
1330ns	0000008c	:	ADD	R3	R3	R4		;	R3 =0x0000000a=10
1350ns	00000090	:	CMP	R3	R1			;	SW=0x40000000
1370ns	00000094	:	JLE	0x00fffff0				;	PC=0x00000088
1390ns	00000088	:	ADD	R2	R2	R3		;	R2 =0x00000037=55
1410ns	0000008c	:	ADD	R3	R3	R4		;	R3 =0x0000000b=11
1430ns	00000090	:	CMP	R3	R1			;	SW=0x00000000
1450ns	00000094	:	JLE	0x00fffff0				;	PC=0x00000098
1470ns	00000098	:	ST	R2	R15	0x000d	;	R2 =0x00000037=55	
1490ns	0000009c	:	STB	R3	R15	0x000d	;	R3 =0x0000000b=11	
1510ns	000000a0	:	POP	R14				;	R14=0x00000070, SP=0x000000

b6

```
1530ns 000000a4 : RET ; PC=0x00000070
1550ns 00000070 : POP R14 ; R14=0xffffffff, SP=0x000000
ba
1570ns 00000074 : RET ; PC=0xffffffffffff
```

結語

從本章的內容中，您應該可以瞭解到直接使用高階的 Verilog 流程式語法來設計處理器，像是 `cpu0mc.v` 與 `cpu0sc.v`，都是相當容易的事，這完全是因為 verilog 支援了相當高階的運算，像是「`+, -, *, /, &, |, ^, <<, >>`」等運算的原故。

不過、在上述程式當中，我們並沒有支援「硬體中斷」的功能，也沒有實作「軟體中斷」SWI 的函數呼叫，這樣 `CPU0sc.v` 就只能是一顆單工 (Single Task) 的處理器，而無法支援多工 (Multi Task) 的功能了。

在下一章當中，我們將繼續擴充 `CPU0sc.v` 這個程式，加入支援「軟硬體中斷」的功能，以及支援「浮點運算」的功能，該程式稱為 `CPU0ic.v` (`i` 代表 Interrupt, `c` 代表 cache memory)。

然後我們將再度用 16 進位的方式，寫出一個機器語言的程式，可以同時執行兩個「行程」(Task)，並且每隔一小段時間就利用 硬體中斷進行「行程切換」，以示範如何設計一個可以支援「多工」

能力的 CPU 處理器。

參考文獻

- 陳鍾誠的網站：使用 Verilog 設計 CPU0 處理器
- 陳鍾誠的網站：CPU0-Mini 處理器設計

速度議題

乘法與除法

for 迴圈的實作方法

最簡單的乘法器是移位乘法器，這種乘法器基本上只用了一個加法器和一個移位器所組成，電路上而言相當簡單，但缺點是執行速度不快，以下是一個 32 位元的移位乘法器之程式碼。

參考：<http://www.edaboard.com/thread235191.html>

檔案：shift_mult.v

```
module multiplier(output reg [63:0] z, input [31:0] x, y);
reg [31:0] a;
integer i;

always @(x , y)
begin
    a=x;
```

```
z=0; // 初始化為 0
for(i=0;i<31;i=i+1) begin
    if (y[i])
        z = z + a; // 請注意，這是 block assignment =，所以會造成延遲，長度越長延遲越久。
        a=a << 1;
    end
end
endmodule

module main;
reg [31:0] a, b;
wire [63:0] c;

multiplier m(c, a, b);

initial begin
    a = 17;
```

```
b = 7;  
#10;  
$display("a=%d b=%d c=%d", a, b, c);  
end  
  
endmodule
```

其編譯執行結果如下：

```
D:\Dropbox\Public\web\oc\code>iverilog -o mul32a mul32a.v  
  
D:\Dropbox\Public\web\oc\code>vvp mul32a  
a= 17 b= 7 c= 119
```

但是、以上的這個程式，採用了 for 迴圈，而且使用 = 這種阻隔式的方式，應該會相當耗費電路資源，筆者認為上述電路應該會用到 32 組 32 位元的加法器，這樣實在是難以接受的一種情況，所以我們接下來將採用只有一組加法器，然後用多個時脈的方式來設計乘法器。

參考：<http://www.edaboard.com/thread86772.html>

1. for loop verilog synthesis

It is synthesizable but it is always advised that for loops are not to be used in RTL coding. This is because it consumes lot of resources (like area etc.etc) . However u can use it in behavioral coding becuse we do not synthesize behavioral codes.

2. verilog for loop syntax

In verilog,synthesizable of for loop and while loop depends on which tools you are using . But it is better dont use it in RTL because it reflects replica of hardware.

32 個時脈的實作方法

```
module multiplier(output reg [63:0] z, output ready, input [31:0] x  
, y, input start, clk);  
    reg [5:0] bit;  
    reg [31:0] a;  
  
    wire ready = !bit;
```

```
initial bit = 0;

always @( posedge clk ) begin
    if( ready && start ) begin
        bit = 32;
        z = 0;
        a = x;
    end else begin
        bit = bit - 1;
        z = z << 1;
        if (y[bit])
            z = z + a;
    end
    ! [] (./timg/f227758f3c73.jpg) stime, x, y, z, ready, bit);
end
endmodule
```

```
module main;
reg [31:0] a, b;
wire [63:0] c;
reg clk, start;
wire ready;

multiplier m(c, ready, a, b, start, clk);

initial begin
    a = 17;
    b = 7;
    start = 1;
    clk = 0;
    #200;
    start = 0;
    #200;
    $finish;
end
```

```
always #5 clk = !clk;  
// always @(posedge clk)  stime, a, b,  
c, ready);  
  
endmodule
```

執行結果

```
D:\Dropbox\Public\web\oc\code>iverilog -o mul32b mul32b.v
```

```
D:\Dropbox\Public\web\oc\code>vvp mul32b
```

```
      5ns : x=      17  y=      7  z=          0  read
```

```
y=0 bit=32
```

```
      15ns : x=      17  y=      7  z=          0  read
```

```
y=0 bit=31
```

```
      25ns : x=      17  y=      7  z=          0  read
```

```
y=0 bit=30
```

```
      35ns : x=      17  y=      7  z=          0  read
```

y=0 bit=29
 45ns : x= 17 y= 7 z= 0 read

y=0 bit=28
 55ns : x= 17 y= 7 z= 0 read

y=0 bit=27
 65ns : x= 17 y= 7 z= 0 read

y=0 bit=26
 75ns : x= 17 y= 7 z= 0 read

y=0 bit=25
 85ns : x= 17 y= 7 z= 0 read

y=0 bit=24
 95ns : x= 17 y= 7 z= 0 read

y=0 bit=23
 105ns : x= 17 y= 7 z= 0 read

y=0 bit=22
 115ns : x= 17 y= 7 z= 0 read

y=0 bit=21

125ns : x=	17	y=	7	z=	0	read
y=0 bit=20						
135ns : x=	17	y=	7	z=	0	read
y=0 bit=19						
145ns : x=	17	y=	7	z=	0	read
y=0 bit=18						
155ns : x=	17	y=	7	z=	0	read
y=0 bit=17						
165ns : x=	17	y=	7	z=	0	read
y=0 bit=16						
175ns : x=	17	y=	7	z=	0	read
y=0 bit=15						
185ns : x=	17	y=	7	z=	0	read
y=0 bit=14						
195ns : x=	17	y=	7	z=	0	read
y=0 bit=13						
205ns : x=	17	y=	7	z=	0	read
y=0 bit=12						

215ns : x=	17	y=	7	z=	0	read
y=0 bit=11						
225ns : x=	17	y=	7	z=	0	read
y=0 bit=10						
235ns : x=	17	y=	7	z=	0	read
y=0 bit= 9						
245ns : x=	17	y=	7	z=	0	read
y=0 bit= 8						
255ns : x=	17	y=	7	z=	0	read
y=0 bit= 7						
265ns : x=	17	y=	7	z=	0	read
y=0 bit= 6						
275ns : x=	17	y=	7	z=	0	read
y=0 bit= 5						
285ns : x=	17	y=	7	z=	0	read
y=0 bit= 4						
295ns : x=	17	y=	7	z=	0	read
y=0 bit= 3						

305ns : x=	17	y=	7	z=	17	read
y=0 bit= 2						
315ns : x=	17	y=	7	z=	51	read
y=0 bit= 1						
325ns : x=	17	y=	7	z=	119	read
y=1 bit= 0						
335ns : x=	17	y=	7	z=	238	read
y=0 bit=63						
345ns : x=	17	y=	7	z=	476	read
y=0 bit=62						
355ns : x=	17	y=	7	z=	952	read
y=0 bit=61						
365ns : x=	17	y=	7	z=	1904	read
y=0 bit=60						
375ns : x=	17	y=	7	z=	3808	read
y=0 bit=59						

檔案：div32.v

```
// 參考: http://www.ece.lsu.edu/ee3755/2002/107.html
// 參考: http://answers.google.com/answers/threadview/id/109219.html
// a/b = q ; a%b = r;

module divider(output reg [31:0] q, output [31:0] r, output ready,
input [31:0] a,b, input start, clk);
    reg [63:0]      ta, tb, diff;
    reg [5:0]       bit;
    wire           ready = !bit;

initial bit = 0;

assign r = ta[31:0];

always @(posedge clk)
    if( ready && start ) begin
        bit = 32;
```

```
q = 0;
ta = {32' d0, a};
tb = {1' b0, b, 31' d0};

end else begin
    diff = ta - tb;
    q = q << 1;
    if( !diff[63] ) begin
        ta = diff;
        q[0] = 1' d1;
    end
    tb = tb >> 1;
    bit = bit - 1;
end
endmodule

module main;
reg clk, start;
reg [31:0] a, b;
```

```
wire [31:0] q, r;  
wire ready;  
  
divider div(q, r, ready, a, b, start, clk);  
  
initial begin  
    clk = 0;  
    a = 90; b = 13; start = 1;  
    #200;  
    start = 0;  
    #200;  
    $finish;  
end  
  
always #5 clk = !clk;  
always @(posedge clk)  stime, a, b, q  
, r, ready);
```

```
endmodule
```

執行結果

```
D:\Dropbox\Public\web\oc\code>iverilog -o div32 div32.v
```

```
D:\Dropbox\Public\web\oc\code>vvp div32
```

	a	b	q	r
5ns : ready=0	90	13	0	90

15ns : ready=0	90	13	0	90
----------------	----	----	---	----

25ns : ready=0	90	13	0	90
----------------	----	----	---	----

35ns : ready=0	90	13	0	90
----------------	----	----	---	----

45ns : ready=0	90	13	0	90
----------------	----	----	---	----

55ns : ready=0	90	13	0	90
----------------	----	----	---	----

65ns : a=	90 b=	13 q=	0 r=	90 r
eady=0				
75ns : a=	90 b=	13 q=	0 r=	90 r
eady=0				
85ns : a=	90 b=	13 q=	0 r=	90 r
eady=0				
95ns : a=	90 b=	13 q=	0 r=	90 r
eady=0				
105ns : a=	90 b=	13 q=	0 r=	90 r
eady=0				
115ns : a=	90 b=	13 q=	0 r=	90 r
eady=0				
125ns : a=	90 b=	13 q=	0 r=	90 r
eady=0				
135ns : a=	90 b=	13 q=	0 r=	90 r
eady=0				
145ns : a=	90 b=	13 q=	0 r=	90 r

eady=0					
	155ns : a=	90 b=	13 q=	0 r=	90 r
eady=0					
	165ns : a=	90 b=	13 q=	0 r=	90 r
eady=0					
	175ns : a=	90 b=	13 q=	0 r=	90 r
eady=0					
	185ns : a=	90 b=	13 q=	0 r=	90 r
eady=0					
	195ns : a=	90 b=	13 q=	0 r=	90 r
eady=0					
	205ns : a=	90 b=	13 q=	0 r=	90 r
eady=0					
	215ns : a=	90 b=	13 q=	0 r=	90 r
eady=0					
	225ns : a=	90 b=	13 q=	0 r=	90 r
eady=0					
	235ns : a=	90 b=	13 q=	0 r=	90 r

eady=0					
	245ns : a=	90 b=	13 q=	0 r=	90 r
eady=0					
	255ns : a=	90 b=	13 q=	0 r=	90 r
eady=0					
	265ns : a=	90 b=	13 q=	0 r=	90 r
eady=0					
	275ns : a=	90 b=	13 q=	0 r=	90 r
eady=0					
	285ns : a=	90 b=	13 q=	0 r=	90 r
eady=0					
	295ns : a=	90 b=	13 q=	0 r=	90 r
eady=0					
	305ns : a=	90 b=	13 q=	1 r=	38 r
eady=0					
	315ns : a=	90 b=	13 q=	3 r=	12 r
eady=0					
	325ns : a=	90 b=	13 q=	6 r=	12 r

eady=1									
	335ns : a=	90	b=	13	q=	13	r=	6	r
eady=0									
	345ns : a=	90	b=	13	q=	27	r=	3	r
eady=0									
	355ns : a=	90	b=	13	q=	55	r=	2	r
eady=0									
	365ns : a=	90	b=	13	q=	111	r=	2	r
eady=0									
	375ns : a=	90	b=	13	q=	223	r=	2	r
eady=0									
	385ns : a=	90	b=	13	q=	447	r=	2	r
eady=0									
	395ns : a=	90	b=	13	q=	895	r=	2	r
eady=0									

- 參考：[維基百科：布斯乘法演算法](#)

以下範例來自維基百科

Booth 算法範例：

考慮一個由若干個 0 包圍著若干個 1 的正的[[二進制]]乘數，比如 0011110，積可以表達為：

$$M \times "0\ 0\ 1\ 1\ 1\ 1\ 0" = M \times (2^5 + 2^4 + 2^3 + 2^2 + 2^1) = M \times 62$$

其中，M 代表被乘數。變形為下式可以使運算次數可以減為兩次：

$$M \times "0\ 1\ 0\ 0\ 0\ 0\ -1\ 0" = M \times (2^6 - 2^1) = M \times 62$$

事實上，任何二進制數中連續的 1 可以被分解為兩個二進制數之差：

$$(\dots \overbrace{01\dots 1}^n 0\dots)_2 \equiv (\dots \overbrace{10\dots 0}^n 0\dots)_2 - (\dots \overbrace{00\dots 1}^n 0\dots)_2.$$

因此，我們可以用更簡單的運算來替換原數中連續為 1 的數字的乘法，通過加上乘數，對部分積進行移位運算，最後再將之從乘數中減去。它利用了我們在針對為零的位做乘法時，不需要做其他運算，只需移位這一特點，這很像我們在做和 99 的乘法時利用 $99=100-1$ 這一性質。

這種模式可以擴展應用於任何一串數字中連續為 1 的部分（包括只有一個 1 的情況）。那麼，

$$M \times "0\ 0\ 1\ 1\ 1\ 0\ 1\ 0" = M \times (2^5 + 2^4 + 2^3 + 2^1) = M \times 58$$

$$M \times "0\ 1\ 0\ 0 - 1\ 0\ 1\ 0" = M \times (2^6 - 2^3 + 2^1) = M \times 58$$

布斯算法遵從這種模式，它在遇到一串數字中的第一組從 0 到 1 的變化時（即遇到 01 時）執行加法，在遇到這一串連續 1 的尾部時（即遇到 10 時）執行減法。這在乘數為負時同樣有效。當乘數中的連續 1 比較多時（形成比較長的 1 串時），布斯算法較一般的乘法算法執行的加減法運算少。

浮點數運算

浮點運算單元 FALU

```
// 輸入 a, b 後會執行 op 所指定的運算，然後將結果放在暫存器 y 當中
module falu(input [63:0] ia, input [63:0] ib, input [1:0] op, output reg [63:0] oy);
real a, b, y;

always @(a or b or op) begin
    a = $bitstoreal(ia);
    b = $bitstoreal(ib);
```

```

case (op)
  2'b00: y = a + b;
  2'b01: y = a - b;
  2'b10: y = a * b;
  2'b11: y = a / b;
endcase
$display("falu32:op=%d a=%f b=%f y=%f", op, a, b, y);
oy = $realtobits(y);
end
endmodule

module main; // 測試程式開始
  reg [63:0] a64, b64;
  wire [63:0] y64;
  reg [1:0] op;
  real a, b;

  falu falu1(a64, b64, op, y64); // 建立一個 alu 單元，名稱為 alu

```

1

```
initial begin          // 測試程式的初始化動作
    a = 3.14159;   a64 = $realtobits(a);
    b = 2.71818;   b64 = $realtobits(b);
    op = 0;
end

always #50 begin        // 每個 50 奈秒就作下列動作
    op = op + 1;      // 讓 op 的值加 1
end

initial #1000 $finish;  // 時間到 1000 奈秒就結束

endmodule
```

執行結果

```
D:\Dropbox\Public\web\oc\code>iverilog -o flu64 flu64.v
```

```
D:\Dropbox\Public\web\oc\code>vvp flu64
```

```
falu32:op=0 a=3.141590 b=2.718180 y=5.859770
```

```
falu32:op=1 a=3.141590 b=2.718180 y=0.423410
```

```
falu32:op=2 a=3.141590 b=2.718180 y=8.539407
```

```
falu32:op=3 a=3.141590 b=2.718180 y=1.155770
```

```
falu32:op=0 a=3.141590 b=2.718180 y=5.859770
```

```
falu32:op=1 a=3.141590 b=2.718180 y=0.423410
```

```
falu32:op=2 a=3.141590 b=2.718180 y=8.539407
```

```
falu32:op=3 a=3.141590 b=2.718180 y=1.155770
```

```
falu32:op=0 a=3.141590 b=2.718180 y=5.859770
```

```
falu32:op=1 a=3.141590 b=2.718180 y=0.423410
```

```
falu32:op=2 a=3.141590 b=2.718180 y=8.539407
```

```
falu32:op=3 a=3.141590 b=2.718180 y=1.155770
```

```
falu32:op=0 a=3.141590 b=2.718180 y=5.859770
```

```
falu32:op=1 a=3.141590 b=2.718180 y=0.423410
```

```
falu32:op=2 a=3.141590 b=2.718180 y=8.539407  
falu32:op=3 a=3.141590 b=2.718180 y=1.155770  
falu32:op=0 a=3.141590 b=2.718180 y=5.859770  
falu32:op=1 a=3.141590 b=2.718180 y=0.423410  
falu32:op=2 a=3.141590 b=2.718180 y=8.539407  
falu32:op=3 a=3.141590 b=2.718180 y=1.155770
```

使用 Verilog 預設的浮點數運算

檔案： float.v

```
module main;  
real x, y, x2, y2, xyadd, xysub, xymul, xydiv;  
reg [63:0] x1, y1;  
  
initial  
begin  
x=3.14159;  
y=-2.3e4;
```

```
x1 = $realtobits(x) ;
x2 = $bitstoreal(x1) ;
y1 = $realtobits(y) ;
y2 = $bitstoreal(y1) ;
xyadd = x+y;
xysub = x-y;
xymul = x*y;
xydiv = x/y;
$display("x=%f x1=%b x2=%f", x, x1, x2) ;
$display("y=%f y1=%b y2=%f", y, y1, y2) ;
$display("x+y=%f xyadd=%f", x+y, xyadd) ;
$display("x-y=%f xysub=%f", x-y, xysub) ;
$display("x*y=%f xymul=%f", x*y, xymul) ;
$display("x/y=%f xydiv=%f", x/y, xydiv) ;
end
endmodule
```

執行結果：

```
D:\Dropbox\Public\web\oc\code>iverilog -o float float.v
```

D:\Dropbox\Public\web\oc\code>vvp float

x=3.141590 x1=0100000000001001001000011111001111000000011011100001
1001101110 x

2=3. 141590

000 y2=-23000.00000

x+y=-22996.858410 xyadd=-22996.858410

x-y=23003.141590 xysub=23003.141590

x*y=-72256.570000 xymul=-72256.570000

x/y=-0.000137 xydiv=-0.000137

```
// https://instruct1.cit.cornell.edu/courses/ece576/StudentWork/ss86
8/fp/Reg27FP/FpMul.v
// https://instruct1.cit.cornell.edu/courses/ece576/StudentWork/ss86
8/fp/Reg27FP/FpAdd.v
// https://instruct1.cit.cornell.edu/courses/ece576/FloatingPoint/in
dex.html#Schneider_fp

`define F_SIGN          63
`define F_EXP           62:52
`define F_FRAC          51:0

// a = (-1)^a.s (1+a.f) * 2 ^ {a.e-1023}
// b = (-1)^b.s (1+b.f) * 2 ^ {b.e-1023}
// a*b = (-1)^(a.s xor b.s) (1+a.f) (1+b.f) * 2^{(a.e+b.e-1023) - 1
023}
//         z.s = a.s xor b.s   z.f = tail(...)   z.e = a.e+b.e-1023
module fmul(input [63:0] a, input [63:0] b, output [63:0] z);
```

```

wire      a_s = a[`F_SIGN];
wire [10:0] a_e = a[`F_EXP];
wire [51:0] a_f = a[`F_FRAC];
wire      b_s = b[`F_SIGN];
wire [10:0] b_e = b[`F_EXP];
wire [51:0] b_f = b[`F_FRAC];

wire    z_s = a_s ^ b_s; // 正負號 z.s = a.s xor b.s
wire [105:0] f = {1'b1, a_f} * {1'b1, b_f}; // 小數部分: f = {1
, a.f} * {1, b.f}
wire [11:0] e = a_e + b_e - 12'd1023; // 指數部份: e = a.e + b
.e - 1023
wire [51:0] z_f = f[105] ? f[104:53] : f[103:52]; // 若最高位 f
[105] == 1, 則取 z.f[104:53], 否則取 z.f[103:52]
wire [10:0] z_e = f[105] ? e[10:0]+1 : e[10:0]; // 若最高位 f[1
05] == 1, 則取 z.e 要上升 1 (??), 否則不變
wire underflow = a_e[10] & b_e[10] & ~z_e[10]; // underflow
assign z = underflow ? 64'b0 : {z_s, z_e, z_f}; // 若 underflow

```

, 則傳回零, 否則傳回 $z=\{z.s, z.e, z.f\}$ 。

endmodule

```
module main;
real x, y, z;
reg [63:0] x1, y1;
wire [63:0] z1;

fmul f1(x1, y1, z1);

initial
begin
// x=7.00; y=-9.00;
// x=6.00; y=8.00;
// x=301.00; y=200.00;
x=1.75; y=1.75;
x1 = $realtobits(x);
y1 = $realtobits(y);
```

```

#100;
$display("a. s=%b a. e=%b a. f=%b", f1.a_s, f1.a_e, f1.a_f);
$display("a. s=%b b. e=%b b. f=%b", f1.b_s, f1.b_e, f1.b_f);
$display("e=%b \nf=%b \nunderflow=%b", f1.e, f1.f, f1.underflow);
$display("z. s=%b z. e=%b z. f=%b", f1.z_s, f1.z_e, f1.z_f);

z = $bitstoreal(z1);
$display("x=%f y=%f z=%f", x, y, z);
$display("x1=%b \ny1=%b \nz1=%b", x1, y1, z1);
end

endmodule

```

執行結果

D:\Dropbox\Public\web\oc\code>iverilog -o fpu64 fpu64.v

D:\Dropbox\Public\web\oc\code>vvp fpu64

參考文獻

- http://en.wikipedia.org/wiki/Single-precision_floating-point_format

- http://en.wikipedia.org/wiki/Double-precision_floating-point_format

繪圖加速功能 (Graphics)

在 3D 的遊戲或動畫的運算當中，經常要計算向量的加法、內積的浮點數運算，而且這些運算通常獨立性很高，可以採用平行的方式計算，所以就需要在繪圖處理器 GPU 當中內建很多平行的向量式浮點運算器來加速此一計算過程。

舉例而言，當我們想計算下列的 N 維向量加法時，如果採用 GPU 就會快上很多：

```
for (i=0; i<N; i++)
    C[i] = A[i]+B[i];
```

當這樣的程式被放到 GPU 去執行時，可能會翻譯出如下的組合語言。

LD	R1, N
LD	R2, A
LD	R3, B
LD	R4, C
LOOP:	VectorLoad V2, [R2]

```
VectorLoad  V3, [R3]
VectorAdd    V4, V2, V3
VectorStore  V4, [R4]
ADD         R2, R2, 4*L
ADD         R3, R3, 4*L
ADDd        R4, R4, 4*L
SUB         R5, R5, L
CMP          R5, R0
JGE          LOOP
```

如果上述 GPU 的一次可以計算的浮點長度為 L 個，那麼整個計算幾乎就可以快上 L 倍了。

GPU 通常有著與 CPU 不同的指令集，像是 GPU 大廠 NVIDIA 的 CUDA 就是一種 GPU 指令集，NVIDIA 甚至還為 CUDA 設計了一種擴充式的 C 語言 -- 稱為 CUDA C，這種 C 語言可以用標記指定哪些部份應該展開到 CPU 當中，哪些應該展開到 GPU 當中，以及如何有效的運用 GPU 進行向量計算等等。甚至、CUDA 還提供了專用的函式庫，以便讓 CUDA C 能夠方便的呼叫這些常用的繪圖函數。

另外、為了讓各家廠商的 CPU 與 GPU 可以更有效的合作，甚至有組織制定了一種稱為 OpenCL 的標準，可以讓各家廠商的 CPU 與 GPU 可以在一個標準架構下進行協同運算，以便有效的運用這些

異質處理器的效能。

平行處理 (Parallel)

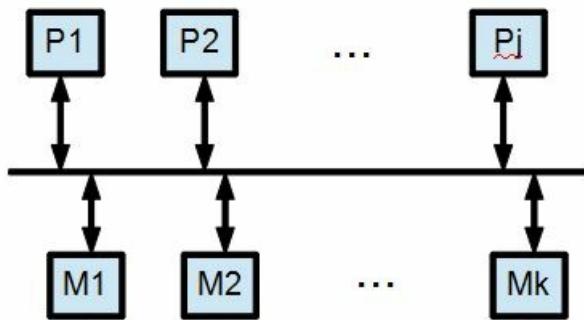
由於散熱等因素無法克服，目前的處理器頻率已經不容易再快速往上調整，但是晶片的密度與容量還在持續增加，於是處理器內的平行，也就是多核心的情況就愈來愈普遍了。

目前雙核心、四核心、八核心甚至 16 核心的處理器已經很常見了，在可見的未來，或許成百上千核心的處理器也會被開發出來，因此如何運用平行技術充份利用這麼多核心將會是一個重要的課題。

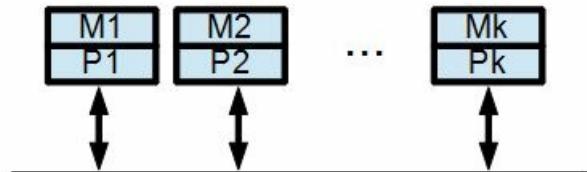
另外、網路雲端運算的需求不斷增強，也會進一步增加多處理器技術的重要性，像是 Google 提出的 MapReduce 就在「大數據」的潮流下愈來愈形重要，而 MPI 與 Hadoop 等分散式平行處理技術也就在這種情況下逐漸普及了。

對於平行計算的架構，有一種相當好的分類方式，是根據「指令與資料是單數還複數」所進行的分類，這種分類法將平行架構分為「單指令多資料」(SIMD)，「多指令單資料」(MISD)，「多指令多資料」(MIMD) 等架構，在加上傳統非平行化的「單指令單資料」(SISD) 等形成一個 2×2 的矩陣。

舉例而言，像是上述的 GPU 向量架構，是屬於 SIMD 的平行架構，而下圖中的 UMA 與 NUMA 架構，則是屬於 MIMD 的平行架構。



(a) UMA



(b) NUMA

當然，在上述架構下，每個處理器理又有自己的 cache，如何保持這些 cache 的一至性就會是個問題，而解決這些問題所依賴的方法仍然是與快取一至性中的所使用的「寫穿」(write-through) 和「寫回」(write-back) 等方法類似，只是規模要放大到整個網路上的快取而已。

結語

目前在本書當中，我們對於「高階處理器」的實作還很缺乏，像是快取、MMU 與 pipeline 等部份的實作都還沒有完成，因此還無法完全實現「開放電腦計畫硬體部份」的目標。

不過在探索過程當中，筆者對硬體領域開始有了更深一層的瞭解，希望未來能夠將這些部份實作好

之後，盡快加入到本書裏面。

到時、隨著本書進化成「第二版、第三版、....」，相信讀者與我對電腦硬體的理解也會變得愈來愈深入了。

附錄

本書內容與相關資源

主題	投影片	實作程式	教學影片
第 1 章. 開放電腦計畫			http://youtu.be/goi-VUwUnjI
第 2 章. 電腦硬體架構	co01_Overview.pdf		http://youtu.be/goi-VUwUnjI
— 計算機結構：簡介 1			http://youtu.be/PX2jDLoVLmg
— 計算機結構：簡介 2			http://youtu.be/_TUGe1GtJ6A
第 3 章. 硬體描述語言 -- Verilog			
- 區塊式設計			
- 流程式設計			
— Icarus			http://youtu.be/wIKqxZ-9cI0

— Verilog		xor3.v	http://youtu.be/m_pM57K1G80
— Quartus II			http://youtu.be/raflIGnoFik
— DE2-70			http://youtu.be/TOz4gyuUOgA
第 4 章. 組合邏輯			
— 多工器		mux.v	http://youtu.be/tMdq_WcMIMI
— 全加器			http://youtu.be/UD3o5XQNkfq
— 4 位元加法器		[adder4.v]	http://youtu.be/t_0Q5Ddi4xE
— 4 位元加減器		[addsub4.v]	
— 4 位元快速加法器			
— N 位元加法器			
— 乘法器		mult4.v	
— 移位乘法器		shift_mult.v	

— 除法器			
— 浮點乘法器		[fmult4.v]	
第 5 章. 算術邏輯單元			
— 算術邏輯單元 ALU		[alu.v]	http://youtu.be/tMdq_WcMIMI
第 6 章. 記憶單元			
— 正反器			
— 框鎖器		latch.v	http://youtu.be/2EsDL8aGHgI
— 計數器		counter.v	http://youtu.be/B1WFOPSL1A
— 狀態機			
— 多工器		mux.v	http://youtu.be/KcbUO_VRQ0U
— 暫存器			
— 暫存器群		regbank.v	http://youtu.be/6zLPnv-PUBA

- 8 位元記憶體		memory8.v	
- 32位元記憶體		memory32.v	http://youtu.be/dGmY6YhTbNw
第 7 章. 控制單元			
第 8 章. 微處理器			
- MCU0 指令集			
- MCU0 迷你版			
- MCU0 迷你版 (區塊設計)			
- MCU0 完整版			
第 9 章. 中斷與輸出入			
- MCU0 中斷處理			
- MCU0 的輸出入			
第 10 章. 記憶系統			

- 記憶體階層			
— 快取記憶體			
第 11 章. 高階處理器			
— 哈佛架構			
— 流水線			
- CPU0 指令集			
- CPU0 迷你版			
- CPU0 完整版			
第 12 章. 速度議題			
— 乘法與除法			
— 浮點運算單元 FPU			
— 繪圖加速功能			

— 平行處理

— 結語