6502 模擬器製作流程
[第一步之前的那一步!]

[第一步 - 收集資料]
1.0 你有什麼?
1.1 別人有什麼?

[第二步 - 整理所獲得的資料]

[第三步 TVGAME 硬體的運作方式]
3.1 硬體週邊分類
3.2 CPU 及 MEMORY MAP
3.2.0 CPU 的功能與地位
3.2.1 MULTI-PROCESSOR 的系統:
3.2.2 看起來不一樣,實際又是大同小異的 CPU
3.2.3 MEMORY MAP 的功能
3.2.4 ROM IMAGE
3.2.5 CPU 與其他週邊溝通方式
3.2.6 DSP 的功能
3.3 PPU(圖形處理單元運作)方式
3.3.0 PPU(圖形處理單元)的功能及地位
3.3.1 PPU 的組成
3.3.2 VBLANK & HBLANK
3.4 SPU(聲音處理單元)運作方式:
3.4.0 SPU(圖形處理單元)的功能及地位:
3.4.1 SPU 的組成:
3.4.2 與 PPU 的不同處:
3.4.3 聲音處理器專用 DSP 的功能:
3.4.4 聲音處理器的指令:
3.5 ROM & RAM AREA
3.5.0 SYSTEM ROM:
3.5.1 沒有 SYSTEM ROM 的基版:
3.5.2 GAME ROM
3.5.3 以光碟做媒介的 ROM -- CD ROM:
3.5.4 GAME ROM 的 MAPPER 問題:
3.5.5 如何開發遊戲:
3.5.6 ROM 中的程式如何控制硬體:
3.6 搖桿控制裝置及錢幣計數器等 NMI 的處理
3.6.0 搖桿通知 CPU 有按鍵按下的原理:
3.6.1 MEMORY MAP 中分配給搖桿控制裝置的方式:
3.6.2 錢幣計數器,START 鍵等 NMI 的處理:
3.6.3 非數位式搖桿的輸入:
3.6.4 錢幣計數器,START 鍵等 NMI 的處理:
3.7 RESET 處理

[第一步之前的那一步!]

為什麼你想做一個模擬器?練習程式語言?還是你想玩很難玩到的遊戲?還是你只
是單純的想做一個模擬器?動機會影響你的熱衷度,同時你也必須選定一個明確的目標,模擬一個 ARCADE 的遊戲,還是遊戲主機?一旦選定了明確的目標,你一定

要下定決心,否則便沒有完成這個模擬器的機會!在撰寫模擬器時,你會遇到很多
困難,像找不到資料;為了除錯,幾天都很難睡覺!就算是寫好了,放出來也還有一
堆莫名其妙的人抱怨一些有的沒的!你唯一的報酬可能就只有成就感而已,或者加上能玩到已經玩不到的遊戲.如果你還願意去寫一個模擬器,我真的十分佩服你!

[第一步 - 收集資料]
1.0 你有什麼?
在你想寫模擬器之前,你必須要真實的檢視你自己的資源,你會什麼程式語言?你對所模擬的硬體認識多少?你了解計算機內部運作的方式嗎?你看不看得懂硬體線路圖?這些會影響你寫不寫得出一個完整的模擬器!其實程式語言的影響也不是那麼嚴重,還是有人可以用 QUICK BASIC 寫出模擬器,只是效率實在不好而已,但是其他的因素卻是影響這個模擬器寫不寫的出來!

1.1 別人有什麼?
你需要去找你想要模擬的硬體的資料,你有哪些要找的呢?
A.使用何種 CPU?基版上面有多少特殊的晶片?時序速度多少?有多少相關資料?
B.有沒有硬體概圖(Schematics)?有沒有 DIP SWITCH 的資料?
C.有沒有 MEMORY MAP?有沒有軟體的其他相關資料?
D.有沒有別人已經寫好的模擬器原始檔?
你可到哪裡找?
A.新聞討論區!
B.模擬器網頁的留言版!
C.模擬器新聞網頁-可以知道是否有模擬器提供其原始檔!
D.模擬器程式發展資源網頁-可以收集到許多前人寫好的文件!
E.別人的模擬器網頁-也許有額外的連結可以讓你找到多一點資訊!
F.如果是遊戲主機,到官方網頁也許有意想不到的資料!
還缺了什麼?
A.CPU 手冊,反組譯工具!
B.也許你會需要電子元件手冊,供你分別或查詢元件的用途及接腳定義等!
C.適當的程式語言,適當的函式庫(節省開發時間)!

[第二步 - 整理所獲得的資料]
在蒐集到所有能收集的資料之後,依照我們要的部份加以分類:
硬體:
A.CPU 種類&速度!次系統處理器種類&速度!
B.Schematics,DIP SWITCH,硬體線路,電路圖等!
C.MEMORY MAP,搖桿接腳定義!

D.基版上的 SYSTEM ROM!
E.特殊晶片的設計資料!例如圖形處理晶片,聲音處理晶片等!
F.畫面處理的方式,圖形及聲音編碼方式等!
軟體:
A.CPU 手冊,反組譯工具,CPU 模擬程式等!
B.遊戲 ROM!SYSTEM ROM!
C.特殊晶片的模擬程式!
D.別人的模擬器程式,可以"偷"或是看看別人的寫法!
E.函式庫手冊等!

[第三步 TVGAME 硬體的運作方式]
3.1 硬體週邊分類
一般說來,你可以把整個 TVGAME 硬體分為:
A.CPU:
一般是廉價的泛用處理器,例如 68000,6502,Z80 等!用來指揮整個硬體的運作!複雜的系統甚至以多處理機協同處理!例如 TAITO 的雙 68000 系統,SS 的雙 SH-2 系統!
B.PPU(圖形處理單元,VDP,GPU):
專職於圖形的處理!包含 PICTURE PROCCESSOR,專用 RAM,專用 ROM,數位類比轉換器等!常見名稱的有 PPU(PICTURE PROCCESS UNIT),GPU(GRAGHIX PROCESS UNIT),VDP(VIDEO DATA PROCCESSOR),以下都使用 PPU 代替!
C.特殊 DSP 晶片
一種能快速計算數值的晶片,用來協助處理資料!
D.SPU(聲音處理單元,APU):
專職於聲音的處理!包含 SOUND PROCCESSOR,專用 RAM,專用 ROM,專用 DSP,聲音晶片,數位類比轉換器等!名稱也很多,SPU(SOUND PROCCESS UNIT),APU(AUDIO PROCEESS UNIT),以下用 SPU 代替!
E.ROM AREA:儲放 ROM 的區域!
1.SYSTEM ROM:
類似於 PC 中 BIOS+OS 的地位,提供開機檢查,常用函式等!
2.GAME ROM:
真正遊戲的程式區域!家用主機把圖形,聲音,及其他資料儲放在一起!
但是 ARCADE 遊戲通常會分開儲放!如果細分的話,可以分為:
a.GRAPHIC ROM:儲放遊戲圖形資料!
b.SOUND ROM:儲放聲音資料!
3.儲放其他資料的 ROM!
F.MAIN MEMORY,WORK RAM
就是主記憶體!
G.光碟機及控制週邊!
只有以光碟作為儲存媒介的主機才有!
H.搖桿等控制裝置:
搖桿輸入,RESET,錢幣計數器等!

I.BACKUP DEVICE:
電池記憶,記憶卡之類的裝置!
J.基版控制週邊,DMA 裝置,外部擴充界面及裝置等!
K.硬體其他週邊,螢幕,電源供應器等!

當然也不是每一個 TVGAME 系統都這麼完整,越早期的家用主機或 ARCADE 基版就簡單一點!現在的家用主機或是 ARCADE 系統基版就十分複雜,這都是你在蒐集資料時一定要找到的,有些是模擬器一定會用到的,甚至影響模擬器的完成度,有些則是模擬器可省略的,例如螢幕,電源供應器等!以下對於會使用在模擬器上的裝置作一下說明!

3.2 CPU 及 MEMORY MAP:
3.2.0 CPU 的功能與地位:
對整個 TVGAME 或是 ARCADE 來說,CPU 應該算是"導演"的工作,而 ROM IMAGE 則是導演手上的劇本,用來指揮週邊的動作.指揮別人,當然可以不用太高級,只要足夠就行了!畫面表現的好壞,聲音處理的好不好,與 CPU 的關係較少!但是如果週邊的處理比 CPU 快太多,讓週邊老是等待也不好,同時,有一部份工作也必須由 CPU 處理,所以 CPU 也不可能太慢!把價格與運算能力做考量依據,才選出適用的 CPU!常見的有 68000,6502,Z80 等!

3.2.1 MULTI-PROCESSOR 的系統:
MULTI-PROCESSOR 的系統就是指這個系統中含有兩個以上的 CPU,例如 SS 就是由兩個 SH-2 所組成的,依照 CPU 分工的不同,可以分為:
A.MASTER/SLAVE(非對稱結構):
只有一個 CPU(MASTER)負責 IO 的工作,計算的工作則每個 CPU 都能執行!同時只有 MASTER CPU 能執型 O.S.,而 SLAVE CPU 則是以中斷要求 MASTER 的服務!
B.SEPERATE EXECUTIVES:
每個 CPU 有自己的 OS,INTERRUPTS,CPU 種類及 OS 種類均允許不同,不能共同處理同一工作!
C.SYMMETRIC ORGANIZATION:
由一個 OS 整合管理所有的處理機,稱為 PROCESSOR POOL,每個 CPU 皆能存取任一 I/O DEVICE 及儲存單元!
如果你想做的是 MULTI-PROCESSOR 的系統,必須要先蒐集 CPU 間的關係,溝通方式,例如 SS 就是一個 MASTER/SLAVE 的系統!

3.2.2 MEMORY MAP 的功能:
正如家家都必須要有不同的地址,才能把郵件送到正確的地方,CPU 也需要相同的機制,用來存取到正確的資料,處理機所使用的方式是加上位址線,連接至位址匯流排,再連接至 RAM,ROM,其他裝置等!位址匯流排寬度決定了 CPU 所能定址的最大空間,這空間,你可稱為"定址空間"!實際上不同的定址模式會影響定址空間,不過這裡我們就假裝不知道吧!

為了存取方便,需要將定址空間做一番規劃,規劃之後的定址空間,稱之為
MEMORY
MAP!把定址空間分成一段一段的,把每一段都賦予不同的用途!例如下面是 SFC
的
MEMORY MAP!

SNES Documentation v2.30: Written by Yoshi

Here's a really basic memory map of the SNES's memory. Thanks to Geggin of Censor for supplying this. Reminder: this is a memory map in MODE 20.		
Bank	Address	Description
$00-$3F	$0000-$1FFF	Scratchpad RAM. Set D-reg here if you'd like (I do)
	$2000-$5FFF	Reserved (PPU, DMA)
	$6000-$7FFF	Expand (???)
	$8000-$FFFF	ROM (for code, graphics, etc.)
$70	$0000-$7FFF	SRAM (BRAM) - Battery RAM
$7E	$0000-$1FFF	Scratchpad RAM (same as bank $00 to $3F)
	$2000-$FFFF	RAM (for music, or whatever)
$7F	$0000-$FFFF	RAM (for whatever)

不只是 TVGAME 及 ARCADE,IBM PC 也有同樣的機制,例如 DOS 的
0KB-640KB-1024KB
規劃!

3.2.3 看起來不一樣,實際又是大同小異的 CPU:
這裡先暫且擱下 MEMORY MAP 不談!回到 CPU,如果你有機會看到 SNK NEOGEO/MVS
系統的基版的話,從文件上知道,它應該是 MOTOROLA 68000 的 CPU,不過你就是
找
不到長方形的 MOTOROLA 68000 CPU,因為 SNK 特別訂製這顆 CPU,把 68000 及兩個
PPU 作在一起,成為一顆 VLSI!實際上,有很多機器都是以這樣的方式製作,訂製
成 VLSI!在寫模擬器時,只需要知道它的解碼方式等同於 MOTOROLA 68000 CPU
就可以了!

還有一種形式是 DECODE 及執行結果等同於其他 CPU,最明顯的例子就是 FC 用的
"
65C02",事實上它與真正的 6502 不太一樣,只是編碼及執行結果等同於 6502!在寫模擬器時,仍然需要知道它的解碼方式等同於 6502 就行了!

3.2.4 ROM IMAGE:
一般 TVGAME 及 ARCADE 遊戲,都是把程式或資料燒錄在 ROM 中-不論是 ROM IC 顆粒
,還是 CD-ROM!很明顯的,除了 CD-ROM 之外,沒有辦法直接讀取其中的指令或資料(即便是 CD-ROM 也可能自己搞奇怪格式,所以也可能無法讀取),所以需要使用 ROM DUMPER,把 ROM 中的指令或資料 DUMP 成二進位制檔案,稱為"ROM IMAGE"!

因為所使用的機器語言不同(或者粗略的稱為 CPU 不同),ROM IMAGE 的內容目前對 PC 並不具任何意義,所能做的也只侷限在 COPY,MOVE,EDIT HEX,或是 ZIP 起來,類似於圖形檔的處理,即便它是一個受歡迎的遊戲!直到有適用的模擬器出現,ROM IMAGE 才有價值--拿來玩遊戲!把 ROM 做成 ROM IMAGE,在大部分情況下都算是比較簡單,所以 ROM IMAGE 多,而對應的模擬器少,作 ROM IMAGE 只要有機器就行,但是沒人寫模擬器就是白搭!

回到 ROM IC 上,ROM 上燒錄的當然是程式或資料,問題是給誰的資料!如果不考慮有些基版有 SYSTEM ROM 的情況(留待 SYSTEM ROM 區再討論)!ROM 一般都是給 CPU 的資料,所以指令都是 CPU 的機器語言,存取到資料區時,即便可能是給另外 PPU 或 SPU 所使用的指令,對於 CPU 而言,這些仍然視作"資料"而不是"指令"!等到這些"資料"傳遞給對應 PPU 或 SPU 時,才會變成"指令"!

3.2.5 CPU 與其他週邊溝通方式:
這裡必須要先解釋"MEMORY MAPPED I/O"!所謂 MEMORY MAPPED I/O"是指 I/O 界面位址和記憶體位址使用相同的位址空間,此時界面暫存器是記憶體系統的一部份!界面單元和記憶體使用相同指令存取,資料位址指到記憶體,即是和記憶體溝通;如果指到界面暫存器位址範圍即是要進行 I/O 動作!

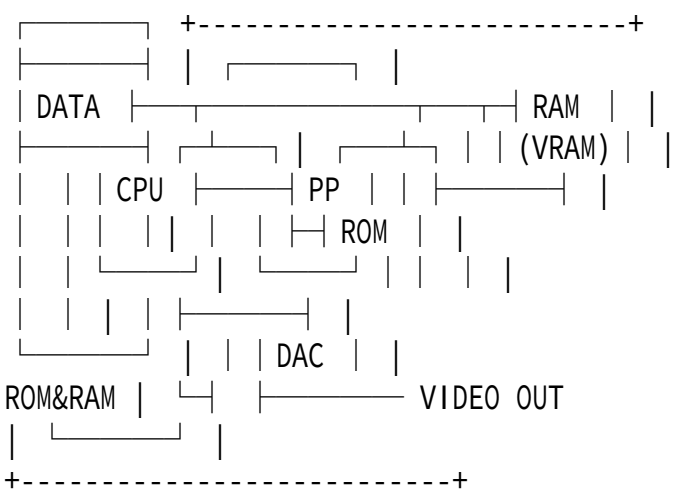
大部份的 TVGAME 及 ARCADE 都是設計成 MEMORY MAPPED I/O,以這種方式對其週邊作存取動作,所以知道你所要模擬的系統的 MEMORY MAP 是很重要的事!尤其是對 PPU,SPU 的存取(設定執行某一特定功能,或傳遞圖形或聲音資料),知道細部的 MEMORY MAP 結構,及每個 BIT 的用途,才能正確的模擬出 PPU,SPU 的功能!

在 MEMORY MAP 中,屬於 PPU,SPU 功能的段落的位址上的字組中,每一個 BYTE 上的每一個 BIT 都有特定的用途,這些就是 CPU 與週邊溝通的窗口!CPU 把設定好的資
料,寫入到正確的位址,就等於是叫 PPU,SPU 執行某一個特定的功能!詳細的執行這些功能的時機,我沒有找到資料!比較可能執行時機可能為兩種,一是 CPU 直接中斷 PPU,SPU,然後 PPU,SPU 檢查這些窗口,執行對應的功能!一是 PPU,SPU 在定期中斷 CPU 時,同時檢查這些窗口,再執行對應的功能!不過寫成模擬器時,都是在 CPU 暫停的期間,PPU,SPU 再去檢查這些窗口,執行對應的功能!

3.2.6 DSP 的功能
在大部分的情況下,CPU 不需要處理大量的數字資料,所以在選擇 CPU 時,一般並不刻意挑選數值處理速度極快而高價的 CPU,而是在 CPU 之外,另外搭配高速的 DSP,協助 CPU 處理數值資料,類似於 PC 上的 FPU,當然 PC 現在都是內建的,在 386 及 486SX 的時代,FPU 是選購項目!在特殊的情況下,DSP 是屬於可加入式的,例如 SFC 便是可在卡匣上加裝 DSP 協助 CPU 處理大量多邊型的運算,而不是固定在基版上,因為它並不是必要的!不過因為需要處理的資料越來越多,新一代的 TV GAME 及 ARCADE 系統都內建了 DSP,以免數值資料處理過慢,拖累整個執行速度!在你找資料時,也必須找到 CPU 與 DSP 溝通的方式,DSP 如何處理資料,以模擬出 DSP 的功能!

3.3 PPU(圖形處理單元運作)方式:
3.3.0 PPU(圖形處理單元)的功能及地位:
PPU 是整個 TV GAME/ARCADE 系統的重心,遊戲畫面能做到多炫,特效多少,完全都是由 PPU 去表現!而 CPU 只是負責指揮的工作,所以使用同一個 CPU 的系統,如果所使用的 PPU 能力不同的話,所呈現的效果也將有很大的不同!這是模擬器所要模擬的重心,整個模擬器完成度幾乎全取決於 PPU 及 SPU 是否能模擬的完美!

3.3.1 PPU 的組成:
包含 PICTURE PROCCESSOR,專用 RAM,專用 ROM,數位類比轉換器等!簡圖如下:

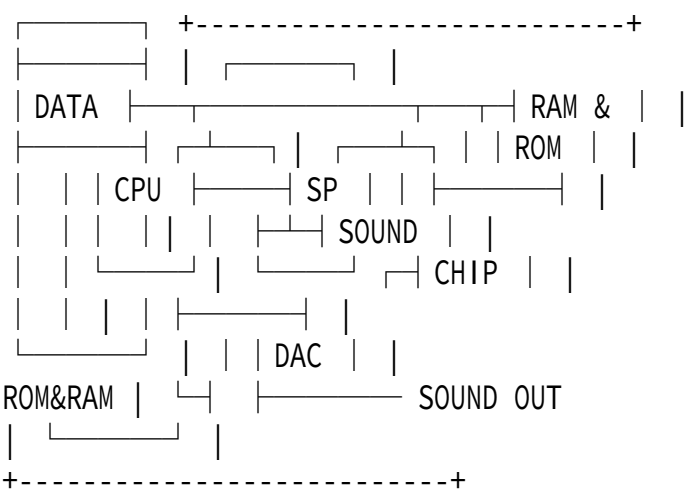


CPU 透過 MEMORY MAP 中的各個窗口,與 PP(圖形處理器,PICTURE PROCCESSOR)溝通並設定欲執行的功能!PP 便到從 ROM&RAM 中搬移所要處理的資料到自己的 RAM 中!接下來是功能解碼,再從 ROM 中找出應該執行的指令,依序執行之後,寫回 RAM
,待螢幕更新週期時,由 DAC(數位-類比轉換器)轉換成 MONITOR 能接受的訊號,顯示到螢幕!

3.3.2 VBLANK & HBLANK:
由於螢幕(MONITOR)的更新是由電子束打在對應的點,讓點上的螢光質發光,藉以顯示出要顯示的圖形!電子束打的方式是一個接著一個,由左至右,由上至下,因為視覺暫留的現象,人眼看到的就是整個螢幕的圖形!電子束從每一行的最右邊回到下一行的最左邊,這一段時間稱為 H-BLANK,水平空白期!同樣的,電子束會從右下角再回到左上角,準備做下一次整個螢幕的更新,這一段時間稱為 V-BLANK,垂直空白期!HBLANK 與 VBLANK 的時間長短與移動的距離有關,行移動的 HBLANK 時間較短,而 VBLANK 的時間較長,所以 PPU 使用 VBALNK 時間做一些工作
是很重要的事!這也是寫模擬器及蒐集資料時非常重要的一部份!

3.4 SPU(聲音處理單元)運作方式:
3.4.0 SPU(圖形處理單元)的功能及地位:
與 PPU 一樣,SPU 用來專職處理聲音.在比較古早的系統中,一般並不另外搭配專用的處理器,而是由 CPU 直接控制.不過因為聲音資料日益增多,於是另外搭配專用的處理器,以減輕 CPU 的負擔!隨著 CPU 速度的加快,所搭配的 SOUND CPU 也日益高級,從 Z80,變成 MOTOROLA 68000,逐漸變快!

3.4.1 SPU 的組成:
包含 SUUND PROCCESSOR,專用 RAM,專用 ROM,專用 DSP,數位類比轉換器等!簡圖如下:



大致上與 PPU 的組成是一樣的!
CPU 透過 MEMORY MAP 中的各個窗口,與 SP(聲音處理器,SOUND PROCCESSOR)溝通並設定欲執行的功能!SP 便到從 ROM&RAM 中搬移所要處理的資料到自己的 RAM 中
!接下來是功能解碼,再從 ROM 中找出應該執行的指令,依序執行並指揮聲音晶片(SOUND CHIP)產生實際的聲音資料,傳遞給 DAC 轉換成聲音訊號!

3.4.2 與 PPU 的不同處:
最大的不同處在於 MONITOR 還有 VBLANK&HBLANK 可以做很多事!SPU 則必須不間斷的產生聲音輸出,同時聲音晶片的模擬也是一大問題,這是模擬器的一大障礙,所以模擬器一般都是發展到一定階段了,機器的等級也到一定速度了,才逐漸加入聲音的支援!

3.4.3 聲音處理器專用 DSP 的功能:
因為所選用的聲音處理器不一定適於處理聲音資料,尤其是 FM 與 PCM 有極大的不同,PCM 需要更強大的數值運算能力,因此有些系統會加入聲音處理器專用 DSP,協助聲音處理器處理 PCM 的運算!這也是蒐集資料時的重要部份!

3.4.4 聲音處理器的指令:
一般來說,CPU 傳給聲音處理器的資料,將會是這個聲音處理器的指令.所以必須加入 CPU CORE 處理指令解碼,例如是 Z80 的,就必須加入一個 Z80 的 CPU CORE,用來處理聲音資料,同樣有頻率及定址空間的設計,整個 SPU 基本上就是一個隱含的完整電腦系統!

3.5 ROM & RAM AREA
3.5.0 SYSTEM ROM:
在某些 TVGAME 或 ARCADE 基版上會加入所謂 SYSTEM ROM,這個 SYSTEM ROM 有幾種
功能:
A.執行開機檢查程式及 O.S.!
B.儲放重要或常用函式!
C.儲放 PPU,SPU 常用函式!
D.儲放中斷向量及中斷處理常式
這樣的作法,明顯的可以減輕 GAME ROM 的成本,不用每一個 GAME ROM 都需要挪出空間放置這些程式,而使得成本降低!簡單的說,SYSTEM ROM 就等於是 PC 上的作業系統!通常設置的目的是方便遊戲開發人員能方便的使用一些特殊或常用函式,基版設計廠商則必須盡量加入需要的功能函式!所以,如果你想寫的主機上有 SYSTEM ROM 的話,你必須想辦法 DUMP 下來,因為不管是 CPU,或者可能有 PPU,SPU 都會存取到這裡,執行這裡的程式!

3.5.1 沒有 SYSTEM ROM 的基版:
大部分為單一遊戲或非系統基版的情況!必須在 GAME ROM 中撥出空間放置等同於 SYSTEM ROM 功能的程式,因為是單一遊戲,可以依個別情況撰寫,空間的損失並不大!通常會在基版開機時,就會從 ROM 中載入並執行開機檢查程式!

3.5.2 GAME ROM
除去了 SYSTEM ROM 的部份,剩下來的就是 GAME ROM 的部份,按照功能可以分成:
A.PROGRAM AREA 程式區:

提供 CPU 指令的區域!
B.GRAGHIX DATA AREA 圖形資料區 :
提供 PPU 圖形資料的部份!
C.SOUND DATA AREA 聲音資料區:
提供 SPU 聲音資料的部份!
D.其他資料區

不同的基版設計,處理這些 ROM 的方式也不同!以家用主機大部分的設計來說,所有的 ROM DATA 是按照其位址分開的,從外觀上不易看出,DUMP 時也都是直接 DUMP 成一整個 ROM IMAGE FILE(目前只有 FC 有分開 DUMP 的 ROM IMAGE FILE 出現
)!其原因應該是卡匣的體積,面積有限,不適於分開放置!

ps: 這也可能是 DUMP 工具的問題,家用主機的 ROM DUMPER(或稱為 ROM COPIER)比
較常見的都並不是一個 ROM IC 一個 ROM IC 的 DUMP!

相對於家用主機的情形,ARCADE 則是能夠從 ROM IC 放置的區域分別出這顆 ROM IC 的內容及用途!一般來說,在 DUMP 這些基版的 ROM IC 時,都會按照其用途或 SCHEMATICS 上的編號命名,方便識別及載入至正確位址!

3.5.3 以光碟做媒介的 ROM -- CD ROM:
CD ROM 的處理方式與一般的卡匣或是基版上的 ROM IC 載入的方式並不相同!因為必須從 CD ROM 中分段載入所需要的程式段,圖形資料段,聲音段!所以分區的情況不明顯,而是以當時遊戲進行的狀況處理!

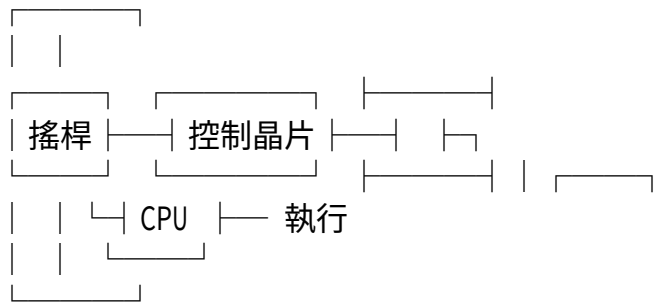
3.5.4 GAME ROM 的 MAPPER 問題:
如果你是準備寫家用主機的模擬器的話,所要面臨的便是可能出現一個 ROM MAPPER 問題,ROM MAPPER 問題是指 MEMORY MAP 中提供給 ROM 的區域比實際 ROM 的容量還小,於是必需存在一個切換 ROM BANK 的暫存器(以 MEMORY MAPPING I/O 方式讀寫)或是其他的機制,利用這些機制與 MEMORY MAP 中的 ROM 區域而使得能讀取到更大的 GAME ROM!這些機制就被稱為 MAPPER,模擬出 MAPPER 的機制就是 MAPPER 問題!目前這個問題仍然困擾著 FC 模擬器的作者們,因為已知的 MAPPER
格式就超過 40 種,不管是不支援某一種 MAPPER 或是 MAPPER 程式寫的不好,這都不能使你的模擬器完美!是你在蒐集資料時必須注意到的!

3.5.5 如何開發遊戲:
一般來說,主機公司會提供開發工具,提供遊戲廠商開發遊戲!遊戲廠商利用開發工具,撰寫高階的原始程式,繪製圖形,編寫樂譜及音效等,經過 DEBUG 後,把全部的檔案經過排列之後做成轉燒成 ROM IC,或是直接壓製成 CD!

3.5.6 ROM 中的程式如何控制硬體:
應該有三種方式:

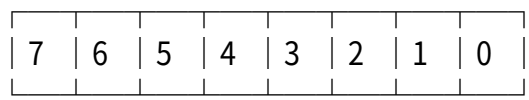
A.直接控制:
在開發時撰寫低階的硬體直接控制程式,理論上效能較快且可能創造出不同的效果!遊戲廠商需對硬體有相當認識,同時開發較不易!
B.呼叫 SYSTEM ROM 中的函式:
算是一種間接控制的方式.寫高階程式但是以呼叫 SYSTEM ROM 中的函式來控制硬體!SYSTEM ROM 中的函式因為容量因素,肯定不會大到哪去也不會太多,遊戲廠商還是大部分需要自己寫!效能亦相當不錯!
C.利用開發工具中內附的函式:
利用開發工具是最方便的方式,開發工具內附的函式到最後仍然會展成前面兩種方式控制硬體!不過開發工具的良窳會影響到遊戲的開發,這點也是很重要!
實際上通常都是這三種方式交叉使用,端看遊戲廠商的能力!

3.6 搖桿控制裝置及錢幣計數器等 NMI 的處理
3.6.0 搖桿通知 CPU 有按鍵按下的原理:
當有按鍵按下時,連接在基版的控制晶片會發出對 CPU 發出一個中斷訊號,並且把按鍵資訊寫入到分配在 MEMORY MAP 中的位址!CPU 接收到這個中斷訊號之後,中斷現在的工作,先處理按鍵輸入!處理完之後,在回到中斷前的程式,並根據接收到的按鍵訊息做反應!看起來會像這樣:

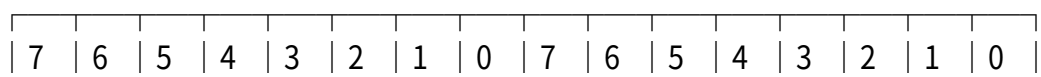


記憶體中存放按鍵資訊的位址(MEMORY MAPPED IO)

3.6.1 MEMORY MAP 中分配給搖桿控制裝置的方式:
因為 TVGAME,ARCADE 一般都是使用數位式的搖桿,也就是只有[按下]及[沒按下]
兩種反應,分別以 1BIT 代替一個按鍵,看起來會像這樣:
上 下 左 右 B1 B2 B3 B4



按鍵多一點的用 2 byte:
上 下 左 右 B1 B2 B3 B4 B5 B6 B7 B8 B9 B10 B11 B12



更多的按鍵的,像麻將,當然就繼續加下去!幾 P 就加幾組了!

3.6.3 非數位式搖桿的輸入:
就流程上基本上是與數位式搖桿相同!問題在於如何輸入資料於給定的位址中,隨著不同的搖桿會有不同的設計,這是蒐集資料時一定找到的!但是找到之後,更嚴重的問題是如何對應到 PC 的指向裝置!要是不幸找不到對應的裝置,你基本上就直接放棄好了!

3.6.4 錢幣計數器,START 鍵等 NMI 的處理:
NMI 是指"NON-MASKABLE INTERRUPT(NON-MASKED INTERRUPT)",也就是不可遮罩的中斷!這與一般中斷有何不同呢?相對於 NMI,其他的中斷是屬於可以被暫時忽略(稱為 MASK 掉),CPU 可稍後再處理!而 NMI 是 CPU 無論如何都必須放下手邊
的工作,必須優先處理這個中斷!在 TVGAME 系統中,PPU 的 VBLANK 就是屬於一個 NMI,因為它必須螢幕更新等動作,此時 PPU 必須搶到週邊的使用權,它會強迫 CPU 暫停手邊的工作!同樣的,在 ARCADE 系統中,像投錢,1P select,1P start,2P select,2P start,這種動作當然是一定要優先處理!不過.有些產生 NMI 的裝置,如 PPU,還是可以利用一些暫存器的設定,使它暫時不要產生 NMI,這些暫存器的位址,設定的方式,也是在蒐集資料時要找到的!

3.7 RESET 處理:
3.7.0 加入 RESET 的原因:
不管是 TVGAME 還是 ARCADE,遊戲通常都是不會停止的!問題是,還是有某些情況必須要重新開機一次,例如:進到硬體測試畫面,設定 DIP SWITCH,使用某些密技,打不贏人家耍賤,就是想重新開機等等!更重要的是,不斷開關電源會可能導致硬體損壞!所以必須要有一種機制可以暫時切斷電源,使系統重新開機,這種機制便是 RESET!
3.7.1 RESET 的流程:
RESET 的第一步當然是暫時切斷電源!重新初始化 CPU,週邊等!載入 SYSTEM ROM 進行開機測試,檢查是否有需要處理特殊程序,如:進到硬體測試畫面,設定 DIP SWITCH!如果沒有,再載入 GAME ROM,開始遊戲!

3.8 DIP SWITCH:
3.8.0 加入 DIP SWITCH 的原因:
因為 ARCADE 系統是一個賺錢的工具,一成不變的遊戲在被人摸熟時,就沒有人玩了,同時還有不少遊戲的參數要設定,所以需要一種方便的儲存遊戲參數的工具,於是在基版上加上 DIP SWITCH,只要開關撥一撥,就能設定參數!不需要外加電源!但是新一代的基版有些就不是使用 DIP SWICH,而是利用 EEPROM 儲存,這樣就不須要用手撥 DIP SWITCH 了!

3.8.1 DIP SWITCH 的作用方式:
說穿了,也只不過是一些邏輯電路!但是利用程式檢查電路短路與否,賦予實際的意義,例如:投多少 COIN 才算一次 CREDIT;設定出紅血看起來暴力一點;或是無敵之類的!不過少了設定 DIP SWITCH 的功能,並不會有什麼問題,只是沒那麼完美罷了!但是 DIP SWITCH 是用"手"設定的,所以如果要做 DIP SWITCH 就要做設定函式代替用手撥開關!當然你還是需要找到該遊戲 DIP SWITCH 圖,才能知道各開關的定義!

3.9 BACKUP DEVICE
3.9.0 需要 BACKUP DEVICE 的原因:
不是每一種遊戲都能一次玩完的,例如 RPG,SLG 等,於是需要一種能保存資料的裝置!古早之前,是利用特殊編碼的 CODE 來保存資料!不過這不是辦法,越複雜的遊戲或是需要保存的資料越多,CODE 會變得越來越長!於是出現劃時代的裝置---利用電池提供 SRAM 的電源,把資料儲存在 SRAM 中!一般稱為"電池記憶",但是用電池還是有問題--電池是有壽命的!新一代的便是使用 FLASH ROM,不須外加電源,可重複寫入!

3.9.1 存取 BACKUP DEVICE 的方式:
和其他裝置一樣,在 MEMORY MAP 有保留給 BACKUP DEVICE 的空間,只要把資料搬過去就行了!在此之前,當然是需要先知道 MEMORY MAP 的空間在哪裡!

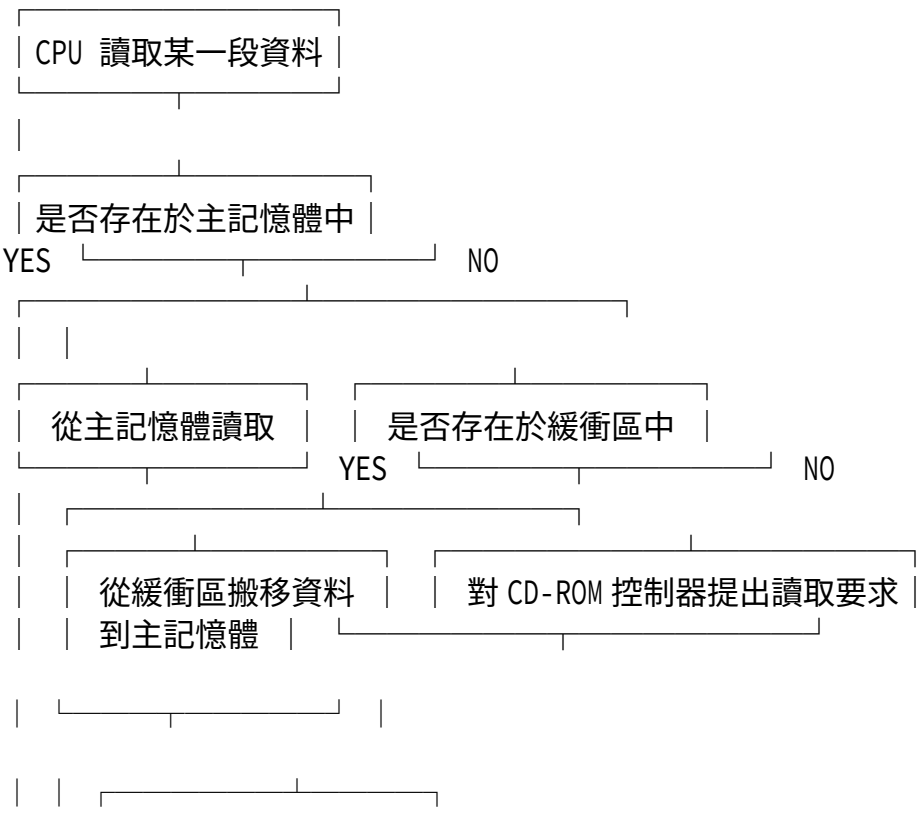
3.A DMA CONTROLLER
3.A.0 什麼是 DMA(DIRECT MEMORY ACCESS) CONTROLLER?
詳細的原理及機制請找計算機組織之類的書!總之就是不想讓資料傳輸通道閒著及減輕 CPU 的負擔,使 CPU 不需要自己處理資料的搬移!於是設計了一個專門處理資料搬移的處理器,便稱為 DMA CONTROLLER,處理的範圍是 RAM 與週邊 I/O!因為它也是一個處理器,所以 CPU 與 DMA CONTROLLER 的溝通是有一定程序的!如果想寫的系統也具有 DMA 控制器的話,就必須蒐集有關 DMA 控制器的資料!

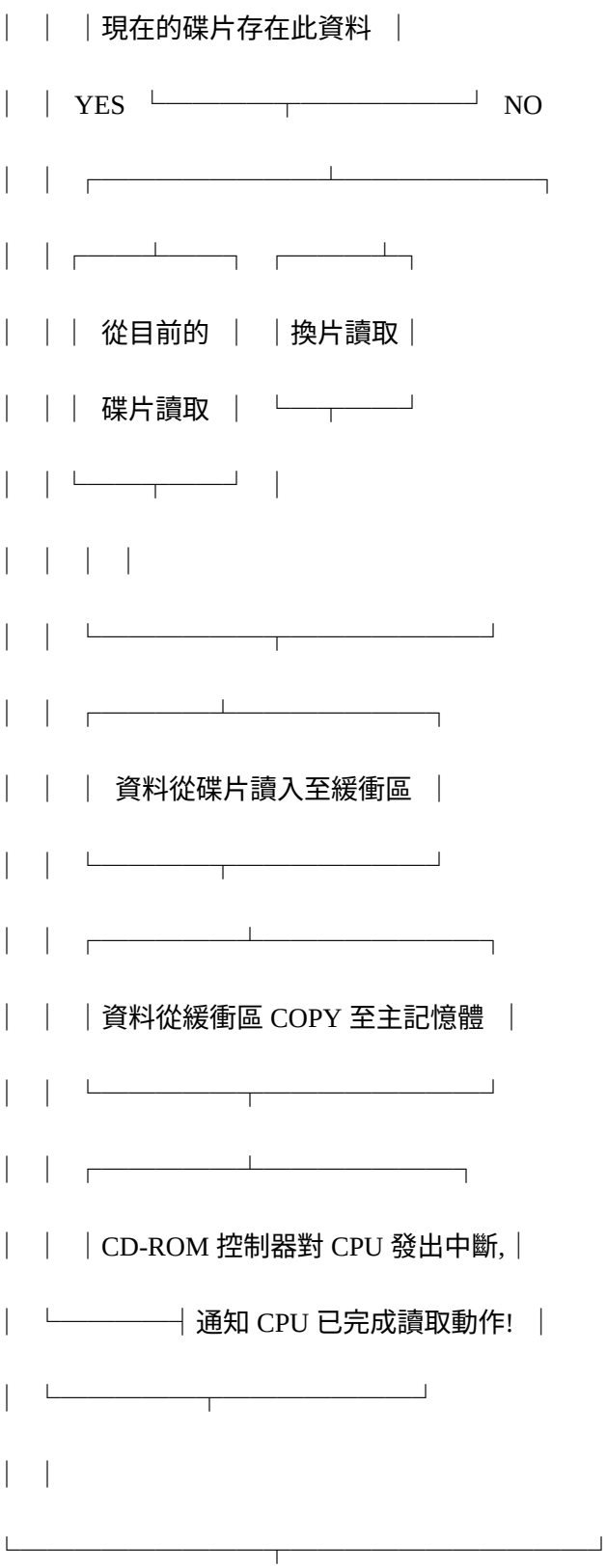
3.A.1 DMA 的運作方式(從書上抄來的):
COMPUTER ORGANIZATION & DESIGN --- DAVID A. PETERSON/JOHN L. HENNESSY
DMA 傳送中有三步驟:
A.由裝置本身提供 DMA,而操作命令是根據記憶體位址,位元數目並在裝置上執行資料的傳送.
B.DMA 在裝置上開始操作命令並重才匯流排.當資料可用時(從裝置或記憶體),它傳送此資料.而 DMA 裝置對讀取或寫入提供記憶體位址.若在匯流排上需求資料超過一次以上,DMA 單元將產生下一個記憶體位址並啟動下一次傳送.DMA 使用這種機制,在不用干擾處理機情況下,它能傳送數千位元組資料.在很多 DMA 控制器包含一些緩衝區允許它們在傳送資料或等待成為

匯流排主控器時來處理延滯時間的可塑性.
C.一旦 DMA 傳送完成時,控制器將中斷處理器,然後處理器查詢 DMA 裝置或記憶體,是否整個操作命令成功地完成.

3.B 光碟機裝置
3.B.0 使用 CD-ROM DRIVER 的原因:
隨著遊戲的聲光效果越來越好,GAME ROM 的容量也隨之不斷上升!原來所使用的 ROM IC 在應用上已經有所困難!這個困難包括兩方面,卡匣的尺寸會隨著 ROM IC 的增加而變大,另一方面,容量越大的遊戲,不是 ROM IC 用的更多,就是必須改用大容量的 ROM IC.而不管哪一種,顯而易見地都會帶來成本上升的影響,造成卡匣價錢上升!就目前而言,解決的方式有兩種:一是對資料進行壓縮,仍然使用 ROM IC 製成卡匣的型式,優點是保有卡匣不須等待,速度較快的優勢,缺點是就算是壓縮過後,還是不一定能解決容量的問題,語音及動畫都是屬於不易再壓縮的資料,只好忍痛刪除,例如 N64!另一種解決的方式是改用 CD-ROM 作儲存媒體,優點是再沒有容量上的限制,只要增加 CD-ROM 片數即可,同時成本低廉,價錢便宜!缺點是 CD-ROM 的存取時間遠大於卡匣!例如:一堆次世代機!

3.B.1 CD-ROM 的存取方式:
CD-ROM 的容量有 650Mbyte,大部份使用 CD-ROM DRIVER 的系統,其主記憶體通常並不足以容?#123;所有資料存放於主記憶體中,於是必須以分段讀取的方式存取!




現在的碟片存在此資料
YES
NO
從目前的
碟片讀取
換片讀取
資料從碟片讀入至緩衝區
資料從緩衝區 COPY 至主記憶體
CD-ROM 控制器對 CPU 發出中斷,
通知 CPU 已完成讀取動作!
CPU 完成資料讀取動作

這一個流程圖實際上還忽略了 CD-ROM DRIVER 硬體的動作,這才是真正最耗費時間的地方!

另外這其實不太像 TV GAME 主機會幹的事,這樣的檢查算起來還是太多,應該是跳過第一個判斷,因為需常駐的資料區及指令區不動,剩下的 MAIN MEMORY 全部使用畫面處理及聲音上,舊資料全部放棄,從 CD ROM 中讀取新資料,自然就不用檢查資料在不在!優點是分配處理畫面及聲音的 MEMORY 夠大,可以作出超炫 CGA 及全程語音,缺點是 CD ROM DRIVER 的 SEEK TIME 過慢及 BUFFER 過小時,等待時間會很長,又需要利用畫面處理的技巧解決及妥善安排檔案位置解決!

無論如何你需要知道系統對 CD-ROM 存取的詳細流程,所發出的中斷要求,所分配的 MEMORY MAP 位置,容量等!

3.C 其他週邊

對於 ARCADE 系統中,當然還有其他的週邊,我建議可以到亞站及巴站的 ARCADE 版中,找尋相關的資料!

對於 TV GAME 系統,所剩下的就是接到 TV 上了!

[第四步 模擬器的運作方式]

4.0 模擬器與真實的硬體還是有差距的

姑且不論模擬的像不像,好不好,從執行時的情況看來,真實的硬體是允許平行處理的,而模擬器卻是以循序的方式,輪流使用 PC CPU 的資源!有很多硬體的機制是模擬器做不來的,全盤以硬體的角度看模擬器,會發現有很多東東是無法

實作的,單從模擬器的角度看硬體,也會發現文件實在漏掉太多東西!以下的大部分內容是以解讀 SNES9X 1.10,MAME 35b5 的原始檔內容所撰寫的,就 SFC 的硬體而言,算是滿符合上面所分類的項目,除了缺少 CD-ROM DRIVER 的部份之外,此外有很多都是我個人的想法,不一定是正確的,這點一定要知道!

4.1 CPU 的處理

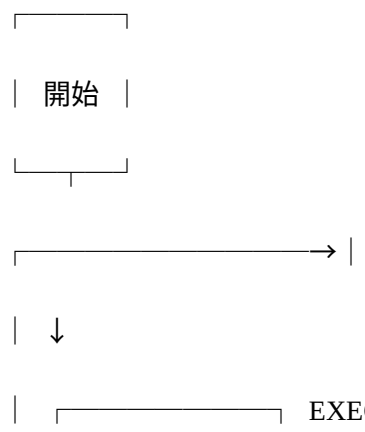
當你選定了 CPU 之後,自然便是需要模擬那個 CPU 的運行!模擬器的 CPU 與真實的 CPU 有很大的不同,可以分為:

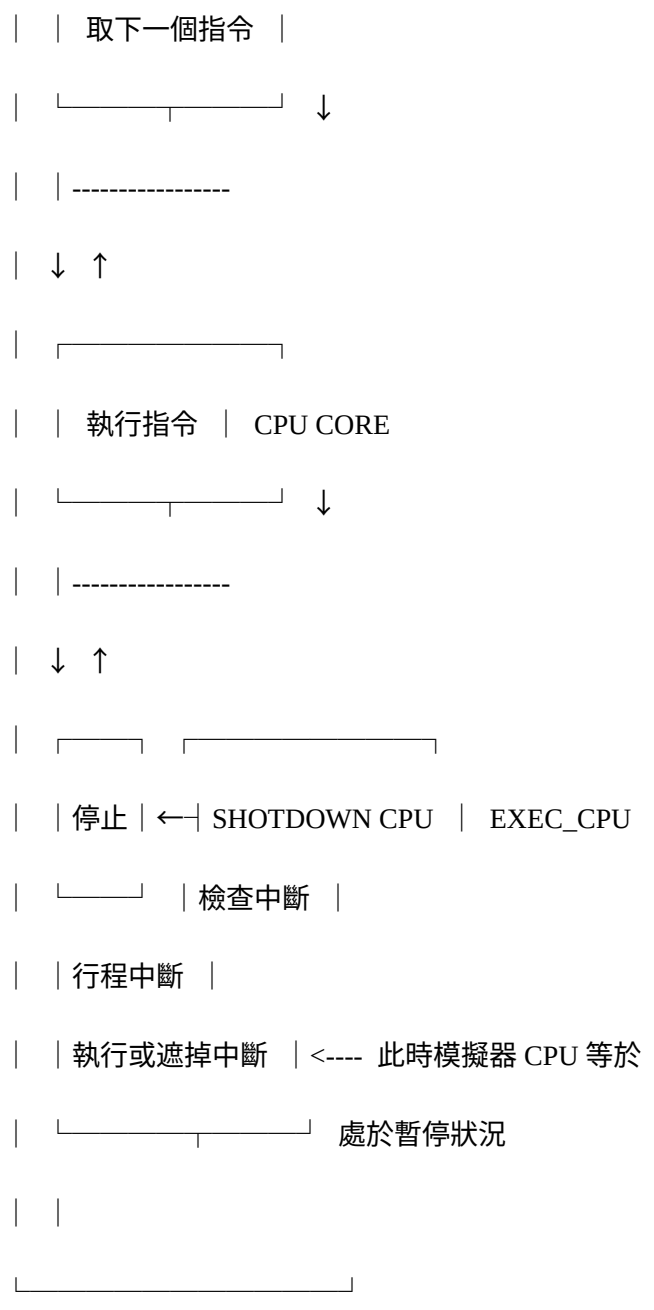
A.CPU CORE:負責解譯指令!

B.EXEC_CPU:負責流程控制!

CPU CORE 與 EXEC_CPU 組成了模擬器的核心,CPU 與外界的接觸及排程由 EXEC_CPU 控制,在 NMI,SPU,其他裝置何時執行,在其餘時間再把 ROM 中的指令交由 CPU CORE 解譯及執行!執行的流程是由 EXEC_CPU 檢查需不需要執行 NMI,SPU,及其他裝置,若需要執行,則執行 NMI 等相關程式,若不需要執行,則從 ROM 中抓取指令交由 CPU CORE 解譯及執行,CPU CORE 處理完再交回 EXEC_CPU 檢查需不需要執行 NMI,SPU,及其他裝置,直到程式結束為止!

以流程圖表示:





此時模擬器 CPU 等於又開始執行下一個指令

4.1.0 CPU CORE

CPU CORE 的名稱應該是指這個程式承繼了真正 CPU 的核心,實際上也是,整個 CPU CORE 只保留了暫存器,定址模式,OPCODE 的解譯及執行(含 ALU,CU,IU,OU),PC(PROGRAM COUNTER,程式計數器)改為整體變數(或 EXEC_CPU 的區域變數)等,去

除掉所有的中間暫存器(IR,MBR,MAR)!但是因為缺少了時序週期,必須增加另外一個變數 CLOCK_COUNT,當成時序週期來用,並對所有的 OP-CODE 加上一個會用掉多少 CLOCK 的數值,當執行某一個 OP-CODE 之後,CLOCK_COUNT 必須減掉這個數值,表示經過了多少個 CLOCK!

有許多 CPU CORE 的作者,同樣的也會有不少不同種類的 CPU CORE!不同的 CPU CORE 效果也大不相同!其中的差別在於:

A.所使用的語言不同:

以組合語言及 C 語言兩種為主,一般以組合語言所寫的速度較快!原因是可以避免掉編譯時所產生多餘的碼,不過實際上還必須評估每個 OP-CODE 花多少碼模擬,執行將花費多少時間等!

B.對 OP-CODE 解譯的方式不同:

怎麼做到有效率的模擬及正確的模擬 OP-CODE,一直是 CPU CORE 作者的目標,問題是每個人對於 OP-CODE 的解釋不同,實作出來的 CPU CORE 便會發生解譯不同的情形!這種解譯不同的情況,嚴重時,會導致模擬的效果不正確!

4.1.1 套用現成的 CPU CORE :

有現成的 CPU CORE 時,當然就是想辦法直接套用!要注意套用的程序,例如,必須先初始化 CPU CORE,設定一個定址空間交給 CPU CORE 等!所以在套用之前,一定需要先看過說明檔,以求了解套用的程序!

4.1.2 自己做一個 CPU CORE:

只好自己做一個!做一個 CPU CORE 模擬器,說難不難,說簡單也是騙人!當然一個簡單的方式便是修改別人的 CPU CORE 的 OP-CODE 部份,修改成為你心目中認

為的 OP-CODE 模擬方式!我沒有實際做一個 CPU CORE 模擬器的經驗,只能有觀察別人的模擬器中提供一些經驗!

需要提供幾個重要的函式,這是 CPU CORE 的外觀:

A.RESET_CPU(INITIAL_CPU):

把 CPU CORE 的狀態恢復至初始時的狀態!各暫存器,旗標,程式計數器!

B.CPU_SHUTDOWN:

釋放 CPU CORE 所佔用的記憶體!

C.OP-CODE DECODER&EXECUTOR:

解譯並執行 OPCODE 的程式!寫法很多,CASE-SWITCH,或是函式陣列等!不管是 CASE-SWITCH 或是函式陣列都是利用轉譯前 OP-CODE 的十六進位數值當成判斷的依據!

就結構上來說,需要實作出這是 CPU CORE 的內在:

A.CPU 的暫存器,旗標等!你可以直接定義一個 CPU 的結構,這樣應用上較為便利!

B.CPU 的定址模式!直接定址指令為何,間接定址指令如何處理等!

C.CPU 所用的每個 OP-CODE,實作的方式滿多的,需要 ALU 的 OP-CODE 直接用四則運算代替等!你必須要先知道這個 CPU 提供多少種 OP-CODE,每個 OP-CODE 的運作方式,OP-CODE 如何解碼等,再一一實作出每個 OP-CODE 來!

我目前的程度,只能提供意見到這裡!

4.1.3 MULTI-PROCCESSOR 的場合:

面對多處理機的系統時,不可避免的問題就是:

A.必須實作出好幾個 CPU!

B.必須實作 CPU 的溝通方式!

對於問題 A 而言,可以以定義出 CPU 結構再宣告多個 CPU 結構當成有多個 CPU,而 CPU CORE 共用同一份!對於問題 B 而言,如果是 MASTER/SLAVE 系統,就是在 MASTER CPU 的 EXEC_CPU 中多實作出處理 SLAVE CPU 中斷及其他溝通的方式等!

4.1.4 CPU 與其他週邊溝通方式的處理

簡單的說,就是處理中斷的方式!真實的硬體是可以平行運作的,但是模擬器可不行,至少在沒辦法寫成幾乎可平行運作又可相互溝通的執行緒前不行!由前面的流程圖可以發現,必須在 EXEC_CPU 中排入中斷檢查,中斷執行,和一切週邊狀態檢查及更新的程式片段!整理一下,包括:

A.SPU 狀態的檢查:

SPU 狀態的檢查在整個模擬器佔了很重要的地位,因為聲音不能有任何中斷現象,所以幾乎在每個函式都必須執行一次,在較長的函式中甚至必須執行許多次,以確保聲音不會中斷!

B.搖桿狀態檢查:

檢查搖桿按鍵是否有按下的訊息傳來!

C.NMI 中斷檢查:

檢查是否有 NMI 中斷發生,依其優先順序執行!

D.一般中斷檢查:

檢查是否有中斷發生,依其優先順序執行!

E.PPU 定期資料更新:

簡單的說就是畫面的更新!前面 CPU CORE 段落中說到必須將每個 OP-CODE 加上一個用掉多少 CLOCK TIME 的常數,在把 PPU 定期資料更新週期定義成多少個 CLOCK COUNT,每執行一個 OP-CODE 就扣掉一些,等到變成負數或等

於零時,就表示定期資料更新週期到了,就去執行資料更新的函式!這就

是沒有時序產生器以產生時序的變通方法!

F.SPU 定期資料更新:

若 SPU 中含有 TIMER,則必須於一定時間時更新狀態!這可保持整個模擬器

以一定的時序同步執行,因為有些裝置是需要同步執行的!這個 TIMER 的

計算方式是以相對於多少 CPU CLOCK TIME 或是以相對於多少 CLOCK COUNT

來計算!

4.1.5 DSP 的處理

實際上有兩種類型的 DSP,這兩種在處理有很大的分別:

A.模擬這個 DSP 的外在行為:

模擬時需要做一個 DSP 的結構與 CPU 溝通,但是實際上的內部執行只是呼叫

函式而已!例如給 DSP 算一個 COS 值等!這是模擬這個 DSP 的外在行為就可以

了!

B.模擬這個 DSP 的內在行為:

最明顯的例子就是 SFC 的 SuperFX 晶片,這其實是一個 RISC 的處理器,必須

模擬出這個處理器的內在行為,OP-CODE 等,就像模擬一個 CPU 一樣!

4.2 MEMORY MAP 的處理

簡單的說,就是配置所需的記憶體,包含 ROM,RAM,SRAM,VRAM(PPU 用)等!並載入

至對應的區域內!同時在模擬器結束時必須釋放出所配置的記憶體!需要實作

出:

A.INIT MEMORY MAP:

也就是初始化 MEMORY MAP,對 ROM,RAM,SRAM,VRAM(PPU 用)都配置足夠大的記憶體!不過倒是不需要只配置一大塊記憶體當成整個 MEMORY MAP,分成 RAM 區,SRAM 區,ROM 區也可以,但是存取時必須視為一整塊 MEMORY MAP!另外還有一些需要配置的記憶體區塊,也能在此一同處理!

B.FREE MEMORY MAP:

在模擬器結束時,必須釋放出原來配置的記憶體,在這個函式中處理!

C.LOAD ROM IMAGE FILE:

檔案處理的部份不在這邊做,這裡直接用檔案指標參數操作!把檔案內容搬移到 MEMORY MAP 中分配給 ROM 的區域!一般而言,除了 CD-ROM 之外,都是配置出足夠的大小,很明顯的,如果 ROM IMAGE 有 50MBYTE,就需要配置出 50MBYTE!這裡需要注意一下,有些系統的 ROM 區域是分段的,LOW ROM&HIGH ROM,也需特別處理!

D.LOAD/SAVE SRAM FILE:

與 LOAD ROM IMAGE FILE 處理的方式一樣!比較不一樣的是,SRAM FILE 要隨時準備存取,ROM IMAGE FILE 一般只有在更換遊戲時才會更動!

E.LOAD SYSTEM ROM FILE:

與 LOAD ROM IMAGE FILE 處理的方式一樣!

F.FIX ROM SPEED:

有時候,會需要模擬 ROM 存取的速度!

G.特殊晶片的 ROM IMAGE 處理:

有些晶片或是 PPU,SPU 會有用到專用的 ROM,原則上仍然等同 LOAD ROM IMAGE FILE 的方式處理!

這裡提供的還是原則性的寫法,詳細的內容,還是得多看看別人的處理方法!

4.3 PPU 的處理

4.3.0 PPU 的處理

在大部分的情況而言,通常都不會知道 PPU 是什麼晶片,所用的 OP-CODE 等,也找不到相關的資料,通常找到的是 CPU 透過 MEMORY MAP 的窗口與 PPU 溝通的資料,所以模擬的方式一般都是模擬外在的行為,而不是模擬內在的操作情形!無論如何,需要提供:

A.RESET PPU(INIT PPU):

重新把每個 MEMORY MAP 所對應的功能的參數,恢復成初始的狀態!

B.CPU 設定 PPU 的函式:

也就是處理那些 MEMORY MAP 的功能的函式!與 CPU 的 OP-CODE 不太一樣,這裡較常用 CASE-SWITCH 的結構!原因是速度較快吧!以 MEMORY MAP 所分配的"位址"當成 SWITCH 的判斷依據!

C.CPU 讀取 PPU 處理後狀態的函式:

也就是處理完那些 MEMORY MAP 的功能之後,預備由 CPU 讀取的函式!也是使用 CASE-SWITCH 的結構!這裡不一樣的地方是,並不是每個功能都有傳回資料,某些 CASE 的處理就可直接跳過!也是以 MEMORY MAP 所分配的"位址"當成 SWITCH 的判斷依據!

不過這都只是外觀而已,真正的還是模擬每個功能的程式,一般找得到的資料會像這個樣子,以 SNES9X 為例:

SNES Documentation v2.30: Written by Yoshi

|rwd2?|Address|Title & Explanation |

||||||----------|

||||||

||||||__ ?: Don't know what the statistics on this register are |

|||||____ 2: 2 byte (1 word) length register |

||||_____ d: Double-byte write required when writing to this register |

|||______ w: Writable register |

||_______ r: Readable register |

| |

|Words in brackets ([]) are the official "names" of the registers |

|Words in braces ({}) are different from the "real" SNES manual |

|Bits define 1 as "ON/ENABLE" and 0 as "OFF/DISABLE," unless otherwise stated|

|Registers without any bits/defined-data can be assumed to be 8 bits in size |

|and should only be read once. |

|rwd2?|Address|Title & Explanation |

|---|

| w |$2100 |Screen display register [INIDISP] |

| | |x000bbbb x: 0 = Screen on. |

| | | 1 = Screen off. |

| | | bbbb: Brightness ($0-$F). |

| | | |

| | | |

按照說明寫成對應的程式:

==========
==

```
case 0x2100:
// Brightness and screen blank bit
if (Byte != Memory.FillRAM [0x2100])
{
FLUSH_REDRAW ();
if (PPU.Brightness != (Byte & 0xF))
{
IPPU.ColorsChanged = TRUE;
PPU.Brightness = Byte & 0xF;
S9xFixColourBrightness ();
if (PPU.Brightness > IPPU.MaxBrightness)
IPPU.MaxBrightness = PPU.Brightness;
}
if ((Memory.FillRAM[0x2100] & 0x80) != (Byte & 0x80))
{
IPPU.ColorsChanged = TRUE;
PPU.ForcedBlanking = (Byte >> 7) & 1;
}
}
break;
```

==========

關於如何寫才算完美,對不起,我沒有辦法說明!而實際上,這就是一個模擬器

好壞的差別!

需要補充的是,通常在 CPU 設定 PPU 的函式中會把像 SPU,DMA 等同樣使用 CASE-SWITCH 結構的函式放在同一個 CASE-SWITCH 結構中!只是執行的函式本體另外寫而已!

4.3.1 實際繪圖的動作:

上面拉拉雜雜一堆,就整個 PPU 而言,仍然只是外觀而已!只是設定 PPU 去做某一件事,設定它的參數,你還是得實作出真正繪圖的動作!這裡還是受限於我的知識不足,等到以後再補充!另外,需要做一個初始動作的函式,這是處理 PPU 內部的初始化!

4.3.2 SCREEN REFRESH

SCREEN REFRESH 有兩種:

A.模擬硬體處理 SCREEN REFRESH 的動作:

這裡還是受限於我的知識不足,等到以後再補充!

B.實際顯示出畫面於 MONITOR 上:

這裡還是受限於我的知識不足,等到以後再補充!

4.4 SPU 運作方式:

4.4.0 SPU(圖形處理單元)的功能及地位:

4.4.1 SPU 的組成:

4.4.2 與 PPU 的不同處:

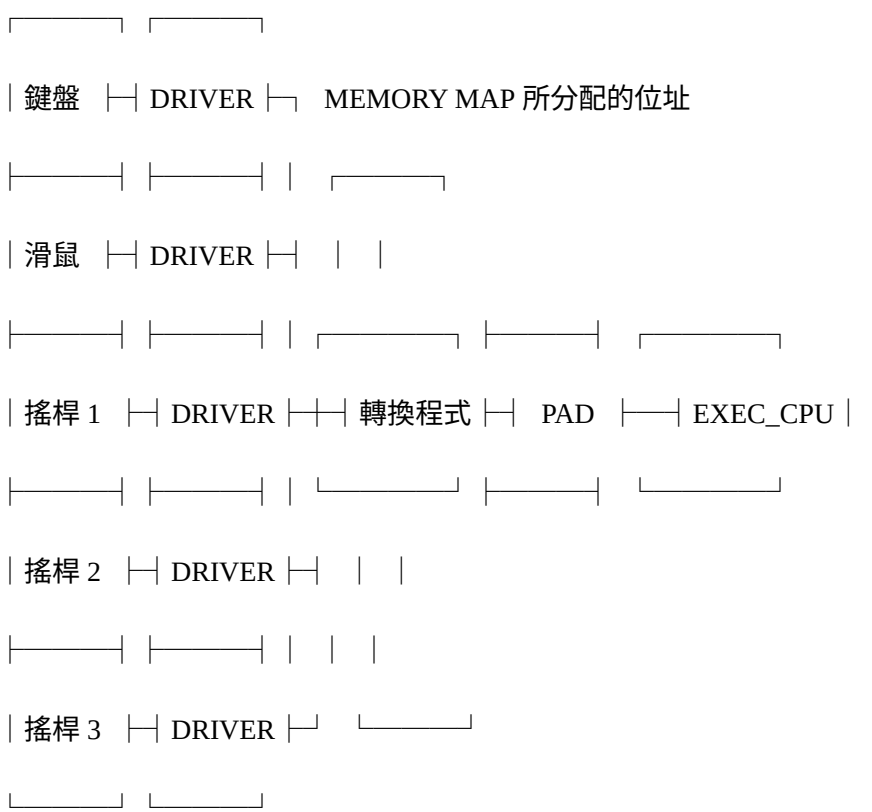
4.4.3 聲音處理器專用 DSP 的功能:

4.4.4 聲音處理器的指令:

4.5 搖桿控制裝置及錢幣計數器等 NMI 的處理

4.5.0 MEMORY MAP 中處理搖桿控制裝置的方式:

除了一些特殊控制器之外,搖桿上的每個按鍵或是方向鍵都是用 1BIT 表示其狀況--按下或是沒按下!直接由 MEMORY MAP 所分配的位址下手,但是因為 PC 上可拿來控制的裝置太多了,需要一個中間程式轉換,大致如下:



搖桿?表示

另一種搖桿,不是另一隻

這樣畫是稍微不正確,不過講解很方便!要加入一種新的裝置,就需要有專用的 DRIVER 才能支援到!所謂專用的 DRIVER 是指如何讀取訊號,怎樣解讀所得到的訊號是什麼意義,負責這種工作的程式,一些特殊的指向裝置都會提供專用的

開發工具,這些開發工具便會提供如何解讀所得到的訊號,其實這就是我所指的 DRIVER!DRIVER 是對應 WINDOWS 的說法!搖桿的動作不正常,亂跳啦,不支援某種搖桿啦,就是 DRIVER 層級的問題!重新對應所按的按鍵,或是選擇不同的裝置輸入訊號,就是轉換程式的問題!控制裝置所找到的 DRIVER 不同的話,支援的情況就有很大的差別!就 WINDOWS 模擬器而言,比較新的通常都使用 DIRECT X 的 DIRECT INPUT 支援不同的指向裝置,這樣非常方便,只要符合 DIRECT INPUT 的函式呼叫,就通用所有指向裝置!但是 DOS 模擬器並沒有這麼簡單,必須一種一種的支援!

另外我把指向裝置切換及按鍵重定義視作轉換程式!轉換程式的目的在於適當的把所選擇的指向裝置上所按的按鍵轉換成所模擬 TV GAME 主機的按鍵資訊字組!很明顯的,具有三種功能:

1.選擇所支援的指向裝置!

2.可自由定義所選擇的指向裝置上的按鍵對應所模擬主機上的按鍵!

3.轉換所選擇的指向裝置上的按鍵定義成為所模擬主機使用的資訊!

不過其實還有第四種功能:

4.熱鍵(HOT KEY)處理界面!

轉換程式一般都是利用 CASE-SWITCH 結構設計,不論是處理何種指向裝置的 DRIVER 提供的訊息!而 K/B 的所使用的 CASE-SWITCH 結構,還可以加入熱鍵處理,例如呼叫 GUI,RESET,FRAMESKIP,抓圖,當然還有 EXIT--退出模擬器,處理方式就是增加 CASE,當 KB 有按鍵按下時,屬於模擬器搖桿的部份就組合成正確的字組填入 MEMORY MAP 的位址,其他的則檢查是否屬於熱鍵,是則執行對應的熱鍵處理函式,再不屬於熱鍵的就直接忽略掉!

不管是什麼指向裝置,到了轉換程式最後都會變成所模擬主機的按鍵資訊字組!然後設定一個中斷 FLAG,等到模擬器控制權又回到 EXEC_CPU 時,將會檢查這個中斷 FLAG,再讀取這個按鍵資訊字組,然後 CPU CORE 會根據這個字組,執行 ROM 中的程式!

4.5.1 特殊指向裝置的處理:

下次再寫!

4.5.2 錢幣計數器,START 鍵等 NMI 的處理:

不管是何種 NMI,都是設定一個中斷 FLAG,等到模擬器控制權又回到 EXEC_CPU 時,將會檢查這個中斷 FLAG,再讀取這個按鍵資訊字組,然後 CPU CORE 會根據這個字組,執行中斷服務常式或 ROM 中的程式!

4.6 RESET 處理

算起來這個最簡單,把所有相關的初始函式,全部關在一起呼叫一次,就是 RESET 了!如果要處理一些特殊動作,也是在這裡處理,當所有初始函式執行完成之後,再執行想要做初期奇怪動作的函式!這個功能的呼叫方式通常是設置在按鍵讀入的處理函式中,在 CASE-SWITCH 結構中,多一個 CASE 處理 RESET!

4.7 DIP SWITCH 作用處理方式:

這個也是滿簡單的,設計一個以位元操作的方式定義出 DIP SWITCH 的結構,最

好是附以顯而易見的定義,最後是設計一個專門處理設定 DIP SWITCH 功能的函式,有專用的畫面及控制方式等!比較懶的方式是,讓使用者自己計算出要設定 DIP SWITCH 功能的十六進位數值,模擬器只要讀入這個數值即可!這個功能的呼叫方式通常是設置在按鍵讀入的處理函式中,在 CASE-SWITCH 結構中,多一個 CASE 處理 DIP SWITCH!

4.8 DMA CONTROLLER 運作處理方式

模擬器是不可能會有 DMA 這種機制的,當 DMA 功能啟動時,實際執行時是從 EXEC_CPU 函式中直接轉移到 DMA 控制函式,進行字組的搬移,但是要定時或完成一個動作時,需要模擬 DMA CONTROLLER 對 CPU 發出中斷的動作,這個動作就是整個 DMA 函式的核心,而資料搬移動作反而是最不重要的,因為這個動作就是資料搬移的動作:

TARGET_MEM = SOURCE_MEM;

還有一個動作要處理,就是 CLOCK TIME 的動作,不論 DMA 多快,都不可能在不經過任何 CLOCK TIME 就能完成,我們需要模擬 CLOCK TIME 經過的動作!

你會發現 EXEC_CPU 轉移到 DMA 控制函式時,整個模擬器 CPU 是處於暫停的狀態,這也是符合 DMA 運作的機制!

前面都是 DMA CORTROLLER 的內部機制:

A.模擬 DMA CONTROLLER 對 CPU 發出中斷的動作!

B.資料搬移動作!

C.模擬 CLOCK TIME 經過的動作

還必須提供 CPU 使用 DMA 的機制,CPU 對 DMA CONTROLLER 的控制方式還是以 MEMORY

MAP 中分配給 DMA CONTROLLER 的窗口設定 DMA CONTROLLER 的參數,在 DMA

CONTROLLER 動作時便按照這些參數運作,需要實作出以下的函式:

A.RESET_DMA:

恢復 DMA CONTROLLER 成初始狀態!

B.DMA 傳輸模式:

必須提供所使用 DMA 的所有傳輸模式!CYCLE STEALING 就是搬移一個字組

就把控制權交回 EXEC_CPU,BURST MODE 除了資料搬移外,還需要處理經過

多少 CLOCK TIME!

4.9 光碟機裝置存取方式:

有空再寫! :)

4.A 檔案處理:

4.A.0 檔案載入

有空再寫! :)

4.A.1 LOAD HISCORE FILE

有空再寫! :)

4.C.2 LOAD/SAVE 隨時記憶檔

有空再寫! :)

4.B 特殊功能:

4.B.0 抓圖

有空再寫! :)

4.A.1 抓音樂檔

有空再寫! :)

4.B.2 GUI

有空再寫! :)

4.B.3 CHEAT CODE

有空再寫! :)

4.B.4 CHEAT TOOL

有空再寫! :)

4.B.5 DISASM

有空再寫! :)

4.C 可攜性版本,WINDOWS 版本,更先進的模擬器架構

有空再寫! :)

4.C.0 可攜性的研究

有空再寫! :)

4.C.1 WINDOWS 版本

有空再寫! :)

4.C.2 更先進的模擬器架構

有空再寫! :)

[第五步 如何偷別人的程式及與別人交換意見]

5.0 如何偷別人的程式

5.0.0 模仿是進步最快的方式:

如果能找到現成的模擬器原始碼,可以幫助你了解許多事,包括如何套用別人的 CPU CORE?如何寫模擬器?不同硬體間如何交換資訊?當然更快的方式便是直接修改別人的原始檔,不過這似乎是滿沒品的!

5.0.1 哪裡找得到模擬器原始碼?

一般都會出現在提供模擬器作者程式設計資料的網頁上,或是從模擬器新聞性網頁中也會有模擬器作者釋出原始檔的消息,可以連到該模擬器的網頁上直接下載!當然還有一個很爛的方式--直接向作者要,不過肯直接給的應該不多吧!

5.0.2 看得懂別人的程式也不是件容易的事:

不要以為有原始檔就 OK 了!事實上想看懂別人的程式,還是必須要下一番功夫的!如果弄到的是公開釋出的原始檔,正常來說,註解及格式都會相當完整,看懂的機會較高!有幾個 TIPS 可以參考:

A.從檔名下手:

公開釋出的版本對檔名的分類會很清楚,從檔名猜看看會得到不小幫助!

B.從 INCLUDE 檔下手:

如果是 C/C++語言的話,從 INCLUDE 檔也可以找到不少有用的資訊,因為有許多結構及函式宣告都會出現在對應的 INCLUDE 檔中!

C.從函式名稱下手:

嚴謹的程式寫法對函式名稱非常注重,直接從函式名稱猜也行!

D.去除不必要的#ifdef 等條件編輯的程式區塊:

條件編輯在除錯及特殊用途時非常有用,但是對想偷程式的人來說,便是累贅了,適當的去除它們會使程式看起來簡單一些!

5.1 和別人交換意見:

一般說來,找一個模擬器的討論區,或是直接寫信給作者,把所發生的問題"具體"的描述出來,都應該收到一定程度的答案!當然描述的越清楚,或是附上一些相關的程式片段或是圖片等,會更容易幫助別人釐清問題!

另外一種方式是尋找同好一起發展模擬器,雙方在一起腦力激發,會比一個人閉門造車好一點!

[第六步 實際撰寫一個模擬器]

呵呵,我沒有經驗,等我有經驗再補這一段!原則上,可以參考一些書籍中有關程式風格的章節!寫的規矩一點會有助於追蹤 BUG!

[第七步 測試,除錯及版本更新]

7.0 測試與除錯是一條漫長的道路:

很多功能都只是覺得應該是這樣,所以便照著自己的想法寫程式碼,因為找資料時實,在不容易找到太細部的資料,但是缺少了這些資料,有些功能其實只是沒用到特殊的地方,而寫程式時並不曉得,萬一有遊戲用到,模擬的效果便很差!問題是就算知道有 BUG,就有能力改嗎?如果有原來的硬體執行看看,也許會有些啟發!或是去找別人的程式看看!

7.1 自己測試的盲點:

因為個人的偏好,可能會只針對想測試的遊戲測試,對於不喜歡的遊戲可能就沒有測試的很完整,如果你找得到朋友幫你測試看看,也許找得到別的 bug!

7.2 版本更新:

當每解決一個 BUG,每更新一個地方,應該隨時記錄下來,一方面作為自己參考所用,一方面提供別人使用時的參考!因為可能當你解決這個 BUG 時,又引發另一個 BUG 出現,如果有記錄版本更新的項目,也可藉此了解 BUG 出現的方向!

7.3 最少保留一份每一個版本的原始檔及執行檔:

這是一方面可以在發展機器發生問題時,還有機會拯救回來,另一方面舊版本可能在某些地方反而優於新版本,保留舊版本的原始檔可以交互參考!

[第八步 撰寫說明文件]

撰寫說明文件是相當重要的事,除非你只是給自己玩而已,否則當需要介紹給別人時,就要有一份完整的說明文件!一般常見的說明文件有幾種:

A.README:

包括測試環境,相容遊戲列表,按鍵說明,選項設定說明,更新項目,版本沿革,還有相關協助人員的列表!當然相容遊戲列表,更新項目也可以直接分離出來!

B.FAQ:

列舉出這個模擬器常見的問題及解決之道!

如果需要翻譯成不同語言時,也許應該請別人代筆,即使沒有請別人代筆,也應該請別人看看寫的會不會怪怪的!

[第九步 公開你的版本]

9.0 建立個人網頁:

建立個人網頁是一個最快也是相當不錯的方式,可以把想讓 FANS 知道的消息,說明文件,執行效果的圖檔(SCREENSHOT),提供各版本下載,或作一個留言版,增加與喜歡這個模擬器的 FANS 的互動!

9.1 投稿至模擬器新聞性網頁!

直接投稿至模擬器新聞性網頁也是相當不錯的選擇,因為新聞性網頁每天觀看的人很多,宣傳的效果很好,如果再加上模擬器實在做得不錯,一夜成名並不是夢想!

[後記]

看到這裡,應該對整個模擬器的發展有點初步的認識!但是想以此寫成一個模擬器,實際上還有大的障礙要通過!當然這篇文章還有很多錯誤的地方,你應該保持處處存疑的態度,以免造成你的誤解,而影響你寫模擬器的進度!最後我希望這篇文章能對你有些許的幫助!

[常用名詞解釋]

1.EMULATOR & SIMULATOR

這兩者中文都叫"模擬",兩者有何不同呢?

EMULATOR 注重在內部的運作機制的一致!就巨觀的觀察來看,是模擬整個硬體系統,使得同一個硬體系統的程式能在這個程式上正常運作--一如在原本的硬體系統上執行一樣!

SIMULATOR 注重在外在操作與反應上的一致!就巨觀的觀察來看,是改寫程式,使這個程式表現的方式類似於所模擬的事物!例如飛行模擬 FLIGHT-SIM,不管模擬器再真實,都沒有辦法真正模擬出飛行的感覺!

而如果我們微觀的觀察 EMULATOR 的運作時,實際上,與 SIMULATOR 的差異只有程度上的不同!真正在做 CPU 模擬器的 OP CODE 時,還是以模擬操作與反應的一致!

2.SCHEMATICS 概圖:

利用電子元件的所繪製的線路圖!

3.DIP(Dual In-line Package) switch DIP 指撥開關

一種位於電路板上的開關,用來選擇硬體設定參數!DIP(雙排插腳包裝)是指這種開關的外盒是 DIP!

4.MEMORY MAP 記憶體對映

主記憶體的分段(SEGMENT)裡,用來定義哪些區域做什麼用途.

5.ROM IMAGE

一般 TVGAME 及 ARCADE 遊戲,都是把程式或資料燒錄在 ROM 中-不論是 ROM IC 顆粒,還是 CD-ROM!很明顯的,除了 CD-ROM 之外,沒有辦法直接讀取其中的指令或資料(即便是 CD-ROM 也可能自己搞奇怪格式,所以也可能無法讀取),所以需要使用 ROM DUMPER,把 ROM 中的指令或資料 DUMP 成二進位制檔案,稱為"ROM IMAGE"!

6.MEMORY MAPPED I/O

MEMORY MAPPED I/O 是指 I/O 界面位址和記憶體位址使用相同的位址空間,此時界面暫存器是記憶體系統的一部份!界面單元和記憶體使用相同指令存取,資料位址指到記憶體,即是和記憶體溝通;如果指到界面暫存器位址範圍即是要進行 I/O 動作!

7.OP CODE:

CPU 中,指明將進行什麼運算(如 ADD,I/O).以二進制代碼來指定一個運算,稱為

運算碼(OP CODE)!

如何寫一個電腦模擬器

前言

怎樣,你決定寫一個軟體模擬器嗎?很好的,這篇文件或許能給你一些幫助!它

涵蓋了一些當人們問及如何寫模擬程式時的通用技術性問題.它同時也提供你

模擬器內部運作的?#123;圖,讓你多少能夠依循!

提綱:

什麼能被模擬呢？

什麼是" emulation "? 和"simulation "有何不同?

模擬有專利的硬體合法嗎?

什麼是"interpreting emulator"? 和"recompiling emulator" 有何不同?

我想要寫一個模擬器.我該從哪開始?

我該使用何種程式語言?

我要從哪裡找到被模擬的硬體的資料?

製作步驟

我該如何模擬一個 CPU?

我該如何處理被模擬記憶體(emulated memory)的存取?

週期行程(Cyclic tasks):這是什麼?

程式寫作技巧

我該如何最佳化 C 程式碼?

什麼是 LOW/HIGH-ENDIANESS?

如何使程式具有可攜性?

為什麼我應該使程式模組化?

其他

什麼能被模擬?

基本上,任何要被模擬的系統都有一個微處理器.當然,只有執行較具彈性或

較不具彈性程式的裝置是我們感興趣模擬的.包含:

電腦.

計算機.

家用遊戲平臺.

大型電玩.

其他.

記下你能模擬的任何電腦系統是必要的,即使它十分複雜(如 Commodore Amiga

computer).雖然這樣的模擬程式效能也許不太好.

什麼是" emulation "? 和"simulation "有何不同?

" emulation "是意圖模擬一項裝置的內部設計."simulation "是意圖模擬

一項裝置的功能!譬如，一個程式能夠模擬"小精靈(PACMAN)"大型電玩硬體和

執行真正"小精靈(PACMAN)"ROM,這個程式就是一個 EMULATOR!一個為了你的電

腦而寫的"小精靈(PACMAN)"遊戲，但使用圖像類似於真正的大型電玩,這個遊

戲程式就是一個 SIMULATOR!

模擬有專利的硬體合法嗎?

這檔事其實是界於一個灰色地帶,模擬有專利的硬體似乎是合法的,但是不包

含非法使用這些用於有專利硬體上的資訊!你應該也發現到伴隨著模擬器一起

散佈 SYSTEM ROMS(BIOS,或其他)是違法的,如果這些 ROM 是有版權的!

什麼是"interpreting emulator"? 和"recompiling emulator" 有何不同?

有三種基本規劃可以用以設計一個模擬器.他們也能相互結合以獲得最佳效

果!

直譯型(Interpretation)

一個模擬器執行的方式是從記憶體一個 BYTE 一個 BYTE 的讀入被模擬的程式碼

,再解碼這被模擬的程式碼,再執行適當的指令在被模擬的暫存器,記憶體,及 I/O

,這種模擬器就稱為直譯型模擬器!這類的模擬器一般的演算法如下:

```
while(CPUIsRunning) /* 利用一個檢查 CPU 是否執行的旗標的無限迴圈 */
{
Fetch OpCode /* 抓取運算碼 */
Interpret OpCode /* 解譯運算碼 */
}
```

這種方式的優點包含易於除錯,具可攜性,易於同步(synchronization,你

可以簡化計算所經過的時序週期,並且固定你的模擬器在同一週期數的延遲)

明顯的缺點就是效能.直譯型模擬器花費相當多 CPU 時間,並且你可能需要

十分快的電腦來執行你的程式,才能獲得尚可的速度.

靜態重編譯型(Static Recompilation)

在這個技巧中,你試圖將一個由被模擬程式碼寫成的程式,翻譯成你的電腦所使用的組合語言碼.結果將成為一個無須任何特殊工具就可以在你的電腦執行的可執行檔.靜態重編譯型模擬器聽起來似乎不錯,不過這通常並不可行的!例如,你不能完全先行編譯會自己修正的程式碼(self-modifying code),因為不執行這些程式碼,是不可能知道它們將會變成什麼的!為了避免這種狀況,你也許可以將直譯型模擬器或動態重編譯型模擬器與靜態重編譯型模擬器組合起來!

動態重編譯型(Dynamic Recompilation)

動態重編譯型模擬器與靜態重編譯型模擬器在本質上是相同的,只不過動態重編譯型模擬器出現在程式執行時.為了代替一次就重編譯好所有的程式碼,我們可以只在遇到"CALL"或是"JUMP"指令時,立即再作一次重編譯工作!為了增加速度,這種技巧也能與靜態重編譯型模擬器組合使用.你能在這裡找到更多有關動態重編譯型模擬器的資料:

http://www.ardi.com/MacHack/machack.html white paper by Ardi,

creators of the recompiling Macintosh emulator.

我想要寫一個模擬器.我該從哪開始?

為了寫模擬器,你必須要有良好的電腦程式設計及數位電子學的一般知識背景.有組合語言程式設計的經驗亦十分重要.你先要做到:

選擇所使用程式設計語言.

找到所有有關被模擬的硬體的資料.

重寫一個 CPU 的模擬器或是取得已寫好的 CPU 模擬器程式碼.

最少先部份地寫一些草擬碼來模擬硬體的一部份,由這點看來,在寫模擬器時同時作一個擁有允許暫停模擬及看看這個程式正在幹嘛這種功能的小型內建除錯

器,這樣是非常有幫助的.同時你也需要被模擬系統所使用的組合語言的反組譯

程式.如果找不到現成的反組譯程式,就自己作一個.試著在你的模擬器上執行

程式看看.使用反組譯程式及除錯器了解程式是如何使用硬體的,並且適當的?#123;

整模擬器的程式碼.

我該使用何種程式語言?

最明顯的選擇方案是 C 及 組合語言.

以下是他們的優缺分析:

組合語言

優:1.可以產生較快速的程式碼.

2.用以模擬的 CPU 的暫存器可以直接用來儲存被模擬的 CPU 暫存器的資料.

3.許多運算碼可以用相似的用以模擬的 CPU 的運算碼來模擬.

缺:1.這種程式碼沒有可攜性,也就是不能在不同架構的電腦上執行.

2.程式碼的除錯與維護不太容易.

C

優:1.可以做成具可攜性的程式碼,如此便能使用在不同的電腦及作業系統.

2.相對來說,程式碼的除錯與維護較為容易.

3.可以很快的測試對真實硬體如何運作的不同假設.

缺:一般而言,C 程式碼通常比純組合語言程式碼慢一些.

對於寫一個可運作的模擬器來說,對所選擇的程式語言有深入的認識是絕對

必要的,因為這是十分複雜的計畫,並且你的程式碼應該盡可能的最佳化使之執

行的更快!說的明白一點,作電腦模擬器不應是一個學習程式語言的計畫.

我要從哪裡找到被模擬的硬體的資料?

以下列出來的地方,你也許想要看看.

Newsgroups

comp.emulators.misc

這是一個有關電腦模擬一般性討論的 Newsgroup.許多模擬器作者也閱讀這裡的討論,不過這裡的廢話多了一點!在你想刊登問題在這個討論區之前,請先看看這裡:

http://www.why.net/home/adam/cem/ c.e.m FAQ

comp.emulators.game-consoles

跟 comp.emulators.misc 差不多,不過特別注重家用遊戲平臺模擬器.在你想刊登問題在這個討論區之前,也請先看看這裡:

http://www.why.net/home/adam/cem/ c.e.m FAQ

comp.sys./emulated-system/

comp.sys.*類階層包含許多特殊電腦系統的討論區.

藉由讀這些 newsgroups,你可以獲得很多的有用的技術性資訊.典型的例子如:

comp.sys.msx MSX/MSX2/MSX2+/TurboR computers

comp.sys.sinclair Sinclair ZX80/ZX81/ZXSpectrum/QL

comp.sys.apple2 Apple][

etc.

鄭重呼籲,先看看這些 FAQ 在你想在這些新聞討論區發問之前.

Alt.folklore.computers rec.games.video.classic

FTP

ftp://x2ftp.oulu.fi/ ---- 包含遊戲平臺及遊戲程式設計,

位置在 Oulu , Finland

ftp: // ftp.spies.com/ ----- 大型電玩硬體資料庫

ftp: // ftp.komkon.org/pub/EMUL8/ ----- 電腦歷史及模擬器資料庫,

位置在 KOMKON

WWW

http://www.why.net/home/adam/cem/ comp.emulators.misc FAQ

http://www.komkon.org/fms/ My Homepage

http://valhalla.ph.tn.tudelft.nl/emul8/arcade.html

Arcade Emulation Programming Repository

http://www.classicgaming.com/EPR/

Emulation Programmer's Resource

我該如何模擬一個 CPU?

首先,如果你只需要模擬一個標準的 Z80 或 6502 CPU,你可以用我寫的 CPU

模擬器! http:// www.komkon.org/fms/EMUL8/

請先確定一下使用條件,在加入它們之前.

對於那些想要寫自己的 CPU 模擬核心或是對於 CPU 模擬核心感興趣的人,以下我提供一個典型 C 語言 CPU 模擬器.在真實的模擬器之中,你也許想要保留它的一部份,或是加入一部份到你自己的 CPU 模擬器.

```
Counter=InterruptPeriod;
PC=InitialPC;

for(;
{
OpCode=Memory[PC++];
Counter-=Cycles[OpCode];

switch(OpCode)
{
case OpCode1:
case OpCode2:
...
}

if(Counter<=0)
{
/*  檢查所發生的中斷並執行這些中斷  */
/*  週期行程(Cyclic tasks)置於這裡  */
```

```
...
Counter+=InterruptPeriod;
if(ExitRequired) break;
}
}
```

首先,我們要給定一個初值到 CPU 週期計數器"Counter",及程式計數器"PC"!

```
Counter=InterruptPeriod;
PC=InitialPC;
```

"Counter"包含 CPU 週期數到下一個可能的中斷.注意那些當"Counter"到期時,不需要發生的中斷:你可以使用在很多其他用途上,例如將計時器同步化,或者更新螢幕上的掃描線."PC"包含我們所模擬的 CPU 將讀入的運算碼記憶體位址.

在初值給定後,我們就可以開始主要的迴圈:

```
for(;
{
```

也可以用這樣的方式,

```
while(CPUIsRunning)
{
```

"CPUIsRunning"是一個布林變數.這有相當的優點,你可以隨時藉由設定

"CPUIsRunning"為"0"終止這個迴圈.不幸的,每次迴圈經過都必須檢查這個變數,這將花費很多 CPU 時間,應該盡量避免這樣的方式.也不要做這樣的迴圈:

```
while(1)
{
```

因為有些編譯器會產生檢查"1"是不是表示布林"true"的程式碼.你一定不希望編譯器產生這種每經過迴圈一次都要做一次不必要工作的程式碼.

現在,我們進到迴圈了,第一件事就是先讀入下一個運算碼,同時更改程式計數器"PC".

```
OpCode=Memory [ PC++ ] ;
```

注意這裡,雖然這是從被模擬記憶體中讀入指令的最簡單及最快的方式,不過這不一定可行就是.更通用的存取記憶體的方式稍後會介紹.

在抓取完運算碼之後,我們減少 CPU 週期數"Counter",減少的數量是這個運算碼所需要的週期數.

```
Counter-=Cycles [ OpCode ] ;
```

"Cycles [ ]"表格包含了每個運算碼的 CPU 週期數.提防某些運算碼可能因為參數的不同而有不同的週期數,例如條件跳躍或是副程式呼叫.雖然在程式中稍後能夠再?#123;整.

現在到了解譯運算碼及執行運算碼的時候了:

```
switch(OpCode)
{
```

一般人有個誤解,以為用 switch()結構是沒有效率的,因為這會編譯成一串 if() ... else if() ...的敘述.當 CASE 很少時,這倒是真的.不過在大型結構中(超過 100-200 個以上的 CASE)會編譯成一個"JUMP TABLE",這反而十分有效率.

有兩種選擇方式來解譯運算碼.第一種方式是作一個函式表(FUNCTION TABLE)並且呼叫適當的函數.這種方式比用 SWITCH() 還沒效率,因為函式呼叫花費更大.第二種方式是作一個標籤表(LABEL TABLE)並且利用"GOTO"敘述.這種方式比用 SWITCH() 有效率一點,不過只在編譯器支援"預先計算標籤(precomputed labels)這項功能才能執行.有的編譯器不允許你做一個標籤地址陣列(array of label addresses).

在我們成功的解譯並執行運算碼之後,再來是檢查需不需要任何岔斷.在這個時刻,你也可以完成任何需要與系統時鐘同步的行程.

```
if(Counter<=0)
{
/*  這裡執行檢查岔斷及完成其他硬體模擬的步驟  */
...
Counter+=InterruptPeriod;
```

```
if(ExitRequired) break;

}
```

這些週期行程(CYCLIC TASKS)稍後會說明.

注意這裡,我們並不是簡單的使用" Counter=InterruptPeriod; ",而是使用

" Counter+=InterruptPeriod ":這將使得週期計算更為精密,因為在"Counter"

中可能負的週期數出現.

也注意這裡:

```
if ( ExitRequired ) break ;
```

因為每經過一次迴圈就檢查一次是否離開程式,這樣滿浪費的,我們只在

"Counter"到期時做這項工作:這樣還是會離開模擬器,當你設定

"ExitRequired=1"時,不過這不會花費太多 CPU 時間.

我該如何處理被模擬記憶體(emulated memory)的存取?

存取被模擬記憶體最簡單的方式是如同一塊平坦的 byte(words 或其他)陣列,

瑣碎的處理它,如:

```
Data=Memory[Address1]; /*  從  Address1  讀入  */

Memory[Address2]=Data; /*  寫入到  Address2 */
```

如此簡單的記憶體存取實際上並不總是可能的,原因如下:

分頁記憶體(Paged Memory)

定址空間可能是零散的在於可切換頁(switchable pages,或稱 BANK).這常被做為擴充記憶體的方式,當定址空間是小的(64KB).

對映記憶體(Mirrored Memory)

一塊記憶體區域可在幾種不同的位址存取.例如,你寫入到位置 $4000 將也出現在 $6000 及 $8000.ROM 也可能被對應到不完全的位址解碼.

ROM 保護(ROM Protection)

某些卡匣型的軟體(如 MSX 遊戲)會嘗試寫入到自己的 ROM,並且在寫入成功時拒絕再繼續執行.通常這是為了防拷.為了使這樣的軟體在你的模擬器上繼續執行,你應該阻止寫入到 ROM 中.

Memory-Mapped I/O

在這種系統中,可能有 Memory-Mapped I/O 裝置.存取到這樣的記憶體位置會產生"特殊效果",因此這樣的存取方式應該要追蹤.

為了應付這種問題,我們介紹兩種對應的函式:

```
Data=ReadMemory(Address1); /* 從 Address1 讀入 */
WriteMemory(Address2,Data); /* 寫入到 Address2 */
```

所有特殊處理,如頁存取,對映,I/O 處理等等,都可以由這些函式處理.

ReadMemory() 及 WriteMemory() 兩種函式在模擬過程中佔用很大一部份,因為他們很常被呼叫.所以,他們一定要盡可能的做得有效率.這裡是用以存取分頁定址空間的函式範例:

```
static inline byte ReadMemory(register word Address)
{
return(MemoryPage[Address>>13][Address&0x1FFF]);
}
static inline void WriteMemory(register word Address,register byte Value)
{
MemoryPage[Address>>13][Address&0x1FFF]=Value;
}
```

注意"inline"這個關鍵字.他告訴編譯器將函式直接展開到程式碼,而不是以函式呼叫的方式.如果你所用的編譯器不支援"inline"或"_inline",嘗試利用"static function"來定義這樣的函式:某些編譯器(如 WATCOM C),會以 INLINE 的方式最佳化短的"static function".

在大部分的情況下,ReadMemory() 函式比 WriteMemory() 函式使用的頻率要高出許多倍,你也應該謹記在心.所以,把 ReadMemory() 函式程式碼盡可能寫得的比 WriteMemory() 函式短,這樣是很值得的.

在記憶體對映(memory mirroring)上還有一個小小的重點:

有此一說,許多電腦都有對映 RAM(mirrored RAM),你可以發現某個數值寫入到某個位址卻也將會出現在其他位址.你可以在 ReadMemory()函式中處理這樣的狀況,但這實在不太值得,因為 ReadMemory()函式呼叫的頻率實在大過 WriteMemory() 函式(譯按:原意應該是"因為有很多位址都能讀到同一個數字,所以只要讀取一個位址就可以,不要讀取好幾個位址才決定這個數字, ReadMemory() 函式使用的頻率太多,多加入不必要的程式碼將嚴重損及效率")更有效率的方式將是撰寫 WriteMemory() 函式時加入記憶體對映的功能.

週期行程(Cyclic tasks):這是什麼?

週期行程是在模擬機器中應該要定期處理的事情,例如:

螢幕更新(Screen refresh)

VBlank 及 HBlank 岔斷

更新計時器(Updating timers)

更新音效參數(Updating sound parameters)

更新鍵盤/搖桿狀態(Updating keyboard/joysticks state)

其他

為了模擬這些行程,你應執行在 2.5 MHz 的?50Hz 的更新頻率(PAL 影像盲邦?, VBlank 岔斷應該以這樣社 W 率發生

2500000/50 = 50000 CPU cycles

現在,如果我們假設螢幕是 256 條掃描線,但是只有 212 條掃描線是真的顯示在螢幕上的(i.e. 剩下 44 條被 VBlink 吃了),我們得到你的模擬器應當花費在更新每條掃描線是

該以適當的 CPU 週期數固定它們.例如,如果 CPU 建議

速度,螢幕顯示使朮?50000/256 ~= 195 CPU cyles

在這之後,你就應該產生一次 VBlink 岔斷,並且直到 VBlink 完成前,不做

任何事.(i.e.

(256-212) *50000/256 = 44*50000/256 ~= 8594 CPU cycles

小心的計算每個行程所需的 CPU 週期數,然後把他們的最大公約數定成

"InterruptPeriod(中斷週期)"並且固定所有其他行程到"InterruptPeriod"

(在每次"Counter"到期時,這些行程不需要執行).

我該如何最佳化 C 程式碼?

首先,正確的選擇編譯器所提供的最佳化選項能增加額外的程式碼效能.在我

的經驗中,以下的選項組合可以給你最佳的執行速度:

Watcom C++ -oneatx -zp4 -5r -fp3

GNU C++ -O3 -fomit-frame-pointer

Borland C++

如果你發現有其他更好的選項組合,不管是以上的編譯器或是其他不同的編

譯器,請你告訴我.

一些迴圈展開(loop unrolling)的小小建議:

將編譯器最佳化的"迴圈展開"選項設成"on"可能是很有用的.這項選項將嘗試轉換短迴圈變成平滑的程式片段.我的經驗顯示,雖然,這項選項不會產生任何效能改進.但是設定成"ON"時也可在某些極特殊的條件中中斷你的程式碼.

最佳化 C 的原始程式碼比選擇編譯器選項稍微來得好,並且就一般而言,用編譯器編譯程式碼這是一個與 CPU 有關的最佳化方式.幾個一般性適用於所有 CPU 規則如下.不要把它們的運作方式當做絕對的事實,因為你的用途可能不同:

使用程式分析工具(profiler)!

你的程式在程式分析工具程式(記住 GPROF)下執行的情形會顯示很多很好的事情,是以前你從未懷疑的.你會發現,看起來微不足道的程式片段執行的頻率比其他部份多的太多了,並且使整個程式慢下來.將這些程式片段最佳化或是改以組合語言撰寫會大幅增加效能.

避免使用 C++

避免使用任何結構,這會強迫使用 C++編譯器代替 C 編譯器去編譯你的程式:C++編譯器通常會產生額外的程式碼.

整數的尺寸(Size of integer)

嘗試只使用一種 CPU 所支援的基底整數尺寸,i.e. 整數(int)尺寸相對於"short" 或"long"整數的尺寸.如果使用混合的整數尺寸,這將使編譯器產生一堆不同整數長度間轉換的程式碼.這也會增加計憶體存取時間,因為某些 CPU 當以基底整數讀寫資料時會因為對齊基本基底位址邊界而使執行速度加快.

暫存器定位(Register allocation)

在每一區中,盡可能的減少使用變數,把最常使用的變數宣告為暫存器變數"

register variable"(雖然大部分新的編譯器可以自動地把變數變成暫存器變

數).這在擁有較多通用暫存器的 CPU(PowerPC)比在只有少數專用暫存器的 CPU

(Intel 80x86)更有意義.

展開小的迴圈(Unroll small loops)

如果你想要使小迴圈在很短的時間內執行,以手動的方式展開小迴圈成為平

滑的程式片段會是個好主意.看看上面有關自動迴圈展開的討論.

位元移位(Shifts) VS. 乘/除法

當你需要乘或除以 2 的次方數時,應該以位元移位代替(J/128==J>>7).

在大部分 CPU 下,這些方式會執行得較快.同樣的,使用位元"AND"運算代替取餘

數的運算(J%128==J&0x7F).

什麼是 LOW/HIGH-ENDIANESS?

所有的 CPU 一般都可分成兩種,這與他們如何在記憶體儲存資料有關.

High-endian CPU

這種 CPU 是把一個 word 的 higher byte 放在前面的儲存方式儲存資料.例如,如

果要儲存 0x12345678 到這樣的 CPU,這塊記憶體看起來會像這樣:

0 1 2 3

+--+--+--+--+

|12|34|56|78|

+--+--+--+--+

Low-endian CPU

這種 CPU 是把一個 word 的 lower byte 放在前面的儲存方式儲存資料.跟前面一樣的例子在這種 CPU 會看起來非常不一樣:

0 1 2 3

+--+--+--+--+

|78|56|34|12|

+--+--+--+--+

典型的 High-endian CPU 的例子是 6502 and 65816 ,Motorola 680x0 系列, PowerPC ,及 Sun SPARC .Low-endian CPU 包含 Zilog Z80,大部分 INTEL 晶片(包含 8080 及 80x86),DEC Alpha,等等.

當你寫一個模擬器,你必須注意被模擬及模擬別人的 CPU 是何種 ENDIANESS,我們說,你想要模擬 Z80 CPU,這是一個 Low-endian CPU,也就是儲存 16-BIT WORD 時 LOWER BYTE 在前面.如果你使用的是 Low-endian CPU(例如,INTEL 80x86),在這個例子,每件事都發生的很正常.如果你使用的是 High-endian CPU(PowerPC), 突然就出現了放置 16-bit Z80 資料到記憶體的問題.如果你的程式必須在兩種機器下執行,你需要某種方式感覺 endianess.

一種處理 endianess 問題的方法如下:

```
typedef union

{

short W; /* Word 存取 */

struct /* Byte 存取... */

{

#ifdef LOW_ENDIAN

byte l,h; /* ...在 low-endian 機器 */

#else

byte h,l; /* ...在 high-endian 機器 */

#endif

} B;

} word;
```

正如你所看到的,一個"WORD"的資料利用 W 可以整個存取.每次你的模擬程式需要分開存取 BYTE 時,你可以使用 B.l 及 B.h 存取資料.如果你的程式將在不同平臺編譯時,在執行任何真正重要的事情時,你可能先想要測試一下它是否以正確的 endianess 方式編譯的.

這裡是這種測試的一種方式:

```
int *T;



T=(int *)"\01\0\0\0\0\0\0\0\0\0\0\0\0\0";

if(*T==1) printf("This machine is high-endian.\n");

else printf("This machine is low-endian.\n");
```