The TOY86 Machine and Its Assembly Language

柯向上・邱亮德

目錄

1 引言

本文描述我們擴充 cyyTOY 組合語言，進而實作 TOY86 的過程與成果， 括奠基於 classical TOY 的 ExtTOY 組合語言、TOY86 架構、和 TOY86 組合語言。TOY86 以 classical TOY 為基礎加以擴充，主要吸收 x86 架構的常用指令。TOY86 的設計理念有三：

- **控制規模**：我們希望 TOY86 能取代 TOY 在教學上的地位，因此規模不能太過龐大。
- **收納精華**：在前項原則的限制下，能加進 TOY86 的功能必然有限，因此設計時必須選擇最精要的指令。
- **保留美感**：Classical TOY 的設計單純而調和，相當適合用於教學。TOY86 假使在美感上不及 TOY，那麼即使功能較強，也無法取而代之。

雖然「讓 TOY86 取代 TOY 在教學上的地位」這樣的目標相當困難，成功率不高，但以此目標作為 TOY86 設計上的指引，應能導出不錯的成果。

2 Extended TOY 組合語言

開始設計 TOY86 之前，我們首先為 classical TOY 設計一個功能較強的組合語言，以收投石問路之效。我們稱之為 Extended TOY 組合語言，簡稱 ExtTOY。ExtTOY 組譯的目標平台仍是 classical TOY，但 ExtTOY（在組合語言層級）可用較少指令完成 classical TOY assembly（後稱 cyyTOY）中須以繁瑣指令完成的動作。

ExtTOY 基本上相容於 cyyTOY，除了下述三點：

- Labels 必須明確冠以 `export` 關鍵字，方能由其他組譯單元指涉。
- 除了 `lda` 以外，所有 format 2 的指令都必須以 label 指涉位址。若要指涉位址 `0xFF`，可使用預先定義好的 `stdio` 標籤（e.g. `ld RA, stdio`）。
- `RF` register 保留供內部使用。

ExtTOY 主要加入 stack directives 及 procedure directives，並允許某些 operations 使用特殊的 operands。這些功能主要是為了方便撰寫 procedure 而設計。

2.1 Stack Directives

Stack directives　括 push 和 pop，各有兩種形式。push 可將 register operand 或 memory operand 推入 stack，而 pop 可將 stack 頂端元素彈出到某個 register，或單純彈出。ExtTOY 的 stack 是以 RE 為 stack pointer，指向**下一個放置元素的位置**。程式啟動時，RE 會被初始化為 0xFE，由高位址往低位址成長。push 和 pop 等價於下列的 cyyTOY 程式碼：

push RD	push [RD + c]	pop RD	pop

lda RF, 1
sti RD, RE
sub RE, RE, RF
	lda RF, c
	add RF, RD, RF
	ldi RF, RF
	sti RD, RE
	lda RF, 1
	sub RE, RE, RF
		ldi RD, RE
		lda RF, 1
		add RE, RE, RF
			lda RF, 1
			add RE, RE, RF

2.2 Procedure Directives

ExtTOY procedures 以 proc_name PROC 指令起頭，以 ENDP 指令結尾，不能嵌套（nested）出現。「呼叫 procedure」和「從 procedure 回返」所使用的指令分別是 call proc_name 和 ret。Procedure 表頭的 PROC 之後可加上 cdecl 或 fastcall，指定 stack frame 的建構方式。若未指明，預設使用 cdecl calling convention。cdecl 的舉止完全模仿 x86 上的 cdecl：引數經由 stack 傳遞，進入 procedure 後，首先推入 return address，接著推入當前的 RB，並使 RB 指向這個值；自 procedure 回返時，先讓 RE 指向 RB 所指位置，並將前一個 RB 的值重新存回 RB，接著彈出 return address，最後將控制權交還給呼叫端，由呼叫端清理引數。轉譯為等價的 cyyTOY：

call p	PROC cdecl	ret

jl RF, p	sti RF, RE
	lda RF, 1
	sub RE, RE, RF
	sti RB, RE
	add RB, RE, R0
	sub RE, RE, RF
		add RE, RB, R0
		ldi RB, RE
		add RE, RE, RF
		ldi RF, RE
		jr RF

由上可見，對於 TOY 而言，cdecl 是滿昂貴的 calling convention。又 TOY 的 register 個數較 x86 為多，因此 ExtTOY 另外提供 fastcall calling convention 減輕呼叫 procedure 帶來的負擔。進入 fastcall procedure 時，不建立複雜的 stack frame，只將 return address 推入 stack，引數經由 registers 傳遞。自 procedure 回返時，直接將 stack 頂端的 return address 彈出，將控制權交還給呼叫端。call 指令不需改變。

PROC fastcall	ret
sti RF, RE	lda RF, 1
lda RF, 1	add RE, RE, RF
sub RE, RE, RF	ldi RF, RE
	jr RF

Remark：此處我們使用 “calling convention” 這個詞，但其實並未指明 calling convention 的所有細節，例如 registers 的 volatility、`cdecl` 的引數傳遞順序、`fastcall` 的引數擺放規則等等。這可以簡單制定，但不是重點，因此我們不詳細描述。另外，“calling convention” 的定義層級應該在組合語言之上，而不是從組合語言支援，這些許反映 ExtTOY 的「高階組合語言」性質。

2.3 Special Operands

為了讓 procedure 相關操作較為方便，ExtTOY 在適當場合提供 immediate operands、offset operands、以及 memory operands。

2.3.1 Immediate Operands

操作 stack pointer 時，常需要把 stack pointer 加上或減去某個常數，以騰出 local variables 的空間或清理引數。因此 ExtTOY 的 ALU operations “`aluop RD, RS, RT`” 中，`RS` 和 `RT` 其中一個可以是常數。例如，`sub RE, RE, 5` 轉譯為 cyyTOY 會產生兩個指令：

```
lda RF, 5
sub RE, RE, RF
```

2.3.2 Offset Operands

Offset operands 是為了簡化 local variables 和 parameters 的存取而被引入 ExtTOY。Offset operands 只能用於 `ldi` 和 `sti` 的第二引數，型如 `ldi R1, RB + 2`，轉譯為 cyyTOY：

```
lda RF, 2
add RF, RB, RF
ldi R1, RF
```

2.3.3 Memory Operands

Memory operands 是 offset operands 的進一步簡化，型如 `[RB - 3]`，可使指令在表面上直接存取 memory。Memory operands 可用於 `bz`、`bp`、`push`、以及 ALU operations “`aluop RD, RS, RT`” 中 `RS` 或 `RT` 之一。以 `bz [RB - 3], zero` 為例轉譯為 cyyTOY：

```
lda RF, 3
sub RF, RB, RF
ldi RF, RF
bz  RF, zero
```

使用 memory operands 的成本已經算滿昂貴。

2.4 其他規則

ExtTOY 將 labels 區分為 data、instruction、procedure 三種型別，需指涉 label 的 operations 都只能使用特定型別的 label，表列如下：

`ld`	data
`st`	data
`bz`	instruction
`bp`	instruction
`jl`	instruction
`call`	procedure

如本節最前面所提，label（ 括 instruction label、procedure label、和 data label）的 scope 預設為 local，需冠以 `export` 方能被其他組譯單元存取。然而，procedure 內的 labels（只可能是 instruction labels）不能被匯出（`exported`），也不能從 procedure 之內以 `bz`、`bp`、`jl` 跳躍至 procedure 外的 label，並禁止使用 `jr`。

Procedures 只能經由 `call proc_name` 進入。當一般指令（即不在 procedure 內的指令）與 procedures 混合時，procedures 會被提出放到程式最後，不與一般指令同處一地。若 procedure 未含 `ret` 指令，程式會在執行完 procedure 的最後一個指令後終止。多個組譯單元連結時，相鄰兩單元的一般指令不會接鄰在一起，因此執行到一個組譯單元的最後一個指令時將終止程式，不會接著執行下一個組譯單元的指令。

2.5 示例：河內塔

我們舉經典的遞迴河內塔作為 ExtTOY 的示例。`hanoi` procedure 的 calling convention 為 `cdecl`，接收四個引數，分別為盤子數、起點柱子編號、中間柱子編號、終點柱子編號。

```
hanoi PROC
; parameters
; [RB + 2]   number of disks
; [RB + 3]   source peg
; [RB + 4]   intermediate peg
; [RB + 5]   destination peg

        ; base case
        bz   [RB + 2], end

        ; first recursive call
        push [RB + 4]
        push [RB + 5]
        push [RB + 3]
```

```
        ldi  RD, RB + 2
        sub  RD, RD, 1
        push RD
        call hanoi
        sub  RE, RB, 1

        ; output
        ldi  RD, RB + 3
        shl  RD, RD, 4
        add  RD, RD, [RB + 5]
        st   RD, stdio

        ; second recursive call
        push [RB + 5]
        push [RB + 3]
        push [RB + 4]
        ldi  RD, RB + 2
        sub  RD, RD, 1
        push RD
        call hanoi

end     ret
ENDP

lda  RD, 2
push RD
lda  RD, 1
push RD
push R0
ld   RD, stdio
push RD
call hanoi
```

組譯出來的結果是：

```
10: 7EFE    21: 0000    32: 7F05    43: BD0E    54: 1FBF    65: 7F02
11: 7D02    22: BF0E    33: 1FBF    44: 7F01    55: AF0F    66: 1FBF
12: BD0E    23: 7F01    34: AF0F    45: 2EEF    56: BF0E    67: AD0F
13: 7F01    24: 2EEF    35: BF0E    46: FF22    57: 7F01    68: 7F01
14: 2EEF    25: BB0E    36: 7F01    47: 7F01    58: 2EEF    69: 2DDF
15: 7D01    26: 1BE0    37: 2EEF    48: 2EBF    59: 7F03    6A: BD0E
16: BD0E    27: 2EEF    38: 7F03    49: 7F03    5A: 1FBF    6B: 7F01
17: 7F01    28: 7F02    39: 1FBF    4A: 1FBF    5B: AF0F    6C: 2EEF
18: 2EEF    29: 1FBF    3A: AF0F    4B: AD0F    5C: BF0E    6D: FF22
19: B00E    2A: AF0F    3B: BF0E    4C: 7F04    5D: 7F01    6E: 1EB0
1A: 7F01    2B: CF6E    3C: 7F01    4D: 5DDF    5E: 2EEF    6F: AB0E
1B: 2EEF    2C: 7F04    3D: 2EEF    4E: 7F05    5F: 7F04    70: 7F01
1C: 8DFF    2D: 1FBF    3E: 7F02    4F: 1FBF    60: 1FBF    71: 1EEF
1D: BD0E    2E: AF0F    3F: 1FBF    50: AF0F    61: AF0F    72: AF0E
1E: 7F01    2F: BF0E    40: AD0F    51: 1DDF    62: BF0E    73: EF00
1F: 2EEF    30: 7F01    41: 7F01    52: 9DFF    63: 7F01    74: 0000
20: FF22    31: 2EEF    42: 2DDF    53: 7F05    64: 2EEF
```

2.6 ExtTOY 的缺憾

ExtTOY 最大的缺點就是（擴充的）組合語言指令並未適當對應至底層的機器碼，例如 `push [RB + 2]` 這樣一個看似單純的指令，實際上產出的機器碼竟然需要 6 個 instructions，這使得產出的機器碼長度和組合語言程式的長度不甚相稱，無法在組合語言層級反映出「instruction count 大幅上升」的事實，對於 assembly programmer 有點欺騙意味。這個問題在河內塔示例中清楚顯現：組合語言層級看似有效的指令（effective instructions）僅約 30 個，但產出的機器碼長度卻達到 100 個指令。不過 stack 和 procedure operations 又是相當重要的操作，的確應該提供，因此最佳的解決方案是把這些功能設計到硬體架構上，直接從硬體層級支援。這就是設計 TOY86 的動機。

另外，ExtTOY 除了受限於 TOY 的計算能力，在語言設計上也過於拘謹，而抹煞了一些重要功能，例如 pointer to function。TOY86 assembly 設計時將盡量使所有合理行為都能不費力地完成（當然以不妨礙上段所提的相稱感為前提），其中「能實作 C/C++ 所支援的特性」[1]會是主要的參考因素。

3 TOY86 指令集架構

設計 TOY86 的第一步是確定指令集架構（instruction set architecture, ISA），或可類比於軟體工程上的需求分析（requirement analysis）。既然決定擴充架構，不如趁機解決 classical TOY 指令稀少而使用不易的麻煩，適度適量引進 x86 一些常用的指令，並擴充 TOY 的記憶體。

3.1 Registers & Memory

TOY86 演化自 classical TOY，設計時有一些血統上的考量。TOY machine code 的特點之一是相當好寫，以 4 bits 為區塊，每個區塊恰好是一個 hex digit，TOY86 希望能延續這個傳統。這馬上引出一個結論：register 的數量和 memory address space 的大小必須是 2^{2k}，最好是 2^{4k}，否則無法簡單以十六進位編碼。TOY 的 registers 有 $16 = 2^4$ 個，對於 TOY86 仍是合理的數量，只需將每個 register 的長度擴充為 32-bit 即可。不過，TOY86 必須有兩個 registers 對應 x86 的 `esp` 和 `ebp` 以便支援 procedure 操作，因此我們將 TOY 的 `RE` 名稱改為 `SP`（stack pointer）、`RF` 改為 `BP`（base pointer），`R1` 到 `RD` 仍為 general-purpose registers，`R0` 也保留「恆為 0」的特性。

---

[1] 我們曾考慮是否要為 TOY86 實作一部簡單的 C compiler（只選擇 C 容易實作的子集），但一來時間緊迫，二來這主要是 compiler 方面的技術，和主題相距太遠，因而作罷。

接著是記憶體的擴充。TOY 採用 flat memory model，沿用至 TOY86 很理想。除了直覺地擴充 address space 之外，我們也想讓記憶體的存取粒度較為細緻（more fine-grained access），即讓可定址單位降為 1 byte，並能以 1-byte、2-byte、或 4-byte 為單位操作記憶體，這在 architecture level 可藉由「為 memory file 打造 4 個 1-byte read ports」而達成。這樣的擴充便產生 byte ordering 問題，我們決定讓 TOY86 採用 little-endian 次序。

至於 address space 要擴充到多大，考慮 TOY 的 memory file 實作和 register file 相同，如果任意擴充到如 32-bit，從 circuit level 來看不甚實際。又因為「方便以十六進位編碼」的考量，我們將 address 設定為 12-bit。連同先前的「可定址單位為 1 byte」，TOY86 的記憶體共有 1 byte $\times$ 2^{12} = 4096 bytes = 4 KB，相較於 TOY 的 2 bytes $\times$ 2^8 = 512 bytes，規模的成長還算可以接受。Standard I/O 也得隨著擴充修正。很直覺地，我們用最後 4 個 bytes 代表 stdio，也就是 address `0xFFC` 到 `0xFFF` 的 4 個 bytes。

最後一點，TOY 的 register file 和 memory file 在硬體上沒有什麼分別，因此 TOY 指令設計上（有意或無意地）限制了 memory 的用途，所有運算一律在 registers 中進行。ExtTOY 支援 memory operands，但如果 TOY86 從硬體層級支援 memory operands，registers 存在的意義將大幅下降，因為 TOY86 的 memory file 就是容量特大的 register file。因此設計指令時，我們仍將限制 memory 的用途，令其僅能和 registers 交換資料。

3.2 指令集

3.2.1 Arithmetic/Logic Operations

Arithmetic/logic operations 仍維持 TOY 的 three-register 形式，也允許第二運算元使用 8-bit 常數。同時，也可指定實施 1-byte、2-byte、或 4-byte 的計算。因此每個操作會有六個版本。除了 TOY 原先的六個指令 `add`、`sub`、`and`、`xor`、`shl`、`shr` 以外，有些指令可考慮納入 TOY86 指令集，我們考慮的候選指令有 `or`、`not`、`neg`、`inc`、`dec`。這些指令中，最後獲選的僅 `or` 一個，因為其他四個指令都可輕易地以既有指令完成。

Candidate	**Substitute**
`not RX, RY`	`xor RX, RY, 0xFF` [2]
`neg RX, RY`	`sub RX, R0, RY`
`inc RX, RY`	`add RX, RY, 1`
`dec RX, RY`	`sub RX, RY, 1`

[2] 這只適用於 1-byte 操作。不過若要施行 2-byte 或 4-byte 操作，手續也不難，例如 `sub RX, R0, RY` 之後 `sub RX, RX, 1`。

3.2.2 Data Transfer Operations

我們希望提供較為強大的記憶體定址模式，目標是 x86 的 base-index-displacement addressing mode。若能支援這個定址模式，TOY 的 `ldi` 和 `sti` 兩個指令便能合併到 `ld` 和 `st` 裡面。另外 TOY86 也支援 `lea`，能載入 effective base-index-displacement address，這就將 `lda` 的功能吸納在內。因此 TOY86 的 data transfer operations 就是 `lea`、`ld`、`st` 三個。這裡有個麻煩的地方：考慮 `st` 指令的泛化型

```
st RX, addr + RY + RZ * c
```

這必須讀出三個 registers 的值，而 TOY(86) 的 register file 卻只有兩個 read ports。當然，可以增加第三個 read port 而支援這個指令，但我們覺得為了單單一個指令而為 register file 增加一組 read port 不太值得，因此我們把 `st` 指令的 base 部份，也就是上式的 `RY` 拿掉。這使得 `st` 指令成了異類，不過我們認為這個 trade-off 可以忍受。

3.2.3 Jump Operations

TOY 的 conditional jump 指令僅 `bp`、`bz` 兩個，使用上較不方便。因此我們決定將 TOY 的 Cond Eval 單元換成一個 Flag Register（`FR`），儲存 zero、carry、overflow、signed 四個 flags。執行任何一個 arithmetic/logic operation 都（可能）會更改 `FR` 的狀態。

Flag	**Set Condition**	**Hardware Implementation**
zero	計算結果為零。	對 ALU 輸出的所有 bits 施行 NOR。
carry	無號值超出範圍。	取加法器 MSB carry bit 和 subtraction signal 的 XOR 結果。加減法以外的操作一律清除 carry flag。
overflow	有號值超出範圍 — 括兩正數相加結果過大或兩負數相加結果過小。	取加法器 MSB carry bit 和次一位 carry bit 的 XOR 結果。加減法以外的操作一律清除 overflow flag。
signed	有號值為負。	複製 MSB。

以這四個 flags 為基礎，TOY86 支援八個 conditional jump operations，命名慣例和 x86 相同：`j[n]`f，其中 f 為 `z`、`c`、`o`、`s` 四個字元之一。另外也支援 `jmp` 指令執行 unconditional jump。在 TOY86 上不需要 `test` 或 `cmp` 等指令，因為 TOY86 承襲 TOY，register `R0` 恆為 0，因此若要轉譯以下這段 C/C++ code：

```
if (a < b) {
  // less
}
else {
  // greater or equal
}
// other code...
```

假設 `a` 值存放在 register `RA`、`b` 值存放在 register `RB`，則上段程式在 TOY86 assembly 大致上是：

```
            sub R0, RA, RB
            jns else_part   ; if RA - RB >= 0 then jump to else_part
            ; less
            jmp post_if
  else_part ; greater or equal
  post_if   ; other code...
```

x86 的 arithmetic/logic operations 都有連帶效果（side effects），所以需要 `test`、`cmp` 等無連帶效果的指令。而 TOY(86)「`R0` 恆為 0」的設計優雅地省去對此類指令的需求。

最後整理一張比較方法表，假設兩個要比較的數存於 `RA` 和 `RB`，符合條件時跳到 label `condition_met`。

Compare...	Signed Integers	Unsigned Integers
`RA == RB`	`sub R0, RA, RB` `jz  condition_met`	`sub R0, RA, RB` `jz  condition_met`
`RA != RB`	`sub R0, RA, RB` `jnz condition_met`	`sub R0, RA, RB` `jnz condition_met`
`RA < RB`	`sub R0, RA, RB` `js  condition_met`	`sub R0, RA, RB` `jc  condition_met`
`RA > RB`	`sub R0, RB, RA` `js  condition_met`	`sub R0, RB, RA` `jc  condition_met`
`RA <= RB`	`sub R0, RB, RA` `jns condition_met`	`sub R0, RB, RA` `jnc condition_met`
`RA >= RB`	`sub R0, RA, RB` `jns condition_met`	`sub R0, RA, RB` `jnc condition_met`

3.2.4 Stack & Procedure Operations

最後是 TOY86 的重頭戲，關於 stack 和 procedure 的操作。為了這兩類操作，TOY86 將 TOY 的 register `RE` 和 `RF` 分別更名為 `SP` 和 `BP`（編號不變），各對應 x86 的 `esp` 和 `ebp`。

首先考慮 stack operations，最基本的兩個操作是 `push RX` 和 `pop RX`。我們讓 `SP` 指向 ***stack* 最頂端的元素**，由高位址往低位址成長，和 x86 相同。因此 TOY86 開機時，`SP` 的初始值會是 `0xFFC`。`push RX` 指令可推入 1-byte、2-byte、或 4-byte 的 register，在 architecture level 可算出 `SP - c` 的值，然後將此值導到 memory write address（以將 `RX` 寫入`[SP - c]`）和 register write data（以將 `SP - c` 存入 `SP`），沒有問題。

但 `pop RX` 就有麻煩了：TOY86 必須同時把 stack 頂端的值寫入 `RX` 並將更新後的 stack pointer 寫入 `SP`！所以情況和 `st` 指令類似，要嘛多開一個 register file write port，要嘛模仿 C++ STL 讓 `pop` 單純更新 `SP`，讀值另外用 `ld RX, SP + c` 解決，避開多重寫入的情況。然而若如上述縮減 `pop` 的功能，那麼 `pop` 就能直接以 `add` 指令取代，除了字面意義較直觀以外毫無價值。又，如果加入第二個 register file write port，那麼就能支援更複雜的 procedure operations，i.e. `enter` 和 `leave`。因此我們選擇為 register file 加上一個 write port，從而支援 `pop RX`（以及稍後的 `enter` 和 `leave`）。

另外，我們也提供 pushf 和 popf 兩個指令，能將 FR 推入 stack，佔位 1 byte。

至於 procedure operations，TOY86 提供 call addr、call RX、ret、enter 和 leave，語意都和 x86 相同。其中 call 將 address 推入 stack 時，雖然 address 為 3-byte，但為了讓參數和區域變數的存取比較好看，我們讓 address 佔位 4-byte。這些指令初步推想，在 architecture level 都能夠實作，沒有問題。舉最為複雜的 enter 為例：enter 將 BP 推入 stack，然後將 SP 的值賦予 BP。在 architecture level，可將 SP - 4 的值導到 memory write address（BP 的寫入位址）和 register write data（新的 SP 值），將 SP 的值導到另一個 register write data(新的 BP 值)，將 BP 的值導到 memory write data，最後等待 clock signal。

以下是指令集簡表。

Instruction Group	Pseudo-Assembly	Pseudo-Code	Description
Arithmetic/ Logic	add RX, RY, RZ	RX ← RY + RZ	支援 1-byte、2-byte、4-byte 操作。適當改變 FR 的狀態。RZ 的位置可使用 8-bit 常數。
	sub RX, RY, RZ	RX ← RY - RZ	
	and RX, RY, RZ	RX ← RY & RZ	
	or RX, RY, RZ	RX ← RY | RZ	
	xor RX, RY, RZ	RX ← RY ^ RZ	
	shl RX, RY, RZ	RX ← RY << RZ	
	shr RX, RY, RZ	RX ← RY >> RZ	
Data Transfer	lea RX, addr + RY + RZ * c	RX ← addr + RY + RZ * c	支援 1-byte、2-byte、4-byte 操作。
	ld RX, addr + RY + RZ * c	RX ← [addr + RY + RZ * c]	
	st RX, addr + RZ * c	[addr + RZ * c] ← RX	
Jump	jz addr	if zero PC ← addr	語意和 x86 相同。
	jnz addr	if not zero PC ← addr	
	jc addr	if carry PC ← addr	
	jnc addr	if not carry PC ← addr	
	jo addr	if overflow PC ← addr	
	jno addr	if not overflow PC ← addr	
	js addr	if signed PC ← addr	
	jns addr	if not signed PC ← addr	
	jmp addr	PC ← addr	

Instruction Group	Pseudo-Assembly	Pseudo-Code	Description
Stack	`push RX`	`SP ← SP - c` `[SP - c] ← RX`	可推入、彈出 1-byte、2-byte、或 4-byte 的值。語意和 x86 相同。
	`pop RX`	`RX ← [SP]` `SP ← SP + c`	
	`pushf`	`SP ← SP - c` `[SP - 1] ← FR`	
	`popf`	`FR ← [SP]` `SP ← SP + c`	

Procedure	`call addr`	`SP ← SP - 4`
		`[SP - 4] ← PC` [3]
		`PC ← addr`
			一律為 4-byte 操作。語意和 x86 相同。

	`call RX`	`SP ← SP - 4`
		`[SP - 4] ← PC`
		`PC ← RX`	

	`ret`	`PC ← [SP]` `SP ← SP + 4`	

	`enter`	`SP ← SP - 4`
		`[SP - 4] ← BP`
		`BP ← SP - 4`	

	`leave`	`SP ← BP + 4` `BP ← [BP]`	
Halt	`hlt`	`halt`	停止運轉。

4 TOY86 架構模型

本節我們以 classical TOY 的 architecture 為基礎[4]，首先設計 TOY86 的指令格式和 machine code，接著擴充 ALU、memory file、register file 並新造一個 flag register，最後建立 TOY86 的 datapath。

4.1 Instruction Formats & Machine Code

制定指令格式有多重考量和限制，正如 McKevit et al. 在 1977 年的 *8086 design report* 所說：“A custom format such as this is slave to the architecture of the hardware and the instruction set it serves. The format must strike a proper compromise between ROM size, ROM-output decoding, circuitry size, and

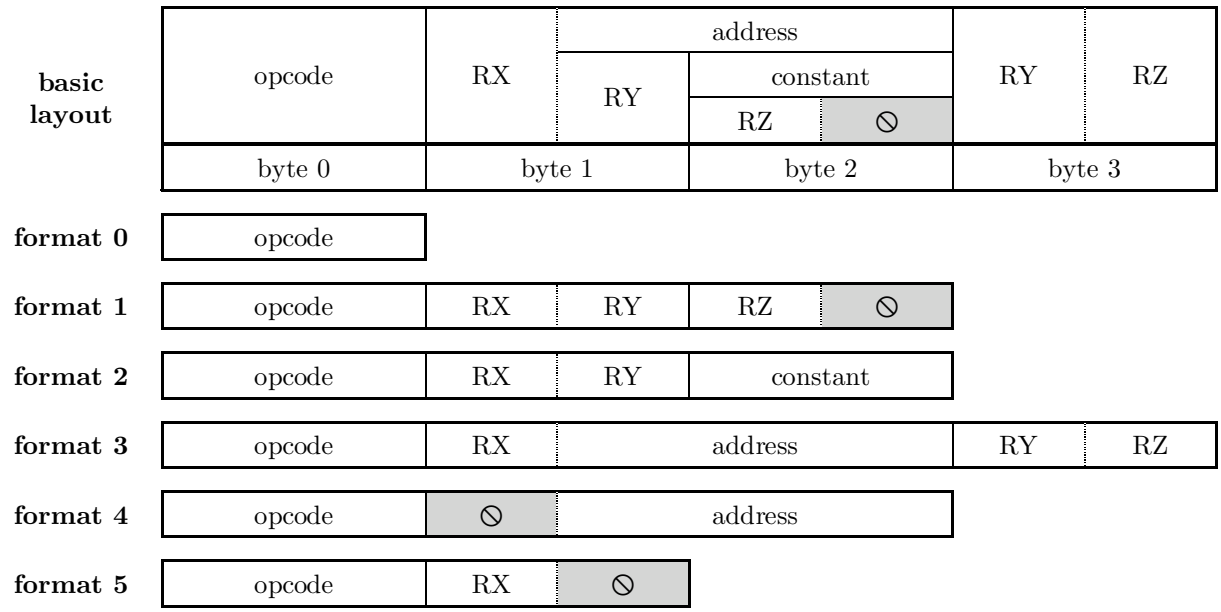
[3] 此行的 `PC` 其實是下一個指令的位址。TOY86 的 `PC` update 發生在 execute stage 最後，所以 `call` 指令推入 stack 的實際上是 `PC + 3` — 3 是 `call` 指令的長度。

[4] 嚴格說來（依照 *Wikipedia* 給的定義），應該是以 classical TOY 的 “***organization***” 為基礎，不過在本文中我們不打算嚴格區分這兩個詞。

machine execution rate." 幸運地，TOY86 是部想像中的機器（hypothetical machine），所以某些因素如執行效率基本上不造成太大影響。我們主要考量的因素有三：machine code 必須不太難寫、code size 不能太大、circuitry complexity 不能太嚇人。

- ***Machine code readability/writability***：這在 3.1 節已經有所顧慮，當時設計讓 register 編號和 address 能輕鬆以 hex digits 表示。這個因素進一步還影響指令格式和 opcode 的設計，這兩者必須遵循某種規律且容易推想。一個直接推論是：opcode 必須是 8-bit，理由和 3.1 節所述相同。
- ***Code size***：觀察 TOY86 指令集，最短可用 1 byte 表述，如 `enter`、`leave`；最長則需要到 4-byte，如 `ld RX, addr + RY + RZ * c`，若把 `c` 編碼到 opcode 裡，整個指令需要 8（`ld` & `c`）+ 4（`RX`）+ 12（`addr`）+ 4（`RY`）+ 4（`RZ`）= 32 bits = 4 bytes。因此我們決定讓指令長度可變，這在 architecture level 很簡單就能實作。另外，像 arithmetic/logic operations 的 three-register 形式，所需的 bit 數是 8 + 4 + 4 + 4 = 20 bits，並未恰好容納於整數個 bytes 之內，勢必會有一些 bubbles。這個因素也要求 bubbles 盡量減少。
- ***Circuitry complexity***：這要求指令格式必須盡量單一同調（uniform）、盡量復用元件、精簡 datapath。也因為這個因素，我們不打算實作 multicycle instructions 或 pipelined architecture — 這兩種實作都能增進執行效率，而剛好我們可以大致忽略執行效率這個因素。

以下是 TOY86 指令的 6 種 formats，奠基於 3 種 basic layout，依照指令需要而增減欄位。



這套指令格式迎合了前兩項要求：基本上很好記，長度可變，而且出現的 bubbles 都無法省略且縮到最小。至於第三項要求必須到 4.4 節方能評估，不過稍微觀察 basic layout，猜測應不至於使電路太過複雜。

接著我們依序描述各類指令的 machine code 與其格式。對於所有指令的 opcode，高端四個 bits 是主碼，代表指令，低端四個 bits 是副碼，代表變異版或參數。若 TOY86 遭遇無效的 opcode，會直接中止執行。為了行文方便，我們稱副碼的 lower half 是 LSB 端的兩個 bits，副碼的 upper half 是 MSB 端的兩個 bits。

4.1.1 Arithmetic/Logic Operations

此類指令各有六個變異版（variants）：可能為 1-byte、2-byte、4-byte 的操作，並有 three-register 和 "two-register, one-constant" 兩種類型。我們將操作的 byte 數編碼在副碼的 lower half：`00` 為 4-byte、`01` 為 1-byte、`10` 為 2-byte，即「byte 個數」二進位表法的最後兩位。副碼的 upper half 則代表類型：`00` 為 three-register、`01` 為 two-register, one-constant。

例：在 stack 中騰出 8 個 bytes 供區域變數使用，所用指令是 `sub SP, SP, 8`，轉譯為 machine code 是 `24EE08`。

Operation	Primary Opcode
`add`	`1`
`sub`	`2`
`and`	`3`
`or`	`4`
`xor`	`5`
`shl`	`6`
`shr`	`7`

Secondary Opcode	1-byte	2-byte	4-byte
three-register	`1`	`2`	`0`
two-register, one constant	`5`	`6`	`4`

Type	Format	Effect
three-register	1	RX ← RY *op* RZ
two-register, one-constant	2	RX ← RY *op* constant

4.1.2 Data Transfer Operations

遵循 arithmetic/logic operations 的先例，操作對象的 byte 數一樣放在副碼的 lower half。Index register 可以乘上一個常數 `c` = `1`, `2`, `4`，我們把關於 `c` 的資訊放在副碼的 upper half。乘以 1、2、4 在 architecture level 實作上就是左移 0、1、2 位，因此副碼 upper half 所編碼的其實是左移的位數，這可簡化 control unit 的設計。

例：`lea RA, 0x3D9 + R1 + R2 * 2`，轉譯為 machine code 是 `84A3D912`。

Operation	Primary Opcode	Format	Effect
`lea`	8	3	RX ← address + RY + RZ * c
`ld`	9	3	RX ← [address + RY + RZ * c]
`st`	A	3	[address + RZ * c] ← RX

Secondary Opcode	1-byte	2-byte	4-byte
$\boldsymbol{c = 1}$	1	2	0
$\boldsymbol{c = 2}$	5	6	4
$\boldsymbol{c = 4}$	9	A	8

4.1.3 Jump Operations

Jump operations 很單純，主碼均為 `B`，副碼代表跳躍種類。副碼若為偶數（LSB 為 0）則為肯定（positive）判斷，若為奇數（LSB 為 1）則為否定（negative）判斷。

Operation	Opcode	Format	Effect	
`jz`	B0	4	if zero	then PC ← address
`jnz`	B1	4	if not zero	then PC ← address
`jc`	B2	4	if carry	then PC ← address
`jnc`	B3	4	if not carry	then PC ← address
`jo`	B4	4	if overflow	then PC ← address
`jno`	B5	4	if not overflow	then PC ← address
`js`	B6	4	if signed	then PC ← address
`jns`	B7	4	if not signed	then PC ← address
`jmp`	B8	4	PC ← address	

4.1.4 Stack Operations

`push` 系列的主碼為 `C`，`pop` 系列的主碼為 `D`，副碼 lower half 指定施行的是 1-byte、2-byte、4-byte 操作，與 arithmetic/logic operations 和 data transfer operations 相同。副碼 upper half 為 `00` 代表推入、彈出一般 register，為 `01` 代表推入、彈出 `FR`。

例：將 `RA` 低位 2 byte 推入 stack 的指令是 `pushw RA`[5]，轉譯為 machine code 是 `C2A0`。

[5] 是的，TOY86 assembly 是從 mnemonic suffix 區分 operand size。無 suffix 是 4-byte，suffix `w` 是 2-byte，suffix `b` 是 1-byte。

Operation	Primary Opcode	Format	Effect
push	C	5	SP ← SP - c [SP - 4] ← RX
pop	D	5	RX ← [SP] SP ← SP + c
pushf	C	0	SP ← SP - c [SP - 4] ← FR
popf	D	0	RX ← [SP] FR ← SP + c

Secondary Opcode	1-byte	2-byte	4-byte
general-purpose registers	1	2	0
flag register	5	6	4

4.1.5 Procedure Operations

Procedure operations 也很單純。call 和 ret 共用主碼 E，副碼 0 和 1 分別是 call 的 address 和 register 形式，2 則是 ret；enter 和 leave 共用主碼 F，兩兩之間以副碼 0 和 1 區分。

Operation	Opcode	Format	Effect
call	E0	4	SP ← SP - 4 [SP - 4] ← PC + 3 PC ← address
call	E1	5	SP ← SP - 4 [SP - 4] ← PC + 3 PC ← RX
ret	E2	0	PC ← [SP] SP ← SP + 4
enter	F0	0	SP ← SP - 4 [SP - 4] ← BP BP ← SP - 4
leave	F1	0	SP ← BP + 4 BP ← [BP]

4.2 Arithmetic/Logic Unit & Flag Register

TOY86 ALU 奠基於 classical TOY ALU，除了原有的 addition (subtraction)、bitwise AND、bitwise XOR、shift left、shift right、copy input 2 以外，增加 bitwise OR 和 copy input 1 兩項功能。TOY ALU 控制訊號線有五條，其中三條是功能選擇、一條是加減法訊號、一條是 shift 方向訊號。TOY86 ALU 僅

加入一個新功能 bitwise OR，因此原本的三條功能選擇線即已足夠。不過 TOY86 ALU 必須支援 1-byte、2-byte、4-byte 的操作，因此我們直接加上三條訊號線，分別代表三種大小。這三條訊號線不影響輸出端，送出的都是完整的 32-bit data，因為 register file 和 memory file 能控制要寫入多少個 bytes。受影響的是與 flag register 的互動。

Flag register `FR` 和 `PC`、`IR` 一樣是 clocked register，並有一個 write signal，內部狀態、輸入輸出都是 4-bit。ALU 操作對 FR 各個 flag 的影響列於 3.2.3 節的表中，也粗描了 circuit level 的實作方式。例如，要得到 k-byte 操作的 zero flag，就對輸出的 k-byte 裡面所有 bits 做 NOR，因此總共會有三個 zero flags，各是由 1-byte、2-byte、4-byte 操作得來的。循同樣模式，每個 flag 都得到三個版本，最後以代表大小的三條訊號線選擇適當版本的四個 flags 送到 `FR`。

4.3 Register File & Memory File

Register file 如 3.2.4 節所述，必須加開一個 write port，而且兩個 write ports 的 write signals 都必須另外把 1-byte、2-byte、4-byte write 的資訊編碼進去，可能各用兩條線然後在內部解碼，或是直接多拉幾條線進去。至於兩個 read ports 和 TOY 的 register file 一樣，直接以最大寬度 32-bit 輸出。

至於 memory file 除了容量擴充、每個可定址單元大小降為 1-byte 以外，輸出一律是 `addr` 到 `addr + 3` 共 4 bytes。一個可能的方式是放四個 multiplexers，適當調整使得這四個 multiplexers 收到同一個 `addr` 時，剛好輸出 `addr` 開始的連續四個 bytes。寫入則和 register file 一樣，必須把「寫入大小」的資訊編碼到 write signal 裡面。Write port 的資料一律是 32-bit，實際寫入多少個 bytes 取決於 write signal 所給的資訊。[6]

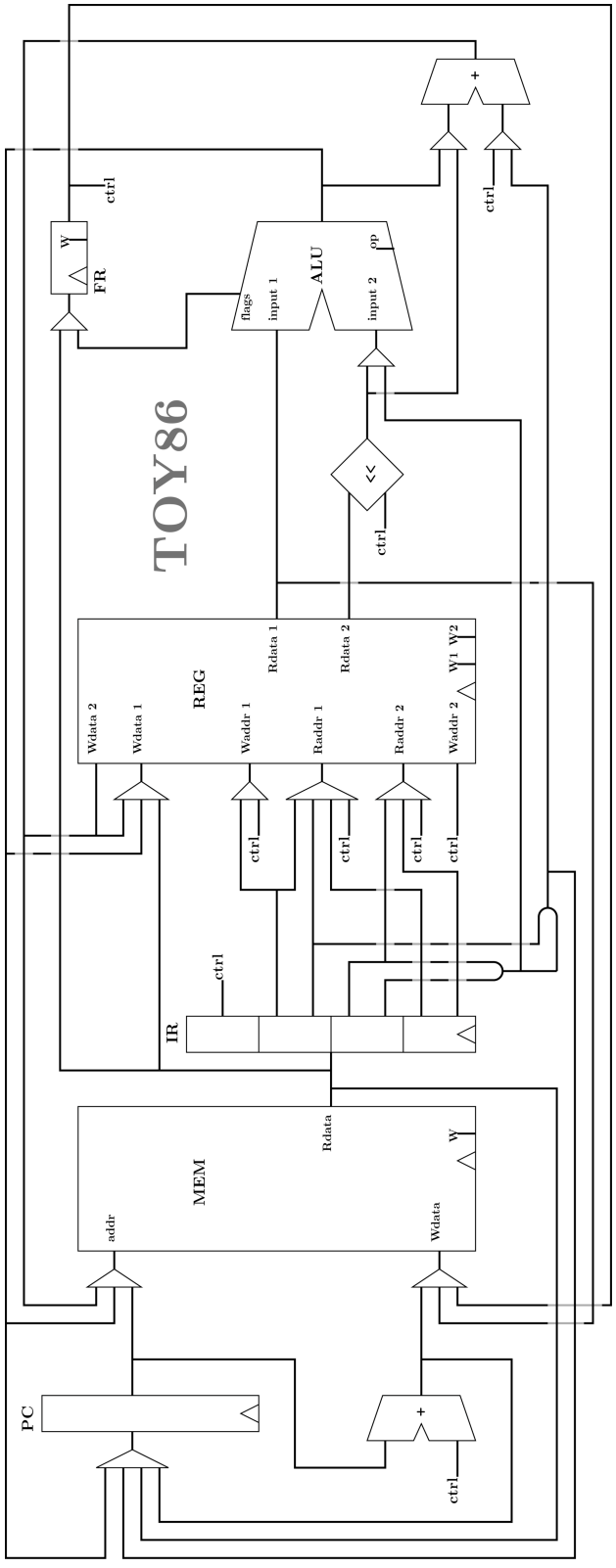
4.4 TOY86 Datapath

最後我們把所有組件連接在一起，括剛剛擴充完成的 memory file、register file、ALU、flag register `FR`，以及 12-bit clocked program counter `PC`、32-bit clocked instruction register `IR`、兩個 12-bit adders（一個用來遞增 program counter，另一個用來計算 effective address）、以及一個 12-bit left-shifter（用以計算 effective address）。TOY86 的運行仍和 classical TOY 一樣是 fetch stage 和 execute stage 交替執行，沒有 pipelining 也不使用 multi-cycle instructions，因此 control unit 完全是 combinational circuit。`PC` 在 execute stage 指向的是目前正在執行的 instruction，且 `PC` update 發生在 execute stage 的 clock edge

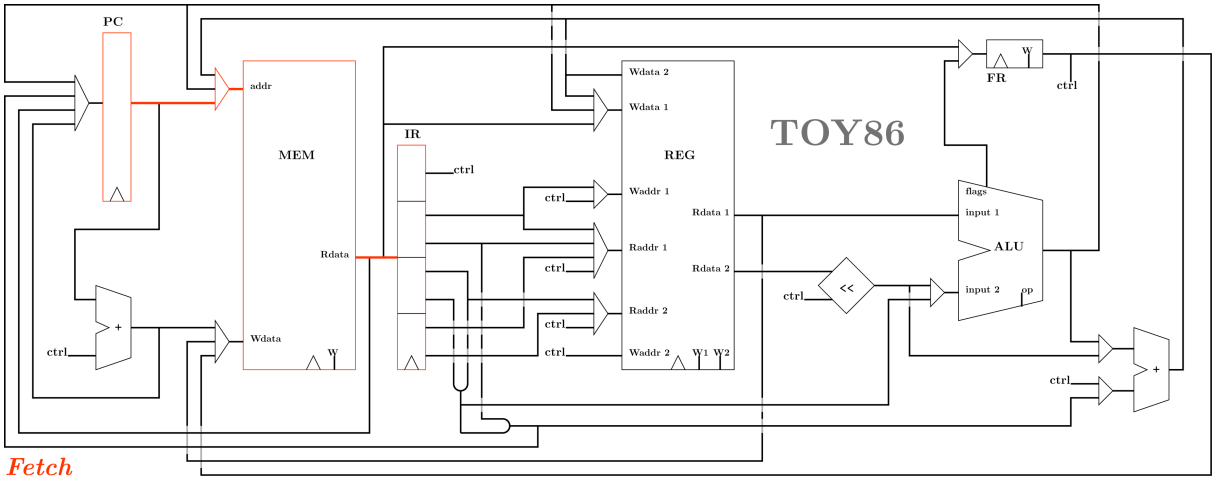
[6] 本節和上節只簡述各個重要元件的擴充，並未指明 digital logic level 的實作細節。一方面時間不足，另一方面這些元件或許到數位邏輯課程再推敲比較適當。唔，目前我們就直覺相信它們能夠運作。:P

上。如此一來，從 IR 的 opcode 部份就能推得指令長度，從而能夠計算下一個 PC 的值，也能自然地支援 jump 系列指令。

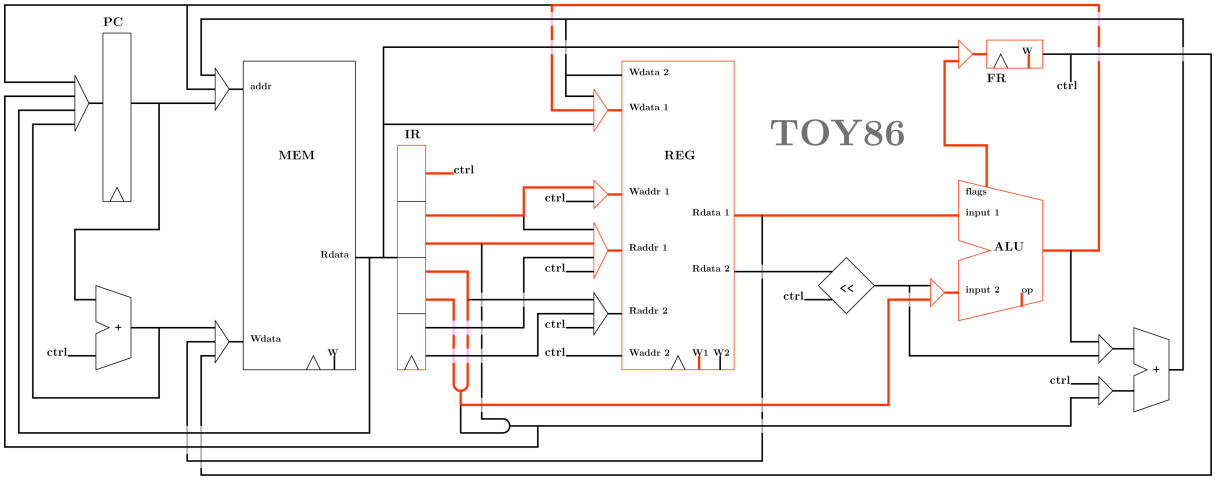
右圖是 TOY86 architecture（datapath）全覽。其中為了簡化， control signals 不細分開來，bus 的寬度均省去不寫。PC 的 I/O 為 12-bit；MEM 的 addr port 為 12-bit、write/read port 為 32-bit；IR 的輸入端為 32-bit，輸出端除了第一個 byte 為 opcode 以外，其餘三個 byte 均以 4-bit 為單位拆成獨立的 bus；REG 的 addr ports 均為 4-bit、write/read data ports 均為 32-bit；ALU 為 32-bit；FR 為 4-bit。若 bus 所連接的兩個 ports 的寬度有落差，那麼可能省去高位或在高位補上零。



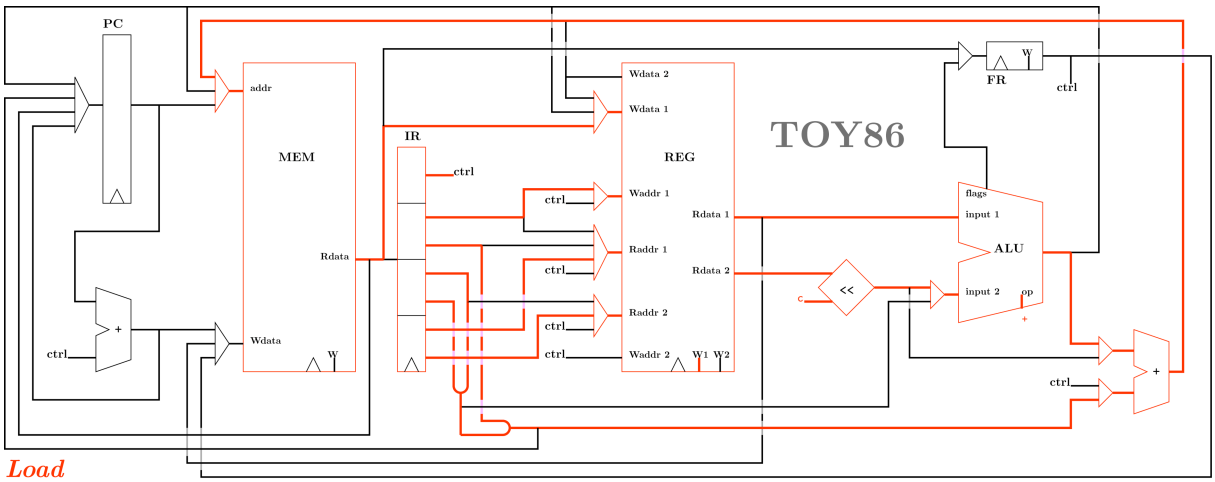
所有指令都經過確認，可在這個 architecture 上運行。以下列出部份指令運行的蹤跡，行為如同第 3 節（最後的總表）所描述。



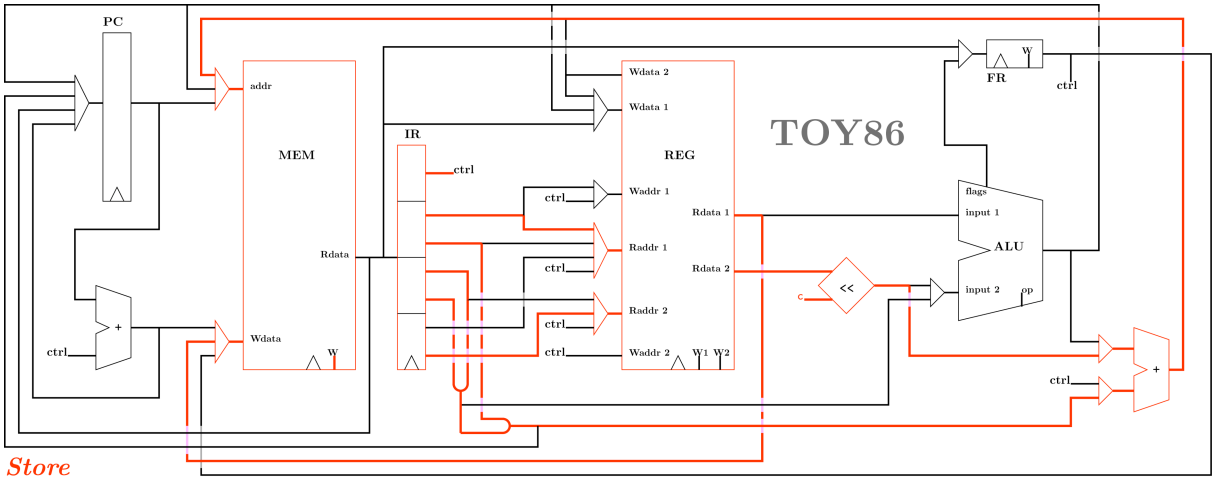
Fetch



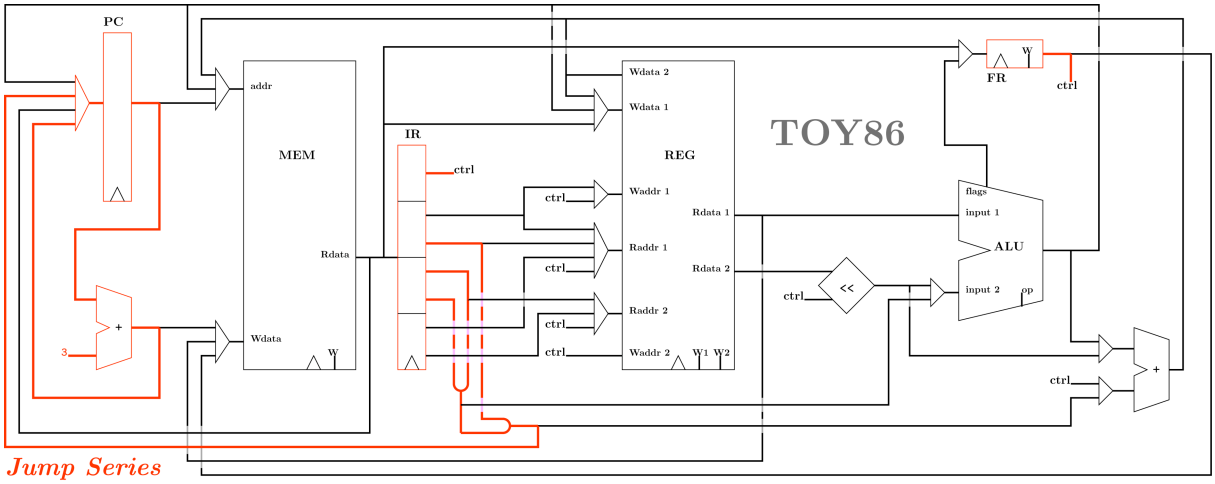
Arithmetic/Logic Operations
two-register, one-constant



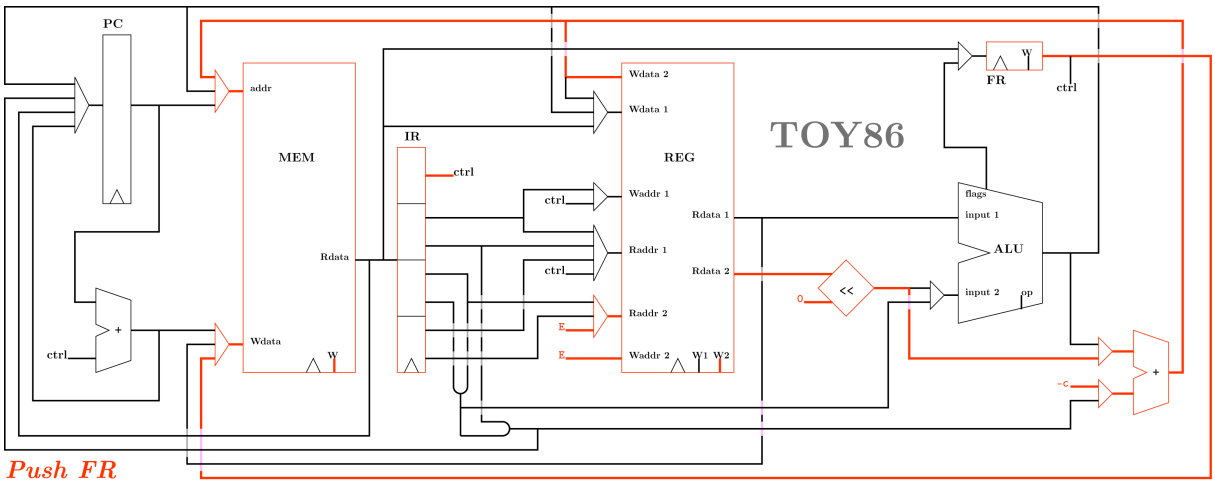
Load



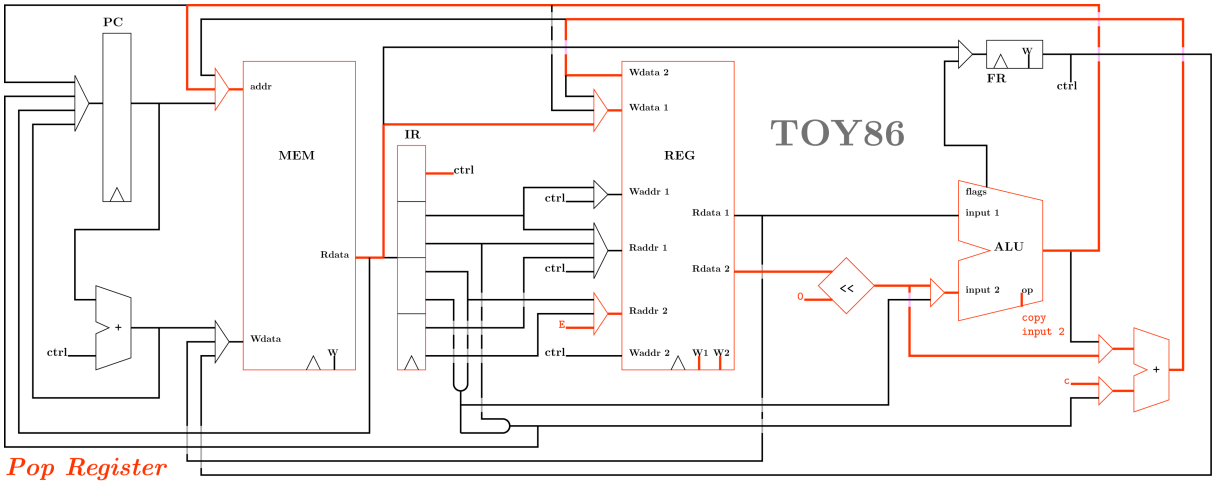
Store



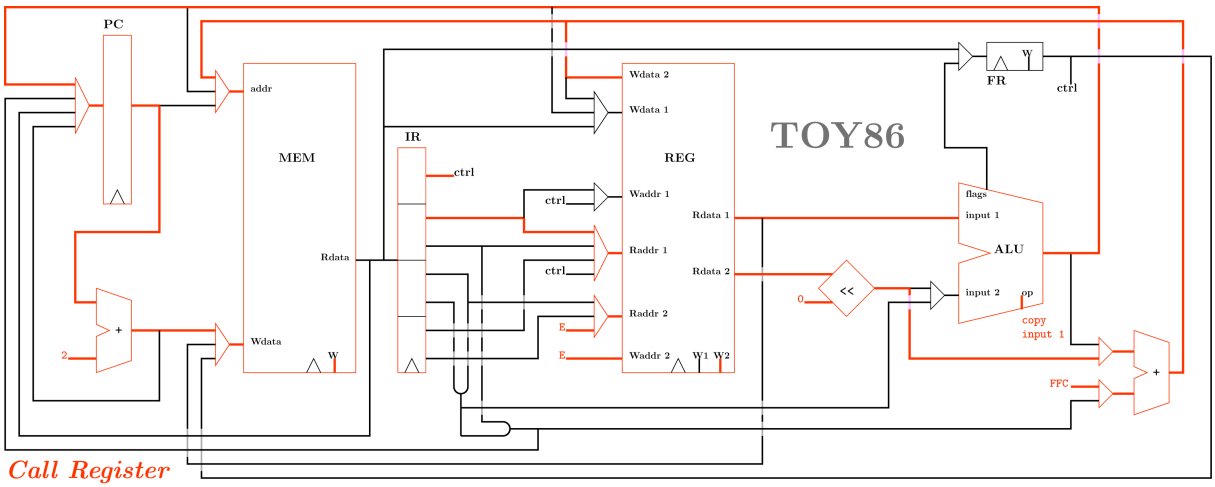
Jump Series



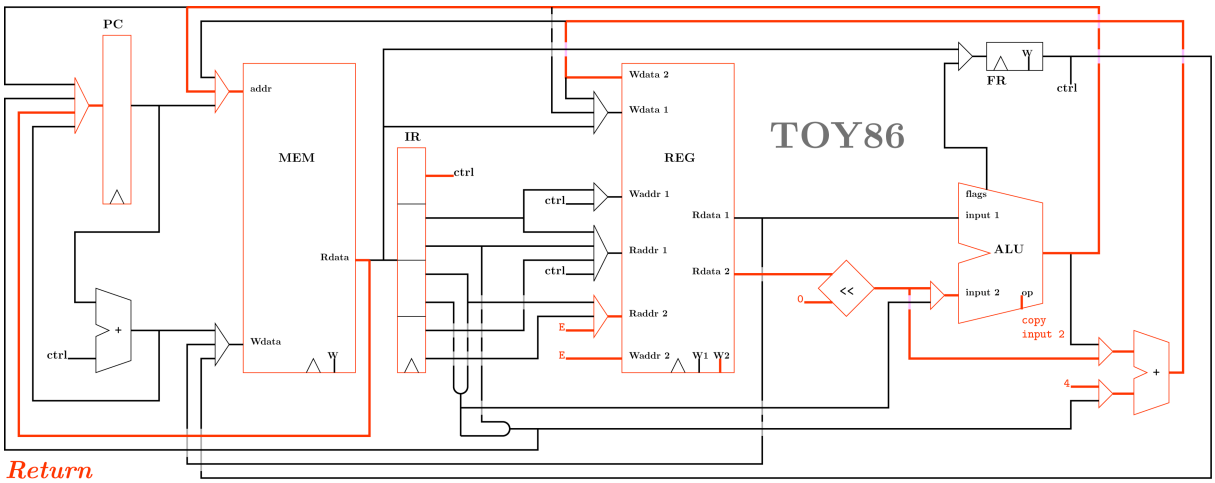
Push FR



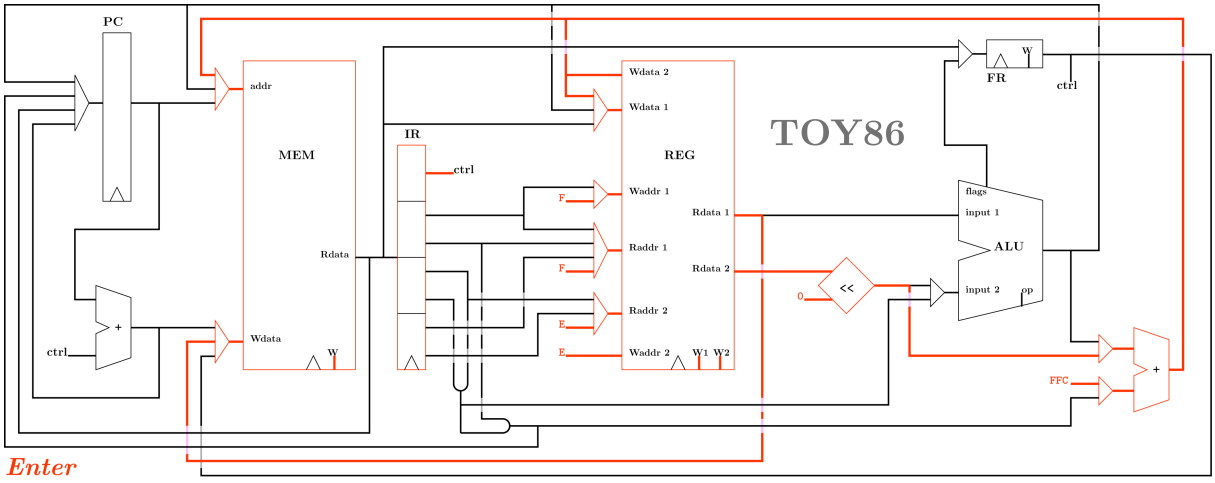
Pop Register

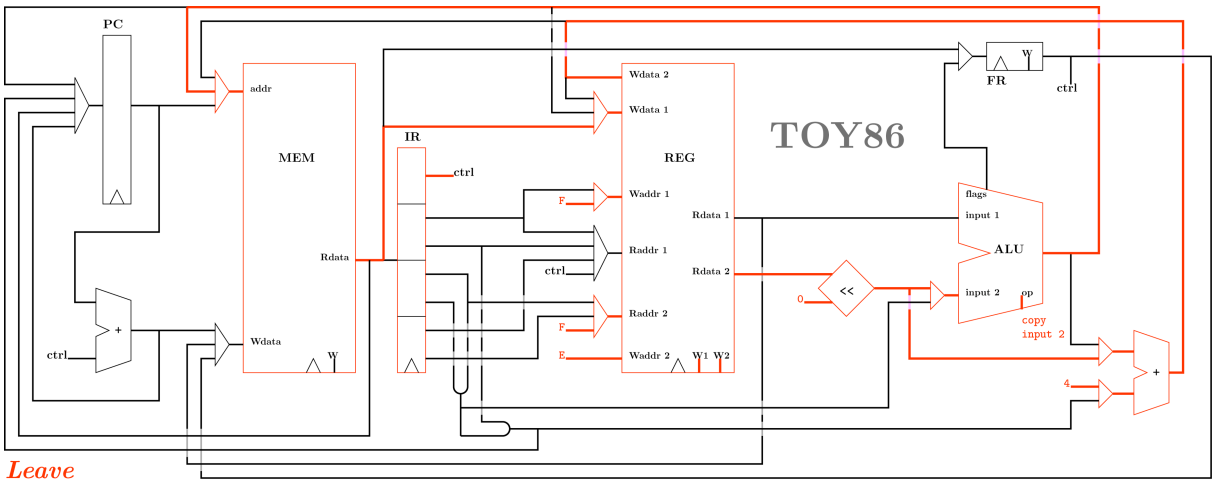


Call Register



Return





5 TOY86 組合語言

繞過漫長的遠路，我們終於回到最初目標 — 以 TOY 為基礎，設計一個更強大的組合語言。TOY86 組合語言所需的指令集已在前兩節獲得硬體層級的支援，assembler 負責處理的各項設施（facilities）也有 ExtTOY 作為參考。以下我們描述 TOY86 組合語言的各個部份。

5.1 Lexical Coventions

TOY86 assembly 的指令（ 括 instructions 和 directives）以「行」為單位，每行最多一個指令，每個指令處於一行，除了新行（newline）字元以外的多餘空白（whitespace）字元都會被忽略。註釋（comments）以 ‘;’ 起首，該行剩餘部分均為註釋。整個語言不區分大小寫（case-insensitive），label 的命名規則與

C/C++ 相同，即以字母（ 括底線）起首，後接任意數量的字母或數字，不能與關鍵字衝突。數值字面常數（numeric literals）可使用 10 進位和 16 進位，後者需冠以 `0x`，否則視為 10 進位。[7]

5.2 Data Declarations

資料宣告必須出現在任何 instructions 和 procedures 之前。資料有三種大小：`BYTE`、`WORD`、`DWORD`，可選擇給定初值或以 `DUP` operator 建立未初始化的陣列。宣告式最前面是一個可有可無的 label 名稱，接著是資料大小指示詞，然後是零或多個初始值。多個初始值之間以逗號分隔。若 label 名稱之前冠以 `export`，則這個 label 可由其他組譯單元指涉，否則其 visibility 僅限於所在檔案。假設以下資料從位址 `0x000` 開始擺放：

```
data_a BYTE  1
data_b WORD  DUP(4)
data_c DWORD 3, 4, 5, 6
       DWORD 7, DUP(2), 8
data_d WORD  9
```

則 `data_a` = `0x000`、`data_b` = `0x001`、`data_c` = `0x013`、`data_d` = `0x033`。`data_c` 並未接著 `data_b` 從 `0x009` 開始擺放，因為 TOY86 和 TOY 一樣，`PC` 從 `0x010` 起跳，因此 `0x010` 必須擺放 `jmp` 指令跳往第一個 instruction。執行檔將含以下這段 data initialization code（或等價的 code）：

```
000: 01
013: 03000000
017: 04000000
01B: 05000000
01F: 06000000
023: 07000000
02F: 08000000
033: 0900
```

Remark：TOY86 data 採用 little-endian byte order，因此 `DWORD 3` 表述為 `03000000`。

5.3 Exposed Instructions

Exposed instructions 是未 覆於 procedures 內的 instructions，用途如同 C/C++ 的 `main` function，是 TOY86 啟動後最先執行的程式片段。一個檔案內的所有 exposed instructions 即使中間任意插入 procedures，也被視為連續排列的 instructions，效果如同所有 procedures 被提到最後，不與 exposed

[7] 以 regex 表示：label 為`[a-z_][a-z0-9_]*`、hex literal 為 `0x[0-9A-F]+`、dec literal 為`[0-9]+`。

instructions 混合。多個檔案連結時，TOY86 會執行第一個檔案的 exposed instructions，且多個檔案的「exposed instruction 區段」不會連續排列。[8]

5.4 Procedures

Procedures 的宣告以 procedure label 名稱起首，然後是關鍵字 `PROC`。Procedure label 之前可冠以 `export`，此時這個 procedure label 可由其他檔案存取，否則僅在目前檔案內可見。Procedure 主體結束後以關鍵字 `ENDP` 結尾。Procedures 不能嵌套（nested）出現。在 procedure 的 scope 之內不能存取 procedure 之外的 instruction labels，而 procedure 內的 instruction labels 亦不為外部所見。簡單地說，「procedure 與 procedure」和「procedure 與 exposed instructions」之間的 instruction labels 各自獨立。

5.5 Instructions

Operand size 可變的指令以 mnemonic suffix 區分之：無 suffix 為 4-byte 操作、suffix `w`（word）是 2-byte 操作、suffix `b`（byte）是 1-byte 操作。每行指令前可加上一個 instruction label，作為 jump 系列指令的目標。若 exposed instruction label 之前冠以 `export`，則這個 label 可由其他檔案的 exposed instructions 存取。Procedure 內的 instructions 不得冠以 `export`。除了 data transfer operations 以外，address 運算元僅能使用 labels。若要存取 stdio 可使用 `stdio` label，這個 label 的值恆為 `0xFFC`，例如 `ld RA, stdio`。

5.5.1 Arithmetic/Logic Operations

和 cyyTOY/ExtTOY 相同，但 constant 僅能放在第三運算元。此類指令區分 operand size，如 `add` 指令有 `add`（4-byte）、`addw`（2-byte）、`addb`（1-byte）三個版本。

5.5.2 Data Transfer Operations

此類指令區分 operand size。第二運算元的形式如 `addr + RY + RZ * c`，其中 `addr` 為任意 label 或 12-bit 常數、`c` = 1, 2, 4。以 '+' 號分隔的三個部份可任意調換次序或省略（但不得全部省略），`RZ * c` 也可寫為 `c * RZ`。`addr` 之前若有其他部份，則可將 '+' 換成 '-'，如 `ld RA, BP - 8`，assembler 轉譯時會將 8 換為 2's complement。

另，`addr` 運算元為 procedure label 時，指令需加上 suffix `p`（在 operand size suffix 之前）。

5.5.3 Jump Operations

（Address）運算元可放置 instruction labels。欲跳躍至 procedure labels 須使用 `call` 指令。

---

[8] 設計與 ExtTOY 相同，參見 2.4 節。Procedure 的設計亦同。

5.5.4 Stack Operations

此類指令區分 operand size。

5.5.5 Procedure Operations

`call addr` 的運算元可放置 procedure labels。`ret`、`enter`、`leave` 僅可出現於 procedure 之內。

5.6 示例：Quicksort

我們首先將以下這段 C quicksort 函式（摘錄自 *K&R* 2/e, p.120-121）轉為 TOY86 assembly。

```
/* qsort:  sort v[left]...v[right] into increasing order */
void qsort(void *v[], int left, int right,
           int (*comp)(void *, void *))
{
    int i, last;
    void swap(void v[], int, int);

    if (left >= right)     /* do nothing if array contains */
        return;            /* fewer than two elements */
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i <= right; i++)
        if ((*comp)(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1, comp);
    qsort(v, last+1, right, comp);
}

void swap(void *v[], int i, int j)
{
    void *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

這段程式　括各種算術、條件、迴圈、陣列存取、遞迴函式呼叫、函式指標呼叫，恰好能夠涵蓋 TOY86 支援的所有指令種類。我們假設在 TOY86 上 `sizeof(void*)`為 2、`sizeof(int)`為 4。

```
export qsort PROC               ; global procedure
          enter

          ; BP + 8   v
          ; BP + 12  left
          ; BP + 16  right
          ; BP + 20  comp
```

```
            ; if (left >= right) return;
            ld   R1, BP + 12      ; R1 = left
            ld   R2, BP + 16      ; R2 = right
            sub  R0, R1, R2
            jns  end

            ; swap(v, left, (left + right)/2);
            add  R3, R1, R2
            shr  R3, R3, 1
            push R3
            push R1
            ld   RB, BP + 8       ; RB = v
            push RB
            call swap
            add  SP, SP, 12

            add  R3, R1, R0       ; R3 = last = left
            add  RC, R1, 1        ; RC = i = left + 1
for_cond    sub  R0, R2, RC
            js   for_end          ; if (i > right) break;

            ; preparing to call comp
            ; save all registers in use
            push R1
            push R2
            push R3
            push RB
            push RC

            ; (*comp)(v[i], v[left])
            xor  R4, R4, R4
            ldw  R4, RB + R1 * 2
            push R4
            ldw  R4, RB + RC * 2
            push R4
            ld   R4, BP + 20
            call R4
            add  SP, SP, 8

            ; restore registers
            pop  RC
            pop  RB
            pop  R3
            pop  R2
            pop  R1


            ; result is in RA
            sub  R0, RA, R0
            jns  for_incr

            ; swap(v, ++last, i);
            add  R3, R3, 1
            push RC
            push R3
```

```
          push RB
          call swap
          add  SP, SP, 12

for_incr  add  RC, RC, 1
          jmp  for_cond

          ; swap(v, left, last);
for_end   push R3
          push R1
          push RB
          call swap
          add  SP, SP, 12

          ; qsort(v, left, last-1, comp);
          ; qsort(v, last+1, right, comp);
          ld   R4, BP + 20
          push R4
          push R2
          add  R3, R3, 1
          push R3
          push RB
          push R4
          sub  R3, R3, 2
          push R3
          push R1
          push RB
          call qsort
          add  SP, SP, 16
          call qsort

end       leave
          ret
ENDP

swap PROC                       ; local procedure
          enter

          ; BP + 8   v
          ; BP + 12  i
          ; BP + 16  j

          ld   R7, BP + 8
          ld   R8, BP + 12
          lea  R8, R7 + R8 * 2  ; R8 = &v[i]
          ld   R9, BP + 16
          lea  R9, R7 + R9 * 2  ; R9 = &v[j]
          ldw  RA, R8           ; RA = v[i]
          ldw  R6, R9           ; R6 = v[j]
          stw  RA, R9           ; v[j] = orig v[i]
          stw  R6, R8           ; v[i] = orig v[j]

          leave
          ret
ENDP
```

然後是測試用的主程式，直接將 “pointer” 視為整數排序。

```
; test program
          ld   RC, stdio
          shl  RC, RC, 1
          sub  SP, SP, RC
          shr  RC, RC, 1
          add  BP, SP, R0
          add  RD, RC, R0
i_loop    jz   i_end
          ldw  RA, stdio
          stw  RA, BP
          add  BP, BP, 2
          sub  RD, RD, 1
          jmp  i_loop
i_end     leap RA, int_compare
          add  BP, SP, R0
          sub  RD, RC, 1
          push RC
          push RA
          push RD
          push R0
          push BP
          call qsort
          add  SP, SP, 16
          pop  RC
          sub  R0, RC, R0
o_loop    jz   o_end
          popw RA
          stw  RA, stdio
          sub  RC, RC, 1
          jmp  o_loop
o_end     hlt

int_compare PROC
          enter
          ; BP + 8  a
          ; BP + 12  b
          ld   RA, BP + 8
          ld   RB, BP + 12
          sub  RA, RA, RB
          leave
          ret
ENDP
```

全部組譯連結出來的（一種）結果是：

```
010: 90CFFC00    ;           ld   RC, stdio
014: 64CC01      ;           shl  RC, RC, 1
017: 20EEC0      ;           sub  SP, SP, RC
01A: 74CC01      ;           shr  RC, RC, 1
01D: 10FE00      ;           add  BP, SP, R0
020: 10DC00      ;           add  RD, RC, R0
023: B00037      ; i_loop    jz   i_end
026: 92AFFC00    ;           ldw  RA, stdio
```

```
02A: A2A0000F    ;           stw  RA, BP
02E: 14FF02      ;           add  BP, BP, 2
031: 24DD01      ;           sub  RD, RD, 1
034: B80023      ;           jmp  i_loop
037: 80A10000    ; i_end     lea  RA, int_compare
03B: 10FE00      ;           add  BP, SP, R0
03E: 24DC01      ;           sub  RD, RC, 1
041: C0C0        ;           push RC
043: C0A0        ;           push RA
045: C0D0        ;           push RD
047: C000        ;           push R0
049: C0F0        ;           push BP
04B: E00200      ;           call qsort
04E: 14EE10      ;           add  SP, SP, 16
051: D0C0        ;           pop  RC
053: 200C00      ;           sub  R0, RC, R0
056: B00065      ; o_loop    jz   o_end
059: D2A0        ;           popw RA
05B: A2AFFC00    ;           stw  RA, stdio
05F: 24CC01      ;           sub  RC, RC, 1
062: B80053      ;           jmp  o_loop
065: 00          ; o_end     hlt

; int_compare PROC
100: F0          ;           enter
101: 90A008F0    ;           ld   RA, BP + 8
105: 90B00CF0    ;           ld   RB, BP + 12
109: 20AAB0      ;           sub  RA, RA, RB
10C: F1          ;           leave
10D: E2          ;           ret
; ENDP

; swap PROC
140: F0          ;           enter
141: 907008F0    ;           ld   R7, BP + 8
145: 90800CF0    ;           ld   R8, BP + 12
149: 84800078    ;           lea  R8, R7 + R8 * 2
14D: 909010F0    ;           ld   R9, BP + 16
151: 84900079    ;           lea  R9, R7 + R9 * 2
155: 92A00080    ;           ldw  RA, R8
159: 92600090    ;           ldw  R6, R9
15D: A2A00009    ;           stw  RA, R9
161: A2600008    ;           stw  R6, R8
165: F1          ;           leave
166: E2          ;           ret
; ENDP

; export qsort PROC
200: F0          ;           enter
201: 90100CF0    ;           ld   R1, BP + 12
205: 902010F0    ;           ld   R2, BP + 16
209: 200120      ;           sub  R0, R1, R2
20C: B702A7      ;           jns  end
20F: 103120      ;           add  R3, R1, R2
212: 743301      ;           shr  R3, R3, 1
```

```
215: C030       ;           push R3
217: C010       ;           push R1
219: 90B008F0   ;           ld   RB, BP + 8
21D: C0B0       ;           push RB
21F: E00140     ;           call swap
222: 14EE0C     ;           add  SP, SP, 12
225: 103100     ;           add  R3, R1, R0
228: 14C101     ;           add  RC, R1, 1
22B: 2002C0     ; for_cond  sub  R0, R2, RC
22E: B60278     ;           js   for_end
231: C010       ;           push R1
233: C020       ;           push R2
235: C030       ;           push R3
237: C0B0       ;           push RB
239: C0C0       ;           push RC
23B: 504440     ;           xor  R4, R4, R4
23E: 964000B1   ;           ldw  R4, RB + R1 * 2
242: C040       ;           push R4
244: 964000BC   ;           ldw  R4, RB + RC * 2
248: C040       ;           push R4
24A: 904014F0   ;           ld   R4, BP + 20
24E: E140       ;           call R4
250: 14EE08     ;           add  SP, SP, 8
253: D0C0       ;           pop  RC
255: D0B0       ;           pop  RB
257: D030       ;           pop  R3
259: D020       ;           pop  R2
25B: D010       ;           pop  R1
25D: 200A00     ;           sub  R0, RA, R0
260: B70272     ;           jns  for_incr
263: 143301     ;           add  R3, R3, 1
266: C0C0       ;           push RC
268: C030       ;           push R3
26A: C0B0       ;           push RB
26C: E00140     ;           call swap
26F: 14EE0C     ;           add  SP, SP, 12
272: 14CC01     ; for_incr  add  RC, RC, 1
275: B8022B     ;           jmp  for_cond
278: C030       ; for_end   push R3
27A: C010       ;           push R1
27C: C0B0       ;           push RB
27E: E00140     ;           call swap
281: 14EE0C     ;           add  SP, SP, 12
284: 904014F0   ;           ld   R4, BP + 20
288: C040       ;           push R4
28A: C020       ;           push R2
28C: 143301     ;           add  R3, R3, 1
28F: C030       ;           push R3
291: C0B0       ;           push RB
293: C040       ;           push R4
295: 243302     ;           sub  R3, R3, 2
298: C030       ;           push R3
29A: C010       ;           push R1
29C: C0B0       ;           push RB
29E: E00200     ;           call qsort
```

```
2A1: 14EE10      ;           add  SP, SP, 16
2A4: E00200      ;           call qsort
2A7: F1          ; end       leave
2A8: E2          ;           ret
; ENDP
```

6 参考資料

1. Bryant, R. E., and O'Hallaron, D. R. *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2002.

2. Hoxey, S., Karim, F., Hay, B., and Warren, H. *The PowerPC Compiler Writer's Guide*. http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF778525699600755 8C6

3. Irvine K. R. *Assembly Language for Intel-Based Computers*, 4th edition. Prentice Hall, 2002.

4. Kernighan, B. W., and Ritchie D. M. *The C Programming Language*, 2nd edition. Prentice Hall, 1989.

5. Patterson, D. A., and Hennessy, J. L. *Computer Organization and Design: the Hardware/Software Interface*, 3rd edition. Morgan Kaufmann, 2004.

6. *Intel® 64 and IA-32 Architectures Software Developer's Manuals*. http://www.intel.com/products/processor/manuals/

7. *Wikipedia*. http://en.wikipedia.org/