

# 你所不知道的 C 語言：編譯器和最佳化原理篇

以 GNU Toolchain 為探討對象，簡述編譯器如何運作，以及如何實現最佳化

Copyright (憲C) 2015, 2016, 2017 宅色夫

直播錄影(上)

直播錄影(下)

## 理解編譯器最佳化有時是為了逃出困境

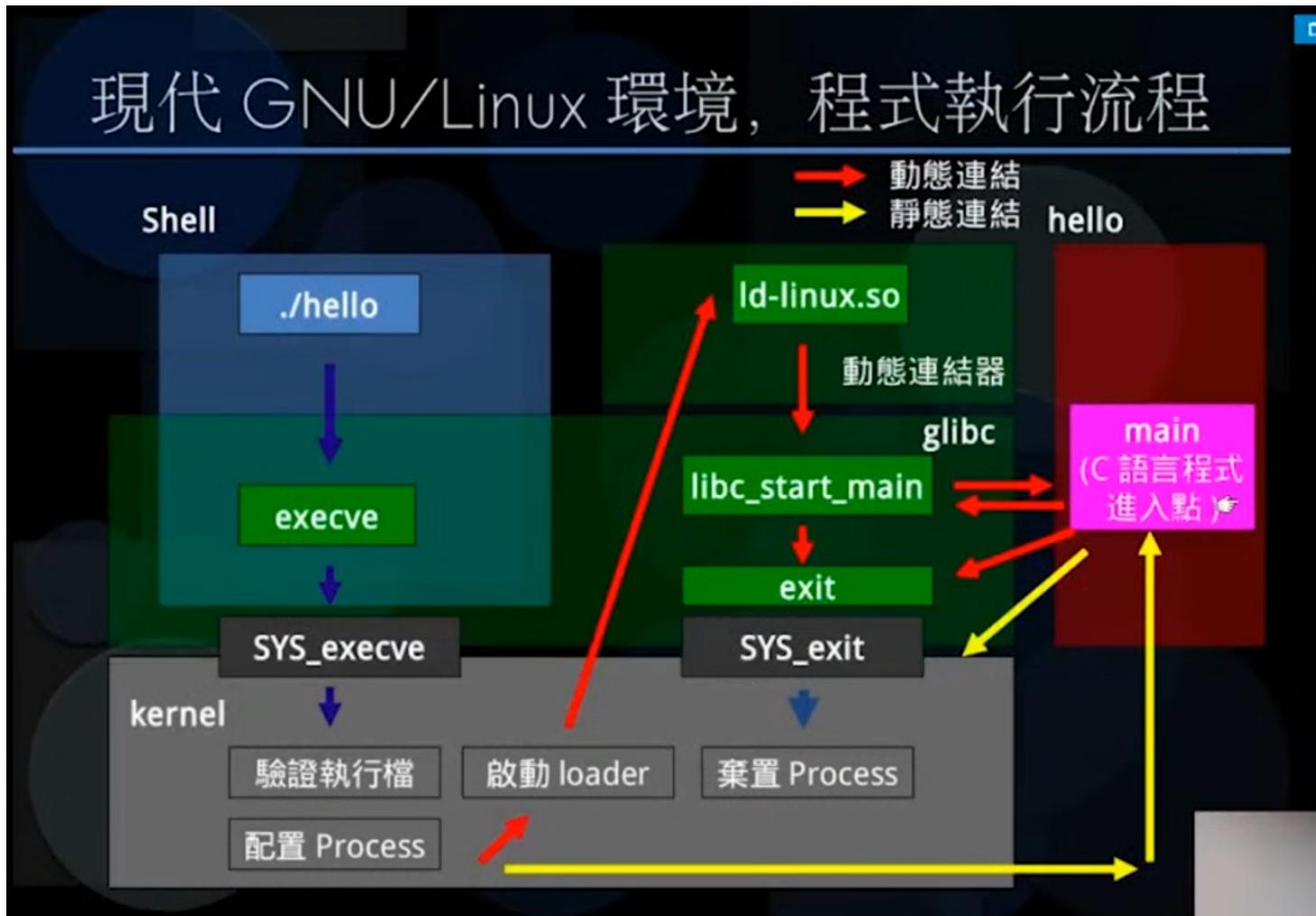
- Who's afraid of a big bad optimizing compiler?
  - 基本的 C 語言 load/store 操作如 `a = b;` 這樣陳述，依據 C 語言標準，編譯器可認定牽涉到的變數在 load 或 store 的同一時刻，沒有其他執行緒在存取它們，因此允許多種最佳化，而目前越來越強大的現代編譯器所做的程式碼最佳化已超出許多人預料。
  - 經過某些最佳化手法後，開發者不能再期望。本文雖針對 Linux 核心，但其實很多場景也適用於其他的並行 (concurrent) 程式設計議題，也包括使用了中斷和 signal 相關程式碼
  - 隨著編譯器越來越強大，就讓我們不禁開始擔心：「這些來自編譯器的最佳化有多危險？」

## 編譯器無所不在

- 編譯器的多元應用(連結失效): 小小一只 Android 手機，裡頭內建了 8 種以上動態編譯器和虛擬機器

- Where can we find compiler
- 為什麼你該理解編譯器的原理
  - 其中一個理由是理解編譯器的限制: [cpp: Implementation limits](#)
- Missed optimizations in C compilers
- What Can Compilers Do for Us?
- Don't Overflow the Stack
  - MISRA C (Motor Industry Software Reliability Association)
- Basics of Compiler Design
  - Lexical analysis
    - Regular expression, NFA (Nondeterministic finite automata)
    - NFA -> DFA
    - lexer
  - Syntax analysis
    - context-free grammar, syntax tree and ambiguity, operator precedence, predictive parsing
    - LL(1) parsing: recursive descent
    - SLR parsing
    - LR-parser generator
  - Scopes & Symbol Tables
  - Interpretation
  - Type checking
  - Code generation: register allocation

## From Source to Binary: How A Compiler Works: GNU Toolchain



[ Page 5 ]

- 這邊提到驗證執行檔後並載入動態函式庫，為何要有 C Runtime?

先想想以下程式的執行: (hello.c)

```
int main() { return 1; }
```

當你在 GNU/Linux 下編譯和執行後 (`gcc -o hello hello.c ; ./hello`)，可用 `echo $?` 來取得程式返回值，也就是 `1`，可是這個返回值總有機制來處理吧？所以你需要一套小程序來作，這就是 C runtime (簡稱 crt)。此外，還有 `atexit` 一類的函式，也需要 C runtime 的介入才能實現。

C 語言和一般的程式語言有個很重要的差異，就是 C 語言設計來滿足系統程式的需求，首先是作業系統核心，再來是一系列的工具程式，像是 `ls`, `find`, `grep` 等等，而我們如果忽略指標操作這類幾乎可以直接對應於組合語言的指令的特色，C 語言之所以需要 runtime，有以下幾個地方：

1. `int main() { return 1; }` 也就是 `main()` 結束後，要將 `exit code` 傳遞給作業系統的程式，這部份會放在 `crt0`
2. exception handling，不要懷疑，C 語言當然有這個特徵，只是透過 `setjmp` 和 `longjmp` 函式來存取，這需要額外的函式庫 (如 `libgcc`) 來協助處理 `stack`
3. 算術操作，特別在硬體沒有 FPU 時，需要 `libgcc` 來提供浮點運算協助
  - C 語言裡面有沒有例外處理？
    - `setjmp`
    - `longjmp`
  - 在 kernel 裡的「驗證執行檔」步驟，是要驗證什麼？
  - UNIX 的「執行檔」有很多種可能，一個是依據特定格式保存的機械碼，也可能是透過額外程式去解析的 shell script，作業系統核心必須得事先解析並確認這個合法的執行檔，才能著手去執行
  - 近來還有對執行檔進行簽章的機制，請見: [ELF executable signing and verification](#)

[ Page 16 ]

- 能夠自己編譯自己原始程式碼的程式: [Self-hosting](#)：作為 toolchain 的一部分，可以產生新版同一程式的程式

- The term **self-hosting** was coined to refer to the use of a [computer program](#) as part of the [toolchain](#) or [operating system](#) that produces new versions of that same program
- 想一個情境，C compiler 的原始程式碼如果用 C 語言撰寫，那要用什麼東西來編譯 C compiler 的原始程式碼呢？如果可以作到這件事，就是所謂的 self-compilation，這不是容易的事，因為編譯器要很可靠才行
- 要知道一件事，由於電腦發展最初是解決軍事需求，後來則應用到人口普查系統，人們一直都希望電腦可以理解人類的語言，這也是為何早期（1950 年代）的程式語言其實很高階的緣故，有透過數學表達的風格（如 LISP），也有類似英文書寫風格（如 COBOL）
- 更有趣的是，就算電腦發展還在真空管的時代，人們就著手研究解析語言學和數學的關聯，這也是為何你在計算理論和編譯器課程（這兩者是資訊工程系和電機系極少沒有重疊的科目）會發現一些語言學的跡象
- 回到編譯器的設計，由於早期硬體限制很多，其實是不可能直接實做出高階語言的編譯器，相反的，早期的工程人員必須漸進地開發相關的工具的程式，所以你可以想像最早人們用機械碼拼湊出簡單的 assembler，然後在這之上發展了簡單的 C compiler，之後再用這個 C compiler 開發出更完整的 C compiler，後者可以編譯更完整的 C 語言程式，然後逐步延展下去。
- Microsoft 舊有的 C# compiler 以 C++ 撰寫，“eat its own dog food” 是各面向的系統開發人員一個重要里程，因此 Microsoft C# 團隊決定以 C# 重新撰寫 C# compiler，並將成果於 GitHub 上開源釋出，如此 C# compiler 是以 C# 撰寫，並能以 C# compiler 編譯出自己的新版本，詳見 [How Microsoft rewrote its C# compiler in C# and made it open source - Project Roslyn](#)。（2018 年 9 月）

[ Page 30 ]

切換到 [Re-introduction to Compiler](#)

code generation 那頁展示 ARMv8-A 組合語言輸出

[ Page 31 ]

下圖說明如果沒有 IR 這樣的中間表示法，每個前端 (source files; human readable programming language) 直接對應到後端 (machine code) 的話，有太多種組合，勢必會讓開發量爆增。但如果先讓前端輸出 IR，然後 IR 再接到後端，這樣整體的彈性就大多了。

延伸閱讀: [The Challenge of Cross-language Interoperability: The increasing significance of intermediate representations in compilers](#)

[ Page 36 - 38 ]

1. Pointer Aliasing 會讓編譯器最佳化功能無用武之地。
2. 針對下面程式碼

```
void foo(int *i1, int *i2, int *res)
{
    for (int i = 0; i < 10; i++) {
        *res += *i1 + *i2;
    }
}
```

- 可能一開始會想優化成下面這樣來加速

```
void foo(int *i1, int *i2, int *res)
{
    int tmp = *i1 + *i2;
    for (int i = 0; i < 10; i++) {
        *res += tmp;
    }
}
```

- 但若考慮到有人這樣寫 `foo(&var, &var, &var)` 來呼叫 `foo()`，那很明顯就會發生問題了，`*res` 值的改變影響 `*i1`、`*i2`。因此編譯器不會採取該最佳化手段，因為不知道 caller 會怎麼傳遞數值。

切到 [Compiler Analysis](#)

[ Page 40 ]

- SSA (Static Single Assignment)
- 每個被 Assign 的變數只會出現一次
- 將被 Assign 的變數給一個版本號碼，再使用  $\Phi$  function 例如:  $ret3 = \Phi(ret1, ret2)$  去判斷從何而來
- SSA 把原本複雜的條件和程式邏輯轉化為 Graph，用讓每個程式邏輯轉成明確的路徑
  - GCC 的 [SSA 說明](#)
  - " The SSA form is based on the premise that program variables are assigned in exactly one location in the program. Multiple assignments to the same variable create new versions of that variable. "
  - SSA 最佳化實例：
  - constant propagation
  - 用法：
    1. 先拆成 basic blocks (單行的指令, 沒有jump or label)
    2. 合併 basic blocks
    3. 最佳化 CFG (減少 goto )
    4. 轉成 SSA form (變數 runtime decided 時, 用函數表示, 並給新的版本號) . 例:  
 $i1 = \Phi(i0, i2)$
    5. constant propagation (變數換成常數)
    6. dead code elimination (remove 不需要的變數)

7. value range propagation (變數用一個範圍的常數來代替)

8. dead code elimination (結果: 直接 `return 0`)

延伸閱讀:

- [Static Single Assignment Form](#), Kenneth Zadeck (GCC Summit 2004)
- [Static Single Assignment Book](#)

[ Page 43 ] GCC 可輸出包含 Basic Block 的CFG

- `gcc -c -fdump-tree-cfg=out test.c`

[ Page 62 ] GCC 後端: Register Transfer Language (RTL)

- LISP-Style Representation (令人喜愛的 S-Expression)
- 生成 RTL 的方式

```
gcc -fdump-rtl-expand xxx.c
```

- RTL Uses Virtual Register
- GCC Built-in Operation and Arch-defined Operation
- Instruction scheduling (Pipeline scheduling)
- Peephole optimizations

[ Page 67 ]

- 最佳化來自對系統的認知

假設我們有兩個**有號整數**: `<stdint.h>`

```
int32_t a, b;
```

然後原本涉及到分支的陳述：

```
if (b < 0) a++;
```

可更換為沒有分支的版本：

```
a -= b >> 31;
```

針對 C 語言編譯最佳化的常見策略：

- Branch Optimization
  - If optimization

```
void f (int *p) {  
    if (p) g(1);  
    if (p) g(2);  
    return;  
}
```

可簡化為以下：

```
void f (int *p) {  
    if (p) {  
        g(1);  
        g(2);  
    }  
}
```

```
    return;  
}
```

- Value Range Optimization

```
for (int i = 1; i < 100; i++) {  
    if (i) g();  
}
```

因為已知變數 `i` 必然是大於等於 `1` 的正整數，於是可刪去 `if` 敘述：

```
for (int i = 1; i < 100; i++) {  
    g();  
}
```

- Branch elimination

```
goto L1;  
    // do something  
L1:  
    goto L2 // L1 branch is unnecessary
```

- Unswitching

減少因為迴圈帶來的分支數量，例如：

```
for (int i = 0; i < N; i++) {  
    if (x) a[i] = 0;  
    else b[i] = 0;
```

可改寫為分支總量更少的形式：

```
if (x)
    for (int i = 0; i < N; i++)
        a[i] = 0
else
    for (int i = 0; i < N; i++)
        b[i] = 0;
```

尾端遞迴 (Tail Recursion) 可透過 goto 替換來減少 stack frame，從而達到最佳化，對照參考 [遞迴呼叫篇](#)。考慮以下程式：

```
int f(int i) {
    if (i > 0) {
        g(i);
        return f(i - 1)
    }
    else {
        return 0;
    }
}
```

可最佳化為以下：

```
int f (int i) {
entry:
    if (i > 0) {
        g(i);
        i--;
        goto entry;
    }
    return 0;
}
```

- Loop Optimizations

- Loop unrolling

當迴圈內的操作彼此獨立時，可將操作展開。例如：

```
for (int i = 0; i < 100; i++) {  
    g();  
}
```

可改寫為以下：

```
for (int i = 0; i < 100; i += 2) {  
    g();  
    g();  
}
```

這樣分支的總數就縮減了。

- Loop Collapsing

考慮到以下程式碼：

```
int a[100][300];  
for (i = 0; i < 300; i++)  
    for (j = 0; j < 100; j++)  
        a[j][i] = 0;
```

原本的巢狀迴圈可改寫為單層迴圈，只要將 array subscripting 變更為指標存取的方式，可對照參考 [指標篇](#)。等價但效率更高的程式碼：

```
int a[100][300];  
int *p = &a[0][0];  
for (i = 0; i < 30000; i++)  
    *p++ = 0;
```

- Loop fusion

兩個獨立的回圈可合併為一，例如：

```
for (i = 0; i < 300; i++)
    a[i] = a[i] + 3;
for (i = 0; i < 300; i++)
    b[i] = b[i] + 4;
```

可改寫為以下更少分支的程式碼：

```
for (i = 0; i < 300; i++) {
    a[i] = a[i] + 3;
    b[i] = b[i] + 4;
}
```

- Access pattern optimization

- Quick Optimization

存取特定物件或結構體的成員，可事先快取，例如：

```
for (i = 0; i < 10; i++)
    arr[i] = obj.i + volatile_var;
```

可改寫為以下更少記憶體操作的程式碼：

```
t = obj.i;
for (i = 0; i < 10; i++)
    arr[i] = t + volatile_var;
```

- Constant Propagation/Constant folding

```
int x = 3;
int y = 4 + x; // 可換為 y = 7;
return (x + y) // 可換為 return 10;
```

- Narrowing

```
unsigned short int s;
(s >> 20)      /* all bits of precision have been shifted out, thus 0 */
(s > 0x10000)   /* 16 bit value can't be greater than 17 bit, thus 0 */
(s == -1)        /* can't be negative, thus 0 */
```

- Integer mod optimization

在許多硬體上，除法指令需要更多的 CPU cycle，因此可考慮功能等價的指令。例如：

```
int f (int x, int y) {
    return x % y;
}
```

可改寫為以下：

```
int f (int x, int y) {
    int tmp = x & (y - 1);
    return (x < 0) ? ((tmp == 0) ? 0 : (tmp | ~ (y - 1))) : tmp;
}
```

關於這類技巧可參照 [bitwise 操作](#)。

- Block Merging

考慮以下程式碼：

```
int a, b;
void f(int x, int y) {
```

```
    goto L1; /* basic block 1 */
L2:           /* basic block 2 */
    b = x + y;
    goto L3;
L1:           /* basic block 3 */
    a = x + y;
    goto L2;
L3:           /* basic block 4 */
    return;
}
```

可改寫為以下：

```
int a, b;
void f(int x, int y) {
    a = x + y; /* basic block 1 */
    b = x + y;
    return;
}
```

- Common SubExpression (CSE)

上述 Block Merging 所舉的案例又可改寫為以下：

```
int a, b;
void f(int x, int y) {
    int tmp = x + y
    a = b = tmp;
    return;
}
```

- Function inlining

倘若函式頻繁地呼叫，可考慮將函式展開到呼叫端函式裡頭，像是巨集展開一般。例如：

```
int add(int x, int y) {
    return x + y;
}
int sub(int x, int y) {
    return add(x, -y);
}
```

於是可改寫為以下：

```
int sub (int x, int y) {
    return x - y;
}
```

[ Page 69 ]

- 編譯器可刪去沒在使用的 static global variable 以節省空間，但不能刪去 (目前) 沒在使用的 non-static global variable，因為無法確定其他的 Compilation Unit 是否會用到此變數
  - 這是為何建議 local function 要宣告成 `static` 的用意！

[ Page 71 ]

- gcc 會把特定的 `printf()` 悄悄換成 `puts()`，這有什麼好處？
- `printf()` 本身就是個解譯器，要處理一堆格式，執行時間和字串長度並未完全相符合。但 `puts()` 就不一樣，只跟何時找到 null terminator 有關，行為明確多了，當然整體執行時間也更短。
- 為何 GCC 算是個 Compiler Driver？在使用上，我們要進行 link 也是會另外使用 ld，gcc 也可以當 linker 嗎？

ld 是真正的 linker，而 gcc 作為 compiler driver，自然也可以呼叫到 ld 作 linking

- 在此範例中最佳化CFG.為何是將bb0最後加一個goto bb2,而bb2搬移到bb3下方.如果原本尚未搬移前多一個goto bb2的動作.這樣最佳化的目的為何？

減少 compare-n-branch 的總次數

右邊的是接近組合語言的寫法

變成 0 的魔法

## 解說：最佳化 CFG

```
int main() {  
bb0:  
    a = 11  
    b = 13  
    c = 5566  
    result = UNDEFINED  
    i = 0  
bb2:  
    if (i < 10000) goto bb3  
    else goto bb5  
bb3:  
    result = (a*b+i) / (a*c)  
    i = i + 1  
    goto bb2  
bb5:  
    return result  
}  
  
int main() {  
bb0:  
    a = 11  
    b = 13  
    c = 5566  
    result = UNDEFINED  
    i = 0  
    goto bb2  
bb3:  
    result = (a*b+i) / (a*c)  
    i = i + 1  
bb2:  
    if (i < 10000) goto bb3  
    else goto bb5  
bb5:  
    return result  
}
```

[ Page 65 ] GCC後端 管線排程

Instruction scheduling 透過將 instruction 重排的方式以減少 pipeline hazard，已達到 ILP(instruction-level parallelism) 的目的

[ Page 66 ]

在資訊領域中，有個術語叫做 [peephole optimization](#)，對應到漢語成語即是「以管窺天」，即逐步擬定策略，消除沒作用或者冗餘的程式碼。以編譯器最佳化來說，此策略就是反覆在已經產生的組合語言或機械碼中，針對目標處理器架構的特性，套用一系列可能會對效能帶來助益的轉換規則，或者透過整體的分析，進行指令轉換，進而提升程式碼效能或空間佔用效率。乍聽之下這樣的轉換很侷限，但累積起來仍有機會對整體帶來顯著效益。

這個 peephole 可看做是個滑動窗口，編譯器在實施 peephole optimization 時，就僅分析在該窗口內的指令列表。每次轉換後，可能還會展現出相鄰窗口之間的特定改善空間，因此該策略仍可反覆套用。peephole optimization 在編譯器最佳化可體現於四個面向：

1. 刪去冗餘指令：包括冗餘的 load 和 store 指令，及無效程式碼 (即不會執行到的程式碼);
2. control flow 最佳化；
3. 弱化特定的作用：利用代價較小的指令或操作，替代原本代價較大的指令或操作；
4. 利用特有指令；



刪除冗餘 load 和 store：考慮到以下 MIPS 組合語言輸出：

```
sh $0, 6($sp)
sh $0, 4($sp)
sh $0, 2($sp)
sh $0, 0($sp)
ldcl $f1, 0($sp)
```

完全可用指令 `xor $f1,$f1,$f1` 取代，這樣可省掉 5 次暫存器存取操作，效能改善的幅度可非常明顯。其中 `sh` 是 store 16bit 到某個位址，`ldc1` 是 load 64bit 到某個暫存器。`xor` 是 bitwise XOR 指令。

但這種指令序列的轉換和合成有個前提，必須保證這些指令按照順序執行，即這些指令之間，不能有其他 label。也就是說，這些指令必須在一個 basic block 塊中。當然，你也可在編譯器較前面階段的最佳化，針對該操作進行變換。這樣到 peephole optimization 時，就不再會有這樣的程式碼。

 弱化特定的作用：考慮到 `x = x + 0`, `x = x * 1` 之類的操作，就能直接避免產生程式碼（因為數值不變），而 `x = x * 2`, `x = x / 2` 之類的操作可改用左移或右移來實作。至於  $x^2$  之類的指數運算可削弱為 `x * x` 的乘法運算，浮點數除以常數的運算可轉換為浮點數乘以常數的倒數。這些都是最佳化策略。

那麼，為何是“peephole”這個看起來不文雅的詞呢？

11 世紀的英格蘭城市 Coventry 裡，人民的生活苦不堪言，這時市長 Leofric the Dane 的妻子 Godiva 請求讓人民減稅；市長有意刁難，就說：如果 Godiva 願意裸體騎馬遊街一圈，這樣才可不加稅。

隔日，Godiva 真的付諸行動，且每戶人家都關窗關門，不偷看美麗又善良的女主人。但有一個叫 Tom 的裁縫師按耐不住，從家中的窗戶往外偷看，之後他就遭到天譴失明了。所以從此時開始，偷窺狂就叫 peeping Tom，門上可偷窺的小洞孔就叫 peephole。

- [peeping Tom 的典故](#)

後來 peephole 廣泛用於電腦科學的術語。

這頁列出的程式碼為 ARM 組合語言，可參照 [手機裡頭的 ARM 處理器](#) 系列講座中關於 conditional execution 的部分

[ Page 73 ]

- 「編譯器無法 inline 一個外部函式，解法為 LTO (Link Time Optimization)」什麼意思？
  - 投影片: [Optimizing large applications](#)
  - 延伸閱讀: [Linktime optimization in GCC](#)

LTO 帶來的提昇，可參考 [gcc-5 的釋出說明](#)：(和 Firefox 有關)

- During link-time optimization of Firefox, this pass unifies about 31000 functions, that is 14% overall.
- About 50% of virtual calls in Firefox are now speculatively devirtualized during link-time optimization.
  - vtable
- GCC 前端:
  - AST
    - ref: [http://www.diku.dk/~torbenm/Basics/basics\\_lulu2.pdf](http://www.diku.dk/~torbenm/Basics/basics_lulu2.pdf)
    - 3.17.2 Abstract syntax
  - "The syntax trees described in section 3.3.1 are not always optimally suitable for compilation. They contain a lot of redundant information: Parentheses, keywords used for grouping purposes only, and so on."
  - Abstract syntax keeps the essence of the structure of the text but omits the irrelevant details. An abstract syntax tree is a tree structure where each node corresponds to one or more nodes in the (concrete) syntax tree"
- GCC中端:
  - Gimple + Tree SSA Optimizer

- Gimple: 只能有兩個運算元
- GCC後端:
  - RTL
  - 使用 virtual register (可有無限多個 registers)
  - register allocation (virtual register => hard register)
  - instruction scheduling (pipeline scheduling)
  - register allocation
  - pipeline scheduling
  - peephole optimization

[ Page 78 ]

- 搭配閱讀 [LLVM 總是打開你的心：從電玩模擬器看編譯器應用實例](#)
- 

- Pointer Aliasing
- Strict Aliasing 延伸

## GNU Toolchain

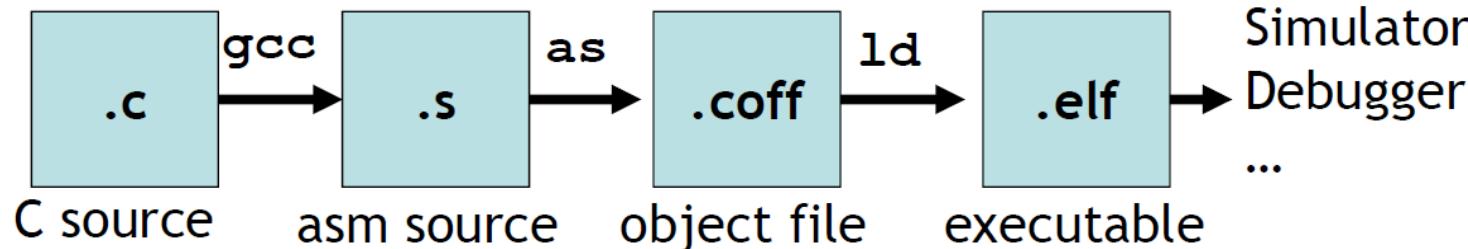
---

- gcc : GNU compiler collection
- as : GNU assembler
- ld : GNU linker
- gdb : GNU debugger

介紹完編譯工具後，來講點大概編譯的流程還有格式

## Compilation flow

先來看這張圖：



.c 和 .s 比較常見先略過，所以就解釋一下 .coff 和 .elf 是什麼：

- **COFF (common object file format)**：是種用於執行檔、目的碼、共享函式庫 (shared library) 的檔案格式
- **ELF (extended linker format)**：現在最常用的文件格式，是一種用於執行檔、目的碼、共享函式庫和核心的標準檔案格式，用來取代COFF

## GAS syntax (AT&T)

```
.file "test.s"
.text
.global main
.type main, %function
main:
    MOV R0, #100
    ADD R0, R0, R0
    SWI #11
.end
```

由一個簡短的 code 來介紹，在程式的 section 會看到 `.`，是定義一些 control information，如 `.file`，`.global` 等

- `%function` : 是一種 type 的表示方式 `.type` 後面可以放 function 或者是 object 來定義之
- `SWI #11` : 透過 software interrupt 去中斷現有程式的執行，通常會存取作業系統核心的服務 (system call)
- `.end` : 表示是 the end of the program

注意，在後來的 ARM 中，一律以“SVC”(Supervisor Call)取代“SWI”，但指令編碼完全一致

[==> ARM Instruction Set Quick Reference Card](#)

以下簡介 ELF 中個別 section 的意義：(注意: ELF section 的命名都由 `.` 開頭)

- `.bss` : 表示未初始化的 data，依照定義，系統會賦予這些未初始化的值 0
- `.data` : 表示有初始過的 data
- `.dynamic` : 表示 dynamic linking information
- `.text` : 表示“text”或者是 executable instructions

## 寫程式的要點

- 程式是拿來實現一些演算法 (想法) 進而去解決問題的
- 可以透過 Flow of control 去實現我們的程式
  - Sequence
  - Decision: if-then-else, switch
  - Iteration: repeat-until, do-while, for
- 將問題拆成多個更小而且方便管理的單元 (每一個單元或稱 function，盡量要互相獨立)
- Think: In C, for / while ?

## Procedures

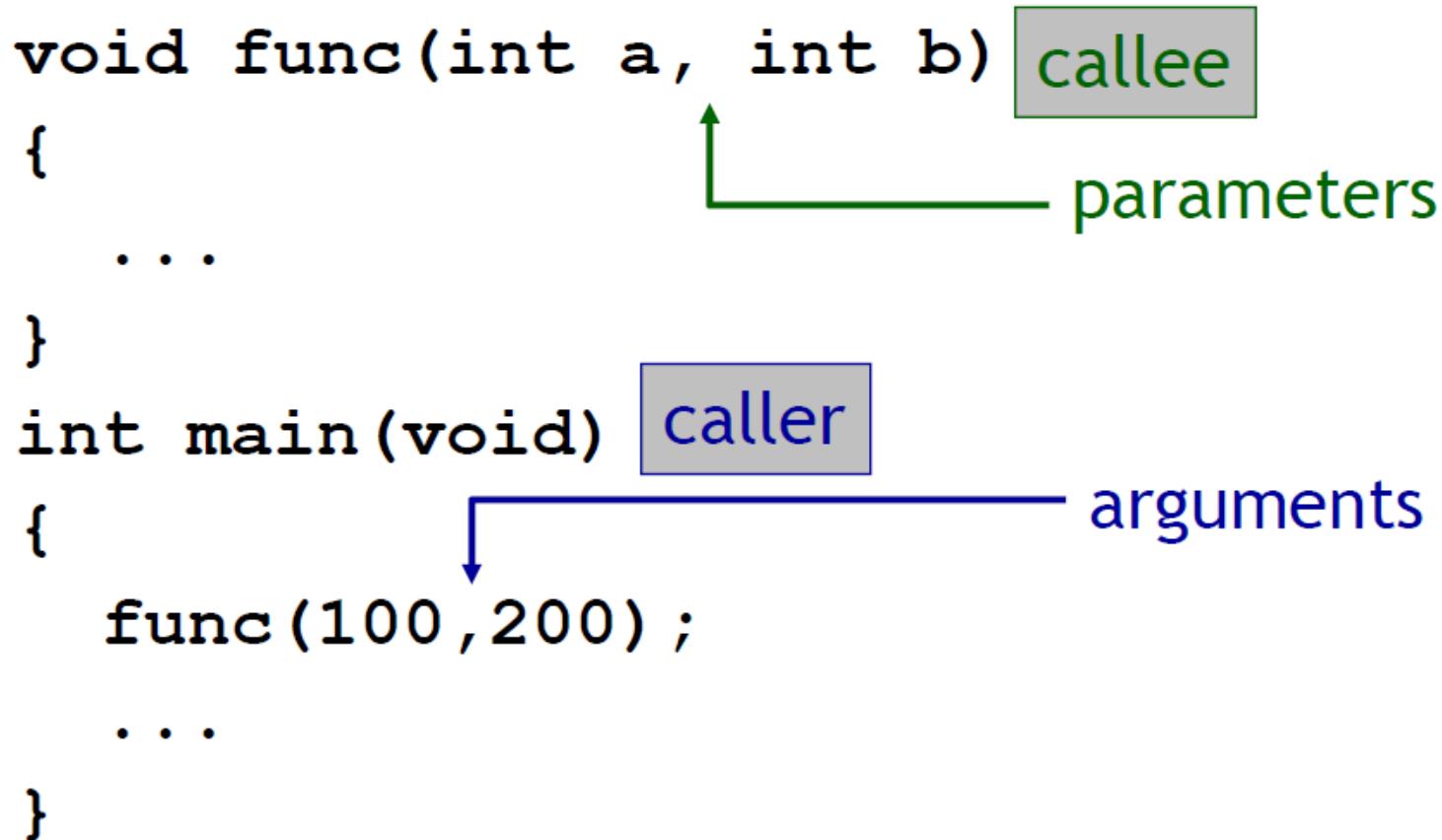
釐清用語：

- Arguments: expressions passed into a function
- Parameters: values received by the function
- Caller & callee
  - 呼叫者(通常就是在其他函式呼叫的function) vs. 被呼叫者(被呼叫的 function)

“argument” 和 “parameter” 在中文翻譯一般寫「參數」或「引數」，常常混淆

“argument”的重點是「傳遞給函式的形式」，所以在 C 語言程式寫 `int main(int argc, char *argv[])` 時，我們稱 `argc` 是 argument count，而 `argv` 是 argument vector

“parameter”的重點是「接受到的數值」，比方說 C++ 有 parameterized type，就是說某個型態可以當作另外一個型態的「參數」，換個角度說，「型態」變成像是數值一樣的參數了。



在撰寫程式常常會使用呼叫 ( call )，在上圖中高階語言直接將參數傳入即可，那麼在組語的時候是如何實作的呢？是透過暫存器？Stack？memory？In what order？我們必須要有個 protocol 來規範

### ARM Procedure Call Standard (AAPCS)

- ARM Ltd. 定義一套規則給 procedure entry 和 exit
  - Object codes generated by different compilers can be linked together
  - Procedures can be called between high-level languages and assembly
- AAPCS 是 [Procedure Call Standard for the ARM® Architecture](#) 的簡稱，從 10 年前開始，全面切換到 EABI (embedded ABI)

- 過去的 ARM ABI 稱為 oabi (old ABI)，閱讀簡體中文書籍時，要格外小心，因為資訊過時
- APCS 定義了
  - Use of registers
  - Use of stack
  - stack-based 資料結構型式
  - argument passing 的機制
    - first four word arguments 傳到 R0 到 R3
    - 剩餘的 parameters 會被 push 到 stack (參數依照反過來的排序丟入堆疊中)
    - 少於 4 個 parameters 的 procedure 會比較有效率

Register	APCS name	APCS role
0	a1	Argument 1 / integer result / scratch register
1	a2	Argument 2 / scratch register
2	a3	Argument 3 / scratch register
3	a4	Argument 4 / scratch register
4	v1	Register variable 1
5	v2	Register variable 2
6	v3	Register variable 3
7	v4	Register variable 4
8	v5	Register variable 5
9	sb/v6	Static base / register variable 6
10	sl/v7	Stack limit / register variable 7
11	fp	Frame pointer
12	ip	Scratch reg. / new sb in inter-link-unit calls
13	sp	Lower end of current stack frame * 特殊功用的 registers * 如果存取正確，可用來當作 temporary variables
14	lr	Link address / scratch register
15	pc	Program counter

- Return value
  - one word value 存在 R0
  - 2 ~ 4 words 的 value 存在 ( R0-R1, R0-R2, R0-R3)

以下測試一個參數數量為 4 和 5 的程式:

```

int add4(int a, int b, int c , int d) {
    return a + b + c + d;
}

int add5(int a, int b, int c , int d, int e) {
    return a + b + c + d + e;
}

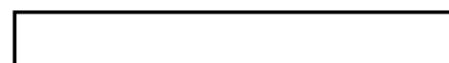
```

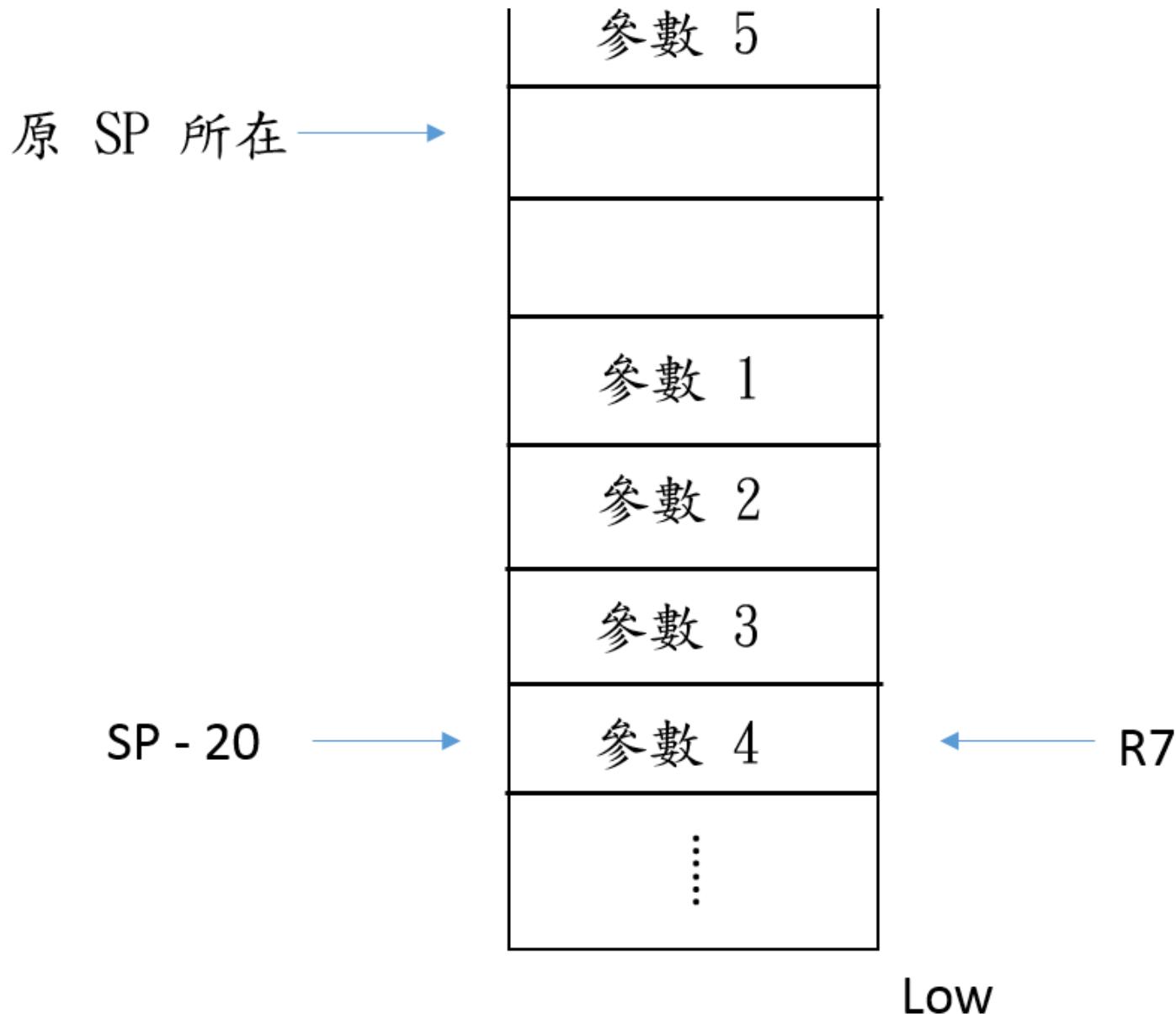
程式編譯後，用 objdump 組譯會得到類似以下：

<pre> 000103c0 &lt;add4&gt;: 103c0: b480      push   {r7} 103c2: b085      sub    sp, #20 103c4: af00      add    r7, sp, #0 103c6: 60f8      str    r0, [r7, #12] 103c8: 60b9      str    r1, [r7, #8] 103ca: 607a      str    r2, [r7, #4] 103cc: 603b      str    r3, [r7, #0] 103ce: 68fa      ldr    r2, [r7, #12] 103d0: 68bb      ldr    r3, [r7, #8] 103d2: 441a      add    r2, r3 103d4: 687b      ldr    r3, [r7, #4] 103d6: 441a      add    r2, r3 103d8: 683b      ldr    r3, [r7, #0] 103da: 4413      add    r3, r2 103dc: 4618      mov    r0, r3 103de: 3714      adds   r7, #20 103e0: 46bd      mov    sp, r7 103e2: f85d 7b04  ldr.w r7, [sp], #4 103e6: 4770      bx    lr </pre>	<pre> 000103e8 &lt;add5&gt;: 103e8: b480      push   {r7} 103ea: b085      sub    sp, #20 103ec: af00      add    r7, sp, #0 103ee: 60f8      str    r0, [r7, #12] 103f0: 60b9      str    r1, [r7, #8] 103f2: 607a      str    r2, [r7, #4] 103f4: 603b      str    r3, [r7, #0] 103f6: 68fa      ldr    r2, [r7, #12] 103f8: 68bb      ldr    r3, [r7, #8] 103fa: 441a      add    r2, r3 103fc: 687b      ldr    r3, [r7, #4] 103fe: 441a      add    r2, r3 10400: 683b      ldr    r3, [r7, #0] 10402: 441a      add    r2, r3 10404: 69bb      ldr    r3, [r7, #24] 10406: 4413      add    r3, r2 10408: 4618      mov    r0, r3 1040a: 3714      adds   r7, #20 1040c: 46bd      mov    sp, r7 1040e: f85d 7b04  ldr.w r7, [sp], #4 10412: 4770      bx    lr </pre>
---	--

紅框標注的是比左邊多出的程式碼，從這裡可以看到參數 1-4 是存在 R0-R3，而第 5 個參數存在原本 sp + 4 的位置，隨著程式碼進行 R0-R3 存在 stack 中，圖下為 stack 恢復前的樣子：

High



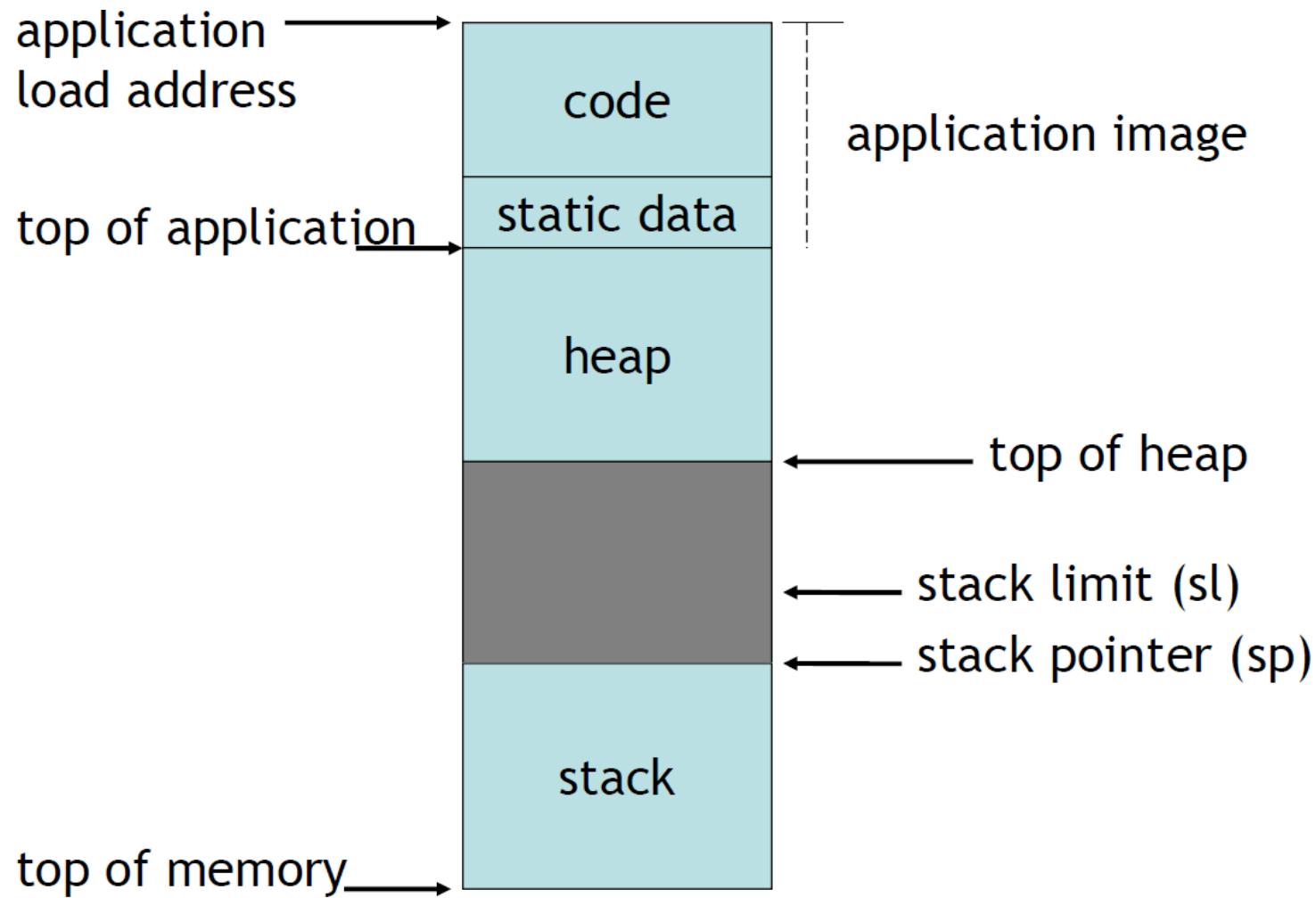


因此若寫到需要輸入 5 個或以上的參數時，就必須存取外部記憶體，這也導致效能的損失。

- 例如 xorg xserver 的最佳化案例

## Standard ARM C program address space

下圖為 ARM C program 標準配置記憶體空間的概念圖:



## Accessing operands

通常 procedure 存取 operands 透過以下幾種方式:

- An argument passed on a register : 直接使用暫存器
- An argument passed on the stack : 使用 stack pointer (R13) 的相對定址 (immediate offset)
- A constant : PC-relative addressing
- A local variable : 分配在 stack 上，透過 stack pointer 相對定址方式存取
- A global variable : 分配在 static area (就是樓上圖片的 static data)，透過 static base (R9) 相對定址存取

用幾張圖來表現出存取 operands 時，stack 的變化:

圖下為 passed on a register :

**main:**

LDR R0, #0

...

BL func

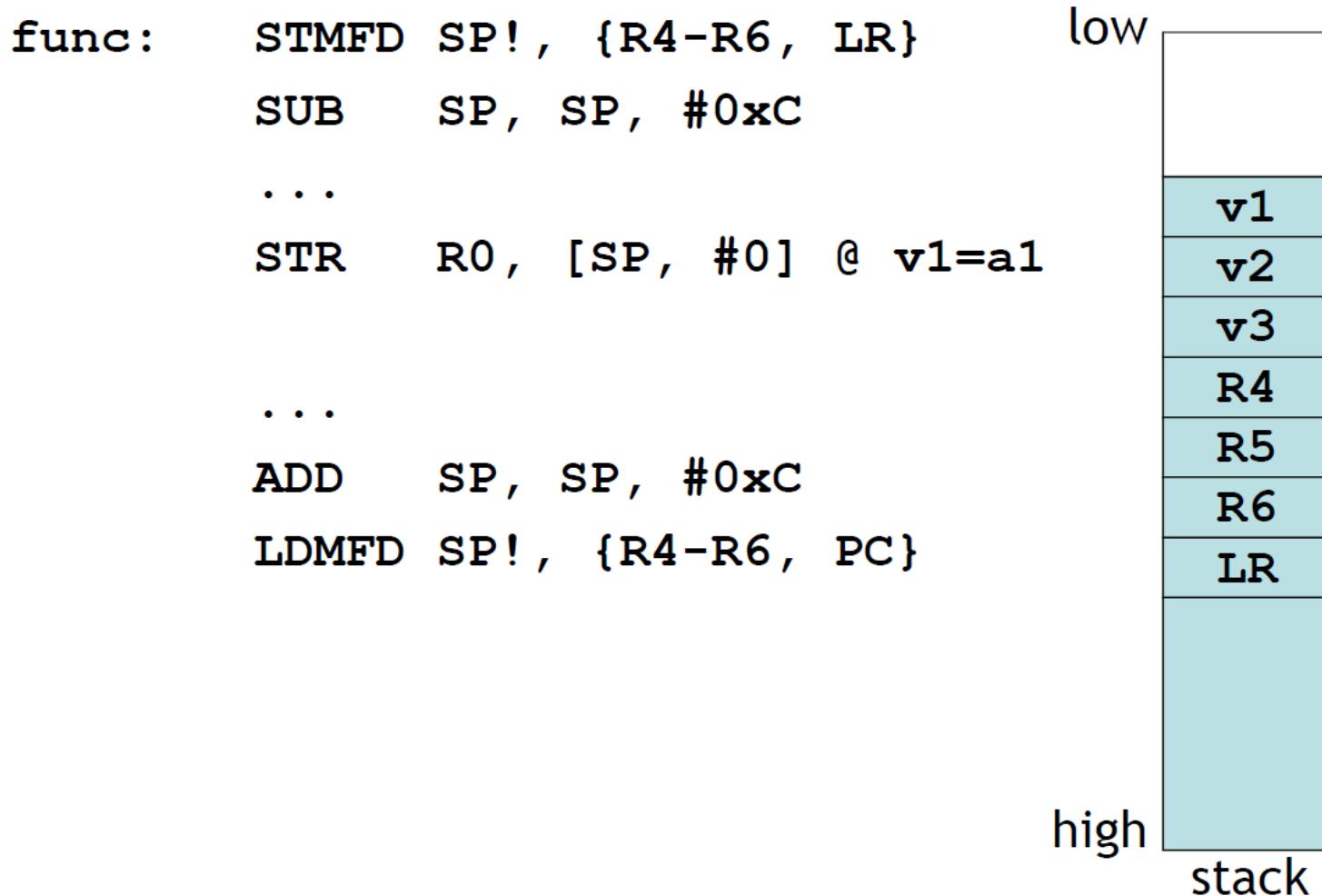
...

low

high

stack

圖下為存取 local variables :



## Target Triple

在使用 Cross Compiler 時，gcc 前面總有一串術語，例如：

- arm-none-linux-gnueabi-gcc

這樣 `arm-none-linux-gnueabi-` 稱為 target triple，通常規則如下：

```
<target>[<endian>] [-<vender>]-<os>[-<extra-info>]
```

- vendor 部份，常見填入 vendor 而已，但沒有一定要有 vendor 的資訊，於是可忽略或填入 none
- extra-info 部份大部份是在拿來描述用的 ABI 或著是 library 相關資訊

先以常見 x86 的平台為例子：

gcc 搭配編譯參數 -v 可見以下：

```
Target: x86_64-redhat-linux
```

<target>-<vendor>-<os> 的形式

Android Toolchain:

```
Target: arm-linux-androideabi
```

<target>-<os>-<extra-info>

androideabi：雖然 Android 本身是 Linux 但其 ABI 細節跟一般 linux 不同

Linaro ELF toolchain:

```
Target: arm-none-eabi
```

<target>-<vender>-<extra-info>

- vender = none

- extra-info = eabi

Linaro Linux toolchain:

```
Target: arm-linux-gnueabihf
```

```
<target><endian>-<os>-<extra-info>
```

- extra-info:
- eabi: EABI
- hf: Hard float, 預設有 FPU
- soft (GPR), softfp (GPR -> float register), hardfp

Linaro big-endian Linux toolchain:

```
Target: armeb-linux-gnueabihf
```

```
<target><endian>-<vender>-<extra-info>
```

- endian = be = big-endian

Buildroot 2015-02

```
Target: arm-buildroot-linux-uclibcgnueabi
```

```
<target>-<vender>-<os>-<extra-info>
```

- extra-info: uclibc 用 uclibc (通常預設是 glibc, uclibc 有些細節與 glibc 不同)

- gnu: 沒有特別意義
- eabi: 採用 EABI

由以上眾多 pattern 大概可以歸納出一些事情：

1. vendor 欄位變化很大，os 欄位可不填；
2. 若不填的話，通常代表 Non-OS (Bare-metal)

source: <http://kitoslab.blogspot.tw/2015/08/target-triple.html>

## 編譯器原理

---

為何我們要理解編譯器？這是電腦科學最早的思想體系

- 從理解動態編譯器 (如 Just-in-Time) 的運作，可一路探索作業系統核心, ABI, 效能分析的原理

[ [Interpreter, Compiler, JIT from scratch](#) ]

- 碎形程式 in C (Brainf\*ck 的版本作為 benchmark)

[ [Virtual Machine Constructions for Dummies](#) ]

[ [Page 14](#) ]

- 為什麼說這是 gray area ? 以及 cell type 指的是什麼?
- cell 是 Brainf\*ck 程式的中間狀態，請對照學習計算理論，裡面提到 Turing machine。而 cell type 自然就是說要如何表達 (represent) cell 呢？8-bit, 16-bit, 32-bit 等等
- 解釋 I/O 的 gray area 之前，先想想為何 C 語言標準函式庫的 `getc()` 返回型態是 `int`，而非 `char` 呢？因為要處理 `EOF` (詳情自行 `man getc`)
- 但 `EOF` 在 Braif\*ck 的 I/O 操作沒有具體定義

# JIT 實做案例

---

- [jit-construct](#) : Brainf\*ck
  - [suhorng-jit-construct / 開發紀錄\(B\)](#)
  - [ARM 版本的 Brainf\\*ck JIT compiler](#)
- [rubi](#) : Ruby-like JIT compiler
  - [Rubi JIT 編譯器的架構與設計原理](#)
- [AMaCC](#) : 極小的 C 編譯器 (實際上是個 JITC)
  - [背景知識解說: 手把手教你構建 C 語言編譯器](#)
  - [C 編譯器原理和案例分析](#)

## TODO

- [Introduction to Compiler Development](#)
- [Missed optimizations in C compilers](#)
- [Header Time Optimization \(HTO\)](#)
- [Traditional Unix Toolchains](#)

---

Published on  HackMD

 182.1k  10  