x86 彙編

Oct 28, 2014

kiterunner_t

TO THE HAPPY FEW

C、彙編代碼、機器代碼之間的關係。電腦實際執行的是機器代碼，通過位元組序列編碼實際的硬體操作（包括資料處理、記憶體管理、記憶體讀寫、網路通信等）。編譯器基於高階語言的語法規範、目的機器的指令集以及作業系統的規範，通過一系列操作產生機器代碼。彙編代碼是機器代碼的文本表示，是人類可讀的機器代碼，因此機器代碼和彙編代碼是一一對應的。GCC（C 語言編譯器）通過前置處理器、彙編器、連結器、載入器等工具將 C 原始程式碼編譯成機器代碼，並通過作業系統在機器上實際運行。本文通過將高階語言與低階語言的對應關係進行翻譯，希望能夠理解高階語言的實際執行過程、在並行時代能夠更直接的與機器進行交流。

學習彙編是為了閱讀和理解編譯器產生的代碼。基於以下一些理由：1) 閱讀編譯器產生的彙編代碼，我們能夠理解編譯器的優化能力，分析代碼中隱含的低效率。2) 高階語言的抽象層會隱藏我們想要理解的有關程式運行時行為的資訊，高階語言通過編譯器實現了低階語言不具備的類型檢查，使人們能更加有效的程式設計。3) 程式遭受攻擊的許多方式中，涉及到程式存儲運行時控制資訊方式的細節，瞭解這些漏洞是如何出現的，以及如何預防，需要具備程式機器級表示的知識。

本文介紹基於 Intel 的彙編知識。Intel 的 32 位元體系結構為 IA-32，通常稱為 x86。Intel64 位元體系結構被稱為 EM64T（之所以不用 IA-64，是因為該名字已被用於另一個沒有被市場認可的 Itanium 了，反而讓 AMD 取得了領先），通常被稱為 x86-64。本文以 x86 和 x86-64 分別表示這兩個概念。

機器代碼通過兩個最基本的抽象模型模型得以在作業系統中無縫的運行：指令集體系結構和虛擬位址空間。

指令集體系結構，即 ISA：定義了機器級程式的格式和行為，描述了處理器狀態、指令的格式，每條指令對狀態的影響。大多數 ISA，將程式的行為描述成循序執行的。處理器硬體遠比描述的精細複雜，它們可以併發地執行許多指令，但是可以採取措施保證整體行為與 ISA 指定的循

序執行完全一致。目前，指令集有 CISC 和 RISC 之分，實際上目前 CISC 在微指令階段也都翻譯成了 RISC 樣子。

ISA 定義了一組在機器代碼中可見的用於保存處理器狀態和執行過程中臨時資料的寄存器，通常包括以下類型：

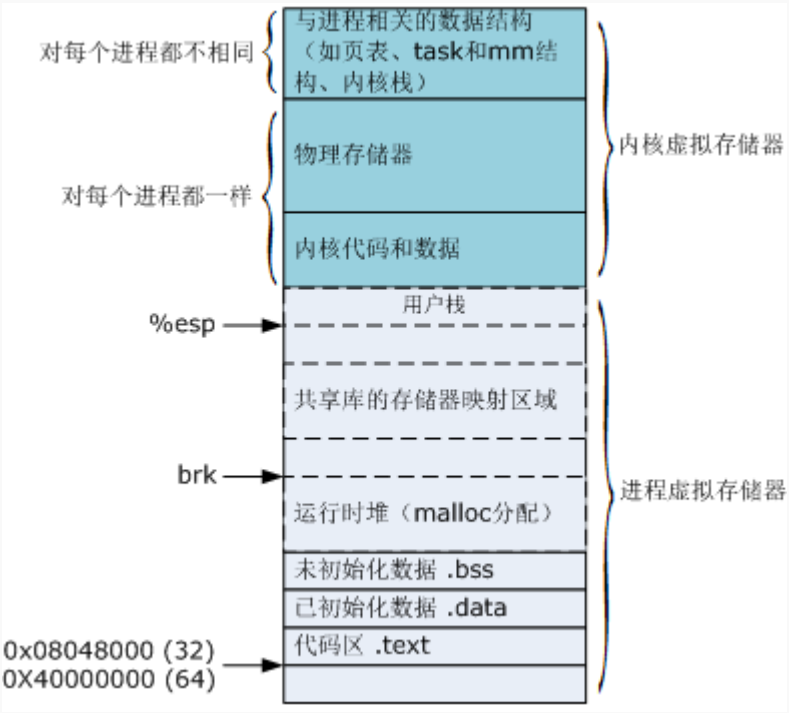
- 程式計數器，指示將要執行的下一條指令在記憶體中的位址（由此可見，代碼也只不過是資料罷了）。
- 整數寄存器檔，這些寄存器用於存儲位址或整數，有的用於記錄某些重要的程式狀態，如棧指標，有的用於保存臨時資料，如過程的區域變數、函數的返回值等。
- 條件碼寄存器，保存最近執行的算術或邏輯指令的狀態資訊，用來實現控制或資料流程中的條件變化。
- 浮點寄存器。

ISA 定義了一些指令類型，包括算術邏輯指令，控制指令，如分支跳轉、迴圈等，資料傳送指令。x86 中，算術邏輯指令的運算元可以是寄存器，也可以是記憶體；而在 RISC 巨集，算術邏輯指令只能是寄存器，故需要用 load/store 指令載入和存儲資料。對於指令運算元個數，早期，X86 電腦寄存器較少，僅有兩個運算元；後來擴展指令 SSE 等才用了 3 個；多數 RISC 採用 3 個運算元。運算元順序，在 x86 目的運算元在前。

指令集除了定義了狀態寄存器和指令，還包括了其他一些，包括定址方式、機器字長、定址方式、大小端等。指令可定址的地方有立即數、寄存器、記憶體。x86 定址方式很複雜，而 RISC 較簡單。大小端問題，Intel 採用小端，很多處理器採用了可配置的大小端方式。機器字長表示處理器一次處理資料的長度，主要由寄存器和運算器決定，常見為 32 位、64 位。32 位元處理器的位址匯流排為 32 位元，可定址範圍是 4GB，資料匯流排高於機器字長，目的是一次讀取更多資料，有 64 位元、128 位等。64 位處理器可定址範圍在目前來看幾乎是無窮大，目前僅用到 48 位元，故位址匯流排是 48 位元，可定址範圍為 256GB。

機器級程式使用的記憶體位址是虛擬位址，提供的記憶體模型看上去是一個非常大的位元組陣列。記憶體系統的實際實現是將多個硬體記憶體和作業系統軟體組合起來。機器代碼本身並不包含資料類型的任何資訊，編譯器負責這個任務。C 語言通過編譯器提供了一種模型，可以在記憶體中聲明和分配各種資料類型的物件（包括基底資料型別和聚合資料類型），在機器代碼看來，只是一串連續的位元組陣列，甚至大多數時候機器代碼對於有、不帶正負號的整數、指標等都不加區分。同時，機器代碼不會感覺到自己使用的是虛擬位址，機器硬體、編譯器、作業系統、

載入器組合起來，為機器代碼提供了一個統一的執行環境，讓機器代碼不用在執行時區分自己究竟放在實際物理記憶體的什麼地方。下圖 1 展示了一個進程的虛擬記憶體鏡像：



組合語言的格式主要有兩種，ATT 與 Intel。主要區別如下：

- Intel 代碼中省略了指示大小的尾碼，如 mov，而不是 movl。
- Intel 代碼中生路了寄存器名字前面的%符號。
- Intel 使用不同的方式來描述記憶體中的位置。如 DWORD PTR [ebp+8]，而不是 8(%ebp)。
- 多個運算元的指令，運算元順序相反。

GCC、objdump 等一些常用工具默認產生的彙編格式是 ATT 的，可以使用編譯-masm=intel 改成 Intel 格式。Microsoft、Intel 文檔使用 Intel 彙編格式。本文採用在 Linux 下通用的 ATT 彙編格式。

GCC 是 Linux 下最通用的 C 語言編譯器，包含了很多優化選項，通常使用被稱為優化級別的優化選項集合來編譯代碼，常用的優化級別為 O1（編譯開發代碼）、O2（編譯生產代碼）。更多的編譯選項可以參考”Using the GNU Compiler Collection”。本文使用的作業系統和編譯器環境如下（32 位和 64 位，都是虛擬機器）：

```
krt@debian:~$ uname -a
```

```
Linux debian 2.6.32-5-686 #1 SMP Sun Sep 23 09:49:36 UTC 2012
i686 GNU/Linux

krt@debian:~$ gcc --version

gcc (Debian 4.4.5-8) 4.4.5
```

```
krt@krt:~/github/krt/l/asm$ uname -a

Linux krt 3.2.0-4-amd64 #1 SMP Debian 3.2.51-1 x86_64
GNU/Linux

krt@krt:~/github/krt/l/asm$ gcc --version

gcc (Debian 4.7.2-5) 4.7.2
```

除了 GCC 編譯器，以下一些工具也需要掌握：gcc/cpp/gas/ld，objdump，readelf，gdb 等。GAS 產生的彙編中，以“.”開頭的行都是指導彙編器和連結器的命令。

本系列文章大致分為以下幾個部分（實際操作的時候，可能有些部分需要分成好幾篇獨立的 post）：

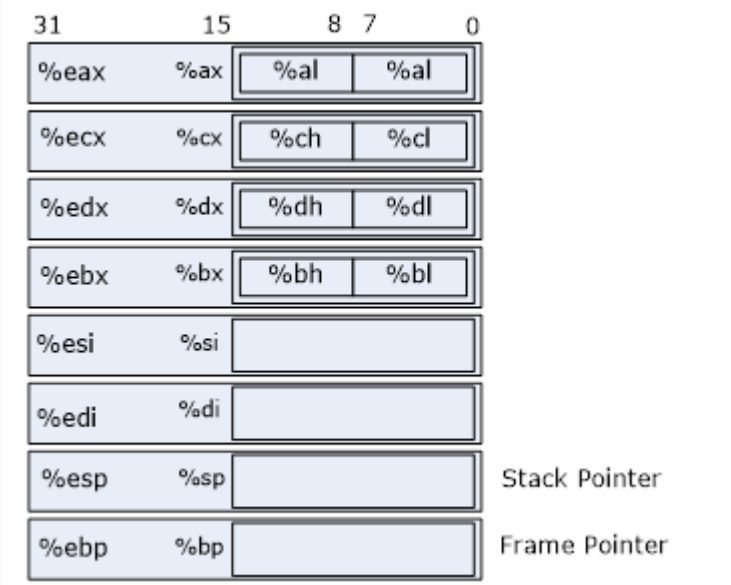
- x86 體系程式設計模型，主要介紹 x86 體系機構的寄存器、定址模式、基本指令等內容；x86 體系程式存儲，主要介紹 x86 下 C 語言各種資料類型的表示；x86 體系程式過程，主要介紹 x86 下函數和流程控制的實現。
- x86-64 體系介紹，與 x86 進行區分。
- 浮點運算，介紹 x86/x86-64 體系下的浮點體系，x87 和 SSE。
- 其他，包括 GCC 內聯彙編、與並行程式設計相關的一些指令，如原子操作、記憶體柵欄等、彙編級別代碼優化的一些技術等等。

代碼參考這裡 https://github.com/kiterunner-t/krt/tree/master/l/coding 和這裡 https://github.com/kiterunner-t/krt/tree/master/l/asm，以及這裡 https://github.com/kiterunner-t/ktest/tree/master/src。

1 x86 程式設計模型/抽象

1.1 x86 體系結構

x86 為開發人員提供了以下處理器可見的寄存器：8 個 32 位元的通用寄存器模型，存儲整數和指標；條件碼寄存器；浮點寄存器 x87/SSE；程式計數器%eip，也稱%pc。通用寄存器如下圖所示：



- %eax, %ecx, %edx 是調用者保存；%ebx, %esi, %edi 屬於被調用者保存。
- %ebp 和%esp 用來保存棧幀位址。
- %eax 用來返回值，若需要 8 個位元組，則聯合使用%edx。
- 8 個寄存器都可以操作其對應的低 16 位，前四個可以訪問低 16 位元的兩個位元組。

指令運算元有三種：立即數，寄存器和記憶體引用，符號表示為$1, R[eax], M[addr]。

1.2 定址模式

通用的定址模式是 Imm(B, I, s), s 為比例因數，必須是 1、2、4 或 8，基址寄存器為 B，變址寄存器為 I，則

格式	操作数值	名称
$Imm	Imm	立即数寻址
B	R[B]	寄存器寻址
Imm	M[Imm]	绝对寻址
(B)	M[R[B]]	间接寻址
Imm(B)	M[Imm + R[B]]	（基址+偏移量）寻址
Imm(B, I)	M[Imm + R[B] + R[I]]	变址寻址
Imm(B, I, s)	M[Imm + R[B] + s * R[I]]	比例变址寻址

變址定址中，立即數 Imm 可以省略，變形為(B, I)。比例變址定址中 Imm 和基址寄存器可以省略，變形為(, I, s), Imm(, I, s), (B, I, s)。相較於 RISC ，x86 的定址模式較複雜，但也對陣列的位址運算等提供了強有力的工具。

1.3 基本指令

x86 體系基本指令如下：

- 資料傳送指令 mov, movs, movz
- 棧操作指令 pushl, popl
- 載入有效位址 leal
- 算術指令 inc, dec, neg, add, sub, imul （缺少除法和求模指令）
- 邏輯操作指令 xor, or, and, not
- 移位操作 sal, shl, sar, shr
- 特殊的算術操作 imull, mull, cltd, idivl, divl

注意這些指令中，有很多指令中的運算元是區分位元組、字、雙字、四字的，分別用尾碼 b、w、l、q 來表示。

指令	效果	描述
leal S, D	D <- &S	加载有效地址
inc D	D <- D+1	加 1
des D	D <- D-1	减 1
neg D	D <- -D	取负
not D	D <- ~D	取补
add S, D	D <- D+S	加
sub S, D	D <- D-S	减
imul S, D	D <- D*S	乘
xor S, D	D <- D^S	异或
or S, D	D <- D|S	或
and S, D	D <- D&S	与
sal k, D	D <- D<<k	左移
shl k, D	D <- D<<k	左移（等同于 sal）
sar k, D	D <- $D \ll_A k$	算术右移，填符号位
shr k, D	D <- $D \ll_L k$	逻辑右移，填 0

leal(load effective address)實際是 mov 指令的變形。C 中的位址引用操作符&就是通過該指令來實現的。其目的運算元必須是寄存器。通常還是用該指令來進行算數運算。

移位元指令的移位元量只考慮低 5 位，即只允許 0 到 31 位的移位。移位量是立即數或者放在單字節寄存器%cl 中。左移的兩個指令效果一樣，都是將右邊填 0。算術右移 sar 填符號位元，而邏輯右移 shr 執行邏輯移位元，填 0。如下面這段代碼所示，左移 32 位等於左移 0 位，左移 33 位等於 33 mod 32，即左移 1 位：

```
/* coding/bit_op.c */

  int w = 0xFFFFFFFF;

  int k = 32;

  printf("    w: %x\n", w);

  printf("w<<32: %x\n", w << k);

  printf("w<< 1: %x\n", w << 1);

  printf("w<<33: %x\n", w << (k+1));
```

符號擴展和零擴展。movs 和 movz 指令將一個較小的資料複製到一個較大的資料位置，高位用符號擴展（movs）或零擴展（movz）進行填

充。零擴展就是所有高位用零填充。符號擴展是目的數所有高位用來源資料的最高位元進行填充。彙編中什麼時候使用算術右移？什麼時候使用邏輯右移？根據有符號數與無符號數？是的。op.c

資料等二元指令限制：(1) 兩個運算元不能都是記憶體；(2) 利用資料傳送指令強制類型轉換時，操作應先改變大小再改變符號。

```
movsbl %al, (%edx)
```

特殊的算術操作指令支援 32 位元數位的全 64 位元乘積及整數除法。

指令	效果	描述
imull S mull S	R[%edx]:R[%eax] <- S * R[%eax]	有/无符号全 64 位乘法
cltd	R[%edx]:R[%eax] <- SignExtend(R[%eax])	将%eax 符号扩展为四字
idivl S divl S	R[%edx] <- R[%edx]:R[%eax] mod S R[%eax] <- R[%edx]:R[%eax] / S	有/无符号除法

除法指令中 GCC 會先產生被除數，無符號除法直接將%edx 置為 0；有符號擴展有兩種方式（這兩種方式都會被 GCC 採用）。

```
movl 8(%ebp), %edx

movl %edx, %eax

sarl $31, %edx
```

```
movl 8(%ebp), %eax

cltd
```

一些指令收集：

```
; pushl %ebp

subl $4, %esp

movl %ebp, (%esp)
```

```
; popl %eax
```

```
movl (%esp), %eax

addl $4, %ebp



leal 7(%edx, %edx, 4) ; x in %edx, so 5x + 7



xorl %edx, %edx ; 清零，指令長度更少
```

2 資料存儲

2.1 指針

指標統一了對不同資料類型中元素的引用，是 C 語言的一個重要特徵。指標就是位址，常由&操作符產生，對應的機器指令是 leal。間接引用就是對該位址的值進行訪問，訪問方式是將指標放在寄存器中，然後通過記憶體引用使用這個寄存器。指標的間接引用表明了對該位址的物件的訪問。用*操作符來表示間接引用。

指標的類型，每個指標都應有一個類型，表明了指向的物件的資料類型。機器碼沒有任何關於類型系統的資訊，這都是由編譯器維護的。void * 為通用指針。

指標的強制轉換只是向編譯器表明該位址上的物件的資料類型變化了，並不改變指針本身的值。

指標運算運算會根據該指標引用的資料類型的大小進行伸縮，陣列的位址就是通過 C 語言的該特性實現的。p + i 值為 p + sizeof(*p) * i。

call 的間接程序呼叫支援函數指標。函數指標 datatype_fp.c，在該段代碼中實際的彙編代碼為

```
movl    $sum, -12(%ebp)

movl    $2, 4(%esp)

movl    $1, (%esp)

movl    -12(%ebp), %eax
```

```
call    *%eax                ; 區域變數 f 代表了函數 sum 的位址，存放在
%eax 中

movl    $2, 4(%esp)

movl    $1, (%esp)

call    sum
```

注：在 C++中支援引用，從底層機器碼的角度來說，引用也是指針。（這句話並不太對）

2.2 代碼

代碼也是資料，在 C 中，通過使用函數指標，可以將代碼作為一種程式存儲的資料類型。在程式中也可以通過某些方式直接修改指令，達到某些目的。參考代碼 ktest_stake.c。在這段代碼中，通過修改 pc 目標函數第一個指令，從而達到跳轉到另一個函數代替執行本函數的目的。0xE9 為 x86 體系上直接跳轉指令的二進位表示。

```
echo "jmp 0x08048000" >jmp.s

gcc -m32 -c jmp.s

objdump -d jmp.o
```

有結果

```
00000000 <.text>:

0: e9 fc 7f 04 08       jmp    0x8048001
```

```
  /* only for x86 32/64 */

  *pc++ = 0xE9;

  t = (int) (s - (d + 5));

  *(int *) pc = t;

  strncpy(pc + 4, KTEST_STAKE_FLAG "1", KTEST_STAKE_FLAG_LEN
+ 1);
```

2.3 陣列

陣列是將標量資料聚集成更大資料類型的方式，C 語言中對陣列元素的訪問是通過指標進行的，IA32 提供的定址模式支援有效的對陣列的進行操作，簡單的一條 mov 和 leal 指令就可以完成訪問陣列位址或記憶體引用。

資料類型為 int，大小為 N 的陣列聲明如下，

```
int arr[N];
```

則在記憶體中分配 sizeof(int) * N 的連續存儲區域；識別字 arr 指向該存儲區域的起始位址；第 i 個元素的位址為 arr + sizeof(int) * i，陣列引用 arr[i]等價於*(arr + i)。假設 arr 的值存在%edx，i 放在%ecx，允許縮放的因數 1、2、4、8 覆蓋了所有基本的單一資料型別，則有

```
arr + i - 1     leal -4(%edx, %ecx, 4), %eax

*(arr + i - 1)  movl -4(%edx, %ecx, 4), %eax
```

二維陣列

```
int arr[M][N];
```

位址計算公式為 arr + sizeof(int)(N * i + j)，編譯器實現時通常會利用移位、加法和伸縮的組合來避免計算位址時的乘法開銷。編譯器經常利用陣列的訪問模式優化多維陣列的位址計算，如行、列、對角線的訪問。如代碼 datatype_array_2d.c

變長陣列。C99 支援變長陣列，允許陣列的維度是運算式，僅在陣列被分配的時候才計算其維度。對於編譯器來說，變長陣列與定長陣列本沒有本質區別，只是動態版本必須用乘法指令計算位址，而不能通過移位、加法等來進行優化。通過探索其訪問模式，編譯器通常也會類似與定長陣列一樣優化其位址的計算。datatype_array_dynamic.c

2.4 結構和聯合

C 語言提供了兩種結合不同類型的資料而創建新的資料類型的機制：結構和聯合。

結構將不同資料類型的物件聚集在一個新的資料類型中，結構中各個欄位用名字來引用。

佈局和欄位引用（其實就是位址計算）。結構的所有欄位都放在記憶體中的一個連續區域中，指向結構的指標是該結構的第一個位元組的位址；編譯器維護了每個欄位的類型資訊，每個欄位的位元組偏移，以該偏移作為記憶體引用指令中的位移產生對結構元素的引用。機器代碼並不維護結構的任何資訊。

```
typedef struct point_s point_t;

struct point_s {

  int x;

  int y;

};



point_t  a;

point_t *pa = &a;

a.y = 1;

(*pa).y = 1; // pa->y = 1; 括弧是必須的，否則會被當成*(pa.y)，這肯定是非法的
```

需要注意的是由於對齊規則，結構體成員之間可能有空洞。相較於 C++ 和 Java，C 的結構更樸實，前者將資料以及對其進行操作的方法放在一起，形成一個規範，就被成為物件。

offsetof 的實現

```
#define offsetof(type, field) &((((type) *) 0)->(field))
```

聯合提供了一種方式規避 C 語言的類型系統（由編譯器維護），用不同的欄位來引用相同的存儲塊。由於將不同資料類型綁在了一塊，位元組順序就是不得不考慮的問題。聯合的大小是其最大成員的大小。

聯合的應用： 1) 減小互斥欄位的的記憶體佔用。 (2) 訪問不同資料類型的位元模式。

2.5 數據對齊

一些電腦體系結構對基底資料型別合法位址做出了一些限制，要求某種類型物件的位址必須是某個值 K（通常是 2, 4, 8）的倍數，一方面簡化了處理器和記憶體系統之間介面的硬體設計，另一方面更重要的是提高了記憶體系統的性能。大多數 IA32 指令無論資料是否對齊都能正常工作（某些實現多媒體操作的 SSE 指令要求 16 位元組對齊，否則無法正常工作）。

Linux 對齊的策略是 2 位元組的資料類型位址必須是 2 位元組的倍數，即最低位為 0；而更大資料類型的地址都必須是 4 的倍數，最低兩位為 0。與 Windows 不同的是對於 8 位元組的資料類型，Linux 也只要求 4 位元組對齊，對現代處理器來說 Windows 的此時 8 位元組對齊更合理。

常見資料類型對齊的說明 (1) IA32 對棧幀的慣例是確保每個棧幀的長度都是 16 位元組的整數倍，因此在建棧指令中會看見分配過多的棧空間。（注：參數傳遞過程中，是否要求所有的欄位都是 sizeof(int)大小對齊？在 datatype_varg.c 中，明顯看見編譯器進行了該強制對齊要求。）(2) 記憶體分配庫設計返回的指標必須滿足最嚴格的對齊限制，對齊為 4（64 位元機器上為 8 位元組對齊），即低 2 位為 0。有些代碼就利用該特性，直接在指標裡面嵌入一些資訊。 (3) 結構體要求每個元素滿足該域的對齊要求，結構體的起始位址也會因結構陣列要求而必須滿足一定的對齊要求。這些通常會造成結構體中間以及末尾的空洞，因此結構體元素需要合理安排，結構體本身需要按照最大元素的對齊要求進行對齊。如下面代碼，s1 因 j 需要滿足起始位址為 4 的倍數，所以在 c 後填充了一個位元組，4 位元組對齊；s2 因結構陣列時 i 需要滿足 4 位元組對齊，所以在 c 後填充了 3 個位元組，4 位元組對齊。

如下代碼所示

```
struct s1 {

  short i;

  char  c;

  int   j;

}; // IA32: sizeof(struct s1) = 8, align 4



struct s2 {

  int    i;
```

```
  double  j;

  char    c;

}; // IA32: sizeof(struct s2) = 16, align 4
```

2.6 C 資料類型組合的提示

大多數體系結構都存在資料對齊的要求，這有助於 CPU 更高效的訪問資料（不用跨越機器字的邊界做多次訪問，目前 x86 體系上訪問非對齊資料也很高效），也有助於緩存的使用。通過多種填充、重排序等手段，通常能以較低的代價獲取到較高的收益。

但在協定處理、網路傳輸 raw 資料、硬體等方面，可能需要違反資料對齊和填充、重排序等技術，通過#pragma pack 指示符，可以要求編譯器按照任意的邊界對資料進行對齊。

3 程式結構

IA32 提供了基本控制指令，編譯器在此基礎上提供了棧模型完成程序呼叫和高階語言的流程控制功能。

3.1 過程/函數

過程完成的主要功能：一是資料和控制流的改變，二是區域變數的分配和釋放。

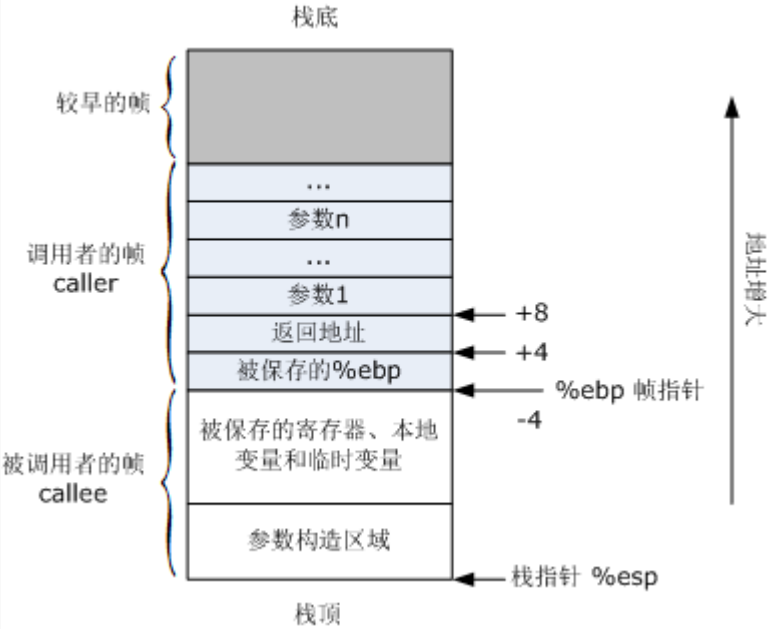
大多數機器只提供控制過程的轉入和轉出指令，區域變數的分配和釋放使用程式棧來實現。IA32 通過棧來支援程序呼叫，編譯器根據一組慣例產生管理棧結構的代碼。單個過程在棧上分配的空間為棧幀，%ebp 為幀指針寄存器，%esp 為棧指針寄存器。棧規則提供了一種機制，每次函式呼叫都有它自己的私有狀態資訊，並提供私有存儲空間，棧空間的分配和釋放與函式呼叫和返回的順序匹配，因此可以利用棧方便的實現自身遞迴（尾遞迴可以優化棧空間的使用）、相互遞迴等更複雜的函式呼叫。

3.1.1 用來實現過程的結構

棧幀。IA32 使用棧來支援程序呼叫。棧的主要功能有：完成過程的參數傳遞、返回資訊、寄存器上下文保存和恢復、本地區域變數的存儲。

需要棧保存區域變數的原因： (1) 沒有足夠的寄存器存放區域變數，這被成為寄存器溢出 register spilling，通常讀記憶體比寫更有效率，故常將唯讀變數溢出。 (2) 有些區域變數是陣列或結構，必須通過陣列或結構引用來訪問。 (3) 需要對區域變數使用位址操作符&，必須能為該區域變數生成一個位址。

棧幀的空間結構。一個過程在棧中存儲的結構被稱為一個棧幀。



棧是從高位址向低位址方向增長的，%esp 指向棧頂，過程執行中會改變，減小%esp 則分配棧空間，增大%esp 則釋放棧空間；%ebp 指向當前棧幀，過程執行時不會改變，棧中變數訪問（除棧頂的 push 和 pop）都是相對於%ebp 來進行的。

棧是向下增長的，棧頂在地址低處。不能訪問棧指標之外的資料，x86-64 中有一個 red zone 可以訪問 128 個位元組的資料。

棧空間分配：將棧指針的值減小或增大完成棧空間的分配和釋放。 一個過程的棧空間分配必須滿足嚴格的對齊要求，GCC 中必須滿足 16 位元組對齊。

寄存器組是唯一能被所有過程共用的資源。在任意給定時刻只能有一個過程是活動的，因此必須按照某種規則來保證程序呼叫時，寄存器的值不會被錯誤的覆蓋。

IA32 採用一致的約定，即寄存器使用慣例，將寄存器分為調用者保存 caller save 和被調用者保存 callee save 兩組，假設在 P 中調用 Q，P 為調用者，Q 為被調用者，則

- %eax, %edx, %ecx 為調用者保存，Q 可以直接使用這些寄存器，而不會破壞 P 所需要的資料；
- %ebx, %esi, %edi 為被調用者保存，Q 在使用這些寄存器之前，必須先保存其值到棧中，Q 返回到 P 之前再恢復回去；
- %ebp, %esp 必須被保存，P 棧幀第一個 4 位元組位置必然為 Q 幀的開始位置；
- 函數返回整數或指標時，使用%eax。

支援程序呼叫的指令

指令	描述	等价指令
call LABEL call *operand	将返回地址%eip 入栈，跳转到调用过程的起始处。返回地址是 call 指令后下一条指令的地址。	
leave	leave 使栈做好返回准备。等价于	movl %ebp, %esp popl %ebp
ret	将返回地址从栈中弹出，并跳转到返回地址执行代码。	
pushl S	双字压栈	R[%esp] <- R[%esp]-4 M[R[%esp]] <- S
popl D	双字出栈	D <- M[R[%esp]] R[%esp] <- R[%esp]+4

3.1.2 過程的實現

一個過程分為三個部分：建棧，主體，結束。參考代碼 proc_32.c

函式呼叫時的棧幀視圖：參數入棧、變參、返回值、傳值調用（傳引用調用）。參數壓棧順序是從右到左，存儲在棧中的位置是最右邊的參數在高地址，而左邊的在低地址，最靠近棧指針的位置。

一個過程的棧幀由 3 部分組成：初始化棧幀，保存被調用者保存寄存器上下文，獲取參數；主體，完成過程的實際計算；結束，恢復棧狀態，過程返回。

程序呼叫前，調用者需要完成：保存調用者保存寄存器上下文，準備參數，保存返回位址。

```
  call next

next:

  popl %eax
```

變參：在 x86 中，函數參數都是通過棧來傳遞的，因此變參實現原理較為簡單，通過類型在棧上就可以推導出參數。參考代碼 kvarg.h.

返回值：返回大型資料（如結構體），不能直接返回陣列，可以將陣列包含在一個結構體內進行返回。結構體指標會作為第一個參數傳遞給被調用者，被調用者 Q 最後返回一個指標到%eax。（C++物件中 this 指標也是如此操作的。）

long long 類型通過%edx, %eax 返回值

GDB 調試代碼時，又是遇到棧被破壞，可以根據棧幀推導出函式呼叫順序。

遞迴、相互調用過程

3.2 流程控制

流程控制主要涉及到以下 3 個方面：

- 機器提供的用於流程控制的條件碼寄存器
- 機器指令 cmp, test, set, 以及其他指令如何設置條件碼
- 編譯器如何利用機器提供的機制實現 C 語言的條件控制。

代碼的執行流程是順序的，機器代碼提供了兩種基本機制實現有條件的行為：測試資料值，根據測試結果改變控制流或資料流程。為實現 C 語言的控制結構，編譯器必須在機器代碼的基礎上產生指令序列。

CPU 維護了一組單個位的條件碼寄存器，用來描述最近一次的算術或邏輯操作的屬性，通過這些寄存器位元可執行條件分支指令。

CF	进位标志位	最近的操作使最高位产生了进位，可用来检查无符号操作数溢出
ZF	零标志位	最近的操作数结果为 0
SF	符号标志位	最近的操作结果为 0
OF	溢出标志位	最近的操作结果有补码溢出——正溢出或负溢出

算術、邏輯和移位元操作都會設置這些條件碼。

逻辑	进位标志和溢出标志会设置为 0
移位	进位标志将设置为最后一个被移出的位，移出标志位 0
inc 和 dec	设置溢出和零标志，但不会改变进位标志

指令	效果	描述
cmp S2, S1	S1-S2	比较
test S2, S1	S1&S2	测试

cmp 和 test 指令只設置條件碼而不改變任何其他寄存器，除此外可以分別等價於 sub 和 and 指令。以下指令通常用來測試寄存器中的的值的符號，test 也通常用於遮罩測試，

```
testl %eax, %eax
```

訪問條件碼：條件碼通常不會直接被讀取，常用的使用方法有 3 種：(1) set 指令，根據條件碼的某個組合，將一個位元組設置為 0 或 1；(2) jmp 有條件跳轉指令，根據條件組合進行代碼跳轉；(3) 條件傳送指令，根據條件組合進行陣列傳送。

這裡機器代碼區分有符號和無符號。

set 指令	jmp 指令	cmov 指令	条件码组合	条件说明	等价后缀
sete D	je Label	cmove S,R	ZF	相等/零	z
setne D	jne Label	cmovne S,R	~ZF	不等	nz
sets D	js Label	cmovs S,R	SF	负数	
setns D	jns Label	cmovns S,R	~SF	非负数	
setg D	jg Label	cmovg S,R	~(SF^OF) &	有符号>	nle
setge D	jge Label	cmovge S,R	~ZF	有符号>=	nl
setl D	jl Label	cmovl S,R	~(SF^OF)	有符号<	nge
setle D	jle Label	cmovle S,R	SF ^ OF	有符号<=	ng
			(SF^OF) | ZF		
seta D	ja Label	cmova S,R		无符号>	nbe
setae D	jae Label	cmovae S,R	~CF & ~ZF	无符号>=	nb
setb D	jb Label	cmovb S,R	~CF	无符号<	nae
setbe	jbe Label	cmovbe S,R	CF	服务号<=	na
			CF | ZF		

一條 set 指令的目的運算元是 8 個單字節寄存器元素之一，或單字節記憶體位置，將這個位置設置為 0 或 1。為了得到一個 32 位元結果，需要對高 24 位元清零。

a < b 的彙編指令

```
cmpl %eax, %edx

setl %al

movzbl %al, %eax
```

跳轉指令：無條件跳轉可以是直接跳轉，也可以是間接跳轉；條件跳轉只能是直接跳轉。直接跳轉是將目標的位址作為指令的一部分編碼的；間接跳轉是從寄存器或記憶體讀出目標位址。直接跳轉的編碼可以使用絕對位址和 PC 相對位址，彙編器和連結器會根據情形選擇適當的跳轉目的編碼。在生成目標代碼時，彙編器會確定帶標號指令的位址，並將跳轉目標編碼為該指令的一部分。

```
jmp  LABEL

jmp *LABEL
```

PC 相對位址是將目標指令的位址與緊跟在跳轉指令後面那條指令的位址差作為編碼。在理解連結過程中，理解這些機器代碼格式的細節是必須的。

編譯器通過條件測試和跳轉組合來實現 C 語言中的條件和迴圈語句。參考代碼

- flow_dowhile.c
- flow_while.c
- flow_for.c
- flow_continue.c

if/else, do/while, while, for 語句的通用轉換，while 和 for 轉換成 do-while 的等價形式。在彙編逆向工程過程中，確定每個變數和寄存器之間的映射關係是關鍵。

通用模板	汇编伪代码

if (test-expr)
  then-statement
else
  else-statement
	t = test-expr;
	if (!t)
	  goto FALSE;
	then-statement
	goto DONE;
	FALSE:
	else-statement;
	DONE:

通用模板	汇编伪代码

do
  body-statement
while (test-expr);
	LOOP:
	  body-statement
	  t = test-expr;
	  if (t)
	  goto LOOP;

while (test-expr)
  body-statement
	t = test-expr;
	  if (!t)
	  goto DONE;
	LOOP:
	  body-statement
	  t = test-expr;
	  if (t)
	  goto LOOP;
	DONE:

for (init-expr; test-expr;
  update-expr)
  body-statement
	init-expr;
	  t = test-expr;
	  if (!t)
	  goto DONE;
	LOOP:
	  body-statement
	  update-expr;
	  t = test-expr;
	  if (t)
	  goto LOOP;
	DONE:

while 的實現 gcc (Debian 4.7.2-5) 4.7.2。gcc –S –m32 fact_while.c 編譯時產生指令序列

```
jmp TEST_COND;

LOOP:

  body-statement

TEST_COND:

  t = test-expr;

  if (t)

    goto LOOP;
```

continue 語句時，for 不能簡單的直接套用 do-while 的模式，如 flow_continue.c 所示

```
  i = 0;

  if (i >= 10)

    goto DONE;


LOOP:

  if (i & 1)

    goto COND_UPDATE;

  sum += i;


COND_UPDATE:

  ++i;

  if (i < 10)

    goto LOOP;


DONE;
```

各種運算子的翻譯：&&

條件傳送指令只支援 16 位元、32 位元的資料長度，不支援 1 個位元組的條件傳送。執行時，處理器讀取源值（寄存器或記憶體），檢查條件碼，然後或者更新目的寄存器，或者保持不變。

```
v = test-expr ? then-expr : else-expr
```

轉換成

```
vt = then-expr

v = else-epr
```

```
t = test-expr

if (t) v = vt
```

條件傳送指令更好的利用了現代處理器的性能特性，它充分利用 CPU 的流水線和分支預測邏輯功能，因為控制流不依賴於資料，流水線總是滿的，不用預測。錯誤預測一次，意味著 20~40 個時鐘週期的浪費。但 GCC 的 x86 平臺基本不支持條件傳送。我在 x86-64 位平臺上編譯成 32 位彙編代碼時，使用-m32 –O2 支持條件傳送，不加-O2 時不支持。

```
gcc -S -m32 absdiff.c

gcc -S -m32 -O2 absdiff.c
```

但是條件傳送不總是有用的

- 無論測試結果如何，條件傳送中的兩個語句都會進行求值，若語句有副作用或錯誤條件，則可能會導致非法行為。如代碼

  xp ? *xp : 0 (x < y) ? (lcount++, y-x) : x – y;
- 條件傳送也不總是會改進代碼效率，例如 then-expr 或 else-expr 的求值需要大量計算，對應條件不滿足時，工作就白費了。編譯器必須在計算浪費和分支預測錯誤之間取得平衡。

switch 語句：GCC 根據開光情況的數量和值的稀疏程度翻譯開關。當開關數量較多時，並且值的範圍跨度較小，則使用跳轉表；否則可能使用 if-else 類似的跳躍陳述式來實現。跳轉表高效的實現了多重分支。

- flow_switch.c
- flow_switch_impl.c
- flow_switch_2.c

在 flow_switch_impl 中使用了 GCC 對 C 語言的擴展，通過&&運算子獲取標號所示代碼塊的地址，goto 間接跳轉。

4 參考資料

本文參考了以下資料：

- [Bryant2003] 深入理解電腦系統，第二版。Randal E. Brant, David R. O’Hallaron 著，龔奕利、雷迎春譯。第 2 章，第 3 章，第 5 章，第 9 章。
- [片山善夫 2013] C 程式性能優化：20 個實驗與達人技巧。日·片山善夫著，何本華、居福國譯。

- [Stallman2003] Using the GNU Compiler Collection, For GCC version 4.5.0。
- Intel 64 and IA-32 Architectures Software Developer’ s Manual Volume 2, Instruction Set Reference。
- [ESR2014] The Lost Art of C Structure Packing, http://www.catb.org/esr/structure-packing/。