

NASM 示例

NASM 是一个绝佳的汇编器。现在让我们通过一些例子来学习 NASM。然而这里的笔记仅仅只是蜻蜓点水般地涉及了一些皮毛，所以当你看完这个页面后，你需要查看 [官方的 NASM 文档](#)。

准备工作

请注意：除了最后的几个部分，这里列举的所有例子需要运行在一个 64 位的 Linux 系统上。

请确保已经安装好 nasm 和 gcc。

入门

我们的第一个程序将使用 Linux 下的 1 号系统调用来输出一条信息和 60 号系统调用来退出程序。

```
hello.asm
; -----
; 仅使用系统调用来输出 "Hello, World" 到控制台。这个程序仅在 64 位的 Linux 下运行。
; 如何编译执行:
;
;   nasm -felf64 hello.asm && ld hello.o && ./a.out
; -----

global _start
section .text
_start:
; write(1, message, 13)
    mov    rax, 1          ; 1 号系统调用是写操作
    mov    rdi, 1          ; 1 号文件系统调用是标准输出
    mov    rsi, message    ; 输出字符串的地址
    mov    rdx, 13         ; 字符串的长度
    syscall               ; 调用系统执行写操作

; exit(0)
    mov    eax, 60          ; 60 号系统调用是退出
    xor    rdi, rdi          ; 0 号系统调用作为退出
    syscall               ; 调用系统执行退出
message:
    db     "Hello, World", 10      ; 注意到最后的换行

$ nasm -felf64 hello.asm && ld hello.o && ./a.out
Hello, world
```

调用一个 C 语言库

记得 C 程序为何看上去总是从 "main" 函数开始执行吗？实际上，因为在 C 语言库的内部有一个 `_start` 标签！在 `_start` 处的代码首先会做一些初始化的工作，然后调用 `main` 函数，最后会做一些清理工作，最终执行 60 号系统调用。因此，你只需要实现 `main` 函数。我们在汇编中可以这么做：

```
hola.asm
; -----
; 使用 C 语言库在控制台输出 "Hola, mundo"。程序运行在 Linux 或者其他在 C 语言库中不使用下划线的操作系统上。
; 如何编译执行:
;
;   nasm -felf64 hola.asm && gcc hola.o && ./a.out
; -----

global main
extern puts

section .text
main:
    mov    rdi, message        ; 被 C 语言库的初始化代码所调用
    call   puts                ; 在 rdi 中的第一个整数（或者指针）
    ret                          ; 由 main 函数返回 C 语言库例程
message:
    db     "Hola, mundo", 0      ; 注意到在 C 语言中字符串必须以 0 结束

$ nasm -felf64 hola.asm && gcc hola.o && ./a.out
Hola, mundo
```

了解调用约束

当你为 64 位 Linux 写一个集成了 C 语言库的程序时，你必须遵循以下的调用约束条件，详情可以参考 [AMD64 ABI Reference](#)。你也可以从 [Wikipedia](#) 得到这些信息。在这里列出最重要的几点：

- 传递参数时，按照从左到右的顺序，将尽可能多的参数依次保存在寄存器中。存放位置的寄存器顺序是确定的：
 - 对于整数和指针，`rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`。
 - 对于浮点数（float 和 double 类型），`xmm0`, `xmm1`, `xmm2`, `xmm3`, `xmm4`, `xmm5`, `xmm6`, `xmm7`。
- 剩下的参数将按照从右到左的顺序压入栈中，并在调用之后 *由调用函数推出栈*。
- 等所有的参数传入后，会生成调用指令。所以当被调用函数得到控制权后，返回地址会被保存在 `[rsp]` 中，第一个局部变量会被保存在 `[rsp+8]` 中，以此类推。
- 栈指针 `rsp` 在调用前必须进行 16 字节对齐处理。当然，调用的过程中只会把一个 8 bytes 的返回地址推入栈中，所以当函数得到控制权时，`rsp` 并没有对齐。你需要向栈中压入数据或者从 `rsp` 减去 8 来使之对齐。
- 调用函数需要保存如下的寄存器：`rbp`, `rbx`, `r12`, `r13`, `r14`, `r15`。其他的寄存器可以自由使用。
- 被调用函数也需要保存 `XMCSSR` 的控制位和 `x87` 指令集的控制字，但是 `x87` 指令在 64 位系统上十分少见所以你不必担心这点。
- 整数返回值存放在 `rax` 或者 `rdx:rax` 中，浮点数返回值存放在 `xmm0` 或者 `xmm1:xmm0` 中。

下面是一个用来展示如何保存和恢复寄存器的程序：

```
fib.asm
-----
; 一个输出 Fibonacci 数列前 90 项的 64 位 Linux 程序。
; 如何编译执行：
;
; nasm -felf64 fib.asm && gcc fib.o && ./a.out
;

global main
extern printf

section .text
main:
    push rbx           ; 因为需要用 rbx 寄存器所以需要保存

    mov ecx, 90         ; ecx 作为计数器直至减到 0
    xor rax, rax        ; rax 将记录当前的数字
    xor rbx, rbx        ; rbx 将记录下一个的数字
    inc rbx             ; rbx 初始值 1

print:
    ; 我们需要调用 printf 函数，但是我们也在同时在使用 rax,rbx 和 rcx 这三个寄存器。
    ; 调用 printf 函数会破坏 rax 和 rcx 这两个寄存器的值，所以我们要在调用前保存
    ; 并且在调用后恢复这两个寄存器中的数据。

    push rax            ; 调用者保存寄存器
    push rcx            ; 调用者保存寄存器

    mov rdi, format      ; 设置第一个参数 (format)
    mov rsi, rax          ; 设置第二个参数 (current_number)
    xor rax, rax          ; 因为 printf 是多参数的

    ; 栈内已经对齐，因为我们压入了三个 8 字节的数据。
    call printf           ; printf(format, current_number)

    pop rcx              ; 恢复调用者所保存的寄存器
    pop rax              ; 恢复调用者所保存的寄存器

    mov rdx, rax          ; 保存当前的数字
    mov rax, rbx          ; 下一个数字保存在当前数字的位置
    add rbx, rdx          ; 计算得到下一个数字
    dec ecx               ; ecx 减 1
    jnz print             ; 如果不是 0，继续循环

    pop rbx               ; 返回前恢复 rbx 的值
ret
format:
db "%2ld", 10, 0
```

```
$ nasm -felf64 fib.asm && gcc fib.o && ./a.out
0
1
1
2
.
.
.
679891637638612258
110008778366101931
1779979416004714189
```

C 和汇编的联合调用

这是一个实现从三个整型参数中返回最大值函数的程序。

```
maxofthree.asm
-----
; 一个返回三个数字中最大值的 64 位函数。
; 函数有如下形式：
;
; int64_t maxofthree(int64_t x, int64_t y, int64_t z)
;
; 注意到参数通过 rdi, rsi 和 rdx 传递。
; 我们只需要将返回值存放在 rax 中。
;

global maxofthree
section .text
maxofthree:
    mov rax, rdi          ; rax 寄存器初始保存 x 的值
    cmp rax, rsi          ; x 小于 y 吗？
    cmovl rax, rsi        ; 如果是的话，返回值置为 y
    cmp rax, rdx          ; x 和 y 中的最大值小于 z 吗？
    cmovl rax, rdx        ; 如果是的话，返回值置为 z
    ret                  ; 最大值被存放在 rax 中
```

下面是一个调用汇编函数的 C 程序。

```
callmaxofthree.c
/*
 * 这是一个用来展示我们如何调用在汇编语言中
 * 编写的 maxofthree 函数的程序。
 */
#include <stdio.h>
#include <inttypes.h>

int64_t maxofthree(int64_t, int64_t, int64_t);

int main() {
    printf("%ld\n", maxofthree(1, -4, -7));
    printf("%ld\n", maxofthree(2, -6, 1));
    printf("%ld\n", maxofthree(2, 3, 1));
    printf("%ld\n", maxofthree(-2, 4, 3));
    printf("%ld\n", maxofthree(2, -6, 5));
    printf("%ld\n", maxofthree(2, 4, 6));
    return 0;
}
```

```
$ nasm -felf64 maxofthree.asm && gcc callmaxofthree.c maxofthree.o && ./a.out
1
2
3
4
```

命令行参数

在 C 语言中，`main` 是一个古老而简单的函数，其实它自身可以附带一些参数：

```
int main(int argc, char** argv)
```

下面是一个运用这一点实现的简单每行输出一个命令行参数的函数：

```
echo.asm

; -----
; 一个显示命令行参数的 64 位程序。一行一个地输出。
; 在函数入口处, rdi 保存 argc 的值, rsi 保存 argv 的值。
; -----
global main
extern puts
section .text
main:
    push rdi          ; 保存 puts 函数需要用到的寄存器
    push rsi
    sub rsp, 8        ; 调用函数前让栈顶对齐

    mov rdi, [rsi]    ; 需要输出的字符串参数
    call puts         ; 调用 puts 输出 /span>

    add rsi, 8        ; 恢复 rsi 到未对齐前的值
    pop rsi
    pop rdi

    add rsi, 8        ; 指向下一个参数
    dec rdi
    jnz main          ; 如果未读完参数则继续

ret
```

```
$ nasm -felf64 echo.asm && gcc echo.o && ./a.out dog 22 -zzz "hi there"
./a.out
dog
22
-zzz
hi there
```

一个更长一点的例子

注意到就 C 语言库来说，命令行参数总是以字符串的形式传入的。如果你想把参数作为整型使用，调用 `atoi` 函数。下面是一个计算 x^y 的函数。

```
power.asm

; -----
; 一个用于计算  $x^y$  的 64 位命令行程序。
; 格式: power x y
; x 和 y 均为 32 位的正整数
; -----
global main
extern printf
extern puts
extern atoi

section .text
main:
    push r12          ; 调用者保存寄存器
    push r13
    push r14
    ; 通过压入三个寄存器的值，栈已经对齐

    cmp rdi, 3        ; 必须有且仅有 2 个参数
    jne error1

    mov r12, rsi       ; argv

    ; 我们将使用 ecx 作为指数的计数器，直至 ecx 减到 0。
    ; 使用 esi 来保存基数，使用 eax 保存乘积。

    mov rdi, [r12+16]   ; argv(2)
    call atoi          ; y 存放在 eax 中
    cmp eax, 0          ; 不允许负指数
    jl error2
    mov r13d, eax        ; y 存放在 r13d 中

    mov rdi, [r12+8]   ; argv(1)
    call atoi          ; x 存放在 eax 中
    mov r14d, eax        ; x 存放在 r14d 中

    mov eax, 1          ; 初始结果 answer = 1

check:
    test r13d, r13d    ; 减 y 直至 0
    jz gotit
    imul eax, r14d      ; 再乘上一个 x
    dec r13d
    jmp check

gotit:
    mov rdi, answer
    movsxd rsi, eax
    xor rax, rax
    call printf
    jmp done

error1:
    mov edi, badArgumentCount
    call puts
    jmp done

error2:
    mov edi, negativeExponent
    call puts

done:
    ; 恢复所保存的寄存器
    pop r14
    pop r13
    pop r12
ret
```

```

answer:
    db      "%d", 10, 0
badArgumentCount:
    db      "Requires exactly two arguments", 10, 0
negativeExponent:
    db      "The exponent may not be negative", 10, 0

```

```

$ nasm -felf64 power.asm && gcc -o power power.o
$ ./power 2 19
524288
$ ./power 3 -8
The exponent may not be negative
$ ./power 1 500
1
$ ./power 1
Requires exactly two arguments

```

浮点数指令

浮点数参数保存在xmm寄存器中。下面是一个用来计算存放在数组中的浮点数的和的简单的函数：

```

sum.asm

; -----
; 一个返回浮点数数组元素和的 64 位程序。
; 函数声明如下：
; double sum(double[] array, uint64_t length)
; -----


        global sum
        section .text

sum:
        xorssd xmm0, xmm0          ; 初始化累加和为 0
        cmp    rsi, 0               ; 考虑数组长度为 0 的特殊情形
        je     done

next:
        addsd  xmm0, [rdi]          ; 累加当前数组元素的值
        add    rdi, 8               ; 指向下一个数组元素
        dec    rsi                ; 计数据递减
        jnz   next                ; 如果没有结束递减则继续累加
done:
        ret                         ; 返回保存在 xmm0 寄存器中的值

```

这里有一个C程序调用汇编函数：

```

callsum.c

/*
 * 展示如何调用我们在汇编语言中编写的 sum 函数。
 */

#include <stdio.h>
#include <iinttypes.h>

double sum(double[], uint64_t);

int main() {
    double test[] = {
        40.5, 26.7, 21.9, 1.5, -40.5, -23.4
    };
    printf("%20.7f\n", sum(test, 6));
    printf("%20.7f\n", sum(test, 2));
    printf("%20.7f\n", sum(test, 0));
    printf("%20.7f\n", sum(test, 3));
    return 0;
}

$ nasm -felf64 sum.asm && gcc sum.o callsum.c && ./a.out
26.700000
67.200000
0.000000
89.100000

```

数据段

在大多数操作系统中，指令段是只读的，所以你需要使用数据段。数据段仅仅是被用来初始化数据，而且你可以发现一个叫做.bss的段是存放未初始化过的数据的。下面是一个程序用来计算通过命令行参数传递的整数的平均值，并且以浮点数输出结果的程序。

```

average.asm

; -----
; 一个把参数当做整数处理，并且以浮点数形式输出他们平均值的 64 位程序。
; 这个程序将使用一个数据段来保存中间结果。
; 这不是必需的，但是在此我们想展示数据段是如何使用的。
; -----


        global main
        extern atoi
        extern printf
        default rel

        section .text

main:
        dec    rdi           ; argc-1, 因为我们不需要读入程序名称
        jz    nothingToAverage
        mov    [count], rdi   ; 保存浮点数参数的个数

accumulate:
        push   rdi           ; 保存调用 atoi 需要使用的寄存器
        push   rsi
        mov    rdi, [rsi+rdi*8] ; argv[rdi]
        call   atoi           ; 现在 rax 里保存着 arg 的整数值
        pop    rsi
        pop    rdi           ; 调用完 atoi 函数后恢复寄存器
        add    [sum], rax     ; 继续累加
        dec    rdi           ; 递减
        jnz   accumulate     ; 还有参数吗？

average:
        cvtsi2sd xmm0, [sum]
        cvtssd xmm1, [count]
        divsd xmm0, xmm1       ; xmm0 现在值为 sum/count
        mov    rdi, format     ; printf 的第一个参数 [注：输出格式]
        mov    rax, 1             ; printf 第二个参数 [注：会看一个不是参数的参数]

```

```

        sub    rsp, 8          ; 对齐栈指针
        call   printf           ; printf(format, sum/count)
        add    rsp, 8          ; 恢复栈指针
        ret

nothingToAverage:
        mov    rdi, error
        xor    rax, rax
        call   printf
        ret

        section .data
count: dq    0
sum:  dq    0
format: db    "g", 10, 0
error: db    "There are no command line arguments to average", 10, 0

```

```

$ nasm -felf64 average.asm && gcc average.o && ./a.out 19 8 21 -33
3.75
$ nasm -felf64 average.asm && gcc average.o && ./a.out
There are no command line arguments to average

```

递归

可能会让大家吃惊的是，事实上，实现一个递归并不需要什么特别的操作。你仅仅只需要像平时一样小心地保存寄存器的状态即可。

```

factorial.asm

; -----
; 一种递归函数的实现:
;
;  uint64_t factorial(uint64_t n) {
;      return (n <= 1) ? 1 : n * factorial(n-1);
;  }
;

        global factorial

        section .text
factorial:
        cmp    rdi, 1          ; n <= 1?
        jne   L1              ; 如果不是, 进行递归调用
        mov    rax, 1          ; 否则, 返回 1
        ret

L1:
        push   rdi            ; 在栈上保存 n(同时对齐 %rsp 寄存器!)
        dec    rdi            ; n-1
        call   factorial       ; factorial(n-1), 返回值保存在 %rax 中
        pop    rdi            ; 保存 n
        imul  rax, rdi         ; n * factorial(n-1), 保存在 %rax 中
        ret

```

一个递归的例子:

```

callfactorial.c

/*
 * 这是一个调用在外部定义阶乘函数的程序
 */

#include <stdio.h>
#include <inttypes.h>

uint64_t factorial(uint64_t n);

int main() {
    for (uint64_t i = 0; i < 20; i++) {
        printf("factorial(%lu) = %lu\n", i, factorial(i));
    }
    return 0;
}

```

```

$ nasm -felf64 factorial.asm && gcc -std=c99 factorial.o callfactorial.c && ./a.out
factorial(0) = 1
factorial(1) = 1
factorial(2) = 2
factorial(3) = 6
factorial(4) = 24
factorial(5) = 120
factorial(6) = 720
factorial(7) = 5040
factorial(8) = 40320
factorial(9) = 362880
factorial(10) = 39916800
factorial(11) = 479001600
factorial(12) = 6227020800
factorial(13) = 87178291200
factorial(15) = 1307674368000
factorial(16) = 20922789888000
factorial(17) = 355687428096000
factorial(18) = 6402373705728000
factorial(19) = 121645100408832000

```

SIMD 并行

XMM 寄存器一次可以对浮点数进行单个或多个操作。操作形式如下:

```
operation xmmregister_or_memorylocation, xmmregister
```

对于浮点数加法, 指令如下:

```

addpd - do 2 double-precision additions
addps - do just one double-precision addition, using the low 64-bits of the register
addsd - do 4 single-precision additions
addss - do just one single-precision addition, using the low 32-bits of the register

```

TODO - 举一个能一次处理数组中 4 个浮点数的函数的例子

饱和运算

XMM 寄存器也可以进行整数运算。指令有如下形式：

```
operation xmmregister_or_memorylocation, xmmregister
```

对于整数加法，指令如下：

```
paddb - do 16 byte additions  
paddw - do 8 word additions  
paddq - do 4 dword additions  
paddq - do 2 qword additions  
padds - do 16 byte additions with signed saturation (80..7F)  
paddsw - do 8 word additions with unsigned saturation (8000..FFFF)  
paddusb - do 16 byte additions with unsigned saturation (00..FF)  
paddusw - do 8 word additions with unsigned saturation (00..FFFF)
```

TODO - 举一个例子

绘图

TODO

局部变量与栈帧

首先，请阅读 Eli Bendersky 的 [这篇文章](#)，会比这些摘要更加完整。

当一个函数被调用者调用时，调用者会先把参数存入正确的寄存器中，然后再执行 `call` 指令。无法放入寄存器中的参数将会在调用前被推入栈中。所调用的指令会把返回地址存入栈顶。所以如果有以下的函数：

```
int64_t example(int64_t x, int64_t y) {  
    int64_t a, b, c;  
    b = 7;  
    return x * b + y;  
}
```

在函数的入口，`x` 和 `y` 会被分别存在 `edi` 和 `esi` 中，返回地址将会被存在栈顶。局部变量会被存到哪里？无论是否有足够的寄存器，一种简单的选择就是存入函数自己的栈中。

如果程序运行在一个实现了 ABI 标准的机器上，你可以在 `rsp` 保持不变的情况下获取无法在寄存器中保存的参数值和局部变量值，例如：

```
+-----+  
rsp-24 | a |  
+-----+  
rsp-16 | b |  
+-----+  
rsp-8 | c |  
+-----+  
rsp | retaddr |  
+-----+  
rsp+8 | caller's |  
| stack |  
| frame |  
| ... |  
+-----+
```

我们的函数看上去是这个样子的：

```
global example  
section .text  
example:  
    mov    qword [rsp-16], 7  
    mov    rax, rdi  
    imul   rax, [rsp+8]  
    add    rax, rsi  
    ret
```

如果被调用的函数需要调用其他函数，你就需要调整 `rsp` 的值来得到正确的返回地址。

在 Windows 上你可能无法使用这种方法，因为当中断发生的时候，栈指针上的数据会被抹去。而在其他大多数的操作系统中，这件事不会发生，因为有一个 128bytes 的“红色区域”来保护栈指针的安全。在这个例子中，你可以给栈留出空间：

```
example:  
    sub   rsp, 24
```

栈看上去是这个样子的：

```
+-----+  
rsp | a |  
+-----+  
rsp+8 | b |  
+-----+  
rsp+16 | c |  
+-----+  
rsp+24 | retaddr |  
+-----+  
rsp+32 | caller's |  
| stack |  
| frame |  
| ... |  
+-----+
```

下面就是我们的函数。注意到我们需要在返回前替换栈指针！

```
global example  
section .text  
example:  
    sub   rsp, 24  
    mov    qword [rsp+8], 7  
    mov    rax, rdi  
    ...
```

```
        add    rax, rsi
add    rsp, 24
ret
```

在 OS X 下使用 NASM

TODO

在 Windows 下使用 NASM

TODO