目录

底层编程语言：汇编语言asm

- 最新版本：`v1.0`
- 更新时间：`20240430`

简介

介绍编程语言中，相对底层的汇编语言，简称asm。

源码+浏览+下载

本书的各种源码、在线浏览地址、多种格式文件下载如下：

HonKit源码

- crifan/low_level_assembly_asm: 底层编程语言：汇编语言asm

如何使用此HonKit源码去生成发布为电子书

详见：crifan/honkit_template: demo how to use crifan honkit template and demo

在线浏览

- 底层编程语言：汇编语言asm book.crifan.org
- 底层编程语言：汇编语言asm crifan.github.io

离线下载阅读

- 底层编程语言：汇编语言asm PDF
- 底层编程语言：汇编语言asm ePub
- 底层编程语言：汇编语言asm Mobi

版权和用途说明

鸣谢

感谢我的老婆**陈雪**的包容理解和悉心照料，才使得我 `crifan` 有更多精力去专注技术专研和整理归纳出这些电子书和技术教程，特此鸣谢。

其他

作者的其他电子书

本人 `crifan` 还写了其他 `150+` 本电子书教程，感兴趣可移步至：

crifan/crifan_ebook_readme: Crifan的电子书的使用说明

关于作者

关于作者更多介绍，详见：

关于CrifanLi李茂 – 在路上

汇编语言asm概述

- asm汇编开发心得 == 底层编程语言：汇编语言asm
  - 此处整理，底层编程语言，汇编语言=assembly language，简称 asm汇编
    - 相关的内容，包括通用的逻辑和用法
- 关于名称：底层编程语言
  - 编程语言，根据level层次不同，一般分为：
    - Low-level programming language
      - 一般译为：低级编程语言
    - High-level programming language
      - 一般译为：高级编程语言
- 关于编程语言中的：`Low-level` **低级**和 `High-level` **高级**
  - 就其本身含义而言，和中文字面意思的**高级**和**低级**，差异很大
  - 此处主要面向的应用领域不同而已，没有真正的档次上高低之分
  - 为了更加贴近本意，个人建议译为：
    - Low-level programming language：**底层**编程语言
    - High-level programming language: **上层**编程语言
  - 更能反应出两类语言的真正差异：应用领域的不同而已

-》

- **机器语言**：机器才能看懂，即执行对应的指令
- **汇编语言**：人也能看懂，但是很费劲，写起来也很累
  - 用assembler汇编器，把 汇编语言 翻译为 机器语言，让机器读懂和执行，实现程序逻辑
- **高级语言**：人容易看懂，适合人阅读和编写
  - 用compiler编译器，把 高级语言 翻译为 机器语言，让机器读懂和执行，实现程序逻辑

汇编通用知识

常量

- 汇编中的常量值
  - ARM syntax：`#xxx`
    - 举例
      - 10进制
        - `mov x16, #338`
        - `ADD R7, R7, #4 ; a = a + 4`
        - `sub r0, r1, #3 @ r0 = r1 - 3`
      - 16进制：0x开头
        - `svc #0x80`
        - `SUB R8, R7, #0xC ; b = a − 12`
        - `mov r0, #0xFF0 @ r0 = 0xFF0`
  - Intel syntax：h结尾：`xxxh`
    - 举例：
      - `int 80h`
  - AT&T的：`$xxx`
    - 举例
      - `int $0x80`

内联汇编

内联汇编相关关键字

- asm（汇编）关键字
  - GNU/GCC编译器
    - 标准C语言（用-ansi或-std去编译）：用 `__asm__`
    - GNU扩展（GNU extension）：用 `asm`
  - ARM编译器、微软的内嵌汇编
    - `__asm`
- volatile关键字
  - 属于asm的修饰符qualifier
    - GNU/GCC编译器
      - 标准C语言（用 `-ansi` 或 `-std` 去编译）：用 `__volatile__`
      - GNU扩展（ `GNU extension` ）：用 `volatile`

-> 结论：

从兼容性来说，那最好用：

- 前后都带下划线的版本：
  - `__asm__`
  - `__volatile__`

内联汇编的语法

GCC中关于asm的语法

- GCC中关于asm的语法
  - `标准C` 语言（用 `-ansi` 或 `-std` 去编译）：用 `__asm__`
  - `GNU扩展`（`GNU extension`）：用 `asm`

背景：

当用 `标准C`（`-ansi` 或 `-std`），会禁用掉部分关键字：

- `asm`
- `typeof`
- `inline`
  - 特例：当 `-std=c99` 或更新版本，可以用 `inline`
- `restrict`
  - 只有当 `-std=gnu99` 或 `-std=c99`（等价的 `-std=iso9899:1999`）的或更新版 时，才能用 `restrict`

-》想要用上述关键字的话，解决办法是：

在前面和后面分别加上2个下划线

-》对于asm，可以自己加上对应定义：

```
#ifndef __GNUC__
#define __asm__ asm
#endif
```

asm基本语法

- asm asm-qualifiers (AssemblerInstructions)
  - asm
    - 当用 `-ansi` 或 `-std` 时，要写成：**asm**
  - asm-qualifiers
    - `volatile`
      - 其实没啥作用
      - 所有的asm本身就是隐式声明为volatile的
    - `inline`
      - asm汇编代码（块）会尽量少的占用空间
  - AssemblerInstructions
    - 汇编代码
    - 会被assembler汇编器去汇编
    - 可以包含多行汇编代码
    - 多行之间加上换行分隔符
      - 常见的有：`\n\t`
      - 偶尔的有：`分号` = `semicolons`
        - 注意：个别汇编器中分号放在行首意思是注释

基本asm(basic asm) vs 扩展asm(extend asm)

- 名词
  - clobbers =clobbering registers= （asm汇编中）modified registers = （asm汇编中）值被修改了的寄存器
- 背景
  - asm的写法，历史上有多种写法的
  - AT&T 的asm汇编写法 是其中一种
    - -》属于老旧的写法
  - 现在多数是：GNU/GCC的asm的写法

basic asm vs extend asm

- 基本asm=basic asm：直接一行行的写汇编代码
  - 优点
    - 可以放在（C）函数外
      - 放在文件顶部，作为全局（汇编）函数
        - 方便其他文件直接调用
  - 缺点：
    - 编译器（gcc）并不知道汇编内部的逻辑
      - 无法（帮直接汇编的你）做相关的优化和处理
  - 说明：
    - naked的函数，仅支持基本asm
- 扩展asm=Extended Asm：带指定输入输出等寄存器的汇编代码
  - 一般来说生成代码会：更小、更安全、更高效
    - 支持运算符operands：指定 输入寄存器，输出寄存器、受影响的寄存器（clobbered register）
  - 缺点
    - 只能内嵌在（C）函数中

建议

- 如果可以，尽量不要用：内嵌汇编
- 如果要用汇编：尽量不用 `basic asm`，尽量用 `extend asm`
  - 原因
    - 扩展 asm 允许程序员为 asm 指定输入和输出以及它修改的寄存器（clobbers），而基本 asm 则没有。
    - 关于 asm 代码可以覆盖哪些寄存器，不同的 C 编译器使用不同的语义。 gcc 假设没有寄存器被修改。如果您的 asm 修改寄存器而不通知编译器（这需要使用扩展 asm），则会导致未定义的行为。
    - 破坏寄存器（基本 asm 不支持）可以提供比 push/pop 更好的性能。
    - 一些编译器会在调用任何 asm（gcc 称之为“内存”clobber）之前自动将寄存器刷新到内存中。但是 gcc 的基本 asm 不这样做。如果你的 asm 需要这个，你需要在扩展 asm 中使用“内存”clobber。
    - gcc 正在考虑改变基本 asm 的长期语义。它可能很快就会开始破坏一切，而不是不破坏任何东西。这可能会修复现有代码中的细微错误。但是，这有（很小的）机会会导致现有的、正确的功能代码出现问题。此外，这也可能会引入性能问题，因为某些/所有寄存器的内容（即使是您的 asm 不使用的）可能需要围绕您的 asm 语句保存/重新加载。为了“面向未来”您的代码针对这些类型的更改，请使用扩展的 asm 并准确指定需要破坏的内容。
    - 由于 gcc 的基本 asm 没有输入、输出或破坏这一事实，优化器很难在生成的代码中始终如一地定位基本 asm。

basic asm = 基本asm

语法

```
asm("assembly code");
```

或：

```
__asm [volatile] (code); /* Basic inline assembly syntax */
```

举例

单行

```
asm("movl %ecx %eax"); /* moves the contents of ecx to eax */
__asm__("movb %bh (%eax)"); /*moves the byte from bh to the memory pointed by eax */
```

多行

最后加上：`\n\t`

```
__asm__ ("movl %eax, %ebx\n\t"
         "movl $56, %esi\n\t"
         "movl %ecx, $label(%edx,%ebx,$4)\n\t"
         "movb %ah, (%ebx)");
```

解释：编译器`gcc`会把汇编代码发送到汇编器：`as`=`GAS`，其中多行的分隔符就是：`\n\t`

微软的内嵌汇编

```
__asm int 3
```

```
__asm {
   mov al, 2
   mov dx, 0xD007
   out dx, al
}
```

```
__asm mov al, 2
__asm mov dx, 0xD007
__asm out dx, al
```

```
__asm mov al, 2   __asm mov dx, 0xD007   __asm out dx, al
```

```
// asm_overview.cpp
// processor: x86
void __declspec(naked) main()
{
    // Naked functions must provide their own prolog...
    __asm {
        push ebp
        mov ebp, esp
```

```
        sub esp, __LOCAL_SIZE
    }


    // ... and epilog
    __asm {
        pop ebp
        ret
    }
}
```

extend asm = 扩展asm

语法

Extended inline assembly syntax：

```
asm [asm qualifiers] ( AssemblerTemplate
                     : OutputOperandsList
                    [ : InputOperandsList       /* optional */
                    [ : ClobbersRegistersList   /* optional */
                    [ : GotoLabels] ]            /* optional */
                   ])
```

- 说明
  - asm-qualifiers可选值：`volatile`

举例

拷贝src到dst，且给dst加1

```
int src = 1;
int dst;

asm ("mov %1, %0\n\t"
    "add $1, %0"
    : "=r" (dst)
    : "r" (src));

printf("%d\n", dst);
```

fills the fill_value count times to the location pointed to by the register edi

- fills the fill_value count times to the location pointed to by the register edi
  - 同时告诉gcc：
    - 受影响的寄存器有：eax、edi
      - eax、edi寄存器中的内存已无效

```
asm ("cld\n\t"
     "rep\n\t"
     "stosl"
     : /* no output registers */
     : "c" (count), "a" (fill_value), "D" (dest)
     : "%ecx", "%edi"
     );
```

make the value of ’b’ equal to that of ’a’

```
int a=10, b;
asm ("movl %1, %%eax;
     movl %%eax, %0;"
     :"=r"(b)        /* output */
     :"r"(a)         /* input */
     :"%eax"         /* clobbered register */
     );
```

参数和含义解释：

- "b" is the output operand, referred to by %0 and "a" is the input operand, referred to by %1.
- "r" is a constraint on the operands. We'll see constraints in detail later. For the time being, "r" says to GCC to use any register for storing the operands. output operand constraint should have a constraint modifier "=". And this modifier says that it is the output operand and is write-only.
  - 关于操作数的限制constraint，详见
    - 6.1 Commonly used constraints- GCC-Inline-Assembly-HOWTO (ibiblio.org)
- There are two %'s prefixed to the register name. This helps GCC to distinguish between the operands and registers. operands have a single % as prefix.
- The clobbered register %eax after the third colon tells GCC that the value of %eax is to be modified inside "asm", so GCC won't use this register to store any other value.

iOS中ARM内嵌汇编

iOS中内嵌汇编的语法

iOS中内嵌汇编，其实是：XCode（内部的）clang编译器中，内嵌asm汇编

此处目标设备是iPhone7，内部ARM架构

所以汇编是针对于64位的ARM的arm64，调用规范是：

- AAPCS64=Procedure Call Standard for the Arm 64-bit Architecture

具体的asm的语法，也基本上兼容：

- Extended Asm
  - == 上面已总结的：`extend asm`

具体语法详见：

- GCC-Inline-Assembly-HOWTO (ibiblio.org)
- How to Use Inline Assembly Language in C Code — gcc 6 documentation (dmalcolm.fedorapeople.org)

举例

iOS内嵌汇编

```
__attribute__((always_inline)) long svc_0x80_syscall(int syscall_number, const char * pathname, struct stat * stat_in
fo) {
    long ret = 0;
    long long_syscall_number = syscall_number;
    __asm__ volatile(
         "mov x0, %[pathname_p]\n"
         "mov x1, %[stat_info_p]\n"
         "mov x16, %[long_syscall_number_p]\n"
         "svc #0x80\n"
         "mov %[ret_p], x0\n"
        : [ret_p]"=r"(ret)
        : [long_syscall_number_p]"r"(long_syscall_number), [pathname_p]"r"(pathname), [stat_info_p]"r"(stat_info)
    );
    return ret == 0 ? ret : -1;
}
...
openResult = svc_0x80_syscall(SYS_stat64, filePathStr, stat_info);
```

优化后：用寄存器register保存参数变量值

此次之前代码：

```
__attribute__((always_inline)) long svc_0x80_syscall(int syscall_number, const char * pathname, struct stat * stat_in
fo) {
        long ret = 0;
        long long_syscall_number = syscall_number;
        __asm__ volatile(
             "mov x0, %[pathname_p]\n"
             "mov x1, %[stat_info_p]\n"
             "mov x16, %[long_syscall_number_p]\n"
             "svc #0x80\n"
             "mov %[ret_p], x0\n"
            : [ret_p]"=r"(ret)
            : [long_syscall_number_p]"r"(long_syscall_number), [pathname_p]"r"(pathname), [stat_info_p]"r"(stat_info)
            : "x0", "x1", "x16"
        );
```

```
        return ret == 0 ? ret : -1;
}
```

继续优化：用寄存器register保存参数变量值

最后代码是：

```
__attribute__((always_inline)) int svc_0x80_syscall(int syscall_number, const char * pathname, struct stat * stat_info
) {
    register const char * x0_pathname asm ("x0") = pathname; // first arg
    register struct stat * x1_stat_info asm ("x1") = stat_info;  // second arg
    register int x16_syscall_number asm ("x16") = syscall_number; // special syscall number store to x16

    register int x4_ret asm("x4") = -1; // store result

    __asm__ volatile(
         "svc #0x80\n"
         "mov x4, x0\n"
        : "=r"(x4_ret)
        : "r"(x0_pathname), "r"(x1_stat_info), "r"(x16_syscall_number)
//         : "x0", "x1", "x4", "x16"
    );
    return x4_ret;
}
```

C代码调用：

```
#import <sys/syscall.h>

struct stat stat_info;
const char * filePathStr = [filePath UTF8String];
...
openResult = svc_stat64(SYS_stat64, filePathStr, &stat_info);
```

说明：

虽然没有彻底搞懂：

inline asm的extend asm的

- 语法
- 逻辑
- 注意事项

但是至少是能跑通实现：

- svc 0x80的（stat和）stat64的调用
- 返回期望的值

ARM汇编

- ARM汇编
  - 单独的Book
    - 最流行汇编语言：ARM

ARM编译器

- compiler toolchain 编译器工具链
  - 编译器compiler：把C或C++转成机器码
  - 汇编器assembler：把汇编语言转成机器码
  - 链接器linker：把多个机器码模块合成单个可执行文件
- 目前可用的ARM编译器工具链
  - Arm Compiler 6
    - 最新、最高效的 Arm C/C++ 编译工具链，基于 armclang 编译器。 Arm Compiler 6 最大限度地发挥 Arm Cortex 和 Neoverse 处理器和架构的潜力，从 Armv6-M 到 Armv8-A 64 位 Arm：作为 Arm Development Studio 的一部分进行评估
      - Arm Development Studio | Free 30 day evaluation – Arm Developer
  - Arm Compiler 5
    - 上一代Arm C/C++编译工具链，基于armcc编译器。 Arm Compiler 5 为包括 Armv7 在内的遗留项目提供稳定性和出色的代码大小
  - GNU toolchain
    - 一个开源的、社区开发的工具链。 GNU 工具链提供了一种在 Arm 平台上进行开发的低成本机制
      - 资料
        - GCC, glibc performance in 2018 - Tools, Software and IDEs blog - Arm Community blogs - Arm Community

附录

下面列出相关参考资料。

参考资料

- 【已解决】asm汇编代码中的常量立即数操作数的语法
- 【整理】关于iOS内嵌汇编时basic asm和extend asm以及最佳实践
- 【已解决】iOS中ARM64内嵌汇编代码优化：用寄存器register保存参数变量值
- 【已解决】iOS中调用asm汇编关键字：asm **asm** asm**和volatile** volatile__
- 【已解决】iOS中如何inline内嵌汇编调用asm汇编代码
- 【整理】GNU和GCC的asm汇编语法
- 【未解决】iOS的inline汇编中arm代码svc 0x80去syscall调用open打开文件结果异常
-

- 低级语言 - 维基百科，自由的百科全书 (wikipedia.org)
- 机器语言、汇编语言（低级语言）、高级语言*GO_DIE*的博客-CSDN博客_机器语言
- 编程语言（Programming Language）、汇编语言（Assembly Language, ASM）、机器语言（Machine Language/Code）的区别和简介 | 数据学习者官方网站(Datalearner)
- 汇编语言 - 维基百科，自由的百科全书 (wikipedia.org)
- Compiling C and C++ code for Arm
- Assembler User Guide: Numeric constants (keil.com)
- Assembler User Guide: Syntax of numeric literals (keil.com)
- ARM Cortex-R Series Programmer's Guide
- How to load constants in assembly for Arm architecture - Architectures and Processors blog - Arm Community blogs - Arm Community
- Assembler User Guide: Syntax of Operand2 as a constant (keil.com)
- ARMCC: Locate Constants to Fixed Locations
- ARM Compiler armasm User Guide
- How do I do inline assembly on the IPhone? - Stack Overflow
- 'inline-assembly' tag wiki - Stack Overflow
- DontUseInlineAsm - GCC Wiki (gnu.org)
- Extended Asm (Using the GNU Compiler Collection (GCC))
- Procedure Call Standard for the Arm® 64-bit Architecture
- ARM GCC Inline Assembler Cookbook
- ConvertBasicAsmToExtended - GCC Wiki
- ios - fork() implementation by using svc call - Stack Overflow
- linux - ARM inline asm: exit system call with value read from memory - Stack Overflow
- iOS防护----越狱检测2021年版 - 简书 (jianshu.com)
- iOS安全攻防之ptrace等系统函数的调用方式 - 鸟的博客 (gitee.io)
- iOS汇编入门教程之ARM64汇编基础教程 / 张生荣 (zhangshengrong.com)
- 关于在XCode for Simulator中编译的c：inline asm，但无法在Device上编译 | 码农家园 (codenong.com)
- How to Use Inline Assembly Language in C Code — gcc 6 documentation (dmalcolm.fedorapeople.org)
- Arm Compiler User Guide Version 6.16
- Inline Assembler Overview | Microsoft Docs
- __asm | Microsoft Docs
- GCC-Inline-Assembly-HOWTO (ibiblio.org)
- Basic Asm (Using the GNU Compiler Collection (GCC))
- Extended Asm (Using the GNU Compiler Collection (GCC))
- Alternate Keywords (Using the GNU Compiler Collection (GCC))
- Size of an asm (Using the GNU Compiler Collection (GCC))

-