

Box86 相关知识笔记

Box86 是一款类似于 QEMU 的 cpu 模拟器，并且是专用于在 arm 上模拟 i386 架构的开源软件，于此类似的是华为的 exagear（闭源，需要商业授权）。由于 Box86 集成了 DynaRec（动态重新编译器），并且使用某些 host 系统库代替 i386 软件依赖的库，例如 libc, libm, SDL 和 OpenGL，因此与仅使用解释器相比，速度提高了 5 到 10 倍。

一、Box86 简介

Box86 的主要工作逻辑可以概况为如下两部分：

- ① 加载 i386 架构的 ELF 文件
- ② 按顺序执行二进制指令流

其框架逻辑流程如下图所示：

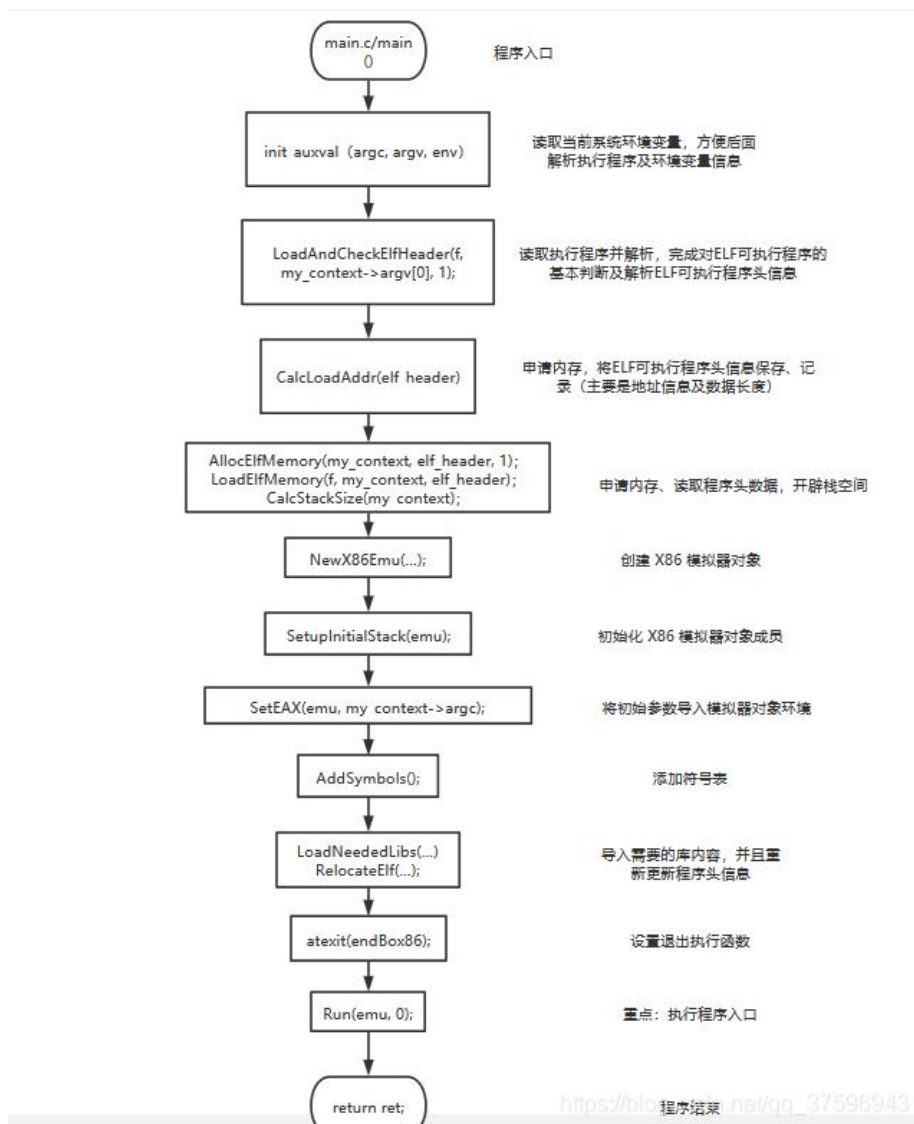


图 1 Box86 框架逻辑流程

BOX86 二进制翻译主体框架如下：

```
int Run(x86emu_t *emu, int step) {
```

```

    uintptr_t ip;
    static const void* baseopcodes[256] = { &&_0x00_0, &&_0x00_1, &&_0x00_2, ... &&_0x0F, ...
    &&_0x66, &&_0x67, ... };
    static const void* opcodes0f[256] = {
        &&_0f_0x38, &&_default, &&_default, &&_default, &&_default, &&_default, &&_default, &&_0f_0x3F,
        //0x38-0x3F
        &&_0f_0x68, &&_0f_0x69, &&_0f_0x6A, &&_0f_0x6B, &&_default, &&_default, &&_0f_0x6E, &&_0f_0x6F,
        //0x68-0x6F
        &&_default, &&_0f_0xD1, &&_0f_0xD2, &&_0f_0xD3, &&_0f_0xD4, &&_0f_0xD5, &&_default, &&_0f_0xD7,
        //0xD0-0xD7
        &&_0f_0xE0, &&_0f_0xE1, &&_0f_0xE2, &&_0f_0xE3, &&_0f_0xE4, &&_0f_0xE5, &&_default, &&_0f_0xE7,
        //0xE0-0xE7
        ..... };
    static const void* opcodes66[256] = { ..... };

x86emurun:
    ip = emu->ip.dword[0];
    opcode = *(uint8_t *) (ip++);
    goto *baseopcodes[opcode];

_0x??:      /* Box86 指令解析逻辑模板 */
    Do something.....
    goto *baseopcodes[(emu->ip.dword[0] = ip, opcode = *(uint8_t *) (ip++))];
_0x66:      /* Prefix to change width of instructions, so here, down to 16bits */
#include "run66.h"
    opcode = *(uint8_t *) (ip++);
    goto *opcodes66[opcode];
_0x67:      /* Prefix to change width of registers */
    emu->old_ip = emu->ip.dword[0];
    emu->ip.dword[0] = ip - 1; // don't count 0x67 yet
    Run67(emu);               // implemented in Run66.c
    ip = emu->ip.dword[0];
    if (emu->quit)
        goto fini;

    goto *baseopcodes[(emu->ip.dword[0] = ip, opcode = *(uint8_t *) (ip++))];

.....
_0x0F:      /* More instructions */
#include "run0f.h"
    opcode = *(uint8_t *) (ip++);
    goto *opcodes0f[opcode];
_default:
    emu->old_ip = emu->ip.dword[0];
    emu->ip.dword[0] = ip;
    UnimpOpcode(emu);
    goto fini;
}

```

二、ELF 相关

Linux 中可执行文件主要是 elf 格式，例如 ls、cd 等命令，当我们在终端中输入 ls 时，系统发生了什么？

1、为什么 Linux 可以运行 ELF 文件？

内核对所支持的每种可执行的程序类型都有个 struct linux_binfmt 的数据结构：

linux_binfmt 定义在 kernel-source/include/linux/binfmts.h 中

```
struct linux_binfmt {
    struct list_head lh;
    struct module *module;
    int (*load_binary)(struct linux_binprm *);
    int (*load_shlib)(struct file *);
    int (*core_dump)(struct coredump_params *cprm);
    unsigned long min_coredump;    /* minimal dump size */
};
```

所有的 linux_binfmt 对象都处于一个链表中，第一个元素的地址存放在 formats 变量中，可以通过调用 register_binfmt() 和 unregister_binfmt() 函数在链表中插入和删除元素，在系统启动期间，为每个编译进内核的可执行格式都执行 register_binfmt() 函数。当实现了一个新的可执行格式的模块正被装载时，也执行这个函数，当模块被卸载时，执行 unregister_binfmt() 函数。

当我们执行一个可执行程序的时候，内核会 list_for_each_entry 遍历所有注册的 linux_binfmt 对象，对其调用 load_binary 方法来尝试加载，直到加载成功为止。

其中的 load_binary 函数指针指向的就是一个可执行程序的处理函数。

那么，要支持 ELF 文件的运行，则必须向内核登记注册 elf_format 这个 linux_binfmt 类型的数据结构，加入到内核支持的可执行程序的队列中。

fs/binfmt_elf.c 中注册如下：

```
static struct linux_binfmt elf_format = {
    .module      = THIS_MODULE,
    .load_binary  = load_elf_binary,
    .load_shlib   = load_elf_library,
    .core_dump    = elf_core_dump,
    .min_coredump = ELF_EXEC_PAGESIZE,
};
```

总结一下 ELF 程序运行流程就是： sys_execve() > do_execve() > search_binary_handler() > load_elf_binary()

2、内核空间的加载过程 load_elf_binary

内核中实际执行 execv() 或 execve() 系统调用的程序是 do_execve()，这个函数先打开目标映像文件，并从目标文件的头部读取 128 个字节（ELF 文件头），然后调用另一个函数 search_binary_handler()，在此函数里面搜索上面提到的 linux_binfmt 对象链表，找到程序匹配的解器。如果匹配成功则调用 load_binary 函数指针所指向的处理函数来处理目标映像文件。

对于 ELF 文件格式，处理函数是 load_elf_binary，其函数的执行过程大致如下：

- ① 填充并且检查目标程序 ELF 头部
- ② load_elf_phdrs 加载目标程序的程序头表
- ③ 如果需要动态链接，则寻找和处理解释器段

- ④ 检查并读取解释器的程序表头
- ⑤ 装入目标程序的段 segment
- ⑥ 填写程序的入口地址
- ⑦ create_elf_tables 填写目标文件的参数环境变量等必要信息
- ⑧ start_kernel 宏准备进入新的程序入口



load_elf_binary.c

动态连接库映像的装入则由 `load_elf_library()` 完成。

3、Elf 之外的其他程序如何运行？

Linux 内核默认支持 elf、flat、aout、script 几种格式，其中 script 的方式最为简单，运行脚本时，内核直接读取脚本的第一行，判断是否如下格式：

```
#!${interpreter}\n
```

如果是的话，直接加载对应的解释器执行，（解释器的执行最终还会转到 elf 加载）。

通过上面的格式可以看出，bash、python、perl 等等脚本文件都属于这个启动流程。

除了内核预定义的几种格式之外，Linux 内核还提供了一种扩展机制——binfmt_misc，使得更多类型的文件得以成为 " 可执行 " 文件，该机制由 binfmt_misc 内核模块提供支持，通过这个模块可以动态注册一些 " 可执行文件格式 "。

- ① 如何查看内核是否已经支持 binfmt ？

`lsmod | grep binfmt`：检查内核模块 binfmt_misc 是否已经加载，有内容输出说明已经加载了，如果没有加载，则可以用 `modprobe binfmt_misc` 来加载它，当然，一般也可以通过 `sudo systemctl restart systemd-binfmt` 来启动/重启它。

`ls /proc/sys/fs/binfmt_misc/`：可以检查内核中目前注册了哪些格式。

- ② 一个示例

```
echo ':DOSWin:M::MZ::/usr/bin/wine:' > /proc/sys/fs/binfmt_misc/register
```

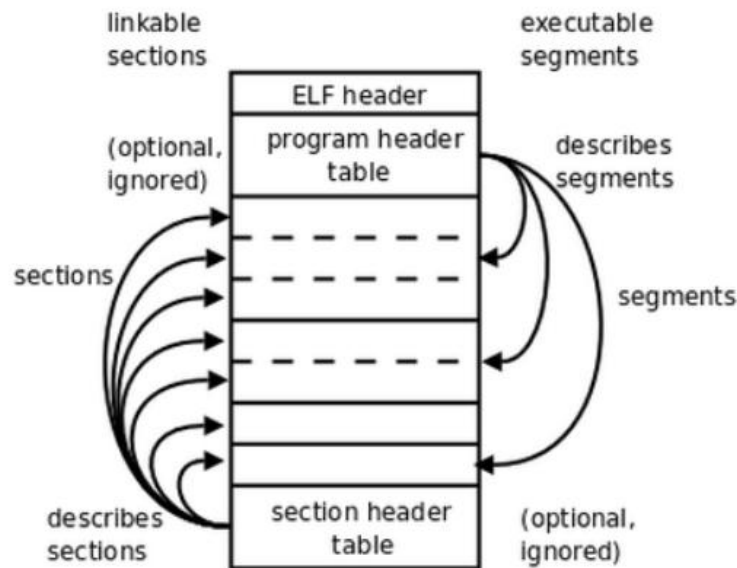
永久生效：

```
$ cat /etc/binfmt.d/wine.conf
```

```
# Start WINE on Windows executables
```

```
:DOSWin:M::MZ::/usr/bin/wine:
```

4、ELF 文件解析



(1) ELF 结构规范

一般的 ELF 文件有三个重要的索引表

- ① **ELF header**: 在文件的开始，描述整个文件的组织。。
- ② **Program header table**: 告诉系统如何创建进程映像。用来构造进程映像的目标文件必须具有程序头部表，可重定位文件不需要这个表。
- ③ **Section header table**: 包含了描述文件节区的信息，每个节区在表中都有一项，每一项给出诸如节区名称、节区大小这类信息。用于链接的目标文件必须包含节区头部表，其他目标文件可以有，也可以没有这个表。

sections 或者 segments: segments 是从运行的角度来描述 elf 文件，sections 是从链接的角度来描述 elf 文件，也就是说，在链接阶段，我们可以忽略 program header table 来处理此文件，在运行阶段可以忽略 section header table 来处理此程序（所以很多加固手段删除了 section header table）。注意：segments 与 sections 是包含的关系，一个 segment 包含若干个 section。

详细规范可以参考: <http://flint.cs.yale.edu/cs422/doc/ELF Format.pdf>

ELF 文件结构对应的实际编程头文件定义如下:

```
#define EI_NIDENT 16
```

```
sizeof Elf64_Ehdr = 64
sizeof Elf32_Ehdr = 52
```

32 位程序的 ELF 头 52 个字节，64 位程序的 ELF 头 64 个字节

定义: <http://sco.com/developers/gabi/latest/ch4.eheader.html#elfid>

```
typedef struct {
    unsigned char
e_ident[EI_NIDENT];
    Elf32_Half    e_type;
    Elf32_Half    e_machine;
    Elf32_Word    e_version;
    Elf32_Addr    e_entry;
    Elf32_Off     e_phoff;
    Elf32_Off     e_shoff;
    Elf32_Word    e_flags;
    Elf32_Half    e_ehsize;
    Elf32_Half    e_phentsize;
    Elf32_Half    e_phnum;
    Elf32_Half    e_shentsize;
    Elf32_Half    e_shnum;
    Elf32_Half    e_shstrndx;
} Elf32_Ehdr;
```

```
typedef struct {
    unsigned char
e_ident[EI_NIDENT];
    Elf64_Half    e_type;
    Elf64_Half    e_machine;
    Elf64_Word    e_version;
    Elf64_Addr    e_entry;
    Elf64_Off     e_phoff;
    Elf64_Off     e_shoff;
    Elf64_Word    e_flags;
    Elf64_Half    e_ehsize;
    Elf64_Half    e_phentsize;
    Elf64_Half    e_phnum;
    Elf64_Half    e_shentsize;
    Elf64_Half    e_shnum;
    Elf64_Half    e_shstrndx;
} Elf64_Ehdr;
```

参考标准: ELF-64 格式规范.pdf



ELF-64
格式规范.pdf

(2) ELF 文件解析示例

\$ hexdump -n 64 hello64 -C

00000000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00 |.ELF.....|

Name	Value	Meaning
ELFOSABI_SYSV	0	System V ABI
ELFOSABI_HPUX	1	HP-UX operating system
ELFOSABI_STANDALONE	255	Standalone (embedded) application

Name	Value	Meaning
EV_NONE	0	Invalid version
EV_CURRENT	1	Current version

Name	Value	Meaning
ELFDATANONE	0	Invalid data encoding
ELFDATA2LSB	1	little-endian
ELFDATA2MSB	2	big-endian

Figure 4-4: e_ident[] Identification Indexes

Name	Value	Purpose
EI_MAG0	0	File identification
EI_MAG1	1	File identification
EI_MAG2	2	File identification
EI_MAG3	3	File identification
EI_CLASS	4	File class
EI_DATA	5	Data encoding
EI_VERSION	6	File version
EI_OSABI	7	Operating system/ABI identification
EI_ABIVERSION	8	ABI version
EI_PAD	9	Start of padding bytes
EI_NIDENT	16	Size of e_ident[]

Name	Value	Meaning
ELFCLASSNONE	0	Invalid class
ELFCLASS32	1	32-bit objects
ELFCLASS64	2	64-bit objects

Name	Value	Position
ELFMAG0	0x7f	e_ident[EI_MAG0]
ELFMAG1	'E'	e_ident[EI_MAG1]
ELFMAG2	'L'	e_ident[EI_MAG2]
ELFMAG3	'F'	e_ident[EI_MAG3]

00000010 03 00 3e 00 01 00 00 00 60 10 00 00 00 00 00 00 |...>.....^.....|

e_type	03 00	<table> <tr> <th>Name</th><th>Value</th><th>Meaning</th></tr> <tr> <td>ET_NONE</td><td>0</td><td>No file type</td></tr> <tr> <td>ET_REL</td><td>1</td><td>Relocatable file</td></tr> <tr> <td>ET_EXEC</td><td>2</td><td>Executable file</td></tr> <tr> <td>ET_DYN</td><td>3</td><td>Shared object file</td></tr> </table>	Name	Value	Meaning	ET_NONE	0	No file type	ET_REL	1	Relocatable file	ET_EXEC	2	Executable file	ET_DYN	3	Shared object file
Name	Value	Meaning															
ET_NONE	0	No file type															
ET_REL	1	Relocatable file															
ET_EXEC	2	Executable file															
ET_DYN	3	Shared object file															
e_machine	3e 00	EM_X86_64 62 AMD x86-64 architecture															
e_version	01 00 00 00	EV_CURRENT 1 Current version															
e_entry	60 10 00 00 00 00 00 00	contains the virtual address of the program entry point. 代码段首地址															

00000020 40 00 00 00 00 00 00 00 98 39 00 00 00 00 00 00 |@.....9.....|

e_phoff	40 00 00 00 00 00 00 00	contains the file offset, in bytes, of the program header table.
e_shoff	98 39 00 00 00 00 00 00	contains the file offset, in bytes, of the section header table.

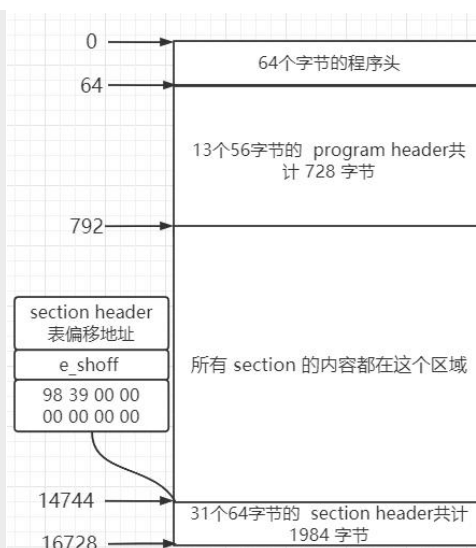
00000030 00 00 00 00 40 00 38 00 0d 00 40 00 1f 00 1e 00 |....@.8...@.....|

e_flags	00 00 00 00	contains processor-specific flags.
e_ehsize	40 00	contains the size, in bytes, of the ELF header. 64 字节
e_phentsize	38 00	contains the size, in bytes, of a program header table entry.
e_phnum	0d 00	contains the number of entries in the program header table.
e_shentsize	40 00	contains the size, in bytes, of a section header table entry.

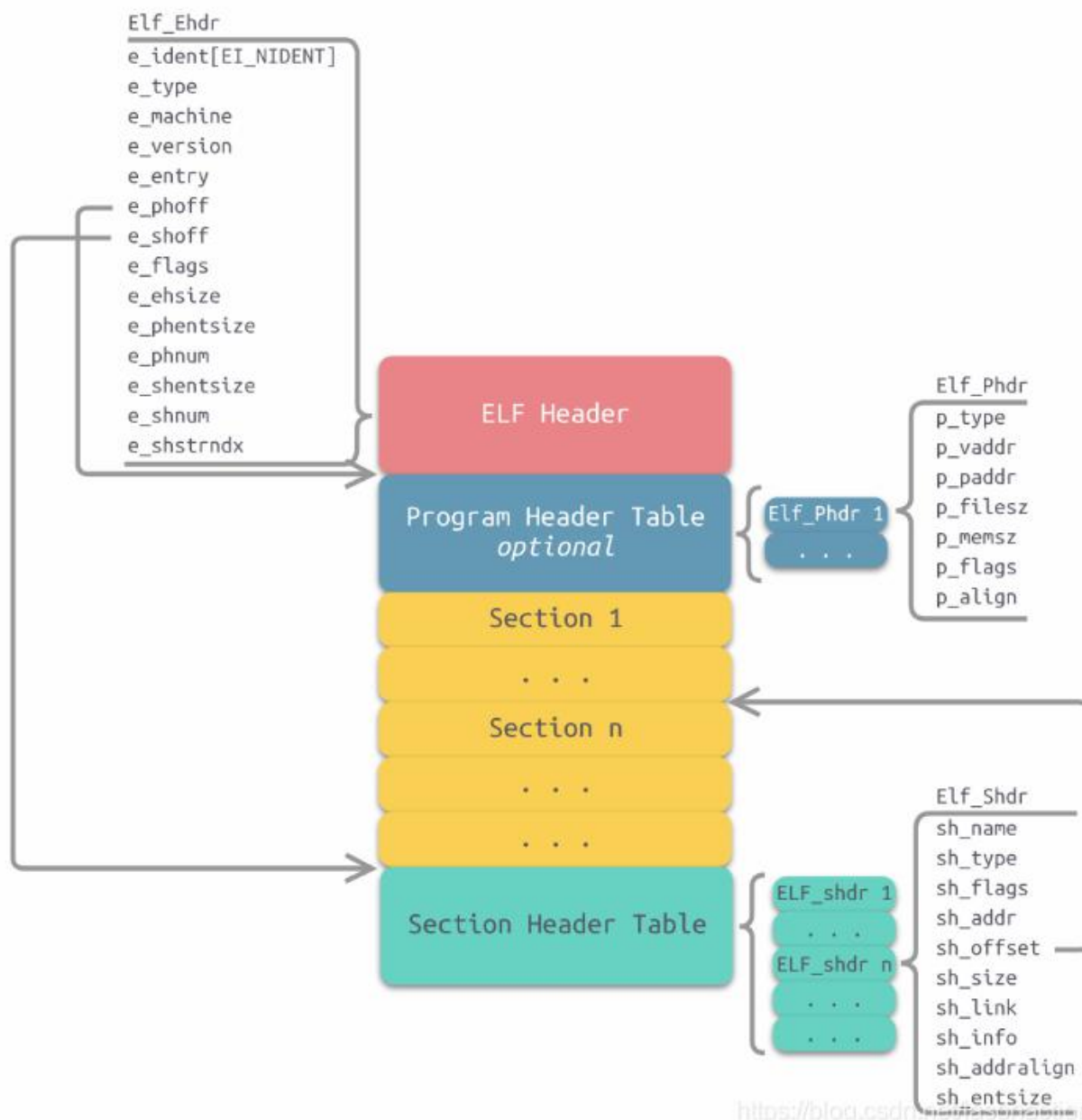
e_shnum	1f 00	contains the number of entries in the section header table.
e_shstrndx	1e 00	contains the section header table index of the section containing the section name string table. -- section name 字符串表对应 section header table 中的索引，也即 30 号 section header 存储的是 section name string.

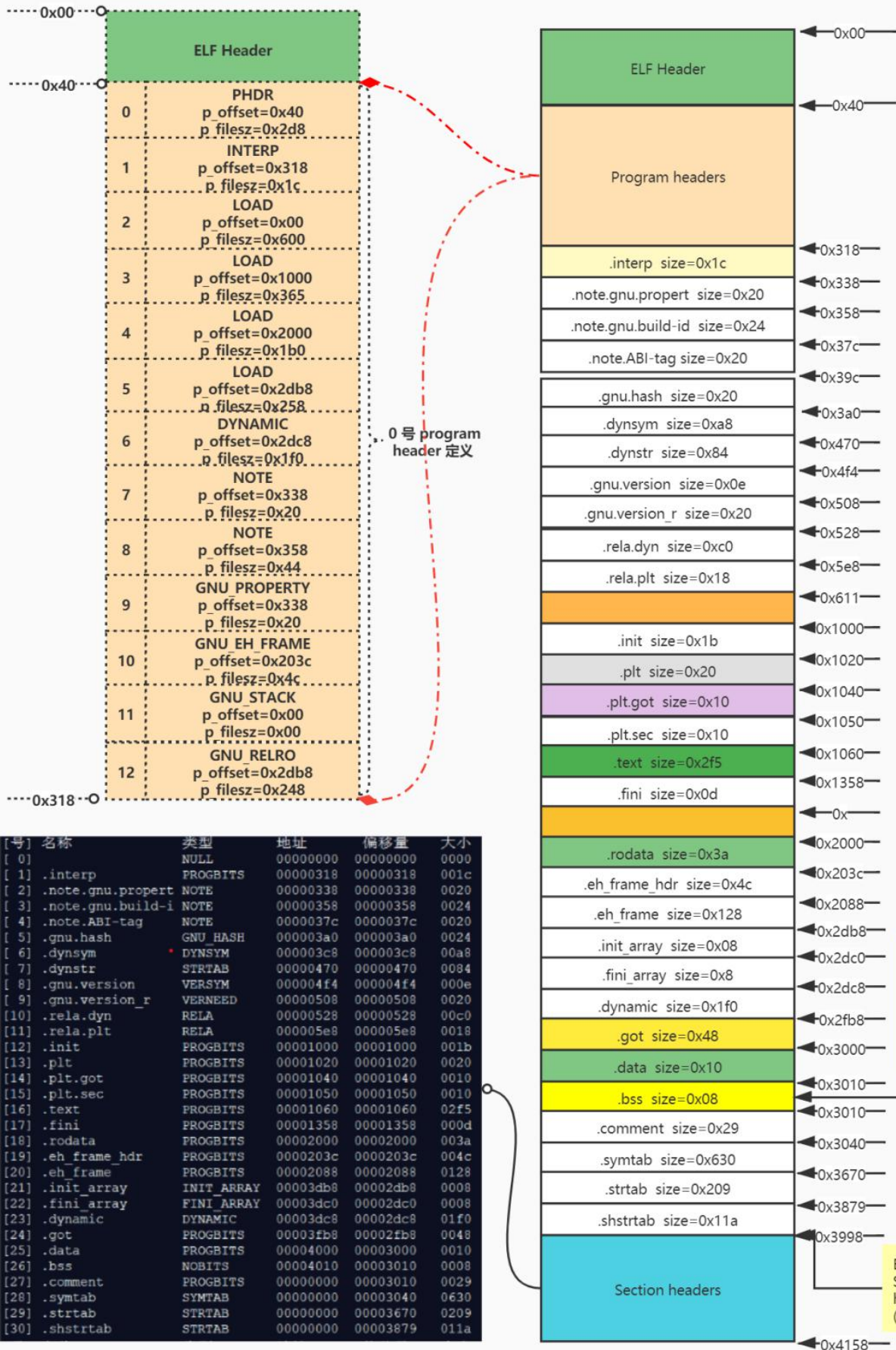
通过上文的解析可以确定，hello64 程序文件的大致结构如下图所示：

```
work@chad-PC:~/ELF-STUDY$ ls -l hello64
-rwxrwxr-x 1 work work 16728 8月 24 11:16 hello64
work@chad-PC:~/ELF-STUDY$ readelf -h hello64
ELF 头:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  类别:                ELF64
  数据:                2 补码，小端序 (little endian)
  Version:            1 (current)
  OS/ABI:             UNIX - System V
  ABI 版本:           0
  类型:               DYN (共享目标文件)
  系统架构:           Advanced Micro Devices X86-64
  版本:               0x1
  入口点地址:         0x1060
  程序头起点:         64 (bytes into file)
  Start of section headers: 14744 (bytes into file)
  标志:               0x0
  Size of this header: 64 (bytes)
  Size of program headers: 56 (bytes)
  Number of program headers: 13
  Size of section headers: 64 (bytes)
  Number of section headers: 31
  Section header string table index: 30
```



ELF 加载的最终结果如下：





三、Intel 指令集

2.1 INSTRUCTION FORMAT FOR PROTECTED MODE, REAL-ADDRESS MODE, AND VIRTUAL-8086 MODE

The Intel 64 and IA-32 architectures instruction encodings are subsets of the format shown in Figure 2-1. Instructions consist of optional instruction prefixes (in any order), primary opcode bytes (up to three bytes), an addressing-form specifier (if required) consisting of the ModR/M byte and sometimes the SIB (Scale-Index-Base) byte, a displacement (if required), and an immediate data field (if required).

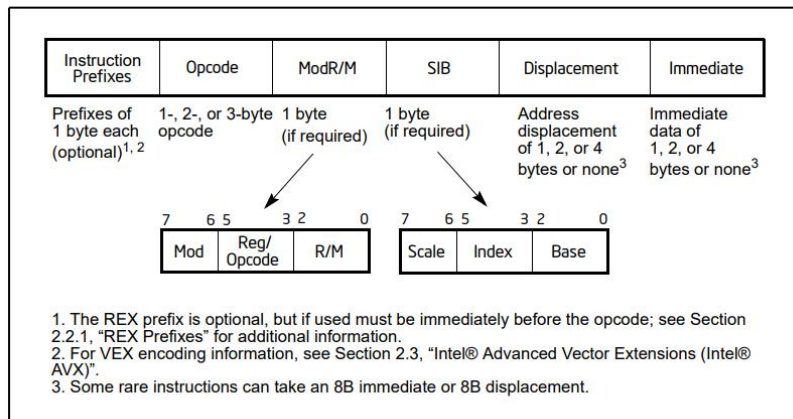


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

机器代码转汇编：<https://onlinedisassembler.com/odaweb/>

x86 汇编指令集大全：<https://www.jianshu.com/p/bbd41e8ebd86>

x86 and amd64 instruction reference：<https://www.felixcloutier.com/x86/>

X86 指令解析：<http://ref.x86asm.net/geek32.html#x11>

x86/x64 SIMD Instruction List (SSE to AVX512)：<https://www.officedaytime.com/simd512e/>

- **Instruction prefixes:** 指令前缀，可选项，每个前缀一个字节，可选 0 个前缀到 4 个不等。详细信息参考 intel manual 2.2 节
- **Opcode:** 操作码，这是唯一不可省略的项，1 到 2 个字节，在某些情况下会有额外的三个位作为补充 opcode，这三个位是 ModR/M 中的 Reg/Opcode，稍后会讲述什么情况下 reg/opcode 作为 opcode 的补充操作码
- **ModR/M (Mod, Reg/Opcode, R/M):** 一共有三个域，mod, reg/opcode, r/m, reg/opcode 在特定情况下作为 opcode 的补充操作码，特定情况下作为第二个操作数寄存器。Mod 域和 R/M 域总共 5 个位，定义了 32 种寻址方式。可选项。
- **SIB (Scale, Index, Base):** 定义 ModR/M 的寻址方式的补充寻址方式，可选项。
- **Displacement:** 偏移，可选，0，1，2，4 个字节
- **Immediate:** 立即数，可选，0，1，2，4 个字节。

在进行下文分析之前，先记住一个结论：

- Opcode ModR/M SIB 这 3 个决定了整个指令的长度
- Opcode 决定 ModR/M 的有无
- ModR/M 决定 SIB 的有无
- Displacement Immediate 的长度由前 3 个 (Opcode ModR/M SIB) 决定

1、Instruction Prefixes: 指令前缀, 可选项 <CHAPTER 2 INSTRUCTION FORMAT - PAGE 1>

也就是说可以有 0, 1, 2, 3, 4 个前缀, 每一个前缀只占用一个字节, 即最多有 4 个字节的前缀。这五种选择有:

- A. 段寄存器前缀 (segment): 2E, 36, 3E, 26, 64, 65——CS, SS, DS, ES, FS, GS
- B. 操作数长度前缀 (operand-size): 可以改变操作数长度 66
- C. 地址长度前缀 (address-size): 可以改变地址长度 67
- D. 总线加锁前缀 (LOCK) / 重复前缀 (rep/repne): F0 F2 F3
- E. Streaming SIMD Extensions prefix, 0FH

下面作出详细的说明:

1) 段寄存器前缀:

```
8B00      MOV EAX,DWORD PTR DS:[EAX]
2E:8B00    MOV EAX,DWORD PTR CS:[EAX]
36:8B00    MOV EAX,DWORD PTR SS:[EAX]
默认情况下是使用了 DS
```

2) 操作数前缀, 允许一个程序在 16/32 位操作数长度之间转换, 默认下是 32 位的, 就是说, 如果加前缀 66H, 就会转换成 16 位的指令:

```
89 C0      MOV EAX,EAX
66 89 C0    MOV AX,AX
```

3) 地址长度前缀, 也是和 2) 一样的, 表示 16/32 位的寻址方式, 在 win32 下编程一般是不用管的:

```
8B00      MOV EAX,DWORD PTR DS:[EAX]——32bit 寻址模式
67: 8B00   MOV EAX,DWORD PTR DS:[BX+SI]——16bit 寻址模式
```

4) 重复前缀, 将会重复操作字符串的每一个元素。

只有 MOVSB, CMPSB, SCASB, LODSB, STOSB, INSB, OUTSB 等字符串操作或 I/O 指令才能使用这些前缀, 举例子:

```
AD      LODSB DWORD PTR DS:[ESI]
F3 AD    REP LODSB DWORD PTR DS:[ESI]
F2 AD    REPNE LODSB DWORD PTR DS:[ESI]
```

5) 总线加锁, 以及 Streaming SIMD Extensions 暂不清楚

总之, prefix 主要起了三个作用: 调整、加强、附加。

2、Opcode: 操作码, 这是唯一不可省略的项, 1、2 或 3 个字节, 在某些情况下会有额外的三个位作为补充 opcode, 这三个位是 ModR/M 中的 Reg/Opcode 域。

```
One-byte Opcode Map:  00H — FFH
Two-byte Opcode Map:  00H — FFH (First Byte is 0FH)
Three-byte Opcode Map: 00H — FFH (First Two Bytes are 0F 38H)
Three-byte Opcode Map: 00H — FFH (First two Bytes are 0F 3AH)
```

随着时间推移、技术进步指令集在不断的扩展, 但是一般而言三字节指令基本不会涉及。

One-byte Opcode Map 示例如下图所示:

一个 1Byte 的定长指令，其 16 进制为类似于“AB”的形式，而第一个 A 是 Opcode Map 表中的行号，第二个的 B 是其列号，行号和列号就能确定一个具体的指令。比如：第 4 行第 5 列索引出来的指令为（绿色圈出来）：inc ebp，第 5 行第 0 列为 push eax 等等。

Table A-2. One-byte Opcode Map (Left)								
	0	1	2	3	4	5	6	7
0	ADD Eb, Gb Ev, Gv Gb, Eb Gv, Ev AL, lb eAX, lv						PUSH ES	POP ES
1	ADC Eb, Gb Ev, Gv Gb, Eb Gv, Ev AL, lb eAX, lv						PUSH SS	POP SS
2	AND Eb, Gb Ev, Gv Gb, Eb Gv, Ev AL, lb eAX, lv						SEG=ES	DAA
3	XOR Eb, Gb Ev, Gv Gb, Eb Gb, Ev AL, lb eAX, lv						SEG=SS	AAA
4	INC general register eAX eCX eDX eBX eSP eBP eSI eDI							
5	PUSH general register eAX eCX eDX eBX eSP eBP eSI eDI							
6	PUSHA/ PUSHAD	POPA/ POPAD	BOUND Gv, Ma	ARPL Ew, Gw	SEG=FS	SEG=GS	Opd Size	Addr Size
7	Jcc, Jb - Short-displacement jump on condition O NO B/NAE/C NB/AE/NC Z/E NZ/NE BE/NA NBE/A							
8	Immediate Grp 1 ^{1A} Eb, lb Ev, lv Ev, lb Ev, lb				TEST Eb, Gb Ev, Gv		XCHG Eb, Gb Ev, Gv	
9	NOP	eCX	eDX	eBX	eSP	eBP	eSI	eDI
A	MOV AL, Ob eAX, Ov Ob, AL Ov, eAX				MOVS/ MOVSB Xb, Yb	MOVS/ MOVSW/ MOVSD Xv, Yv	CMPS/ CMPSB Xb, Yb	CMPS/ CMPSW/ CMPSD Xv, Yv
B	MOV immediate byte into byte register AL CL DL BL AH CH DH BH							
C	Shift Grp 2 ^{1A} Eb, lb Ev, lb		RETN lw	RETN	LES Gv, Mp	LDS Gv, Mp	Grp 11 ^{1A} - MOV Eb, lb Ev, lv	
D	Shift Grp 2 ^{1A} Eb, 1 Ev, 1 Eb, CL Ev, CL				AAM lb	AAD lb		XLAT/ XLATB

注意：这个表是 16 x 16 大小的，在 intel 文档中为了排版清楚被分成了左右两部分，上图是左半部分。故比如查询 Opcode 8B 会发现在上表无法查到，需要到右半部分里查询。

特别注意，标记有 1A 上标的指令需要把 ModR/M 中的第 3, 4, 5 位作为 opcode extension 使用（参见 A. 2. 5. Opcode Extensions For One- And Two-byte Opcodes）

表中 Gb 之类的“G”代表的是操作数的类型，例如 G 代表 REG；小写“b”代表的是此操作数的大小，例如 b 代表 Byte。

常见操作数类型描述：

简写	描述	寻址方式
E	操作数是 REG/MEM	由 ModRM 的 R/M 提供寻址
G	操作数是 REG	由 ModRM 的 REG 提供寻址
I	操作数是 IMM	立即数体现在 Opcode 中
M	操作数是 MEM	由 ModRM 的 R/M 提供寻址，且 MOD! = 11b

熟悉 Opcode Map 后我们可以发现，这些指令是有规律可循的，都遵循寄存器编号的顺序来为汇编指令设计编码的，而寄存器变号我们也有必要提一下：

16进制	000	001	010	011	100	101	110	111
10进制	0	1	2	3	4	5	6	7
8位寄存器	AL	CL	DL	BL	AH	CH	DH	BH
16位寄存器	AX	CX	BX	BX	SP	BP	SI	DI
32位寄存器	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI

3、ModR/M: 一共有三个域, Mod, Reg/Opcode, R/M。

7	6	5	3	2	0
Mod		Reg/Opcode		R/M	

ModR/M, 这个是告诉处理器哪一个寄存器或者内存地址被使用了, 只有一个字节。许多涉及内存操作数的指令都有一个紧挨着主操作码的寻址格式说明字节---ModR/M 字节, ModR/M 字节包含 3 个域信息:

1) mod 域与 r/m 域组合有 32 个可能的值: 8 个寄存器和 24 个寻址模式。

其中的 mod 有 2 bit: 6-7

R/m 有 3 bit: 0-2

所以组合有 5bit, $2^5=32$

2) reg/opcode 域确定寄存器号或者附加的 3 位操作码。reg/opcode 域的用途由主操作码确定。在其中的: 3-5 位。

3) r/m 域确定一个寄存器为操作数或者和 mod 域一起编码寻址模式。有时候有些指令使用特定的 mod 域和 r/m 域组合来表示操作码信息。

这个部分有 2 个表格需要仔细的参考:

16-Bit Addressing Forms with the ModR/M Byte

32-Bit Addressing Forms with the ModR/M Byte

一般而言, 现在 X86 编程只需要 32 位的这个图表了。32 位表如下:

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) /digit (Opcode) REG =	AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111		
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hexadecimal)							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--] ¹		100	04	0C	14	1C	24	2C	34	3C
disp32 ²		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
disp8[EAX] ³ disp8[ECX]	01	000	40	48	50	58	60	68	70	78
		001	41	49	51	59	61	69	71	79

表中特殊字符串释义:

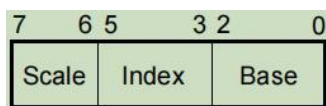
3. 1) [--][--]: 表示使用 SIB 结构。

3. 2) disp32: 表示 32 位偏移。(表中的上标对应注释序号)

3. 3) [--][--]+disp8: 表示使用 SIB 结构, 且 SIB 结构后面有一个 8 位的偏移。

3. 4) [--][--]+disp32: 表示使用 SIB 结构, 且 SIB 结构后面有一个 32 位的偏移。

4、**SIB 字节**，Scale-Index-Base，他的作用是在 ModRM 无法提供更多内存寻址方式时，使用 SIB 进行协助寻址，对 $\text{base} + \text{index} * \text{scale} + \text{disp}$ 这种寻址模式下的内存操作数寻址提供补充定义。



Scale 被认为是 index 寄存器的乘数

<1>00: $2^0=1$ < = > *1

<2>01: $2^1=2$ < = > *2

<3>10: $2^2=4$ < = > *4

<4>11: $2^3=8$ < = > *8

Index 是一个寄存器（除了 ESP），如果索引寄存器是 ESP 就会被忽略，scale 也就被忽略了

Base 是基础寄存器（base register）

这个部分也有 1 个图表需要参考：32-Bit Addressing Forms with the SIB Byte

Table 2-3. 32-Bit Addressing Forms with the SIB Byte										
r32 Base = Base =			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	SS	Index	Value of SIB Byte (in Hexadecimal)							
[EAX]	00	000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	53	54	55	56	57

5、**Displacement**：偏移，可选，0，1，2，4 个字节。

6、**Immediate**：立即数，可选，0，1，2，4 个字节。

7、基于以上知识点 One-Byte Opcode 举例如下：

(1) ADD 指令

Opcode: 030500000000H

LSB address					MSB address
03	05	00	00	00	00

首先，03 不在前缀的取值范围，所以只能是 Opcode。

其次，03 属于 One-Byte Opcode 的取值空间。

然后，查询表 A-2 可知，03 = ADD Gv, Ev 指令。类型 Gv 表示一个通用寄存器，大小是单字数或双字则取决于 operand-size 的属性。类型 Ev 表示该操作数是单字还是双字，是通用寄存器还是内存地址由 ModR/M 指定。

查询表 2-2 可知 ModR/M = 05 (Mod = 00, R/M = 101) 指示 disp32 的位移 (32-bit displacement, reg/opcode[3:5] = 000 说明是 EAX 寄存器 (r32))。

SIB 无

偏移量: 00 00 00 00

最终, 该机器码翻译为汇编如下: ADD EAX, mem_op #mem_op = 00000000H

```
.data:00000000 030500000000 add eax,DWORD PTR ds:0x0
```

(2) Move 指令 r8

解析"88 84 48 12 34 56 78":

MOV—Move		
Opcode	Instruction	Description
88 /r	MOV r/m8,r8	Move r8 to r/m8

- ① Opcode = 88 --> 指令格式: mov Eb, Gb
- ② ModR/M = 84 --> 10 000 100 --> [reg+disp32] (普通格式), al, esp
- ③ 由于 Mod 为 10, 且 R/M 为 ESP, 则属于特殊情况, 不遵循普通格式, 所以下一个字节为 SIB (可确定汇编指令为: mov byte ptr [-][-][disp32], al)

④ [-][-]解析: SIB = 48H --> 01 001 000; Scale=1, Index=1 (ECX), Base=0 (EAX)

⑤ disp32 --> 12 34 56 78 为 32 位偏移量

⑥ 得到汇编指令为: mov byte ptr [eax][ecx2][78563412], al ==> mov byte ptr [eax+ecx*2+0x78563412], al

```
.data:00000000 88844812345678 mov BYTE PTR [eax+ecx*2+0x78563412],al
```

(3) Move 指令 r16

解析"8B 45 EC"

8B /r	MOV r16,r/m16	Move r/m16 to r16
8B /r	MOV r32,r/m32	Move r/m32 to r32
8C /r	MOV r/m16,Sreg**	Move segment register to r/m16

- ① Opcode = 8B --> 指令格式: MOVE Gv, Ev
- ② ModR/M = 45 --> 01 000 101 --> [EAX, disp8[EBP]]
- ③ disp8 --> 8 位偏移量, 即 EC; 位移量 8 位和 16 位是以补码形式存储的, 且计算机指令实际是化减为加通过加一个补码来处理运算, 所以反汇编时就是将上述过程逆向进行, 所以, 其真实值=-14h.

④ 得到汇编指令为: MOVE EAX, EBP + disp8 ==> MOVE EAX, [EBP - 14]

```
.data:00000000 8b45ec mov eax,DWORD PTR [ebp-0x14]
```

8、通过上文的学习, 对机器码翻译的基本知识以及具备, 下文解析直接使用工具完成, 不在进行手工一一翻译。

<https://onlinedisassembler.com/odaweb/> 该网站是一个良好的机器码反汇编网站, 同一段机器码翻译用例如下:

Odaweb 输出结果 (x86-64):

```
.data:00000000 e887feffff callq loc_00000000
.data:00000005 89c6 mov %eax,%esi
.data:00000007 488d3d540d0000 lea 0xd54(%rip),%rdi # 0x00000d62
.data:0000000e b800000000 mov $0x0,%eax
.data:00000013 e87bfdffff callq loc_00000000
.data:00000018 b800000000 mov $0x0,%eax
.data:0000001d c9 leaveq
.data:0000001e c3 retq
.data:0000001f 0f1f4000 nopl 0x0(%rax)
```

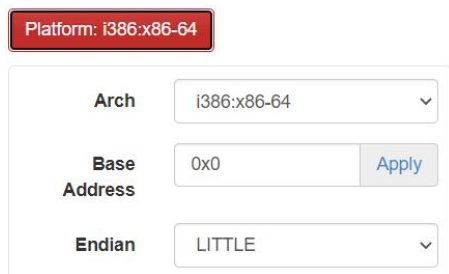
objdump 输出结果 (x86-64):

```

12bd:  e8 87 fe ff ff      callq 1149 <add>
12c2:  89 c6               mov  %eax,%esi
12c4:  48 8d 3d 54 0d 00 00 lea  0xd54(%rip),%rdi    # 201f <_IO_stdin_used+0x1f>
12cb:  b8 00 00 00 00      mov  $0x0,%eax
12d0:  e8 7b fd ff ff      callq 1050 <printf@plt>
12d5:  b8 00 00 00 00      mov  $0x0,%eax
12da:  c9               leaveq
12db:  c3               retq
12dc:  0f 1f 40 00      nopl  0x0(%rax)

```

Odaweb 配置如下：



通过对比可以证明我们可以充分信任该网站的结果。

四、汇编以及编译相关知识学习

(1) AT&T 格式汇编 与 Intel 汇编格式

x86 汇编一直存在两种不同的语法，在 intel 的官方文档中使用 intel 语法，Windows 也使用 intel 语法，而 UNIX/Linux 平台的汇编器一直使用 AT&T 语法。

AT&T 和 Intel 汇编语言的语法区别主要体现在操作数前缀、赋值方向、间接寻址语法、操作码的后缀上，而就具体的指令而言，在同一平台上，两种汇编语言是一致的。下面仅简要列出语法上的两个不同点。

操作数前缀

在 Intel 的汇编语言语法中，寄存器和立即数都没有前缀。但是在 AT&T 中，寄存器前冠以“%”，而立即数前冠以“\$”。在 Intel 的语法中，十六进制和二进制立即数后缀分别冠以“h”和“b”，而在 AT&T 中，十六进制立即数前冠以“0x”：

Intel 语法	AT&T 语法
Mov eax, 8	movl \$8, %eax
Mov ebx, 0ffffh	movl \$0xffff, %ebx
int 80h	int \$0x80

源/目的操作数顺序

Intel 汇编语言的指令与 AT&T 的指令操作数的方向上正好相反：在 Intel 语法中，第一个操作数是目的操作数，第二个操作数源操作数。而在 AT&T 中，第一个数是源操作数，第二个数是目的操作数。

Intel 语法	AT&T 语法
MOV EAX, 8	movl \$8, %eax
mov EAX, [EBP + 12]	movl 12(%ebp), %eax

“l” 是 Longword，相当于 Intel 格式中的 dword ptr 操作限定符；

Gdb 默认是 AT&T 汇编格式，可以通过“set disassembly-flavor intel”命令设置为 Intel 汇编格式。本文统一使用 Intel 汇编格式。linux 内核中使用的是 AT&T 汇编。

更多不同点参考：http://blog.sina.com.cn/s/blog_51e9c0ab010099ow.html

汇编方面知识多而复杂，本文无法一一列举，更多请参考其他教程。

汇编更多知识参考：<https://www.docin.com/p-2142837763.html>

<https://www.cnblogs.com/YukiJohnson/archive/2012/10/27/2741836.html>

(2) 术语

栈：每个进程/线程/goroutine 有自己的调用栈，参数和返回值传递、函数的局部变量存放通常通过栈进行。和数据结构中的栈一样，内存栈也是后进先出，地址是从高地址向低地址生长。

栈帧：（stack frame）又常被称为帧（frame）。一个栈是由很多帧构成的，它描述了函数之间的调用关系。每一帧就对应了一次尚未返回的函数调用，帧本身也是以栈的形式存放数据的。

caller 调用者

callee 被调用者，如在 函数 A 里 调用 函数 B，A 是 caller，B 是 callee

寄存器(X86)

- ESP：（stack pointer）栈指针寄存器，存放着一个指针，该指针指向栈最上面一个栈帧（即当前执行的函数的栈）的栈顶。
- EBP：（base pointer）可称为“帧指针”或“基址指针”寄存器，存放着一个指针，该指针指向栈最上面一个栈帧的底部。
- EIP：指令指针寄存器，它存储的是下一条指令的地址而不是指令本身。
- EAX 是“累加器”（accumulator），它是很多加法乘法指令的缺省寄存器。
- EBX 是“基地址”（base）寄存器，在内存寻址时存放基地址。
- ECX 是计数器（counter），是重复（REP）前缀指令和 LOOP 指令的内定计数器。
- EDX 则总是被用来放整数除法产生的余数。
- ESI/EDI 分别叫做“源/目标索引寄存器”（source/destination index），因为在很多字符串操作指令中，DS:ESI 指向源串，而 ES:EDI 指向目标串。

注意：16 位寄存器没有前缀（SP、BP、IP），32 位前缀是 E（ESP、EBP、EIP），64 位前缀是 R（RSP、RBP、RIP）

函数的参数传递和调用约定

调用约定：声明了函数调用中的参数传递方式和堆栈平衡方式。C 和 C++ 中的调用约定如下类似：

```
void __stdcall add(int a, int b);
```

函数声明中的__stdcall 就是关于调用约定的声明。其中标准 C 函数的默认调用约定是__stdcall，C++全局函数和静态成员函数的默认调用约定是__cdecl，类的成员函数的调用约定是__thiscall。剩下的还有__fastcall，__naked 等。

堆栈平衡方式：因为函数调用过程中，参数需要压栈，所以在函数调用结束后，用于函数调用的压栈参数也需要退栈。那这个工作是交给调用者完成，还是在函数内部自己完成？其实两种都可以。调用者负责平衡堆栈的主要好处是可以实现可变参数（关于可变参数的话题，在此不做过多讨论。如果可能的话，我们可以以一篇单独的文章来讲这个问题），因为在参数可变的情况下，只有调用者才知道具体的压栈参数有几个。下面列出了常见调用约定的堆栈平衡方式：

调用约定	
<code>__stdcall</code>	1) 参数从右向左压入堆栈， 2) 函数自身修改堆栈 3) 函数名自动加前导的下划线，后面紧跟一个@符号，其后紧跟着参数的尺寸
<code>__cdecl</code>	1) 参数压栈顺序是和 <code>stdcall</code> 一样，由右向左压入堆栈。 2) 函数本身不清理堆栈，调用者负责清理堆栈。由于这种变化，C 调用约定允许函数的参数的个数是不固定的
<code>__thiscall</code>	1) 参数从右向左入栈 2) 如果参数个数确定， <code>this</code> 指针通过 <code>ecx</code> 传递给被调用者；如果参数个数不确定， <code>this</code> 指针在所有参数压栈后被压入堆栈。 3) 对参数个数不定的，调用者清理堆栈，否则函数自己清理堆栈
<code>__fastcall</code>	1) 函数的第一个和第二个 <code>DWORD</code> 参数（或者尺寸更小的）通过 <code>ecx</code> 和 <code>edx</code> 传递，其他参数通过从右向左的顺序压栈 2) 被调用函数清理堆栈 3) 函数名修改规则同 <code>stdcall</code>
<code>__naked</code>	编译器不会给这种函数增加初始化和清理代码，更特殊的是，你不能用 <code>return</code> 返回返回值，只能用插入汇编返回结果。这一般用于实模式驱动程序设计

注意，虽然 C 语言里都是借助寄存器传递返回值，但是返回值大小不同时有不同的处理情形。若小于 4 字节，返回值存入 `eax` 寄存器，由函数调用方读取 `eax`。若返回值 5 到 8 字节，采用 `eax` 和 `edx` 联合返回。若大于 8 个字节，首先在栈上额外开辟一部分空间 `temp`，将 `temp` 对象的地址做为隐藏参数入栈。函数返回时将数据拷贝给 `temp` 对象，并将 `temp` 对象的地址用寄存器 `eax` 传出。调用方从 `eax` 指向的 `temp` 对象拷贝内容。

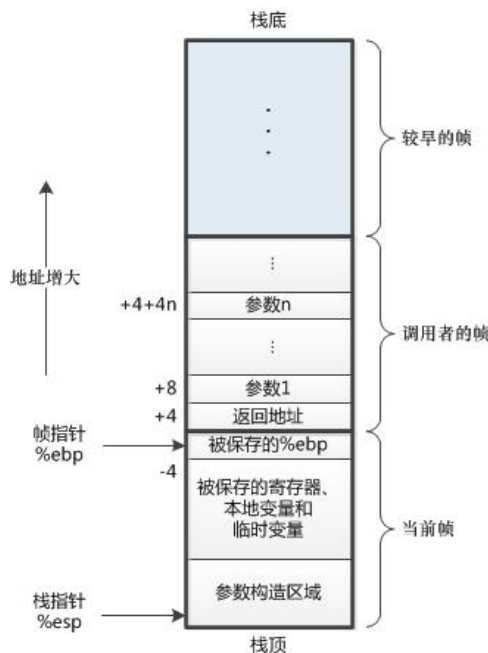
(3) 栈帧的概念：从 `esp` 和 `ebp` 说起

为什么我们需要 `ebp` 和 `esp` 2 个寄存器来访问栈？这种观念其实来自于函数的层级调用：函数 A 调用函数 B，函数 B 调用函数 C，函数 C 调用函数 D...

这种调用可能会涉及非常多的层次。编译器需要保证在这种复杂的嵌套调用中，能够正确地处理每个函数调用的堆栈平衡。所以我们引入了 2 个寄存器：

- 1、`ebp` 指向了本次函数调用开始时的栈底指针，它也是本次函数调用时的“栈底”（这里的意思是，在一次函数调用中，`ebp` 向下是函数的临时变量使用的空间）。在函数调用开始时，我们会使用 `mov ebp, esp` 把当前的 `esp` 保存在 `ebp` 中。
- 2、`esp` 它指向当前的栈顶，它是动态变化的，随着我们申请更多的临时变量，`esp` 值不断减小（正如前文所说，栈是向下生长的）。
- 3、函数调用结束，我们使用 `mov esp, ebp` 来还原之前保存的 `esp`。

在函数调用过程中，`ebp` 和 `esp` 之间的空间被称为本次函数调用的“栈帧”。函数调用结束后，处于栈帧之前的所有内容都是本次函数调用过程中分配的临时变量，都需要被“返还”。这样在概念上，给了函数调用一个更明显的分界。下图是一个程序运行的某一时刻的栈帧图以及一段函数调用的反汇编代码：



```
(gdb) disas
Dump of assembler code for function main:
0x56555522 <+0>:  push    %ebp
0x56555523 <+1>:  mov     %esp,%ebp
0x56555525 <+3>:  sub     $0x10,%esp
0x56555528 <+6>:  call    0x56555554 <__x86.get_pc_thunk.ax>
0x5655552d <+11>:  add     $0x1aaf,%eax
0x56555532 <+16>:  push    $0x8
0x56555534 <+18>:  push    $0x7
0x56555536 <+20>:  push    $0x6
0x56555538 <+22>:  push    $0x5
0x5655553a <+24>:  push    $0x4
0x5655553c <+26>:  push    $0x3
0x5655553e <+28>:  push    $0x2
0x56555540 <+30>:  push    $0x1
=> 0x56555542 <+32>:  call    0x5655554d <add>
0x56555547 <+37>:  add     $0x20,%esp
0x5655554a <+40>:  mov     %eax,-0x4(%ebp)
0x5655554d <+43>:  mov     $0x0,%eax
0x56555552 <+48>:  leave   %eax
0x56555553 <+49>:  ret

End of assembler dump.
(gdb) si
0x5655554d in add ()
(gdb) disas
Dump of assembler code for function add:
=> 0x5655554d <+0>:  push    %ebp
0x5655554e <+1>:  mov     %esp,%ebp
0x5655554f0 <+3>:  call    0x56555554 <__x86.get_pc_thunk.ax>
0x5655554f5 <+8>:  add     $0x1ae7,%eax
0x5655554fa <+13>:  mov     0x8(%ebp),%edx
0x5655554fd <+16>:  mov     0xc(%ebp),%eax
```

(4) “函数调用”的实现

函数调用其实可以看做 4 个过程：

- (1) 压栈：函数参数压栈，返回地址压栈
- (2) 跳转：跳转到函数所在代码处执行
- (3) 执行：执行函数代码
- (4) 返回：平衡堆栈，找出之前的返回地址，跳转回之前的调用点之后，完成函数调用

汇编中有 ret 相关的指令，它表示：

- (1) 取出当前栈顶值，作为返回地址，并将指令指针寄存器 EIP 修改为该值
- (2) 弹出调用者的压栈参数。通过这两个实现函数返回及恢复调用者栈

(5) 一次典型的 C 函数调用过程

在 caller 里：

- 将实参从右至左压栈 (X86-64 下是：将实参写入寄存器，如果实参超过 6 个，超出的从右至左压栈)
- 执行 call 指令 (会将返回地址压栈，并跳转到 callee 入口)

进入 callee 里：

- push ebp; mov ebp, esp; 此时 EBP 和 ESP 已经分别表示 callee 的栈底和栈顶了。之后 EBP 的值会保持固定。此后局部变量和临时存储都可以通过基准指针 EBP 加偏移量找到了。
- sub xxx, esp; 栈顶下移，为 callee 分配空间，用于存放局部变量等。分配的内存单元可以通过 EBP - K 或者 ESP + K 得到地址访问。
- 将某些寄存器的值压栈 (可能)
- callee 执行

- 将某些寄存器值弹出栈(可能)
- `mov esp, ebp; pop ebp;` (这两条指令也可以用 `leave` 指令替代) 此时 `EBP` 和 `ESP` 回到了进入 `callee` 之前的状态, 即分别表示 `caller` 的栈底和栈顶状态。
- 执行 `ret` 指令

回到了 caller 里的代码

具体代码如下:

```
int add(int arg1, int arg2, int arg3, int arg4, int arg5, int arg6, int arg7, int arg8) {
    return arg1 + arg2 + arg3 + arg4 + arg5 + arg6 + arg7 + arg8;
}

int main() {
    int i = add(1, 3, 5, 7, 2, 4, 6, 8);
}
```

x86 版汇编 (gcc -S 生成的汇编与 elf 文件反汇编之后的结果差别很大, 相对来说反汇编结果更容易理解):

```
(gdb) b main
(gdb) r
(gdb) disas
Dump of assembler code for function main:
0x080488da <+0>:    push    ebp
0x080488db <+1>:    mov     ebp, esp
0x080488dd <+3>:    sub     esp, 0x10
0x080488e0 <+6>:    call    0x804890c <__x86.get_pc_thunk.ax>
0x080488e5 <+11>:   add     eax, 0x9071b
0x080488ea <+16>:   push    0x8
0x080488ec <+18>:   push    0x6
0x080488ee <+20>:   push    0x4
0x080488f0 <+22>:   push    0x2
0x080488f2 <+24>:   push    0x7
0x080488f4 <+26>:   push    0x5
0x080488f6 <+28>:   push    0x3
0x080488f8 <+30>:   push    0x1
0x080488fa <+32>:   call    0x80488a5 <add>
0x080488ff <+37>:   add     esp, 0x20
0x08048902 <+40>:   mov     DWORD PTR [ebp-0x4], eax
0x08048905 <+43>:   mov     eax, 0x0
0x0804890a <+48>:   leave
0x0804890b <+49>:   ret
End of assembler dump.
(gdb) disas
```

```
Dump of assembler code for function add:
0x080488a5 <+0>:    push    ebp
0x080488a6 <+1>:    mov     ebp, esp
0x080488a8 <+3>:    call    0x804890c <__x86.get_pc_thunk.ax>
0x080488ad <+8>:    add     eax, 0x90753
=> 0x080488b2 <+13>:   mov     edx, DWORD PTR [ebp+0x8]
0x080488b5 <+16>:   mov     eax, DWORD PTR [ebp+0xc]
0x080488b8 <+19>:   add     edx, eax
0x080488ba <+21>:   mov     eax, DWORD PTR [ebp+0x10]
0x080488bd <+24>:   add     edx, eax
0x080488bf <+26>:   mov     eax, DWORD PTR [ebp+0x14]
0x080488c2 <+29>:   add     edx, eax
0x080488c4 <+31>:   mov     eax, DWORD PTR [ebp+0x18]
```

```
(gdb) print $ebp
$1 = (void *) 0xffffd490
(gdb) print/x *0xffffd498
$2 = 0x1
(gdb) print/x *0xffffd49c
$3 = 0x3
(gdb) print/x *0xffffd4a0
$4 = 0x5
(gdb) i r
eax      0x80d9000    135106560
ecx      0xb8935980  -1198302848
edx      0xffffd524  -10972
ebx      0x80d9000    135106560
esp      0xffffd490   0xffffd490
ebp      0xffffd490   0xffffd490
esi      0x80d9000    135106560
edi      0x80481a8    134513064
eip      0x80488b2    0x80488b2 <add+13>
eflags   0x216       [ PF AF IF ]
cs       0x23        35
ss       0x2b        43
ds       0x2b        43
es       0x2b        43
fs       0x0         0
gs       0x63        99
```

```

0x080488c7 <+34>:  add    edx, eax
0x080488c9 <+36>:  mov     eax, DWORD PTR [ebp+0x1c]
0x080488cc <+39>:  add     edx, eax
0x080488ce <+41>:  mov     eax, DWORD PTR [ebp+0x20]
0x080488d1 <+44>:  add     edx, eax
0x080488d3 <+46>:  mov     eax, DWORD PTR [ebp+0x24]
0x080488d6 <+49>:  add     eax, edx
0x080488d8 <+51>:  pop     ebp
0x080488d9 <+52>:  ret
End of assembler dump.

```

观察上面的汇编代码，可以很容易的与上文的 **C 函数调用过程** 匹配。

Push 与 pushq, pushl 的区别：

GAS (GNU 汇编程序) 汇编指令通常以字母 “b”，“s”，“w” 为后缀，“l”，“q” 或 “t” 确定操作的操作数大小。

```

> b =字节(8 位)
> s =短(16 位整数)或单(32 位浮点)
> w =字(16 位)
> l = long(32 位整数或 64 位浮点)
> q =四(64 位)
> t =十个字节(80 位浮点)

```

五、一个 Bug 分析

Box86 微信点击播放小视频时报错信息：

```

2869|0x3d5c267: Unimplemented Opcode [0F 6F](D3) 0F 3A 0F E0 07 0F 3A 0F
2985|SIGSEGV @0x6286ed70 (???(/usr/bin/box86/0x6286ed70)) (x86pc=0x64a38006/???:"???",
esp=0x979fd1c, stack=0x96a0000:0x97a4000 own=(nil) fp=0x979fd68), for accessing 0x64
(code=1/prot=0), db=(nil)((nil):(nil)/(nil):(nil)/???:clean, hash:0/0)
2985|Double SIGSEGV (code=1, pc=0x6286ed70, addr=0x64)!

```

Unimplemented Opcode [0F 6F](D3) 0F 3A 0F E0 07 0F 3A 0F CA 01 0F

分析 Box86 源码可以确定整理后指令序列如下：

0F 6F D3

0F 3A 0F E0 07 **《——出错位置**

0F 3A 0F CA 01

对应的汇编代码如下：

.data:00000000	0f6fd3	movq mm2,mm3
.data:00000003	0f3a0fe007	palignr mm4,mm0,0x7
.data:00000008	0f3a0fca01	palignr mm1,mm2,0x1

PALIGNR 指令定义如下：

PALIGNR — Packed Align Right

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 3A OF /r ib ¹ PALIGNR mm1, mm2/m64, imm8	RMI	V/V	SSSE3	Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in <i>imm8</i> into <i>mm1</i> .

PALIGNR—Packed Align Right	
mmreg2 to mmreg1, imm8	0000 1111:0011 1010: 0000 1111:11 mmreg1 mmreg2: imm8
mem to mmreg, imm8	0000 1111:0011 1010: 0000 1111: mod mmreg r/m: imm8
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0000 1111:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0000 1111: mod xmmreg r/m: imm8

Table A-5. Three-byte Opcode Map: 08H — FFH (First Two Bytes are 0F 3AH) *

	px	8	9	A	B	C	D	E	F
0									palignr Pq, Qq, lb
66		vroundps Vx,Vx,lb	vroundpd Vx,Wx,lb	vroundss Vss,Vss,lb	vroundsd Vsd,Vsd,lb	vblendps Vx,Hx,Wx,lb	vblendpd Vx,Hx,Wx,lb	vpblendw Vx,Hx,Wx,lb	vpalignr Vx,Hx,Wx,lb

Operation

PALIGNR (with 64-bit operands)

```
temp1[127:0] = CONCATENATE(DEST,SRC)>>(imm8*8)
DEST[63:0] = temp1[63:0]
```

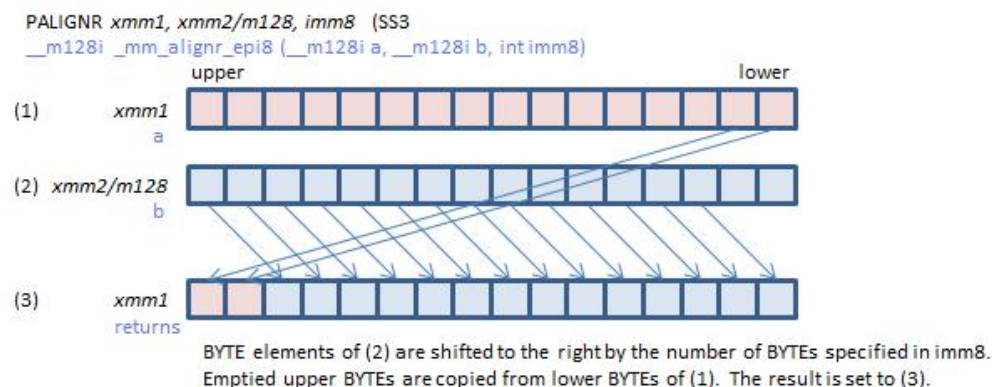
Intel C/C++ Compiler Intrinsic Equivalents

PALIGNR: `__m64 _mm_alignr_pi8 (__m64 a, __m64 b, int n)`

(V)PALIGNR: `__m128i _mm_alignr_epi8 (__m128i a, __m128i b, int n)`

Palignr （包装右对齐）指令详解：

<https://www.officedaytime.com/simd512e/simding/si.php?f=palignr>



在计算机体系中，数据并行有两种实现路径：

MIMD（Multiple Instruction Multiple Data，多指令流多数据流）

SIMD（Single Instruction Multiple Data，单指令流多数据流）。

其中 MIMD 的表现形式主要有多发射、多线程、多核心，在当代设计的以处理能力为目标驱动的处理器的中，均能看到它们的身影。

同时，随着多媒体、大数据、人工智能等应用的兴起，为处理器赋予 SIMD 处理能力变得愈发重要，因为这些应用存在大量细粒度、同质、独立的数据操作，而 SIMD 天生就适合处理这些操作。

SIMD 结构有三种变体：向量体系结构、多媒体 SIMD 指令集扩展和图形处理单元。

(1) MMX 指令——Multi Media eXtension，多媒体扩展指令集

1996 年，MMX 指令集率先在 Pentium 处理器中使用，MMX 指令集支持算数、比较、移位等运算，MMX 指令集的向量寄存器是 64bit。

(2) SSE 指令集系列——Streaming SIMD Extensions，单指令多数据流扩展

SSE 在 1999 年率先在 Pentium3 中出现，向量寄存器由 MMX 的 64bit 拓展到 128bit；

SSE2 在 2002 年出现，包括了 SIMD 的浮点和整型运算的指令以及整型和浮点数据之间的转换；

SSE3 在 2004 年出现，支持不对其访问，处理虚数运算的复杂指令以及水平加减操作运算指令；

SSE4.1 在 2006 年出现，加入了处理字符串文本和面向应用的优化指令；

SSE4.2 指令

(3) 使用 SSE 指令有两种方式：

一是直接在 C/C++ 中嵌入（汇编）指令；

二是使用 Intel C++ Compiler 或是 Microsoft Visual C++ 中提供的支持 SSE 指令集的 intrinsics 内联函数。intrinsics 是对 MMX、SSE 等指令集的一种封装，以函数的形式提供，使得程序员更容易编写和使用这些高级指令，在编译的时候，这些函数会被内联为汇编，不会产生函数调用的开销。想要使用 SSE 指令，则需要包含对应的头文件：

```
#include <mmmintrin.h> //mmx
#include <xmmmintrin.h> //sse
#include <emmintrin.h> //sse2
#include <pmmintrin.h> //sse3
```

<https://blog.csdn.net/fengbingchun/article/details/19293081? t=t>

<https://zhuanlan.zhihu.com/p/325632066>

https://docs.oracle.com/cd/E27071_01/html/E26439/gliwk.html

<https://zhuanlan.zhihu.com/p/31271788>

解决办法

在 run0f.h 中实现如下代码即可：

```
_Of_0x3A: // these are some SSE3 & SSE4.x opcodes
opcode = F8;
switch(opcode) {
    case 0x0F: // PALIGNR GX, EX, u8
        nextop = F8;
        GET_EM;
        uint8_t tmp8u = F8;
        if(tmp8u > 15)
        {
```

```
GM.ud[0] = GM.ud[1] = 0;
}
else
{
    for (int i=0; i<8; ++i, ++tmp8u)
        eam1.ub[i] = (tmp8u>7)?((tmp8u>15)?0:GM.ub[tmp8u-8]):EM->ub[tmp8u];
    GM.ud[0] = eam1.ud[0];
    GM.ud[1] = eam1.ud[1];
}
break;
default:
    goto _default;
}
NEXT;
```