目录

第 18 章 x86-64 位汇编程序设计

x86-64，亦称为 Intel 64 或者 x64（64-bit eXtented），是 x86 架构的 64 位拓展。x86-64 的指令集与 x86-32 指令集兼容。在此之前，Intel 公司推出了 IA-64 架构，但它采用的是全新的指令集，与 x86-32 不兼容，市场反应较为冷淡。目前，在市场上广泛使用的 Intel CPU 采用的是 x86-64 指令集。正是由于 x86-64 与 x86-32 兼容，使得我们在掌握 x86-32 位程序设计后很容易过渡到 x86-64 上来。本章介绍了 x86-64 的运行环境，包括寄存器、寻址方式和指令系统，x86-64 与 x86-32 程序设计的差别和程序示例。

18．1 x86-64 的运行环境

x86-64 指令集在 Pentium 4、Pentium D、Pentium Extreme Edition、Celeron D、Xeon、Intel Core 2、Intel Core i3、Intel Core i5、Intel Core i7 及 Intel Core i9 处理器上使用。从编写汇编语言程序的角度来看，x86-64 与 x86-32 是非常相似的，指令兼容，差异主要体现在引入了 64 位的寄存器，内存寻址采用 64 位地址，指令有一些升级等。

18.1.1 寄存器

1、通用寄存器

x86-64 处理器包含 16 个 64 位的通用寄存器，分别是 rax、rbx、rcx、rdx、rbp、rsi、rdi、rsp、r8、r9、r10、r11、r12、r13、r14、r15。其中前 8 个分别是 32 位的寄存器 eax、ebx、ecx、edx、ebp、esi、edi、esp 的扩展；后 8 个是新增加的通用寄存器。这些寄存器的低双字、最低字、最低字节都可以被独立的访问，各自都有自己的名字。对于前 8 个 64 的寄存器，它们的低双字的名字就是原 32 位寄存器的名字，而 r8-r15 的低双字寄存器为 r8d-r15d。64 位寄存器的最低字也是低双字中的低字，它们的名字分别为 ax、bx、cx、dx、bp、si、di、sp、r8w、r9w、r10w、r11w、r12w、r13w、r14w、r15w。最低字节为：al、bl、cl、dl、bpl、sil、dil、spl、r8b、r9b、r10b、r11b、r12b、r13b、r14b、r15b。

注意，最低字节寄存器的命名和用法发生了较大的变化。首先，原 x86-32 中，si、di、bp、sp 的低字节是没有名字，不能单独使用；而在 x86-64 中，它们有了名字 sil、dil、bpl、spl，即在 16 位寄存器的名字中增加了后缀 l（Low），可以单独使用。另外，新增寄存器的最低字节命名不是以 l 结尾，而是以 b 为后缀。对于 x86-32 中的 ah、bh、ch、dh，它们仍然可以继续使用。当然为统一规范，不建议使用。在机器编码中，通过使用不同的指令前缀来区分 32 位指令和 64 位指令。

2、标志寄存器 rflags

rflags 是一个 64 位的寄存器，是 x86-32 中标志寄存器 eflags 的扩展，其低 32 位与

eflags 完全相同，目前其高 32 位并未使用。

3、指令指针 rip

rip 的作用与 x86-32 中的 eip 是相同的，用来保存当前将要执行的指令的偏移地址。与 eip 一样，不允许在程序中直接使用 rip 的名字，它的值由 CPU 自动维护，不论是顺序执行的指令，还是 call、ret、jmp 和其它条件转移等指令，都会自动的更新 rip。在 x86-64 处理器中，16 位的段寄存器 cs、ds、es、ss、fs、gs 仍保持不变。

4、浮点及多媒体寄存器

对于 x87 FPU 而言，仍然使用 32 环境下的浮点寄存器 st(0)-st(7)。对于 MMX 技术而言，也还是使用原来的 8 个 64 位寄存器 mm0-mm7。对于 SSE，仍使用的 128 位的寄存器 xmm0-xmm7。在 SSE2 中，新增了 8 个 128 位的寄存器 xmm8-xmm15。在 AVX 中，除了原来的 8 个 256 位寄存器 ymm0-ymm7 外，增加了 8 个 256 位的寄存器 ymm8-ymm15。

18.1.2 寻址方式

与 x86-32 处理器一样，x86-64 处理器的寻址方式也是六种：立即寻址、寄存器寻址、直接寻址、寄存器间接寻址、变址寻址、基址加变址寻址。寻址方式没有大的变化，但要注意一些细节。

(1) 在 x86-64 处理器中，内存的地址是 64 位的，因而对应内存寻址的四种方式中，计算出的地址是 64 位的。

(2) 不能使用 32 位的寄存器用于寄存器间接寻址、变址寻址和基址加变址寻址，而应该使用 64 位的寄存器。其中，基址寄存器可以是任意一个通用的 64 位寄存器，而变址寄存器（或称为索引寄存器）是除 rsp 之外的任意一个通用寄存器。

(3) 比例因子仍为 1、2、4、8。

(4) 偏移量为 8 位、16 位、32 位的有符号常量值，而不能使用 64 位的偏移量。

由于偏移量大小的限制，因而使用带变量的变址寻址、基址加变址寻址，如 x[rax]、x[rax+rbx*4]，就会出现问题。变量 x 对应的是其偏移地址，是 64 位的。为了解决这一问题，能够使用以前的方式编写程序，可以在 VS2019 平台中设置有关选项：“项目属性->链接器->系统->启用大地址”,选择“否(/LARGEADDRESSAWARE:NO)”。

（5）立即数的变化

在 mov 指令中，立即数可以是 64 位的，也可以是 8 位、16 位、32 位的。但在其他指令中，立即数不能是 64 位的。对于 32 位立即数，自动采用有符号扩展方式扩展为 64 位有符号数。

例如， 指令“mov rax,0F00000H”的机器码为：48 B8 00 00 00 F0 00 00 00 00 ，执行后 (rax)=00000000F0000000H。在指令中使用的是 64 位立即数。

指令 “add rax,90000000H”的机器码为：48 05 00 00 00 90，对应于：add rax,0FFFFFFFF90000000H。在指令中使用 32 位立即数，但自动采用有符号扩展为 64 位

数。相加的结果为 (rax)=0000000080000000H。

注意，在 VS2019 平台中，在不启用大地址模式的情况下，仍然支持 x86-32 下的用法，即使用 32 位的寄存器作为基址寄存器和变址寄存器。

18.1.3 指令系统

(1) x86-64 指令基本上向下兼容 x86-32 的指令集

对于原 x86-32 位微处理中的指令绝大多指令予以保持。若不涉及到 64 位的地址和 64 位的操作数，依旧可以使用原 x86-32 位指令。例如下列指令依然可用。

```
mov al, 8        ; 一般数据传送指令
xchg  eax,ebx    ; 数据交换指令
add  ah,10h      ; 加法指令
imul  bx,2       ; 有符号乘法指令
shl  ax, 2       ; 逻辑左移指令
jmp  l1          ; 转移指令
```

(2)对大多数的 x86-32 指令进行了升级

升级的指令主要是增加了 64 位数的处理能力，同时在内存地址升级到 64 位。例如：

```
mov  rax,1234567887654321h          add  rax,rbx      sub  rax,1234h
lea  rax, x       cmp  rcx,[rax]    and  [r10+5],dl   shl  rax,4
movsx rax,x       movzx rbx,y       push r10          pop  r11
```

(3)一些指令默认使用的寄存器升级到 64 位的寄存器

例如，push 和 pop 指令使用 rsp 执行栈顶；loop 指令使用 rcx 来控制循环次数；rep/repe/repne 重复前缀使用 rcx 来控制循环次数；串操作指令中 movsb/movsw/movsd、cmpsb/cmpsw/cmpsd、lodsb/lodsw/lodsd 等指令使用 rsi、rdi 来指向操作数的地址。

(5) 增加了一些新的指令

cdqe：将 eax 中的双字符号扩展为 rax（convert doubleword to quadword）

movsq/cmpsq/lodsq/scasq/stosq: 串操作指令，一次处理 4 个字

此外新增的指令还有：syscall、sysret、cmpxchg16b、swapgs 等。

(6) 删除了几条指令

有几条在 x86-32 位平台下很少使用的指令不再支持,例如:pusha、popa、pushad、popad、aaa、aas 等。

此外，在 x86-64 环境中，仍然可以使用 x87 FPU、MMX、SSE、AVX 的指令集进行浮点数运算、单指令多数据流的组合数据运算,这对遗留的代码升级会很简便。在新代码的开发中，建议直接使用最新技术的指令集，如 AVX，而不使用 x87 FPU 和 MMX 等指令集。

18．2　64 位的程序设计

18.2.1　64 位平台下与 32 位平台下的差别

虽然从机器语言的角度来看，x86-32 与 x86-64 平台下程序的差别不大，但是使用 VS2019 开发汇编语言程序时，还要使用编译和链接器将源程序翻译成机器语言程序。两者的编译链接器不同，分别是 ml.exe 和 ml64.exe。两种编译器对伪指令的支持有较大的差别。ml64.exe 对于很多高级用法不再支持，具体如下。

① 不再支持处理器选择伪指令 ， 无 “.686P”、“.xmm”的用法；

② 不再支持存储模型说明伪指令，无“.model”的用法；

③ 不再支持 invoke 伪指令，函数调用的参数传递由编程者自己掌控；

④ 不再支持 条件流控制伪指令；

⑤ 不支持 “end 表达式”的用法，要在项目属性中设置程序的入口点；

由于是自己设置程序运行的入口点（在“项目属性→链接器→高级→入口点”输入），入口点的名称可自由设定。

当子系统选择为“窗口 (/SUBSYSTEM:WINDOWS) ”时，有一个缺省的入口点 WinMainCRTStartup，即程序中有 “WinMainCRTStartup proc …. WinMainCRTStartup endp”。当子系统选择为“控制台 (/SUBSYSTEM:CONSOLE) ”时，缺省的入口点是 mainCRTStartup。

⑥ 不支持在 C 语言程序中内嵌汇编；

⑦ 不支持 “.stack”定义堆栈段。

由于 ml64 对高级伪指令用法不再支持，使得汇编语言程序的编写会显得麻烦一些。除此之外，在 C 语言函数、Windows API 函数调用方面也出现了较大的变化。下面通过一个具体的例子来比较 32 位和 64 位平台的区别。

例如，设定义有 int x,y,z,u,v;

```
printf("%d %d %d %d %d \n", x, y, z, u, v);
```

在 win32 平台下的翻译结果如下。

```
00AD1433 8B F4              mov         esi,esp
00AD1435 8B 45 A4           mov         eax,dword ptr [v]
00AD1438 50                 push        eax
00AD1439 8B 4D B0           mov         ecx,dword ptr [u]
00AD143C 51                 push        ecx
00AD143D 8B 55 8C           mov         edx,dword ptr [z]
00AD1440 52                 push        edx
00AD1441 8B 45 BC           mov         eax,dword ptr [y]
00AD1444 50                 push        eax
00AD1445 8B 4D C8           mov         ecx,dword ptr [x]
00AD1448 51                 push        ecx
00AD1449 68 58 58 AD 00     push        0AD5858h
00AD144E FF 15 14 91 AD 00  call        dword ptr ds:[0AD9114h]
```

```
00AD1454 83 C4 18           add         esp,18h
00AD1457 3B F4              cmp         esi,esp
00AD1459 E8 D8 FC FF FF     call        __RTC_CheckEsp (0AD1136h)
```

Win32 平台中，函数参数采用堆栈传递参数，在执行 call 语言之前，从右向左逐个将参数压栈。此外，函数调用者检查了栈是否平衡，即调用函数前的栈顶指针与调用函数后的栈顶指针是否相同。实现方法也不难，在传递参数前有“mov esi,esp”，函数执行完后由“cmp esi,esp”，并在__RTC_CheckEsp 中根据标志位判断是栈否平衡。

在 x64 平台下，其翻译的结果发生了变化，结果如下。

```
000000013F77108B  8B 44 24 54          mov    eax,dword ptr [v]
000000013F77108F  89 44 24 28          mov    dword ptr [rsp+28h],eax
000000013F771093  8B 44 24 50          mov    eax,dword ptr [u]
000000013F771097  89 44 24 20          mov    dword ptr [rsp+20h],eax
000000013F77109B  44 8B 4C 24 5C       mov    r9d,dword ptr [z]
000000013F7710A0  44 8B 44 24 4C       mov    r8d,dword ptr [y]
000000013F7710A5  8B 54 24 48          mov    edx,dword ptr [x]
000000013F7710A9  48 8D 0D 50 7F 00 00 lea    rcx,[$xdatasym+0DA0h (013F779000h)]
000000013F7710B0  FF 15 72 A1 00 00    call qword ptr [__imp_printf (013F77B228h)]
```

从 C 语言程序的编译结果来看：

① 前 4 个参数依次在 rcx、rdx、r8、r9 中；edx、r8d、r9d 分别是 rdx、r8、r9 的低双字；更进一步，如果只有一个参数，就只使用 rcx 存放该参数；如果有第二参数，第二个参数就被存放到 RDX 中，依次类推。

② 当参数多于 4 个时，多的那部分参数从右至左入栈。但是这些参数放入栈的位置等同于所有参数都入栈的位置。超出 4 个参数的部分压入栈中，但为前 4 个参数在栈中保留了空间。在执行 call 指令之前，栈中数据存放结果如下。

执行 call 时，此处将要存放断点地址	
将要在 printf 函数体中，预留存放 rcx	← rsp
预留存放 rdx	rsp+8
预留存放 r8	rsp+10h
预留存放 r9	rsp+18h
参数 u	rsp+20h
参数 v	rsp+28h

③ 每个参数占 8 个字节；

④ 长度不足 64 位的参数不进行零扩展，因此其高位的值是不确定的。在函数的实现体中应正确使用相应的参数；

⑤ 函数调用者不再进行栈的平衡检查。

在 64 位平台下，当汇编语言调用 C 和 API 函数时，必须遵循上面的约定，即将参数放

入指定的寄存器中或者精确控制放入栈中的位置。对 printf 函数而言，跟踪进入该函数，立即就可以看到如下语句：

```
mov         qword ptr [rsp+8],rcx
mov         qword ptr [rsp+10h],rdx
mov         qword ptr [rsp+18h],r8
mov         qword ptr [rsp+20h],r9
```

后面也会使用栈中的值，如果未能正确将参数放入指定的寄存器中，程序的运行结果也是非预期的。

当然，从机器语言的角度来看，不论是 mov、push、pop、call、ret，还是其他指令，都有严格规定的语义，这些指令对参数传递的方法是没有约束的。只要函数（子程序）的编写者和调用者之间约定好参数和结果传递的规则即可。因此，单纯用汇编语言开发程序时，可以自己随意控制参数的传递，可以使用寄存器、约定单元、栈或者几种方式的组合，并没有用 rcx、rdx、r8、r9 等传递参数的要求。如果在汇编语言中调用 C 语言函数，或者 C 语言程序中调用汇编语言编写的函数，就必须遵循前面看到的参数传递规则，遵循调用者和被调用者之间的约定，这些约定与编译器是有很大关系的。

18.2.2 显示一个消息框

下面给出一个完整的程序示例，弹出一个消息框。

```
.data
  MessageBoxA proto
  lpContent db  'Hello x86-64',0
  lpTitle   db  'My first x86-64 Application',0
.code
start proc
   sub  rsp, 28h
   xor  r9d, r9d
   lea  r8,  lpTitle
   lea  rdx, lpContent
   xor  rcx,rcx
   call MessageBoxA
   add  rsp,28h
   ret
start endp
end
```

在程序中，遵循了 API 函数 MessageBoxA 调用时的参数约定规则。此外，在程序的开头有“sub rsp,28h”,实际上就是为函数 MessageboxA 的 4 个参数加上该函数调用的断点地址（8*5=28h）留出空间。最后再使用“add rsp,28h”，使得栈保持了平衡。这样执行 ret 才能正确返回到调用子程序 start 保存的断点处。如果没有“sub rsp,28h”或者留出的空间不够，在程序运行中会出现异常。

此外，在外部函数说明中使用了“MessageBoxA proto”。由于编译器不支持 invoke 伪指令，编译器也不用关心 MessageBoxA 有几个参数、参数传递顺序、如何消除参数所占的空

间等，因而在说明外部符号时可以简化，不再列出语言类型及各参数的类型。另外，也可以采用“extrn MessageBoxA : proc”或者“extern MessageBoxA : proc”的形式。

值得注意的是，虽然在32位平台和64位平台上调用的C语言库函数和Windows API函数名相同，但是它们是两个不同的版本。

18.2.3 浮点数运算

下面是一个简单的浮点数相加(z=x+y)然后显示结果的完整程序。在程序中使用了 AVX指令，与x86-32平台上所使用的指令完全相同。只是实现printf函数的库有所变化。

```
    extern ExitProcess :proc
    extern printf   :proc
    includelib libcmt.lib
.data
    lpFmt   db  "%f",0ah, 0dh, 0
    x  real4  3.14
    y  real4  5.701
    z  real4  0.0
.code
main  proc
    vmovss xmm0, x
    vaddss xmm0,xmm0, dword ptr y
    vmovss z, xmm0
    cvtss2sd xmm0, z
    movd rdx, xmm0
    lea  rcx, lpFmt
    sub  rsp, 28h
    call printf
    add  rsp,28h
    mov  rcx,0
    call Exitprocess
main  endp
end
```

18.2.4 程序自我修改

下面给出了一个简单的程序自我修改的完整例子。在程序的运行中，修改了机器码，使得程序运行结果发生了变化。

```
 extern  MessageBoxA : proc
 extern  ExitProcess : proc
 extern  VirtualProtect : proc
.data
 szMsg1   db 'before Modify : Hello ',0
 szMsg2   db 'After  Modify : Interesting ',0
```

```
 szTitle  db 'Modify Program Self',0
 oldprotect   dd ?
.code
 mainp proc
  sub  rsp, 28H
  lea  r9,  oldprotect  ; 对应参数 : pfloldProtect 指向前一个内存保护值
  mov  r8d, 40H         ; 对应参数 : flNewProtect  要应用的内存保护的类型
  mov  rdx, 1           ; 对应参数 : dwsize,要更改的内存页面区域的大小
  lea  rcx, ModifyHere  ; 对应 lpAddress，要更改保护特性的虚拟内存的基址
  call VirtualProtect
  add  rsp, 28H
  lea  rax, ModifyHere
  inc  byte ptr [rax]   ; jz,jnz 的机器码分别 为 74H，75H
                        ; 可比较有此语句和无此语句程序运行结果的差异
  lea  rdx, szMsg1
  xor  eax, eax
ModifyHere:
  jz   next
  lea  rdx, szMsg2
next:
  sub  rsp, 28H
  mov  r9d, 0
  lea  r8,  szTitle
  mov  rcx, 0
  call MessageBoxA
  add  rsp, 28h
  mov  rcx, 0
  call ExitProcess
mainp endp
end
```

注意，在工程中的“项目属性->链接器->高级->入口点”中设置入口点 mainp。运行该程序将显示一个对话框“After Modify : Interesting”。从程序的执行流程来看，“xor eax, eax”后，一定有(eax)=0，ZF=1。单纯地看“jz next”的转移条件成立，要转移到 next 处，从而显示“before Modify : Hello”。但是，实际上，之后有语句“lea rax, ModifyHere”和 “inc byte ptr [rax]”，修改了程序，使得转移语句变成了“jnz next”,从而使得显示的串地址为 szMsg2。如果注释掉程序中的语句“inc byte ptr [rax]”，显示的结果就是“before Modify : Hello”。

习题 18

18.1 在支持 x86-64 位的指令的 CPU 中有哪 16 个 64 位的通用寄存器？有哪 16 个 32 位的通用寄存器？16 位和 8 位的通用寄存器有有哪些？

18.2 x86-32 与 x86-64 的寻址方式有何异同？

18.3　x86-32 与 x86-64 的指令系统有何异同？

18.4　x86-32 与 x86-64 的子程序调用有何差别？

上机实践 18

18.1　编写一个 C 语言程序，分别采用 32 位平台和 x64 平台进行编译。比较生成的机器代码的差别。

18.2 在 x64 平台下编写一个汇编语言源程序，实现对一组数据排序然后输出的功能。

18.3 在 VS2019 平台中，在不启用大地址模式的情况下，设有程序片段

```
mov    rax, 1234567887654321H
lea    eax, x
mov    cl, [eax]
lea    rax, x
```

其中 x 是数据段定义的一个变量。

是比较两条 lea 指令的机器码中有何差别？机器码中所反映出的变量 x 地址的偏移量是以什么为参照的？

在启用大地址模式的情况下，上述程序能够编译生成执行程序，但是执行到 mov cl,[eax]时，程序崩溃，试解释原因。