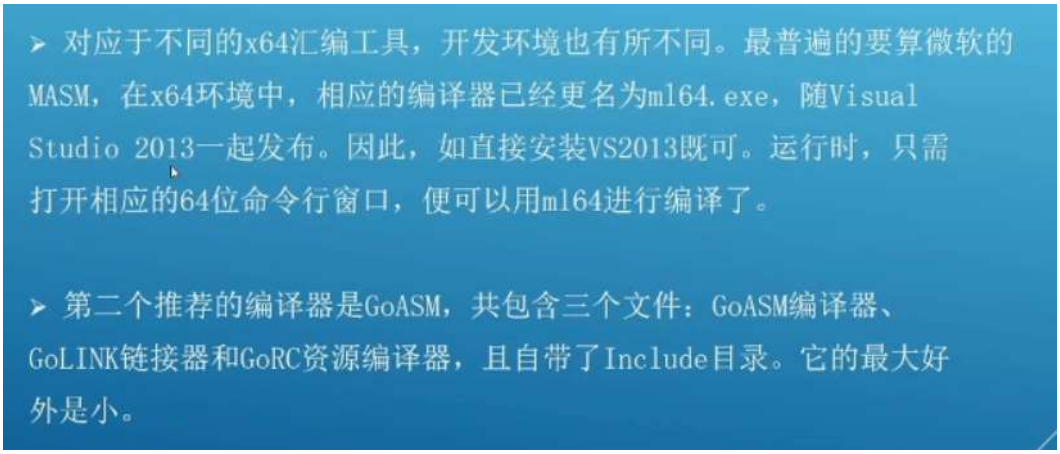
2021/05/24_x64汇编与逆向_第1课_x64汇编寄存器的变化、调用约定

笔记本: x64汇编与逆向
创建时间: 2021/5/24 星期一 15:06
作者: ileemi

64位汇编开发

手册参考：AMD64、Inter64
AMD有些指令在InterCPU上不能正常运行。使用汇编指令前要确保Inter、AMDCPU间的兼容性。

编译器的选择：



x32位上使用的大部分伪指令在x64汇编中不支持使用（invoke、if、else等），API的调用只能使用 "call"。同时微软还提供了 "编译器内部函数"。

编译器内部函数：大多数函数都包含在库中，但也有一些函数是在编译器中生成的（即内部函数）。 这些被称为内联函数或内部函数。微软文档说明：

https://docs.microsoft.com/zh-cn/cpp/intrinsics/compiler-intrinsics?view=msvc-160

寄存器的变化

AMD指令手册第3章

通用寄存器

x32:

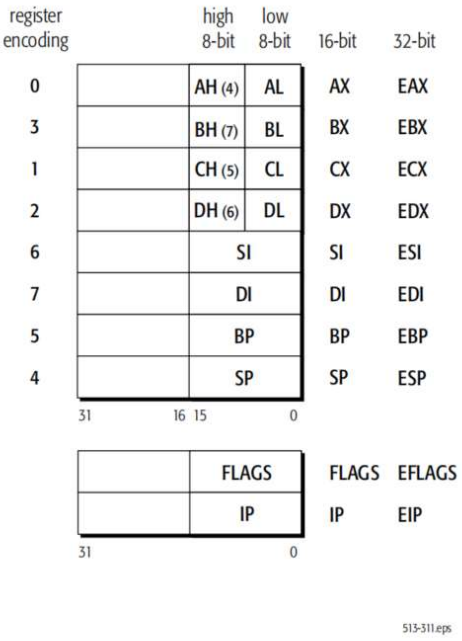


Figure 2-2. General Registers in Legacy and Compatibility Modes

x64：

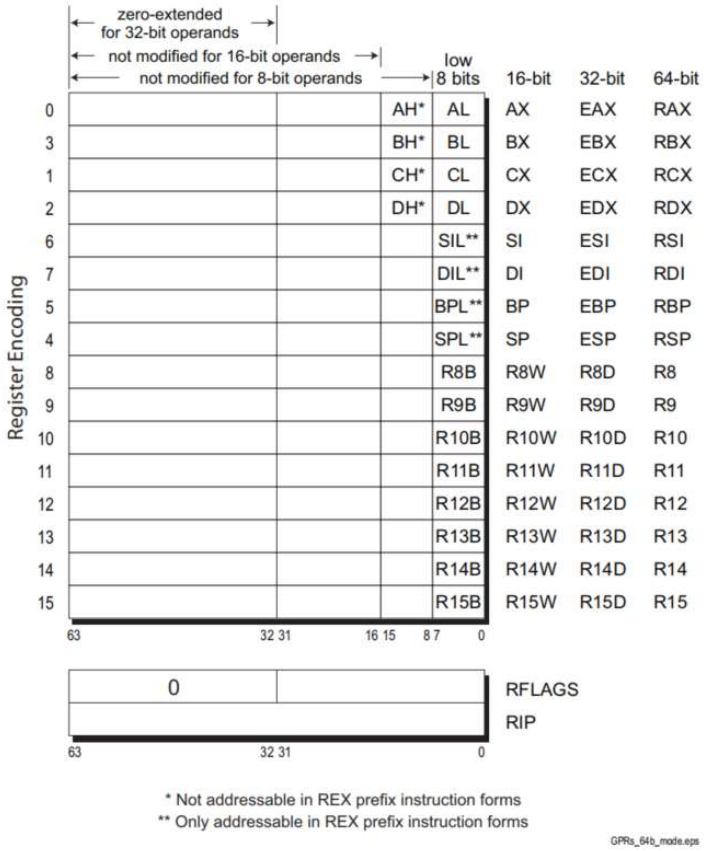


Figure 2-3. General Registers in 64-Bit Mode

- 低8位、16位做修改，高位值不变
- 低32位修改，高32位会置0：mov rax, 0 == mov eax, 0 --> 立即数只需要32位

> 通用寄存器扩展到16个，大小64位。增加 R8~R15寄存器，新增加寄存器高低位的访问：R8（64位），R8d（低32位），R8w（低16位），R8b（低8位）

段寄存器

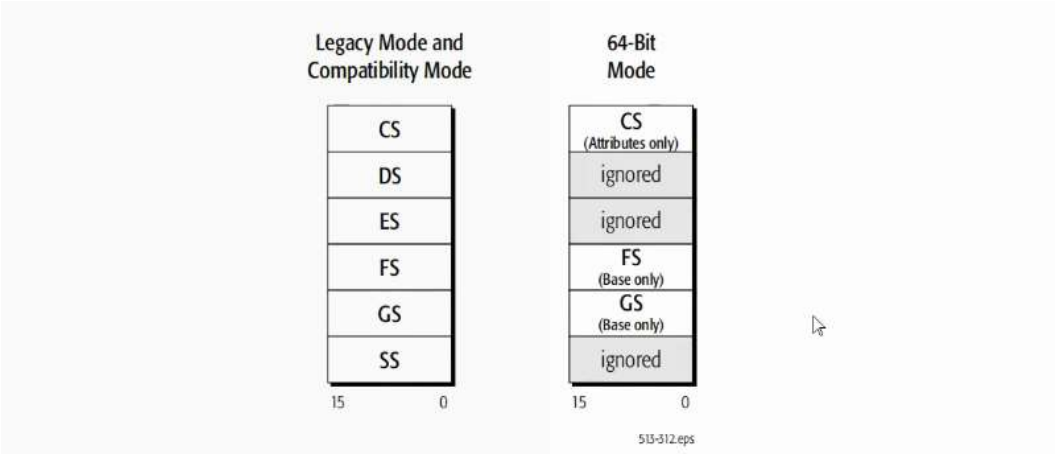


Figure 2-4. Segment Registers

多媒体指令寄存器

多媒体指令寄存器扩展了8个，扩展到16个，大小256位：xmm8~xmm15

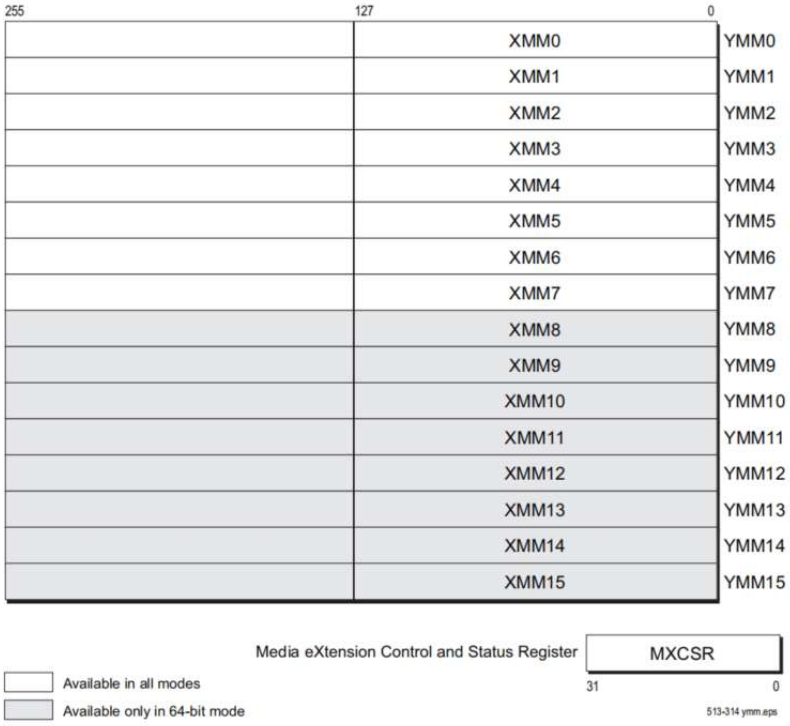


Figure 2-8. SSE Registers

x64调用约定

> 默认情况下，x64 应用程序二进制接口 (ABI) 使用四寄存器 **fast-call** 调用约定（所有函数都是fastcall）。 系统在调用堆栈上分配空间作为影子存储，供被调用方保存这些寄存器。

微软文档说明：https://docs.microsoft.com/zh-cn/cpp/build/x64-calling-convention?view=msvc-160#alignment

参数

- 整数参数在寄存器 RCX、RDX、R8 和 R9 中传递，超过时参数入栈
- 浮点数参数在 XMM0L、XMM1L、XMM2L、XMM3L 中传递
- 16 字节参数按引用传递

```
// 所有整数
func1(int a, int b, int c, int d, int e, int f);
// a in RCX, b in RDX, c in R8, d in R9, f then e pushed on stack

// 所有浮点数
func2(float a, double b, float c, double d, float e, float f);
// a in XMM0, b in XMM1, c in XMM2, d in XMM3, f then e pushed on stack
```

```
// 整数和浮点数混合
func3(int a, double b, int c, float d, int e, float f);
// a in RCX, b in XMM1, c in R8, d in XMM3, f then e pushed on stack

// __m64、__m128  和聚合
func4(__m64 a, __m128 b, struct c, float d, __m128 e, __m128 f);
// a in RCX, ptr to b in RDX, ptr to c in R8, d in XMM3,
// ptr to f pushed on stack, then ptr to e pushed on stack
```

返回值

- 整型：RAX 返回
- 浮点：XMM0 返回

用户定义类型可以从全局函数和静态成员函数通过值返回。 若要将用户定义类型通过值返回到 RAX 中，其长度必须为 1、2、4、8、16、32 或 64 位。 它还必须没有用户定义的构造函数、析构函数或复制赋值运算符。 它不能具有私有或受保护的非静态数据成员，也不能具有引用类型的非静态数据成员。 它不能具有基类或虚拟函数。 而且，它只能有同样满足这些要求的数据成员。

对齐值

> 大多数结构都按其自然对齐方式对齐。 主要的例外是堆栈指针
> 和 malloc 或 alloca 内存；为了提高性能，它们对齐到 **16** 字节。 若要对齐到 16 字节以上，**则必须手动完成。** 由于 16 字节是 XMM 运算的常见对齐大小，因此该值应当适用于大多数代码。

call指令会将返回地址入栈，会导致栈地址不是模16。每个函数中在调用别的函数前必须保证栈顶地址是模16（不保证，栈会崩。最低位为0就是模16），申请的栈模8（直接看最低位是否为8即可）。所以在编写64位程序时，不允许使用内联汇编和裸函数，只能使用联合编译（可以控制栈对齐）。

> 为了支持不定参，由调用方平衡堆栈。
> ;pop rbx ;amd ok intel error

movaps（a-对齐）：访问的内存地址需要模16

对于原型函数，在传递参数之前，所有参数都将转换为所需的被调用方类型。 调用方负责为被调用方的参数分配空间。 调用方必须始终分配足够的空间来存储 4 个寄存器参数（sub rsp, 20H），即使被调用方不使用这么多参数。调用之前，必须将除前 4 个参数外的其他参数存储在影子存储后面的堆栈中

有call的情况下，栈空间就必须抬28H，做为预留参数栈空间。

```
    ;push rbx
    ;push rcx
    sub  rsp,  38h  ;预留参数栈空间

    mov  rcx, NULL
    mov  rdx, offset MY_MSG
    mov  r8,  offset MY_TITLE
    mov  r9d, MB_OK
    mov  qword ptr [rsp+20h], 1
    mov  qword ptr [rsp+28h], 2
    call MessageBoxA      ;mov qword [rsp+0h], rcx
                          ;mov qword [rsp+10h], rdx
                          ;mov qword [rsp+18h], r8
                          ;mov qword [rsp+20h], r9
    mov  ecx, 0

    mov  qword ptr [rsp+20h], 0
    call ExitProcess    ;5params

    ;pop rbx    ;amd ok  intel error
    add rsp, 38h
main endp

end
```

内存布局

```
//预留参数空间
rsp+00h 0
rsp+08h 0
rsp+10h 0
rsp+18h 0

//参数空间，没有参数时，为局部变量空间
rsp+20h 0
rsp+28h 0
rsp+30h 0

//局部变量空间
rsp+38h 0       local1
//sub    rsp, 58h   ;预留参数栈空间+参数空间+局部变量空间

// 保存的寄存器环境
push rbx
push rcx

// 函数返回值
```

联合编译

```
#include <stdio.h>
#include <windows.h>
extern "C" int MyAdd(int n1, int n2);
```

```
int main()
{
  WM_CLOSE
  printf("1+2=%d\n", MyAdd(1, 2));
  return 0;
}

// 汇编代码
.code
  MyAdd proc
    push rbx ;保存环境
    sub rsp, 30h ;预留空间

    mov [rsp+40h], ecx ;参数1预留空间
    mov [rsp+48h], edx ;参数2预留空间

    mov eax, [rsp+40h]
    add eax, [rsp+48h]

    add rsp, 30h
    pop rbx
    ret
  MyAdd endp
end
```

64位汇编的编写

```
// 编译命令:
ml64 /c Hello.asm
link hello.obj test.res /entry:main /SUBSYSTEM:WINDOWS

extern MessageBoxA:proc
extern ExitProcess:proc
extern DialogBoxParamA:proc
extern GetModuleHandleA:proc
extern EndDialog:proc

includelib user32.lib
includelib kernel32.lib

;寄存器的变化   dword   word  byte
NULL EQU 0
MB_OK EQU 0
```

```
IDD_DIALOG1 EQU 101
WM_CLOSE EQU 0010h

.const
  MY_TITLE db "51asm", 0
  MY_MSG db "Hello World!", 0

.code
DialogProc proc
  cmp edx, WM_CLOSE
  jnz LABEL1
    mov edx, 0
  call EndDialog
  mov eax, 1
  ret
LABEL1:
  mov eax, 0
  ret
DialogProc endp

main proc
  // 预留空间，调用函数之前必须要给预留空间，
  // 且多个函数可以共用，多余的参数在预留空间下面
  sub rsp, 58h  ;预留参数栈空间+参数空间+局部变量空间

  mov rcx, NULL
  mov rdx, g MY_MSG
  mov r8, offset MY_TITLE
  mov r9d, MB_OK
  ;mov qword ptr [rsp+20h], NULL  参数5
  ;mov qword ptr [rsp+28h], NULL  参数6
  ;mov qword ptr [rsp+30h], NULL  参数7
  ;mov qword ptr [rsp+38h], NULL  参数8

  call MessageBoxA
    ;mov qword ptr[rsp+8h], rcx    // 需要保存的参数存放到栈空间中
    ;mov qword ptr[rsp+10h], rdx
    ;mov qword ptr[rsp+18h], r8
    ;mov qword ptr[rsp+20h], r9

  mov ecx, NULL
  call GetModuleHandleA

  mov rcx, rax
  mov rdx, IDD_DIALOG1
  mov r8d, NULL
  mov r9, offset DialogProc
```

```
    mov qword ptr [rsp+20h], NULL
    call DialogBoxParamA

    mov rcx, 0
    ;mov qword ptr [rsp+20h], 0  参数5
    call ExitProcess

    ;pop rbx ;amd ok  intel error
    add rsp, 58h
main endp
end
```

官方文档

- Inter手册：https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html
- AMD手册：https://developer.amd.com/resources/developer-guides-manuals/