X86与x64体系结构

孙聪

网络与信息安全学院

2020-09-07

课程内容

- 概述
- x86与x64体系结构
- 静态反汇编
- PE/ELF文件格式
- Windows系统编程基础
- DLL注入
- API钩取
- 调试工具与调试技术
- 混淆技术

提要

提要

基本概念

x86：基于Intel 8086/8088处理器的一系列向后兼容的指令集体系结构(ISA)的总称

- IA-32：32位版本的x86指令集体系结构
- 具有三种主要操作模式
  - 实模式(Real Mode)：只支持16位指令集和寄存器，MS-DOS的运行环境
  - 保护模式(Protected Mode)：支持虚拟内存、分页等(Windows, Linux, ...)
  - 系统管理模式(System Management Mode)：用于执行嵌入在固件中的特殊代码
- 是一种CISC(Complex Instruction Set Computer)体系结构

x64：又称x86-64，是x86体系结构的扩展，是与x86(IA-32)兼容的64位CPU体系结构

字节序

字节序：多字节数据在内存中存储或在网络上传输时各字节的存储/传输顺序

- 小端序（little endian）：低位字节存储在内存中低位地址，效率较高(Intel CPU使用)
- 大端序（big endian）：低位字节存储在内存中高位地址，符合思维逻辑。RISC架构处理器(如MIPS, PowerPC)采用

字节序

例子

```
BYTE b=0x12;
WORD w=0x1234;
DWORD dw=0x12345678;
char str[]="abcde";
```

TYPE	NAME	SIZE	大端序	小端序
			地址低位 <-->地址高位	地址低位 <-->地址高位
BYTE	b	1	[12]	[12]
WORD	w	2	[12] [34]	[34] [12]
DWORD	dw	4	[12] [34] [56] [78]	[78] [56] [34] [12]
char []	str	6	[61] [62] [63] [64] [65] [00]	[61] [62] [63] [64] [65] [00]

小端序示例

```
#include "windows.h"

BYTE b=0x12;
WORD w=0x1234;
DWORD dw=0x12345678;
char str[]="abcde";

int main(int argc, char* argv[]){
    BYTE lb=b;
    WORD lw=w;
    DWORD ldw=dw;
    char *lstr=str;
    return 0;
}
```

小端序示例

- OllyDbg>LittleEndian.exe

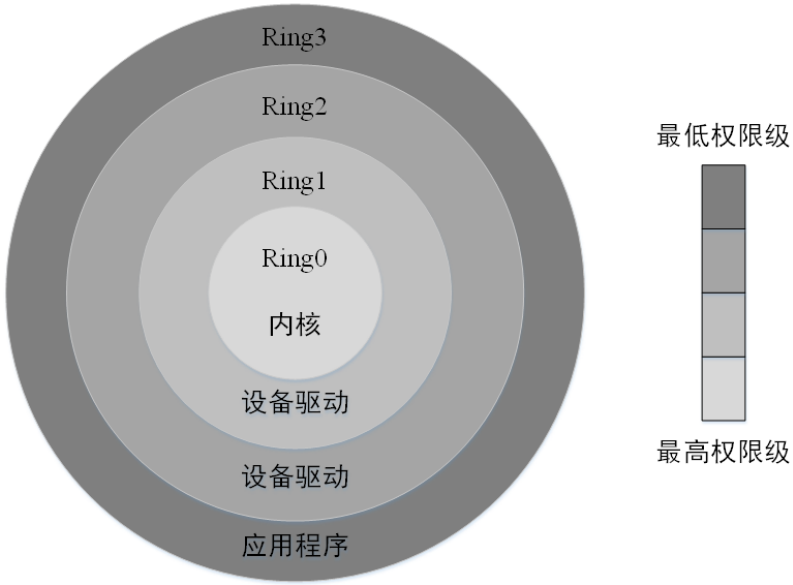
```
01381000  $ 55              PUSH EBP
01381001  . 8BEC            MOV EBP,ESP
01381003  . 83EC 10         SUB ESP,10
01381006  . A0 18303801     MOV AL,BYTE PTR DS:[b]
0138100B  . 8845 F3         MOV BYTE PTR SS:[EBP-D],AL
0138100E  . 66:8B0D 1C3038  MOV CX,WORD PTR DS:[w]
01381015  . 66:894D F4      MOV WORD PTR SS:[EBP-C],CX
01381019  . 8B15 20303801   MOV EDX,DWORD PTR DS:[dw]
0138101F  . 8955 F8         MOV DWORD PTR SS:[EBP-8],EDX
01381022  . C745 FC 243038  MOV DWORD PTR SS:[EBP-4],OFFSET LittleEn  ASCII "
01381029  . 33C0            XOR EAX,EAX
0138102B  . 8BE5            MOV ESP,EBP
0138102D  . 5D              POP EBP
0138102E  . C3              RETN
EAX=00101A12, (ASCII "EFGHIJKLMNOPQRSTUVWXYZ")

littleendian.cpp:13   return 0;
```

Address	Hex dump	ASCII
01383000	6C DF 36 63 93 20 C9 9C FF FF FF FF FF FF FF FF	l?c? 蓽
01383010	FE FF FF FF 01 00 00 00 12 00 00 00 34 12 00 00	? □...□...4□.
01383020	78 56 34 12 61 62 63 64 65 00 00 00 00 00 00 00	xV4□abcde.....

权限级别

- 在保护模式下，存在4个权限级别(Privilege Level)，编号从0~3
- 保护环是x86对权限分隔的支持机制
  - Ring0：最高权限级别，程序能够更改所有系统设置，运行OS内核
  - Ring1/Ring2：很少使用，根据需要可运行操作系统服务或驱动程序
  - Ring3：最低权限级别，程序仅可读写系统设置的一个子集，运行用户态应用程序

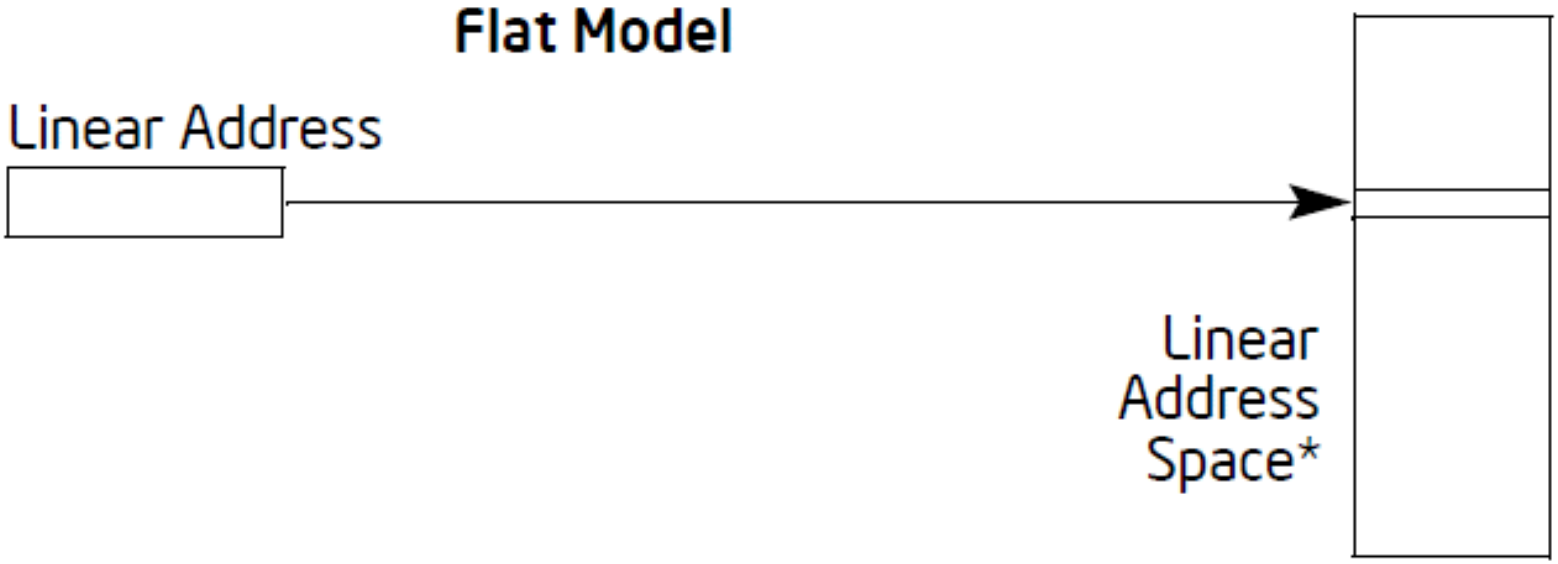


提要

IA-32内存模型

平面内存模型

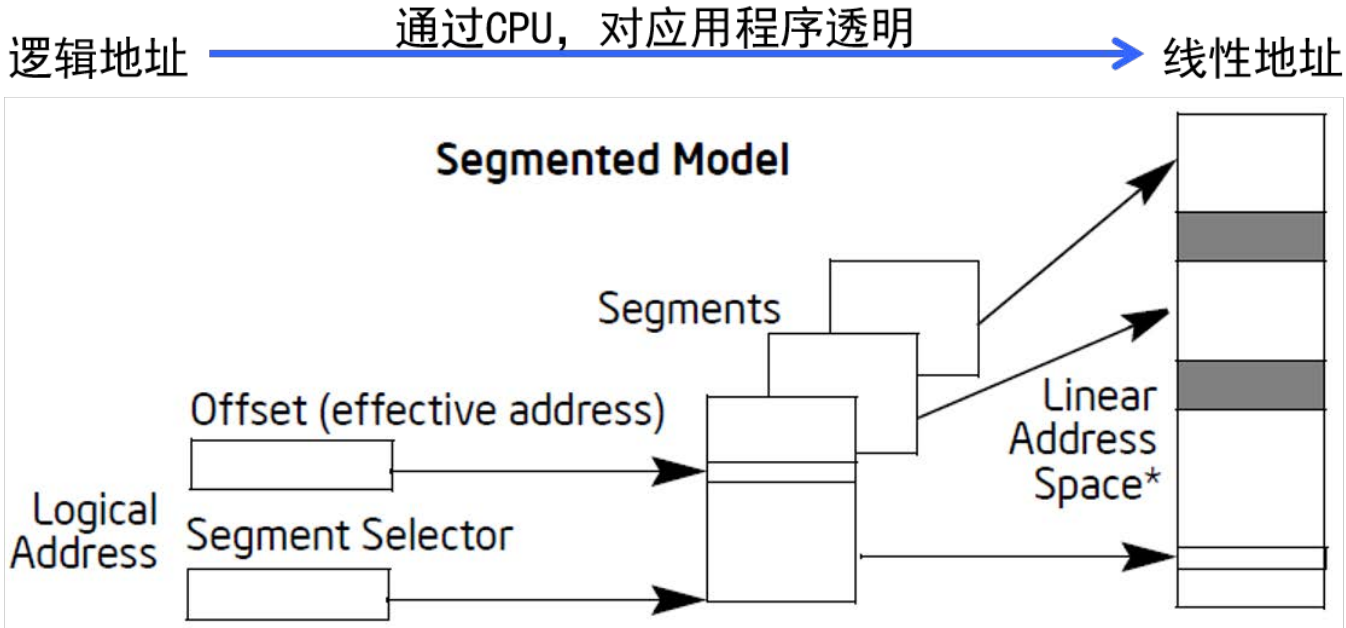
代码、数据和栈在线性地址空间中，线性地址$0 \sim 2^{31} - 1$连续



IA-32内存模型

分段内存模型

- 程序内存由一系列独立的地址空间（称为“段”）组成。代码、数据和栈在不同的段中
- 逻辑地址=段选择器+偏移量



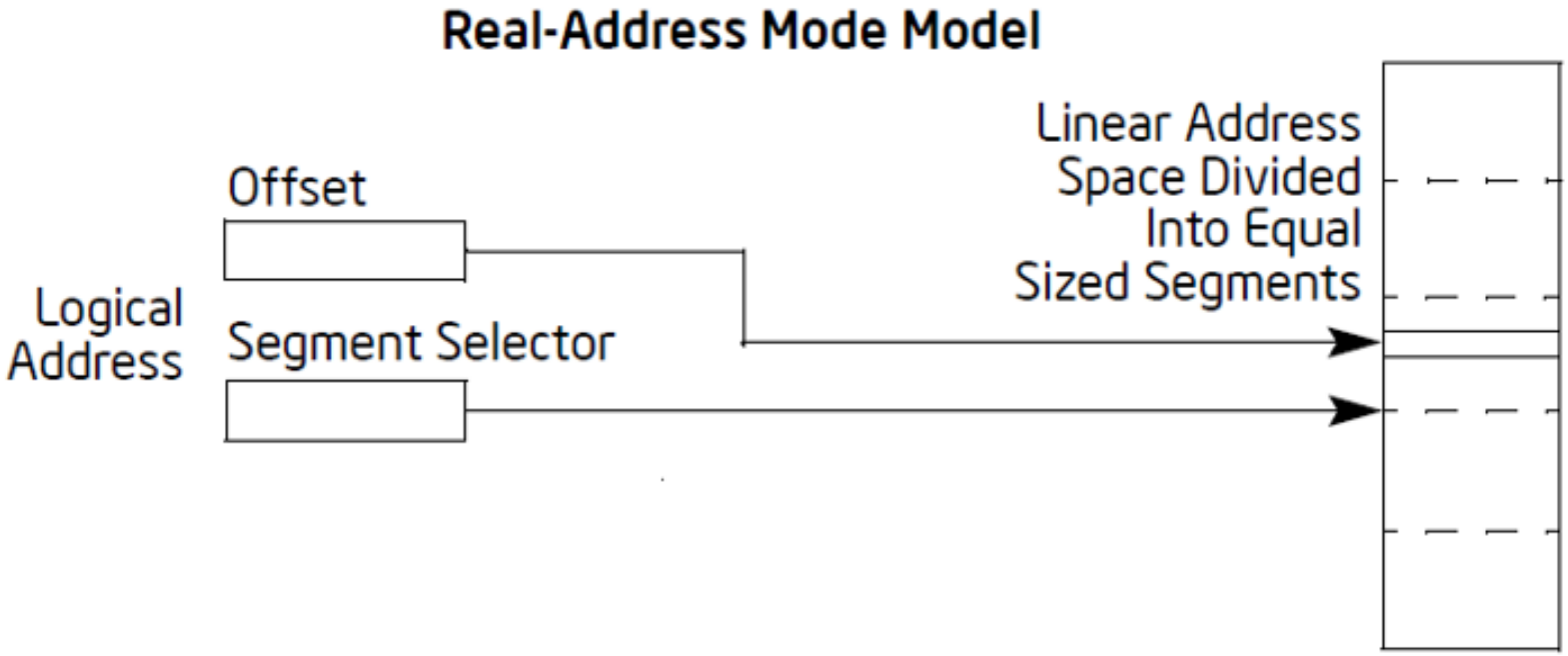
不同操作模式下的内存管理

对于分段内存模型，不同操作模式(实模式/保护模式)下，内存管理方式和寻址模式存在差异

不同操作模式下的内存管理

实模式下的内存管理

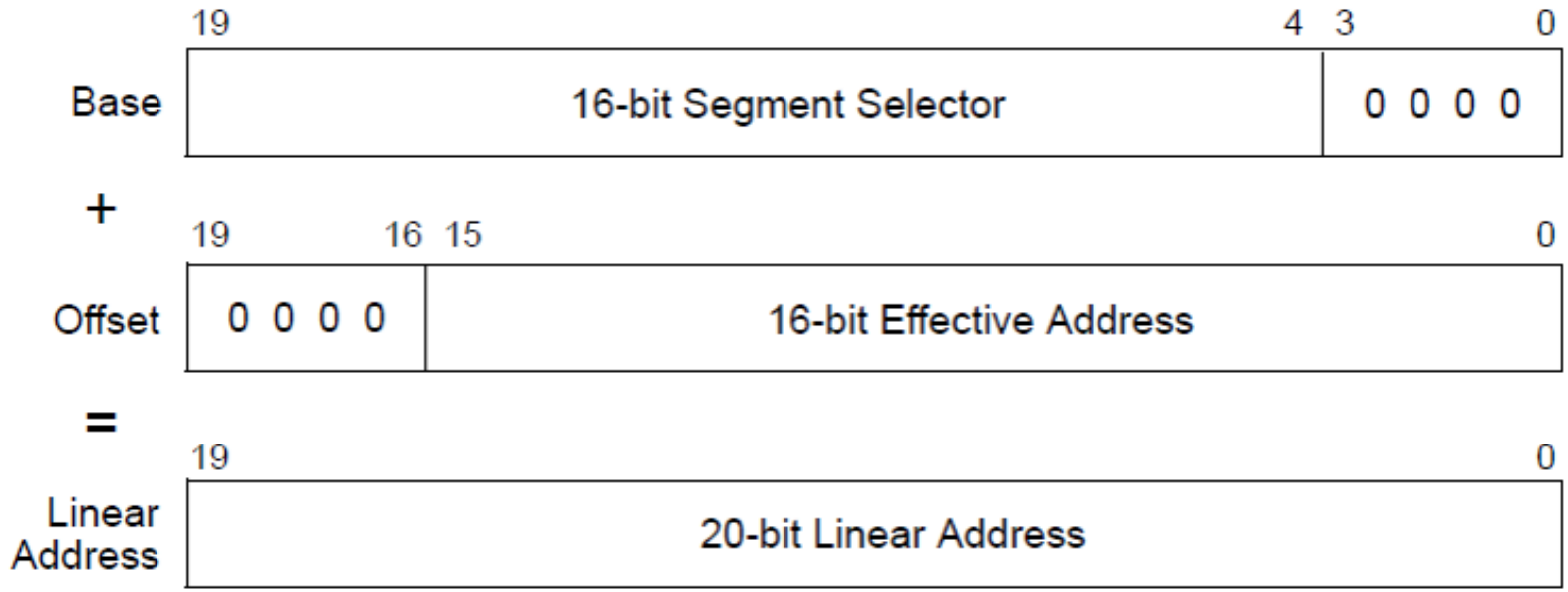
- Intel 8086 CPU使用，是特殊的分段内存模型
- 线性地址空间由一系列64KB大的段组成
- 线性地址范围$0 \sim 2^{20} - 1$



不同操作模式下的内存管理

实模式下的内存管理

- Intel 8086 CPU使用，是特殊的分段内存模型
- 线性地址空间由一系列64KB大的段组成
- 线性地址范围$0 \sim 2^{20} - 1$



不同操作模式下的内存管理

保护模式下的内存管理

- 分段（必须）+ 分页（可选）

不同操作模式下的内存管理

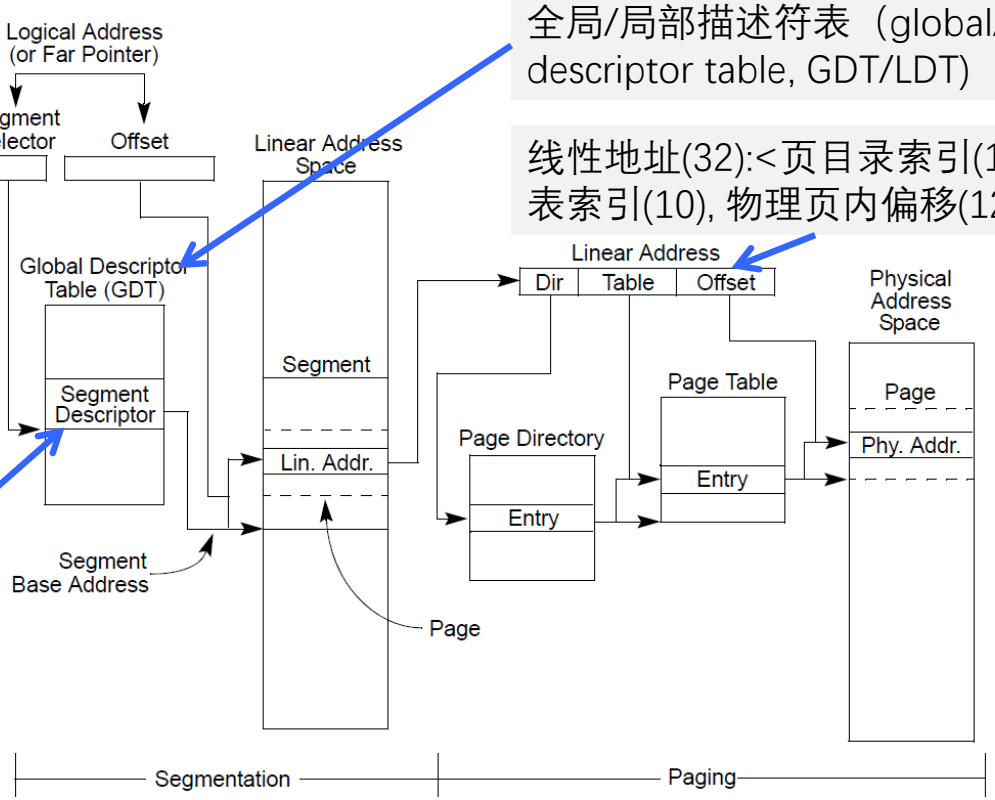
保护模式下的内存管理

段选择器(16):<GDT/LDT索引(13)，GDT/LDT flag(1)，访问权限信息(2)>

段描述符(64):<段线性基址(32), 长度上限, 访问权限, ···>

全局/局部描述符表（global/local descriptor table, GDT/LDT)

线性地址(32):<页目录索引(10), 页表索引(10), 物理页内偏移(12)>



不同操作模式下的内存管理

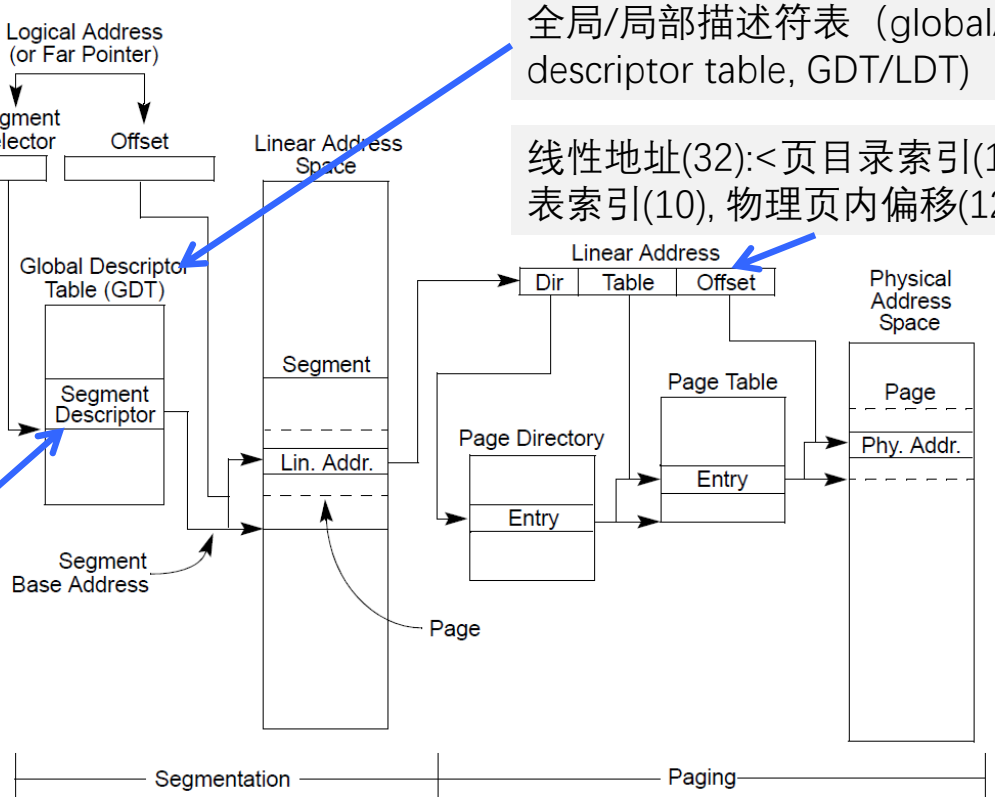
保护模式下的内存管理

段选择器(16):<GDT/LDT索引(13)，GDT/LDT flag(1)，访问权限信息(2)>

段描述符(64):<段线性基址(32), 长度上限, 访问权限, ···>

全局/局部描述符表（global/local descriptor table, GDT/LDT)

线性地址(32):<页目录索引(10), 页表索引(10), 物理页内偏移(12)>



- 程序线性地址空间≤4GB，物理地址空间≤64GB
- 每个段最大2^{32}字节，IA-32程序最多使用16383个段

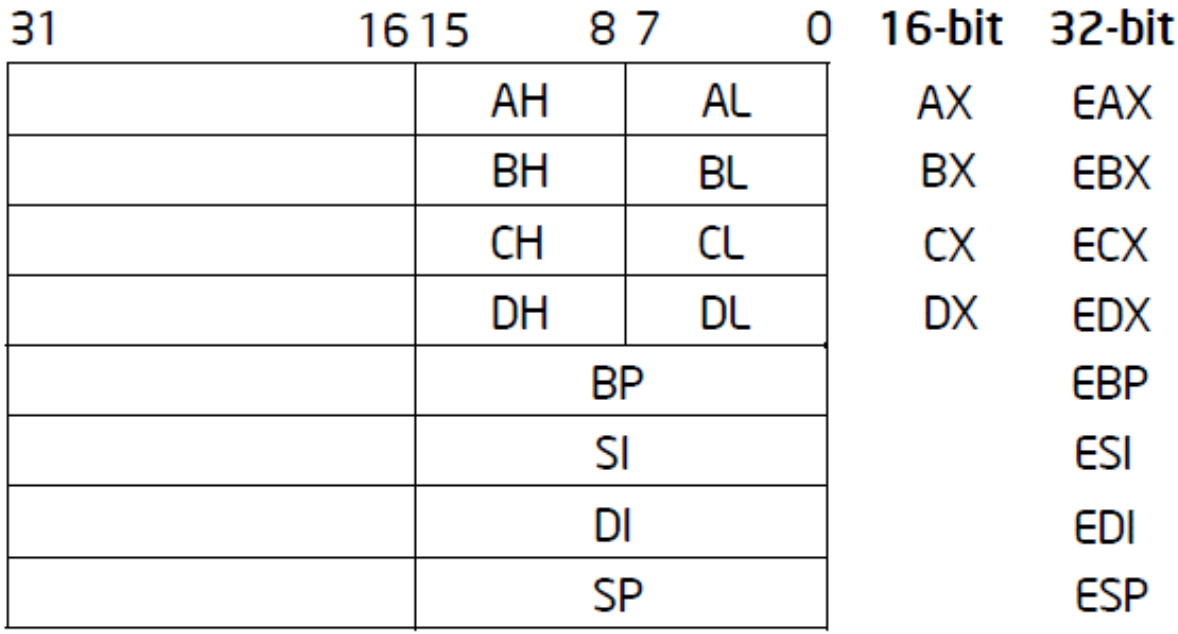
提要

寄存器分类

- 通用寄存器
- 程序状态与控制寄存器
- 指令指针寄存器
- 段寄存器
- 控制寄存器(CR0-CR4)
- 调试寄存器(DR0-DR7)
- 系统表指针寄存器(GDTR, LDTR, IDTR, task register)
- MMX(MM0-MM7, XMM0-XMM7)
- FPU(ST0-ST7, …)

通用寄存器(GPR)

- 8个32位通用寄存器：EAX，EBX，ECX，EDX，EDI，ESI，EBP，ESP
- 其中一些GPR可进一步切分为16位或8位寄存器



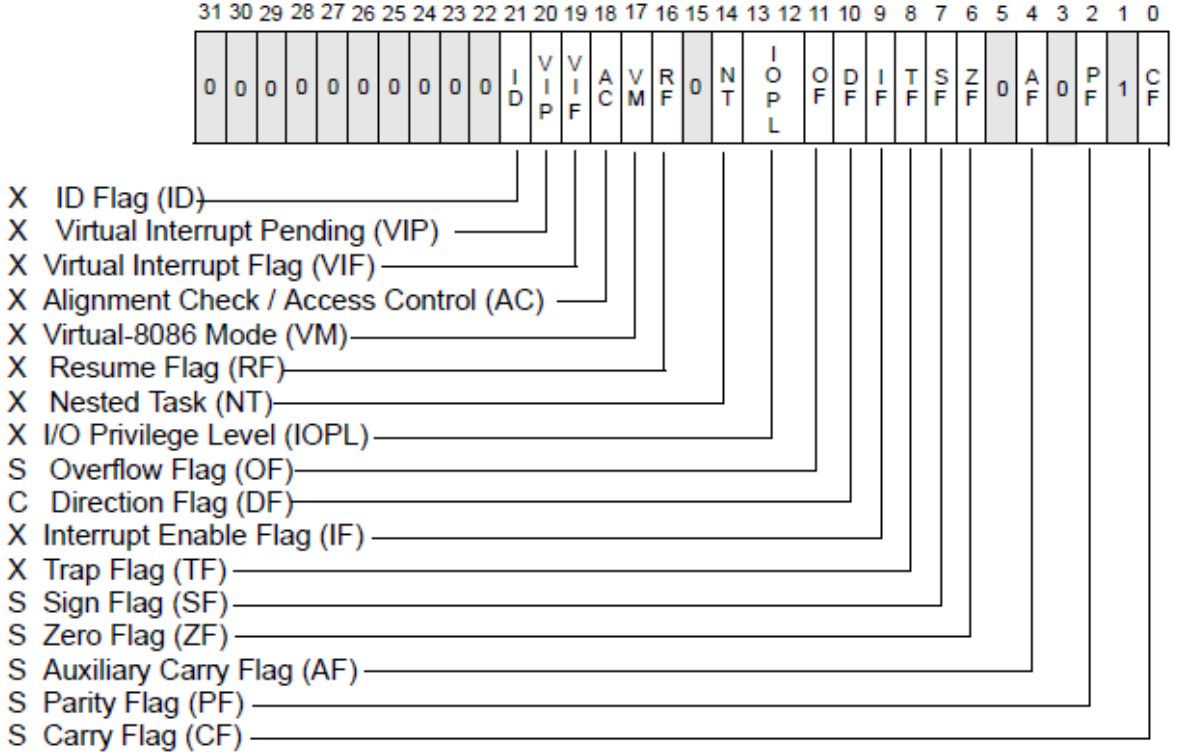
GPR的具体功能

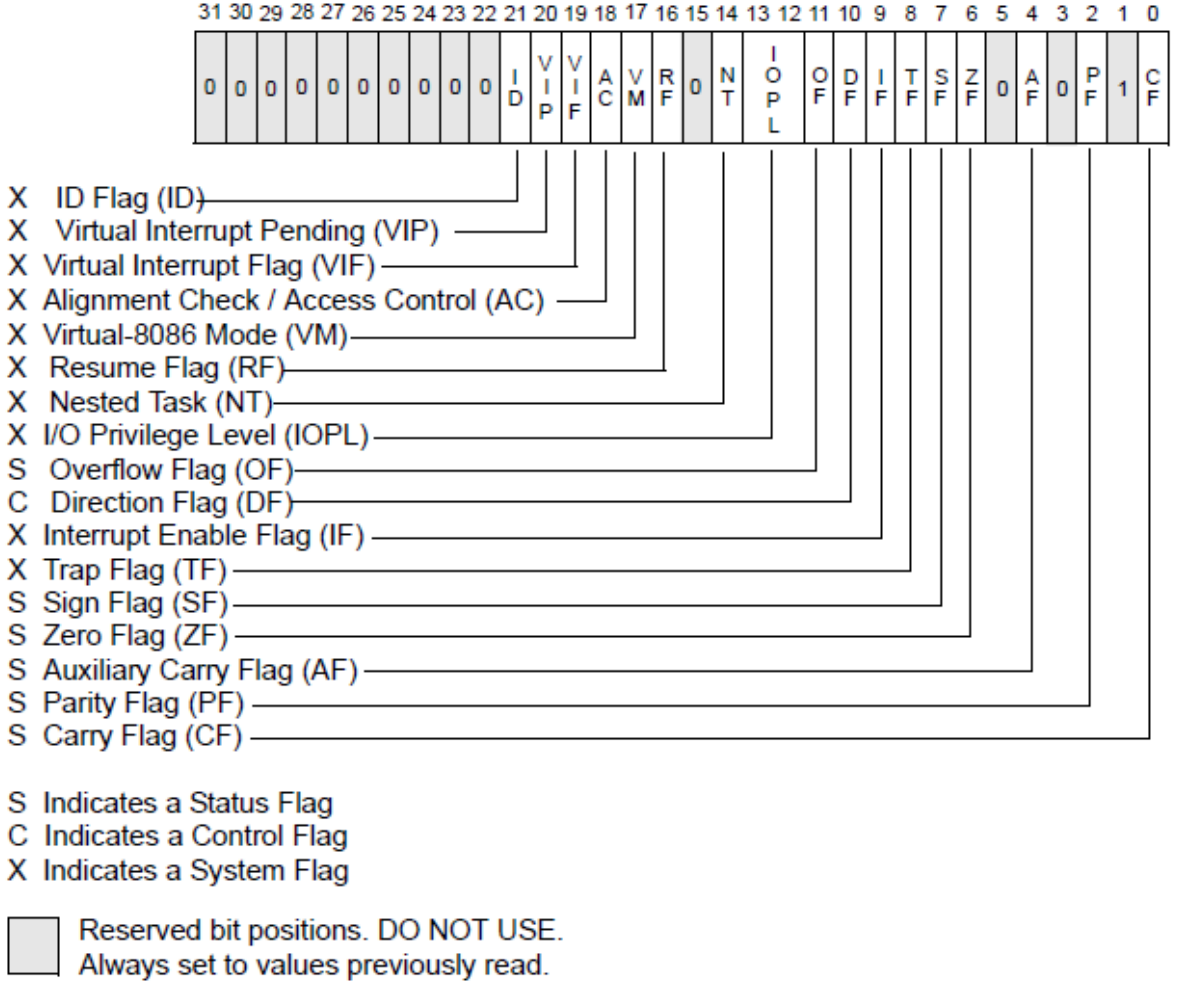
正常编译器的编译结果中

- EAX:（操作数和结果数据的）累加器
- EBX:（在DS段中数据的指针）基址寄存器
- ECX:（字符串和循环操作的）计数器
- EDX:（I/O指针）数据寄存器
- EDI: 变址寄存器，字符串/内存操作的目的地址
- ESI: 变址寄存器，字符串/内存操作的源地址
- EBP:（SS段中的）栈内数据指针，栈帧的基地址，用于为函数调用创建栈帧
- ESP:（SS段中的）栈指针，栈区域的栈顶地址

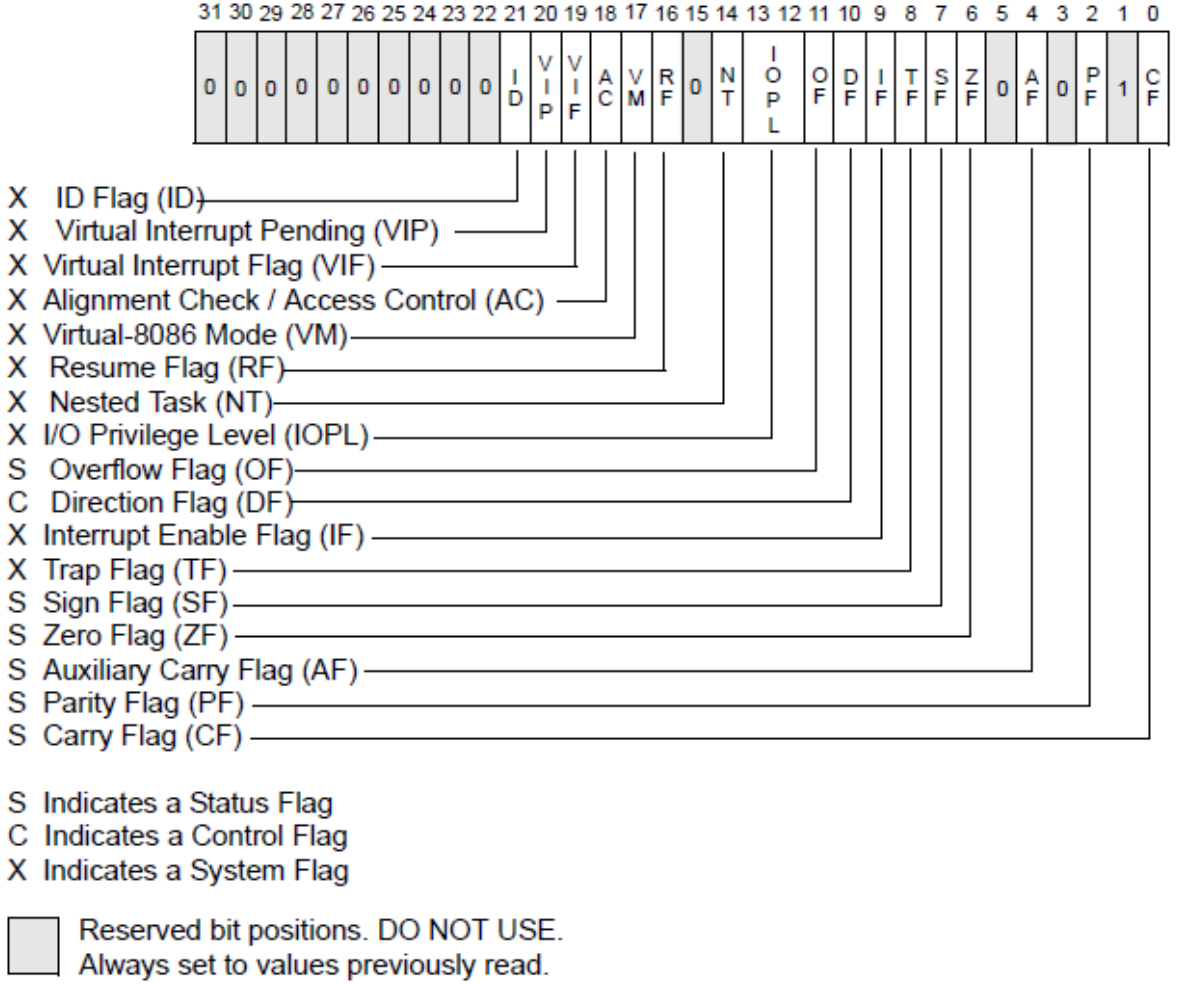
程序状态与控制寄存器(EFLAGS)

- 32位，存储算数操作符状态或其他执行状态
- 含一组状态标识(S)、一个控制标识(C)、一组系统标识(X)
- EFLAGS中的标识主要用于实现条件分支，程序调试与控制
- 关键标识位
  - ZF：零标识(S)
  - OF：溢出标识(S)
  - CF：进位标识(S)
  - SF：符号标识(S)
  - DF：方向标识(C)
  - TF：陷阱标识(X)
  - IF：中断允许标识(X)

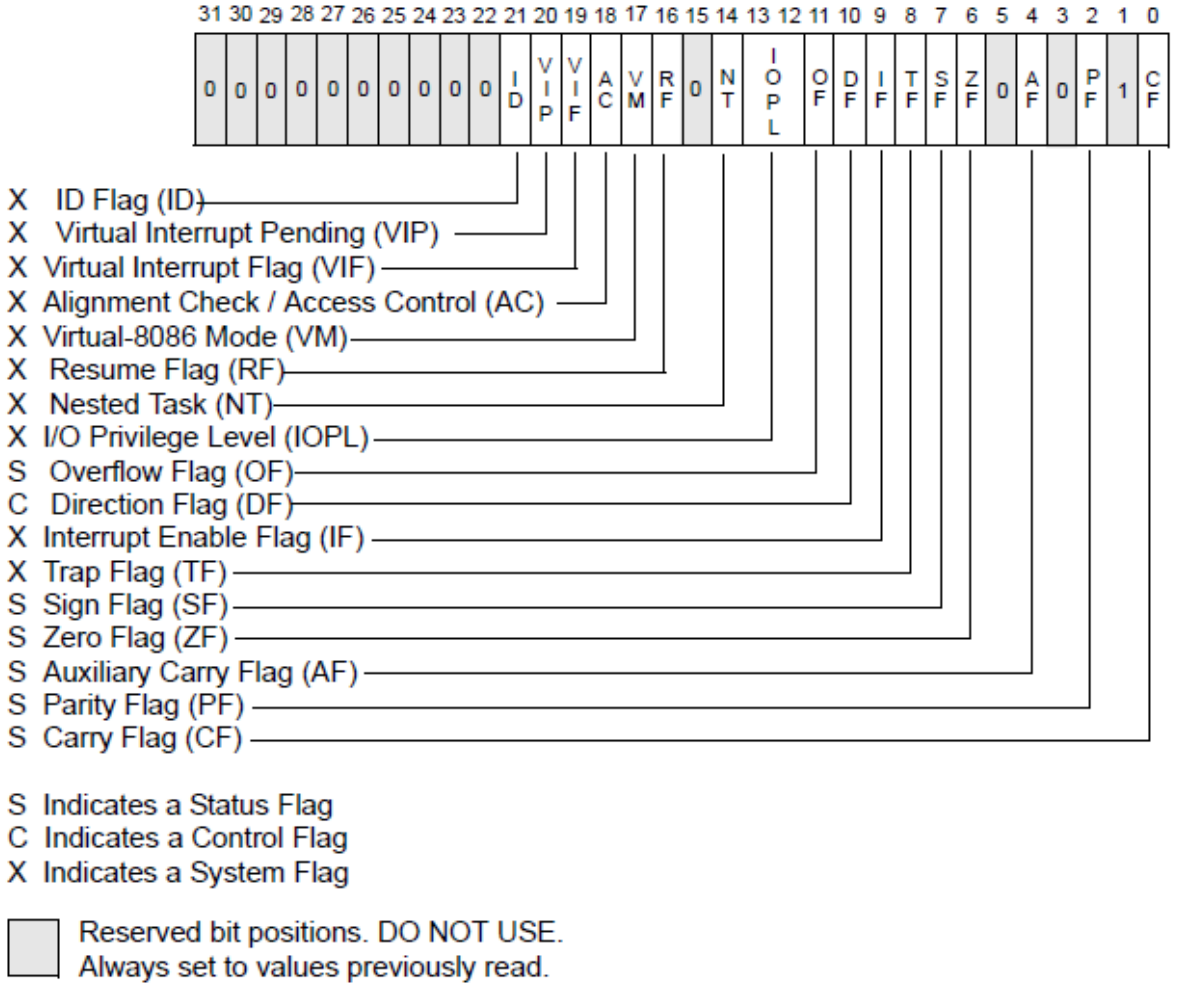

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check / Access Control (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)
S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag
Reserved bit positions. DO NOT USE.
Always set to values previously read.



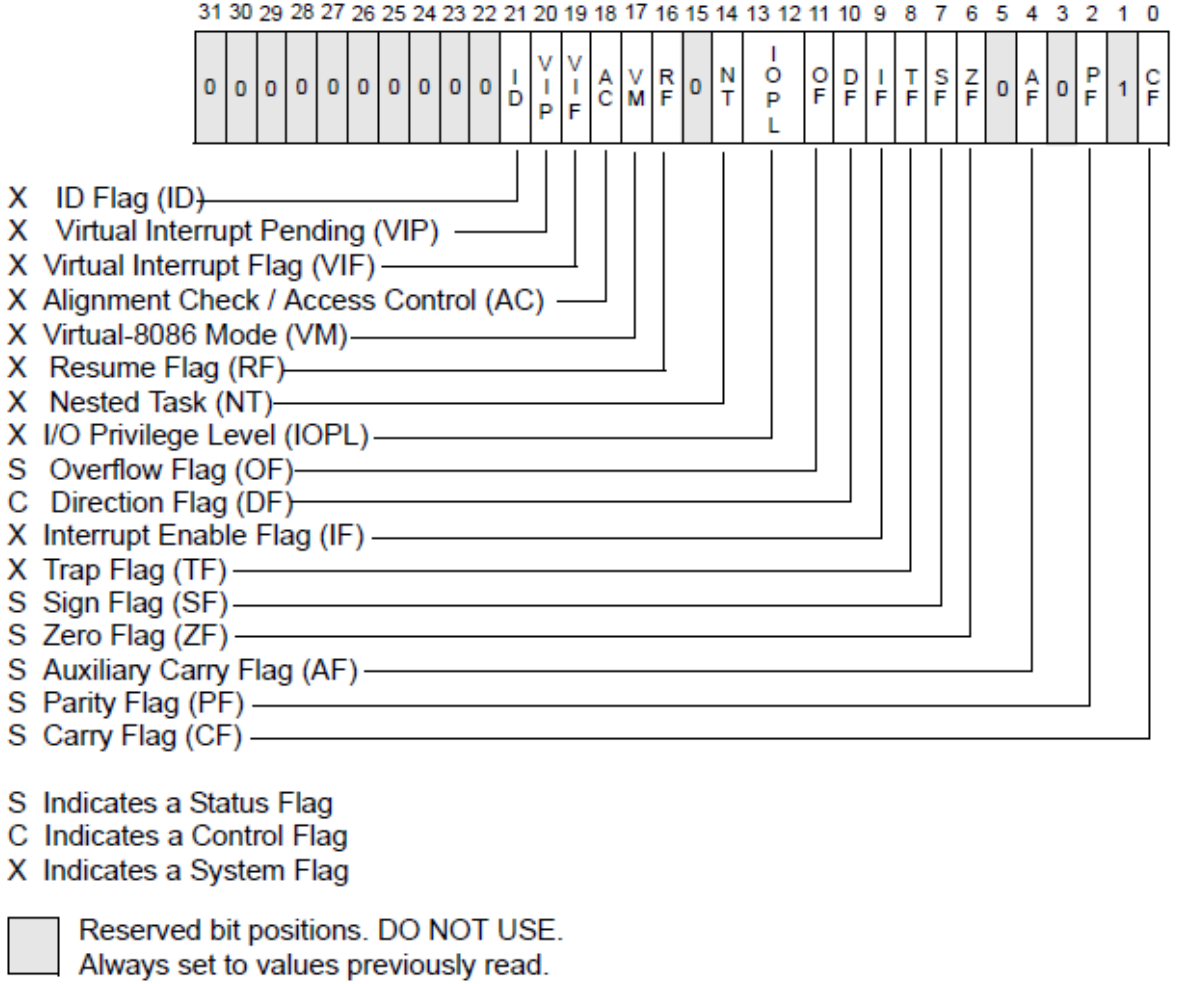
- ZF(零标识)： 若运算结果为0，则ZF值为1，否则ZF值为0



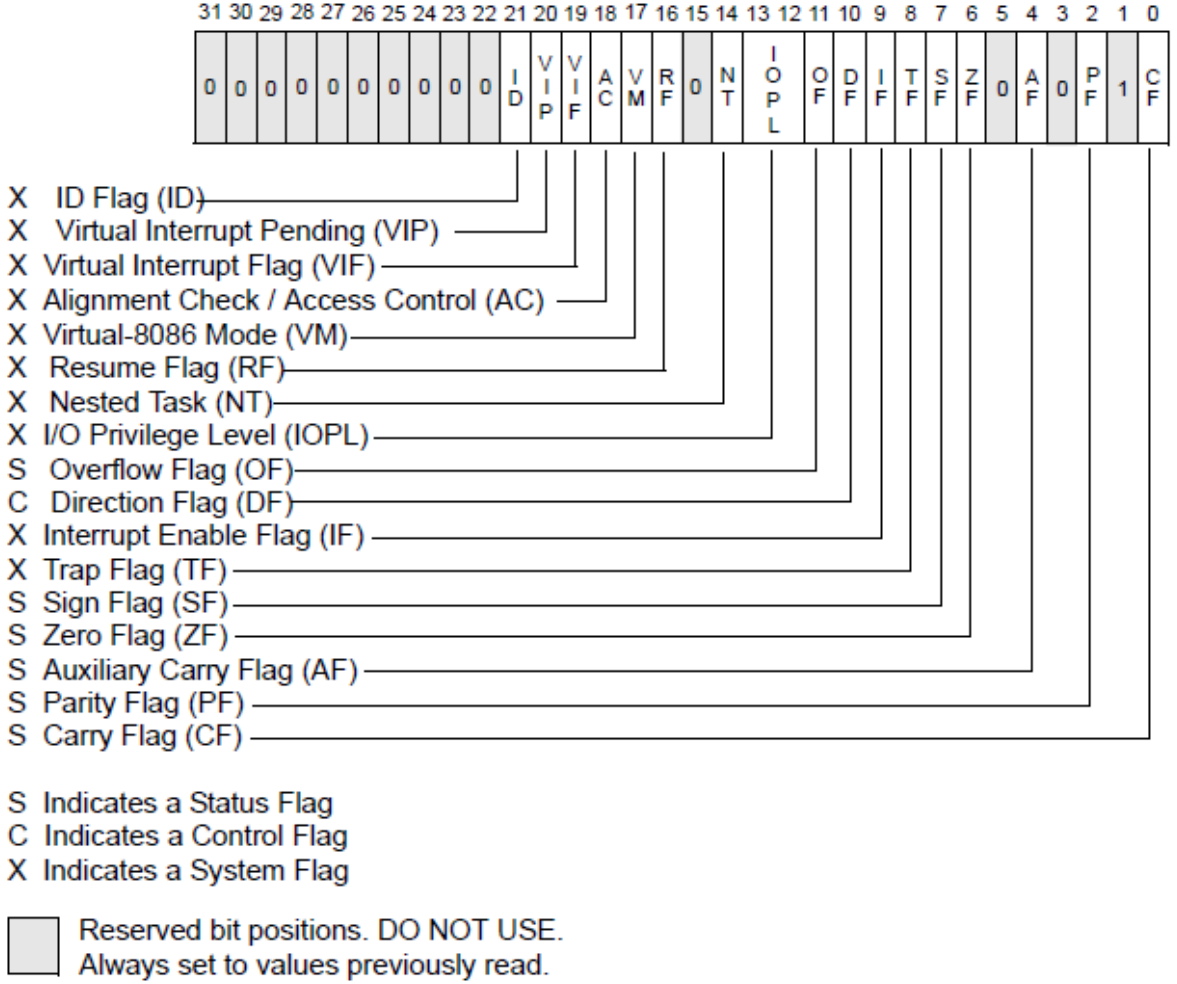
- OF(溢出标识)： 有符号整数溢出时，OF置为1；
  最高有效位(MSB)改变时，OF置为1



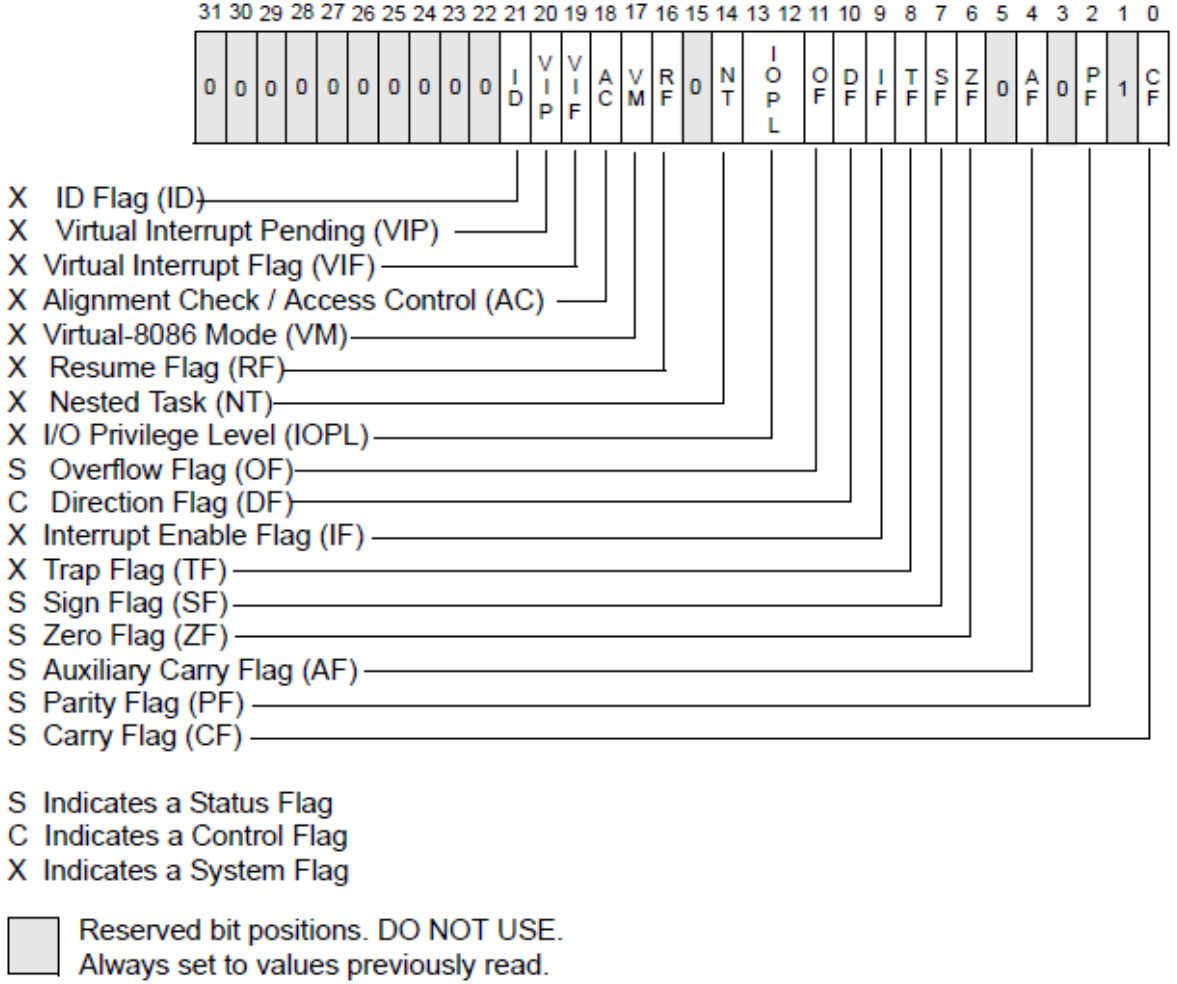
- CF(进位标识)： 无符号整数溢出时，CF置为1



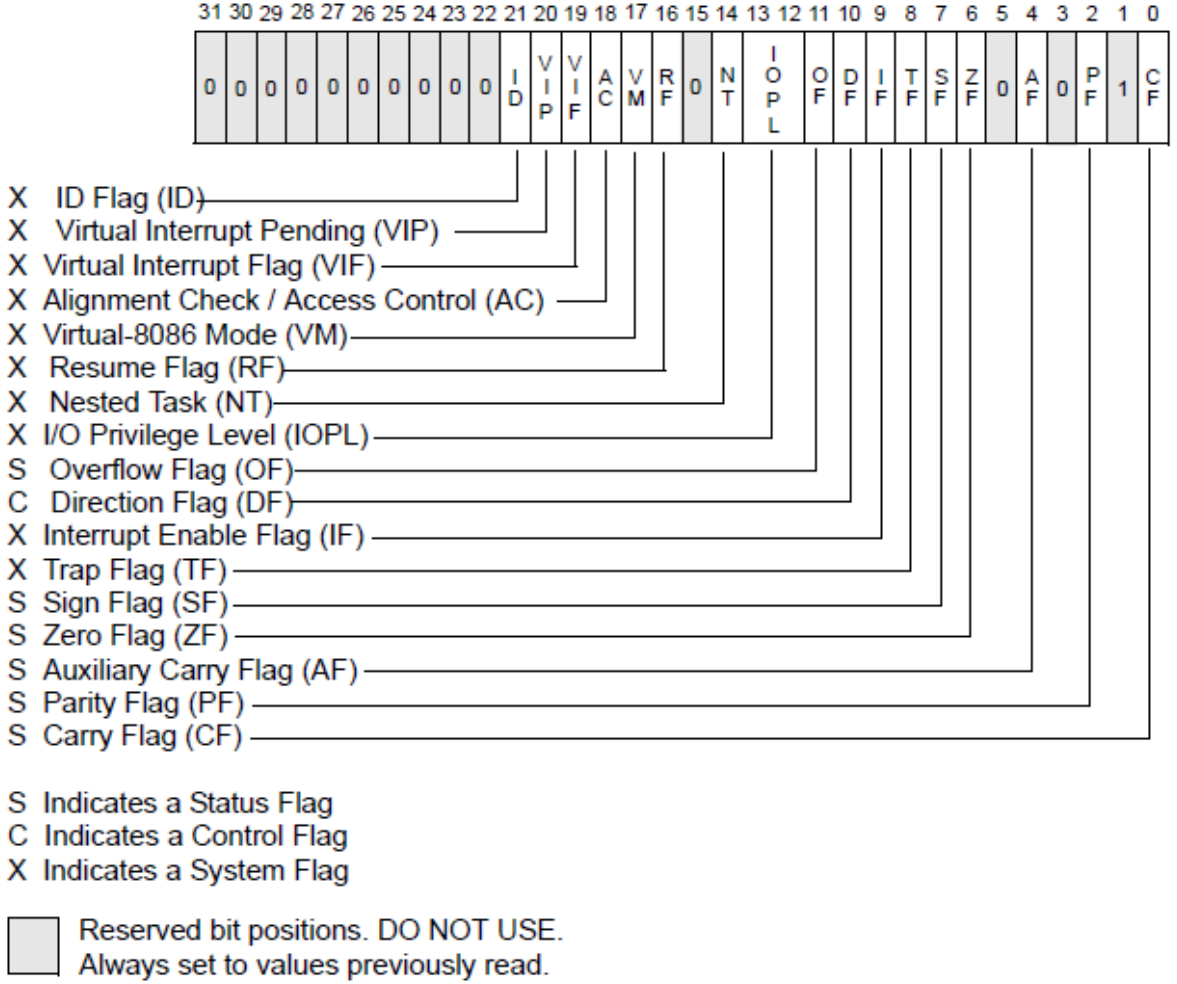
- SF(符号标识)：等于运算结果的最高位(即有符号整数的符号位)；0表示正数，1表示负数



- DF(方向标识)：控制串处理指令处理信息的方向
  - 当DF=1时，每次串操作后，变址寄存器ESI和EDI减小(即串处理沿“高地址→低地址”处理)
  - 当DF=0时，处理方向相反
  - 由STD指令置位，由CLD指令清除



- TF(陷阱标识)：若置1，则CPU执行每条指令后产生单步中断，该特性用于调试器单步执行程序；还可用于检测调试器是否运行



- IF(中断允许标识)： 若置1，则CPU在收到中断请求后，应该对中断请求进行响应处理

指令指针寄存器(EIP)

- 32位，存放指令指针，即当前代码段中将被执行的下一条指令的线性地址偏移
- 程序运行时，CPU根据CS段寄存器和EIP寄存器中的地址偏移读取下一条指令，将指令传送到指令缓冲区，并将EIP寄存器值自增，增大的大小即被读取指令的字节数
- 不能直接修改EIP，修改途径：
  - 指令JMP，Jcc，CALL，RET
  - 中断或异常

段寄存器

6个16位段寄存器

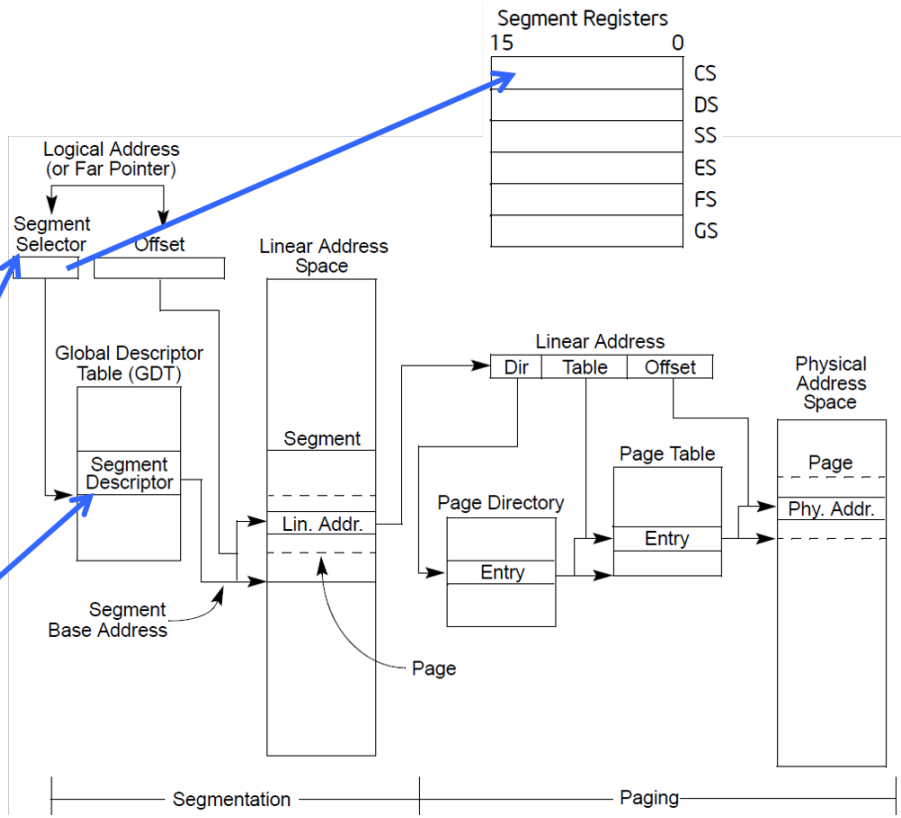
- CS：Code Segment，代码段寄存器，存放应用程序代码所在的段的段描述符索引(该段描述符中包含代码段的线性基址)
- SS：Stack Segment，栈段寄存器，
  存放栈段的段描述符索引(该段描述符中包含栈段的线性基址)
- DS：Data Segment，数据段寄存器，
  存放数据段的段描述符索引(该段描述符中包含数据段的线性基址)
- ES/FS/GS：Extra (Data) Segment，附加(数据)段寄存器，
  存放程序使用的附加数据段的段描述符索引(该段描述符中包含附加数据段的线性基址)
  - DS数据段含有程序使用的大部分数据
  - ES数据段可为某些串指令存放目的数据
  - FS数据段可用于计算结构化异常处理(SEH)、线程环境块(TEB)、进程环境块(PEB)等地址

段寄存器的使用

段寄存器加载段选择器

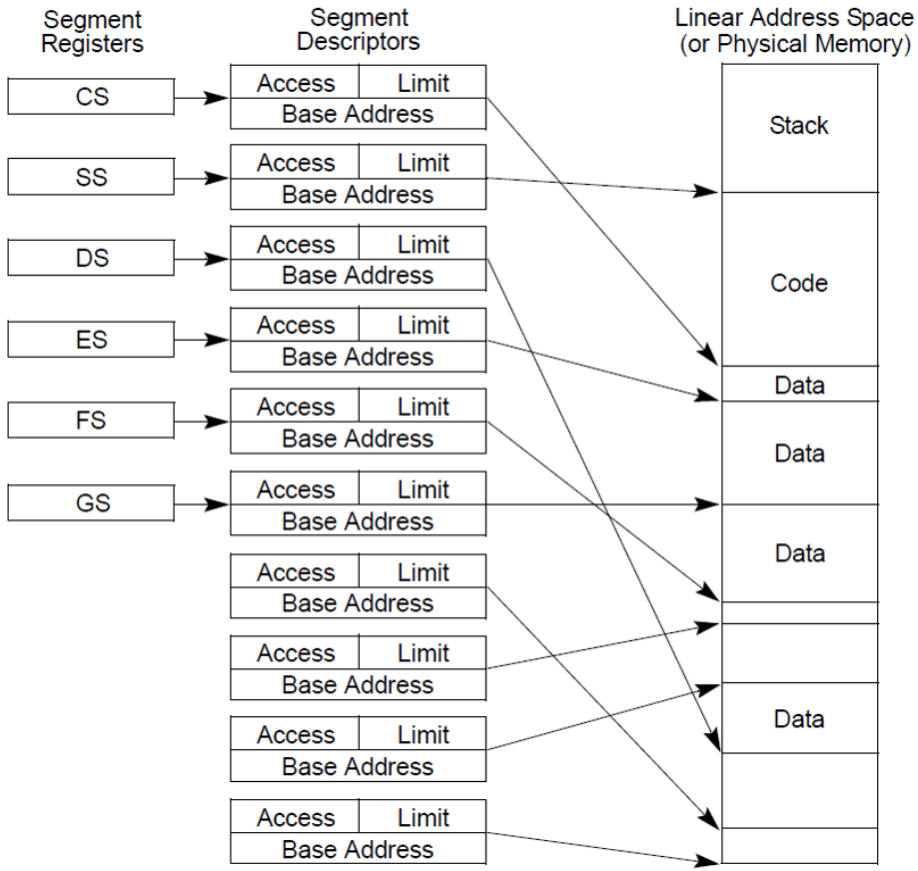
段选择器(16): <GDT/LDT索引(13)，GDT/LDT flag(1)，访问权限信息(2)>

段描述符(64)：<段线性基址(32)，长度上限，访问权限，…>



段寄存器的使用

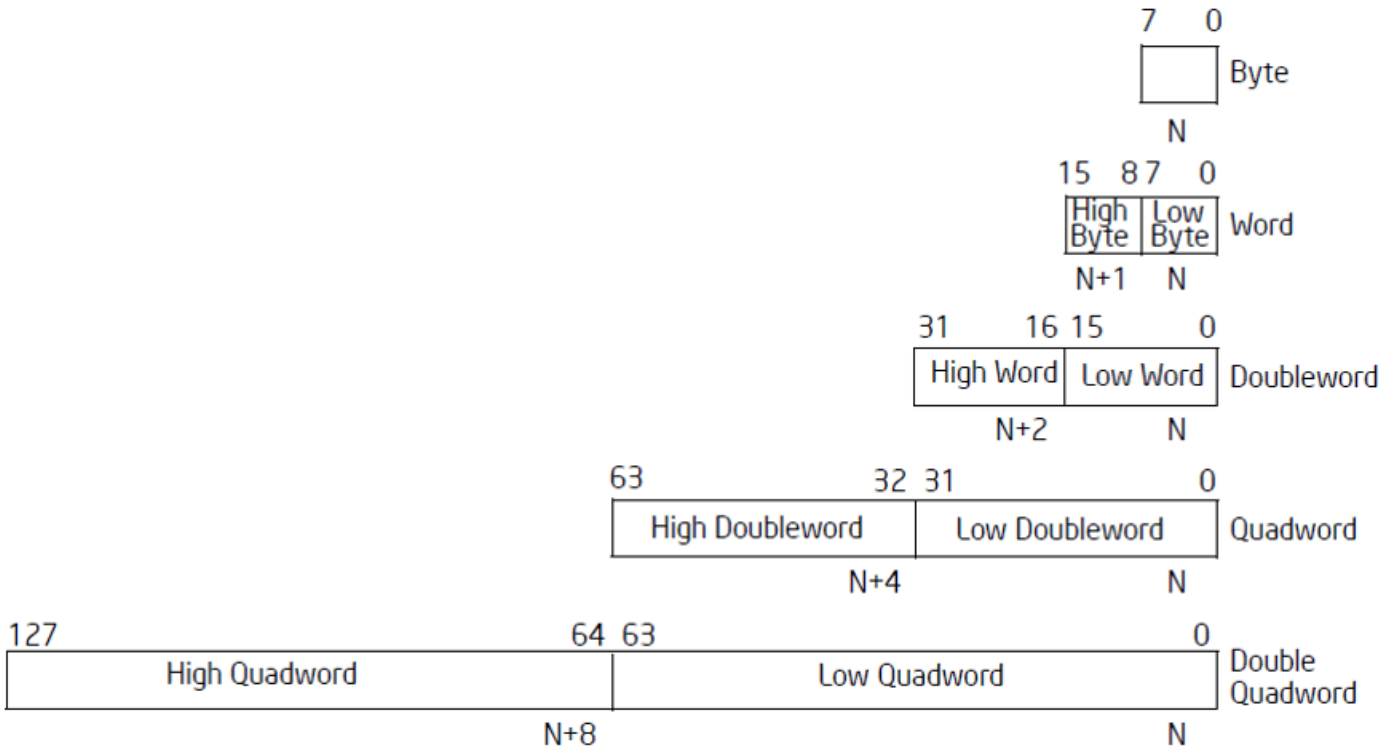
在保护模式的分段内存模型下寻段



- `mov ss:[edx], eax // Segment:[Offset]`

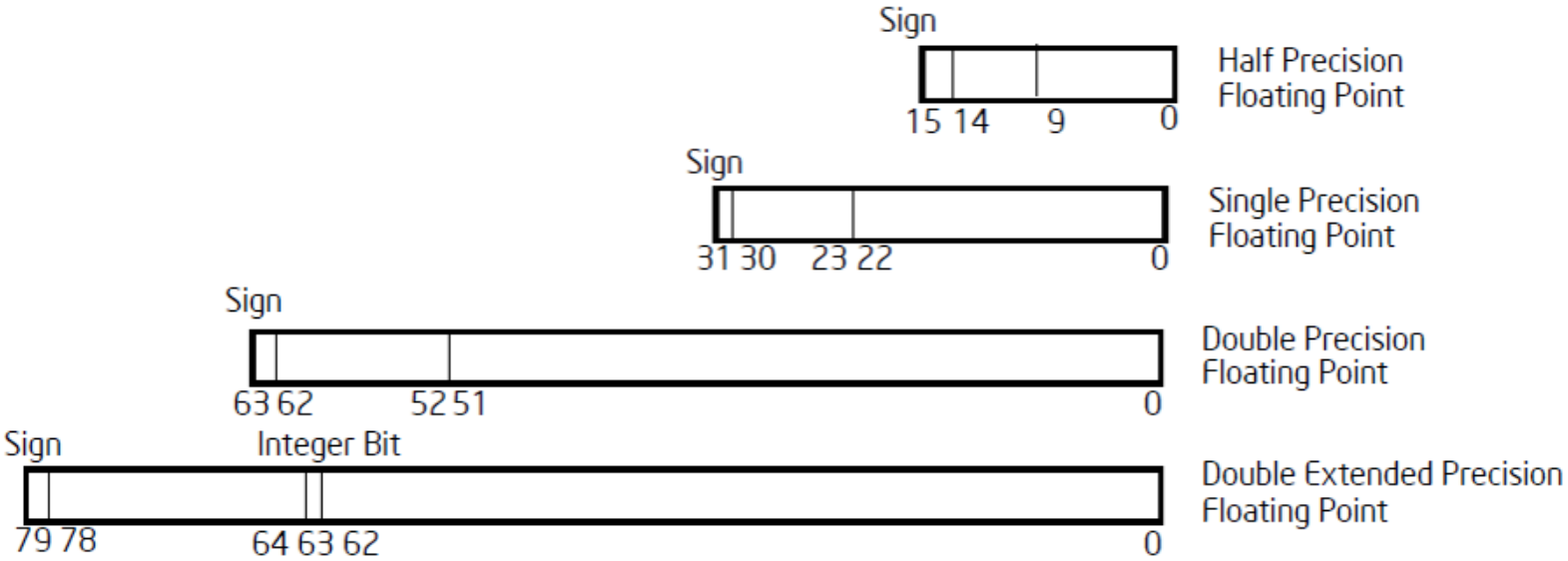
提要

基本数据类型



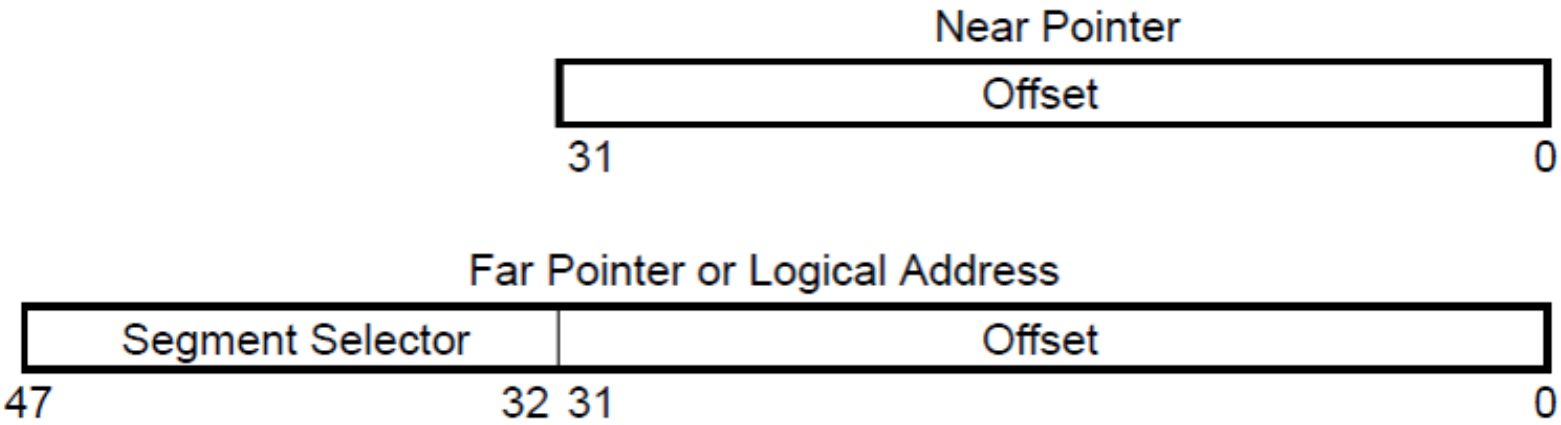
- 寄存器数据类型依照其长度对应于基本数据类型
- x86不存在64位GPR，在某些场景下将EDX:EAX合看作64位，RDTSC指令能将64位值写入EDX:EAX
- 数值数据类型分为有符号/无符号

数据类型——浮点型



Data Type	Length	Precision (Bits)	Approximate Normalized Range	
			Binary	Decimal
Half Precision	16	11	2^{-14} to 2^{15}	3.1×10^{-5} to 6.50×10^{4}
Single Precision	32	24	2^{-126} to 2^{127}	1.18×10^{-38} to 3.40×10^{38}
Double Precision	64	53	2^{-1022} to 2^{1023}	2.23×10^{-308} to 1.79×10^{308}
Double Extended Precision	80	64	2^{-16382} to 2^{16383}	3.37×10^{-4932} to 1.18×10^{4932}

数据类型——指针类型


Near Pointer
Offset
31
0
Far Pointer or Logical Address
Segment Selector
Offset
47
32
31
0

提要

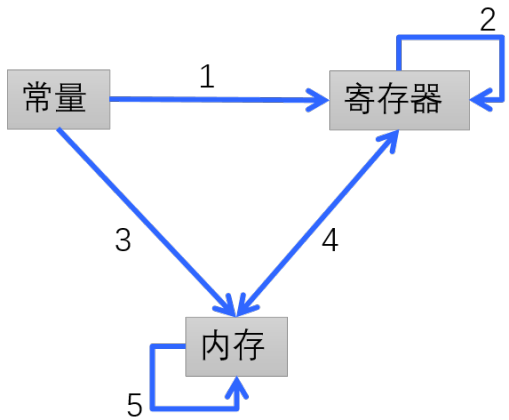
汇编指令格式

- AT&T: source在destination前，
  在较早期的GNU工具中普遍使用（如gcc, gdb等）
- Intel: destination在source前，“[...]”含义类似于解引用，
  MASM, NASM等工具中使用
- 例：

汇编指令格式	示例
AT&T	mov $4, %eax
	mov $4, %(eax)
Intel	mov eax, 4
	mov [eax], 4

数据移动指令

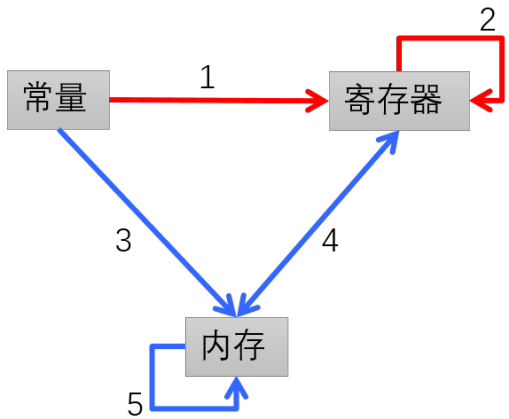
五种数据移动方式



- 前4种被大多数体系结构支持，“内存→内存”为x86特有
- 例：对内存中的值自增

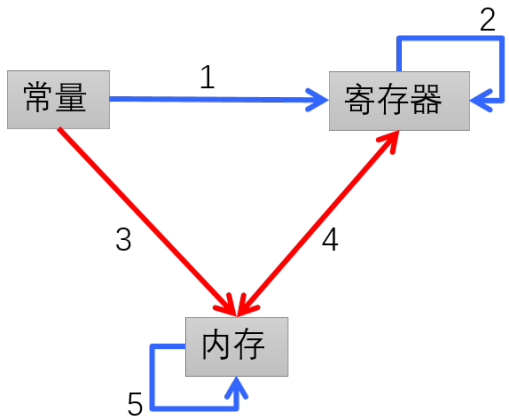
```
//RISC
LDR R4,[R3]
ADDS R2,R4,#1
STR R2,[R3]
//x86
INC DWORD PTR [EAX]
```

数据移动指令MOV



汇编指令	含义
MOV ESI, 0x12345678	将常量0x12345678移入寄存器ESI
MOV EAX, ECX	将寄存器ECX的内容移入寄存器EAX

数据移动指令MOV



汇编指令	含义
MOV EAX, [ECX]	将寄存器EAX设置为地址ECX的内存值
MOV [EAX], EBX	将地址EAX所指的内存值设为寄存器EBX的值
MOV ESI, [0x12345678]	将地址0x12345678的内存值存入寄存器ESI
MOV DWORD PTR [EAX], 1	将地址EAX所指的一块长度为DWORD的内存的值设为1
MOV [ESI+34H], EAX	将地址ESI+34H所指的内存值设为寄存器EAX的值
MOV EAX, [ESI+34H]	将寄存器EAX的值设为地址ESI+34H的内存值
MOV EDX, [ECX+EAX]	将寄存器EDX的值设为地址ECX+EAX的内存值

- 内存可以通过一个基址寄存器和一个偏移量来访问，偏移量可以是寄存器或常量

数据移动指令MOV

已知以下复杂数据结构的基址存在ECX中，分析指令序列含义

```
kd> dt nt!_KDPC
    +0x000 Type :Uchar
    +0x001 Importance :Uchar
    +0x002 Number :Uint2B
    +0x004 DpcListEntry :_LIST_ENTRY
    +0x00c DeferredRoutine :Ptr32 Void
    +0x010 DeferredContext :Ptr32 Void
    +0x014 SystemArgument1 :Ptr32 Void
    +0x018 SystemArgument2 :Ptr32 Void
    +0x01c DpcData :Ptr32 Void
```

汇编指令	伪C代码	含义
MOV EAX, [EBP+0CH]	EAX=*[EBP+0CH];	读取一个内存值到EAX
AND DWORD PTR [ECX+1CH], 0	p->DpcData=NULL;	
MOV [ECX+0CH], EAX	p->DeferredRoutine=EAX;	
MOV EAX, [EBP+10H]	EAX=*[EBP+10H];	读取另一个内存值到EAX
MOV DWORD PTR [ECX], 113H		向数据结构基址写入113H
MOV [ECX+10H], EAX	p->DeferredContext=EAX;	

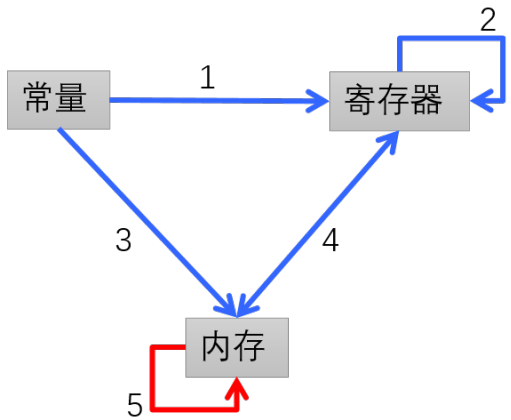
数据移动指令MOV

- 访问数组类型的对象，通过“[基址+索引*数值范围]”
- 在以下例子中，EDI保存结构体基址，EBX保存索引i的值

汇编指令	含义	伪C代码

LOOP_START:
LEA EAX, [EDI+4]
MOV EAX, [EAX+EBX*4]
TEST EAX, EAX
…
JZ SHORT LOC_7F627F
…
LOC_7F627F:
INC EBX
CMP EBX, [EDI]
JL SHORT LOOP_START
	获取bar->array基址
	
	索引i自增
	比较i与bar->size大小
		typedef struct _FOO{
		DWORD size;
		DWORD array[…];
		} FOO, *PFOO;
		PFOO bar= …;
		for (i=…; i< bar->size; i++){
		if(bar->array[i] != 0){
		…
		}
		}

MOVSB/MOVSW/MOVSD



- 在两个内存地址之间移动1字节/2字节/4字节数据
- 使用EDI保存目标地址；使用ESI保存源地址
- 源/目标地址会自动更新，根据EFLAGS中的DF标识决定是自增(DF=0)还是自减(DF=1)
- 用于实现字符串或内存的复制
- 有时可配合REP指令(将指令操作重复ECX次)使用

MOVS(MOVSB/MOVSW/MOVSD)

示例1

汇编指令	伪C代码
MOV ESI, OFFSET _RamdiskBootDiskGuid LEA EDI, [EBP-0C0H] MOVSD MOVSD MOVSD MOVSD	GUID RamdiskBootDiskGuid=…; … GUID foo; Memcpy(&foo, &RamdiskBootDiskGuid, sizeof(GUID));

从ESI到EDI复制4 byte数据，
每次ESI和EDI的地址都加4

ESI为指向_RamdiskBootDiskGuid的指针
EDI是一个栈内地址，EBP：栈帧基址
LEA：load effective address，不去读地址上的内容

示例2

汇编指令	伪C代码
PUSH 8 POP ECX MOV ESI, OFFSET _KeServiceDescriptorTable MOV EDI, OFFSET _KeServiceDescriptorTableShadow REP MOVSD	memcpy(&KeServiceDescriptorTableShadow, &KeServiceDescriptorTable, 32);

数据移动指令SCAS, STOS

SCAS：用AL/AX/EAX减去[EDI]，更新EFLAGS，并对EDI自增/自减

汇编指令	含义
XOR AL, AL	设置AL为0
MOV EBX, EDI	将字符串的起始地址保存在EBX中
REPNE SCASB	比较EDI所指的字节与AL，若不相等则对EDI自增/自减1
SUB EDI, EBX	相减得到字符串中0相对起始地址的偏移量（实现了C语言的strlen()函数）

STOS：将AL/AX/EAX的值写入EDI指向的内存

汇编指令	含义
XOR EAX, EAX	设置EAX为0
PUSH 9	
POP ECX	使ECX为9
MOV EDI, ESI	设置要写入的目标地址
REP STOSD	执行STOSD 9次，将36字节0写入EDI指向的内存目标地址（实现了C语言的memset(edi,0,36)功能）

算数运算指令

汇编指令	伪C代码或含义
ADD ESP, 14H	ESP=ESP+0x14 (有符号运算, 无符号用ADC)
SUB ECX, EAX	ECX=ECX-EAX (有符号运算, 无符号用SBB)
INC ECX	ECX=ECX+1 (无符号操作数)
DEC EDI	EDI=EDI-1 (无符号操作数)
NEG EAX	0-EAX (有符号操作数)
OR EAX, 0FFFFFFFFH	EAX=EAX|0xFFFFFFFF
AND ECX, 7	ECX=ECX&7
XOR EAX, EAX	EAX=EAX^EAX
NOT EDI	EDI=~EDI
SHL CL, 4	CL=CL<<4
SHR ECX, 1	ECX=ECX>>1
ROL AL, 3	将AL循环左移3位
ROR AL, 1	将AL循环右移1位
IMUL reg	EDX:EAX=EAX*reg (有符号运算, 无符号用MUL)
IMUL mem	EDX:EAX=EAX*mem
IMUL reg1, reg2	reg1=reg1*reg2
IMUL reg1, mem	reg1=reg1*mem
IMUL reg1, reg2, imm	reg1=reg2*imm
IMUL reg1, mem, imm	reg1=mem*imm
DIV/IDIV reg/mem	根据除数, 选择AX,DX:AX或EDX:EAX作为被除数, 并将商保存在AL, AX或EAX中, 将余数保存在AH, DX或EDX中

控制流指令

CMP：算数比较

- 比较两个操作数(通过相减)，并设置EFLAGS中的适当标识位
- 常与条件跳转Jcc指令配合使用，跳转依据即CMP运算结果

TEST：逻辑比较

比较两个操作数(通过逻辑AND运算)，并设置EFLAGS中的适当标识位，算法：

```
TEMP <- SRC1 AND SRC2;
SF <- MSB(TEMP);
ZF <- (TEMP == 0? 1:0);
PF <- BitwiseXNOR(TEMP[0:7]);
CF <- 0;
OF <- 0;
```

JMP：无条件跳转到目标指令地址(可用相对地址或绝对地址)

控制流指令

Jcc(cc=conditional code)

Instruction Mnemonic	Condition (Flag States)	Description
Unsigned Conditional Jumps		
JA/JNBE	(CF or ZF) = 0	Above/not below or equal
JAE/JNB	CF = 0	Above or equal/not below
JB/JNAE	CF = 1	Below/not above or equal
JBE/JNA	(CF or ZF) = 1	Below or equal/not above
JC	CF = 1	Carry
JE/JZ	ZF = 1	Equal/zero
JNC	CF = 0	Not carry
JNE/JNZ	ZF = 0	Not equal/not zero
JNP/JPO	PF = 0	Not parity/parity odd
JP/JPE	PF = 1	Parity/parity even
JCXZ	CX = 0	Register CX is zero
JECXZ	ECX = 0	Register ECX is zero
Signed Conditional Jumps		
JG/JNLE	((SF xor OF) or ZF) = 0	Greater/not less or equal
JGE/JNL	(SF xor OF) = 0	Greater or equal/not less
JL/JNGE	(SF xor OF) = 1	Less/not greater or equal
JLE/JNG	((SF xor OF) or ZF) = 1	Less or equal/not greater
JNO	OF = 0	Not overflow
JNS	SF = 0	Not sign (non-negative)
JO	OF = 1	Overflow
JS	SF = 1	Sign (negative)

控制流示例(if-else)

汇编指令	伪C代码

MOV ESI, [EBP+8]
MOV EDX, [ESI]
TEST EDX, EDX
JZ SHORT LOC_4E31F9
MOV ECX, OFFSET _FSRTLFASTMUTEXLOOKASIDELIST
CALL _EXFREETONPAGEDLOOKASIDELIST@8
AND DWORD PTR [ESI], 0
LEA EAX, [ESI+4]
PUSH EAX
CALL _FSRTLUNINITIALIZEBASEMCB@4
LOC_4E31F9:
POP ESI
POP EBP
RETN 4
_FSRTLUNINITIALIZELARGEMCB@4 ENDP
	if(*esi!=0){
	…
	
	EXFREETONPAGEDLOOKASIDELIST(…);
	*esi=0;
	…
	}
	return;

控制流示例(loop)

汇编指令	伪C代码

MOV EDI, DS:__IMP__PRINTF
XOR ESI, ESI
LEA EBX, [EBX+0]
LOC_401010:
PUSH ESI
PUSH OFFSET FORMAT
CALL EDI ; __IMP_PRINTF
INC ESI
ADD ESP, 8
CMP ESI, 0AH
JL SHORT LOC_401010
PUSH OFFSET ADONE
CALL EDI ; __IMP_PRINTF
ADD ESP, 4
	for(int i=0; i<10; i++){
	printf(“%d\n” , i);
	}
	printf(“done!\n”);
	或
	int i=0;
	loop_start:
	if(i<10){
	printf(“%d\n” , i);
	i++;
	goto loop_start;
	}
	printf(“done!\n”);

栈操作与函数调用指令

栈：在IA-32架构下，栈是连续的内存区域，存在于一个栈段内，该栈段可由段寄存器SS指向

- 任意时刻，ESP寄存器所包含的栈指针均指向栈顶位置
- 所有针对栈的指令操作，均基于SS对当前栈的引用
- 通常由高地址向低地址扩展（PUSH时ESP自减，POP时ESP自增）

栈操作与函数调用指令

栈操作指令

- PUSH：将字(或双字)压栈
  - PUSH时先ESP自减，再将指令操作数放入栈中，
    即ESP所指的栈单元内容是栈顶元素，属于栈的一部分
- POP：将字(或双字)弹出栈
- PUSHA/POPA/PUSHAD/POPAD
- ENTER
  - 进入栈帧，等价于PUSH EBP；MOV EBP，ESP
- LEAVE
  - 退出栈帧，等价于MOV ESP，EBP；POP EBP

栈操作与函数调用指令

栈帧：是将调用函数和被调用函数联系起来的机制，栈被分割为栈帧，栈帧组成栈。栈帧的内容包含

- 函数的局部变量
- 向被调用函数传递的参数
- 函数调用的联系信息（栈帧相关的指针：栈帧基址针，返回指令指针）

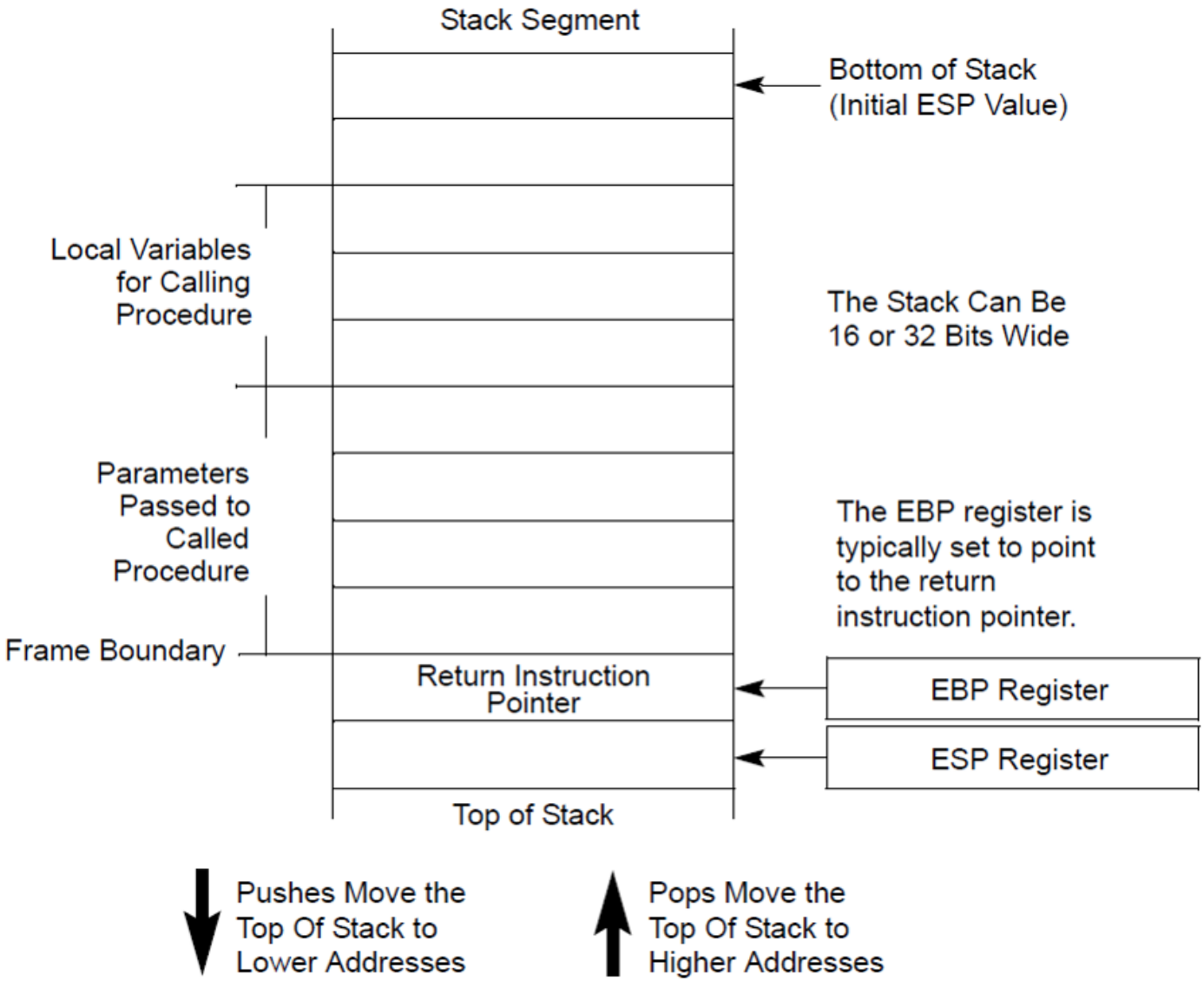
栈帧基指针(在EBP中)

被调用函数栈帧的固定参考点

返回指令指针：在执行被调用函数的第一条指令前，CALL指令将EIP寄存器中的地址压栈，这一被压入栈中的指令地址称为返回指令指针

是被调用函数返回后首先执行的调用函数指令的地址，即调用函数中CALL指令的下一条指令的地址

栈操作与函数调用指令


Stack Segment
Bottom of Stack
(Initial ESP Value)
Local Variables
for Calling
Procedure
The Stack Can Be
16 or 32 Bits Wide
Parameters
Passed to
Called
Procedure
The EBP register is
typically set to point
to the return
instruction pointer.
Frame Boundary
Return Instruction
Pointer
EBP Register
ESP Register
Top of Stack
Pushes Move the
Top Of Stack to
Lower Addresses
Pops Move the
Top Of Stack to
Higher Addresses

栈操作与函数调用指令

近调用/近返回

控制流转移到/出当前代码段中的被调用函数，通常提供对本地函数的访问

远调用/远返回

控制流转移到/出其他代码段中的被调用函数，通常提供对OS函数或其他进程函数的访问

栈操作与函数调用指令

函数的指令框架

```
PUSH EBP
MOV EBP, ESP
…
MOV ESP, EBP ;opt
POP EBP
RETN
```

近调用时的CALL指令语义

- 将EIP的当前值(返回指令指针)压栈
- 将CALL的目标指令（被调用函数的第一条指令）的地址偏移载入EIP寄存器
- CALL指令结束后开始执行被调用函数
  - PUSH EBP：将调用函数的栈帧的EBP压栈
  - ESP -> EBP，确立新栈帧（被调用函数栈帧）的基址针
  - 执行被调用函数的具体功能

栈操作与函数调用指令

函数的指令框架

```
PUSH EBP
MOV EBP, ESP
…
MOV ESP, EBP ;opt
POP EBP
RETN
```

近返回时的RET指令语义

- RETN指令执行之前
  - EBP -> ESP：清空当前被调用函数的栈帧(此时ESP指向的栈顶内容恰好为调用者函数的EBP)(可选)
  - POP EBP：将EBP恢复为调用者函数的原始EBP
- 将栈顶的内容（返回指令指针）弹出到EIP
- 若RETN指令有参数n，则将ESP增加n字节，从而释放栈上的参数
- 恢复对调用函数的执行

栈操作与函数调用指令

函数的指令框架

```
PUSH EBP
MOV EBP, ESP
…
MOV ESP, EBP ;opt
POP EBP
RETN
```

栈操作与函数调用指令

注意

- 对于远调用(调用函数和被调用函数不在同一代码段中)，除需要保存EIP之外还要保存CS
- CPU不跟踪返回指令指针的位置，由程序员确保在执行RET指令时，栈顶内容为返回指令指针
- CPU不要求返回指令指针必须指回调用者函数，在执行RET指令前，返回指令指针可以被修改并指向当前代码段中任一地址(脆弱性)
- 将ESP指向返回指令指针的方法：将ESP指向被调用函数栈帧的EBP，再从栈顶POP出调用者函数的EBP，从而使得栈顶内容变为返回指令指针

栈操作与函数调用指令

练习：
对以下程序在/Od编译优化选项下的编译结果用OllyDbg进行调试，画出程序执行过程中整个栈的变化过程

```
#include "stdio.h"
long add(long a, long b){
    long x=a, y=b;
    return (x+y);
}
int main(int argc, char* argv[]){
    long a=1, b=2;
    printf("%d\n", add(a,b));
    return 0;
}
```

调用惯例(Calling Convention)

是对函数调用时如何传递参数和返回值的约定

- 参数传递用寄存器？用栈？两者都用？
- 参数从左到右/从右到左压栈？
- 返回值存储在栈？寄存器？两者都存？

主要调用惯例

- cdecl：C语言中使用(GCC, GNU libraries)；参数从右到左压栈；EAX, ECX, EDX不保存(调用者应自己保存)；返回值由EAX返回；调用者清理栈 (ESP自增)
- stdcall：常用于Win32 API，被调用者清理栈(RETN n)
- fastcall：类似于stdcall，但使用寄存器ECX、EDX传递函数的前2个参数

中断指令

中断：通常指由I/O设备触发的异步事件

异常(exception)：CPU在执行指令时，检测到一个或多个预定义条件时产生的同步事件

- 故障(fault)：可修正的异常。故障处理后执行产生故障的指令
- 陷入(trap)：调用特定指令（如SYSENTER）时产生的异常。陷入处理后执行产生陷入的指令的下一条指令

中断和异常的处理

中断与一个索引值相关，该索引值是一个函数指针数组(中断向量表IVT/中断描述符表IDT)的索引，当中断发生时，CPU执行对应索引处的函数，然后恢复中断发生前的执行

中断指令

INT n

- 生成一个软件中断，其中n为中断向量编号
- INT3 / INT 3H
  - 显式地调用断点异常处理程序
  - 指令的十六进制值：0xCC或0xCD 0x03
- INT 80H
  - Unix系统调用

IRET

- 从中断处理程序返回被中断函数
- 与RET的区别：IRET还会从栈上恢复EFLAGS

提要

数据类型

C declaration	Intel data type	Assembly code suffix	x86-64 size (bytes)	IA32 Size
`char`	Byte	`b`	1	1
`short`	Word	`w`	2	2
`int`	Double word	`l`	4	4
`long int`	Quad word	`q`	8	4
`long long int`	Quad word	`q`	8	8
`char *`	Quad word	`q`	8	4
`float`	Single precision	`s`	4	4
`double`	Double precision	`d`	8	8
`long double`	Extended precision	`t`	10/16	10/12

寄存器

- 16个64位GPR，注意前缀为R的是64位寄存器

RIP相对寻址

- 允许指令在引用数据时使用相对于RIP的地址
- 常用于访问全局变量，全局变量a常通过a(%rip)进行访问

```
#include <stdio.h>
extern long a;

void func(void) {
        printf("%lu\n", a);
}
```

```
func:
.LFB23:
        subq    $8, %rsp
        movq    a(%rip), %rdx
        movl    $.LC0, %esi
        movl    $1, %edi
```

RIP相对寻址

- 这种访问方式支持位置独立代码PIC
  - 每次运行时，程序的函数和全局变量被OS装载到不同的起始地址
- PIC指令访问全局变量时不使用直接寻址(如“movl g %eax”)，而是使用相对于下一条语句“%rip: movl g(%rip), %eax”的Δ相对量，此相对量与程序被装载到的内存位置无关。例：指令装载到(起始地址+0x80)，该指令将(起始地址+0x1000)处的全局变量g装载到%rax中
  - 非PIC中，指令形如“movq (0x401000), %rax”
    (编译结果为“movq g, %rax”，要求g有固定的地址)
  - PIC中，指令形如“movq 0xf80(%rip), %rax”(编译结果为“movq g(%rip), %rax”)

If the starting point is...	The instruction is at...	g is at...	With delta...
0x400000	0x400080	0x401000	0xF80
0x404000	0x404080	0x405000	0xF80
0x4003F0	0x400470	0x4013F0	0xF80

算数运算、规范地址、系统调用

- 算术运算： 多数算术运算都提升到64位，即使操作数只有32位

```
MOV RAX, 1122334455667788H
XOR EAX, EAX                          ;也会清除RAX的高32位，执行后RAX=0
MOV RAX, 0FFFFFFFFFFFFFFFFH
INC EAX                                ;执行后RAX=0
```

- 规范地址（Canonical Address）
  - X64的虚拟地址宽度为64位，但实际CPU只使用48位地址空间
  - 虚拟地址的规范形式指48～63位都与第47位相同的地址
  - 若代码试图解引用一个非规范地址，则触发系统异常
- 系统调用
  - x86：0x80中断或特殊的SYSENTER指令
  - x64：SYSCALL指令

函数调用

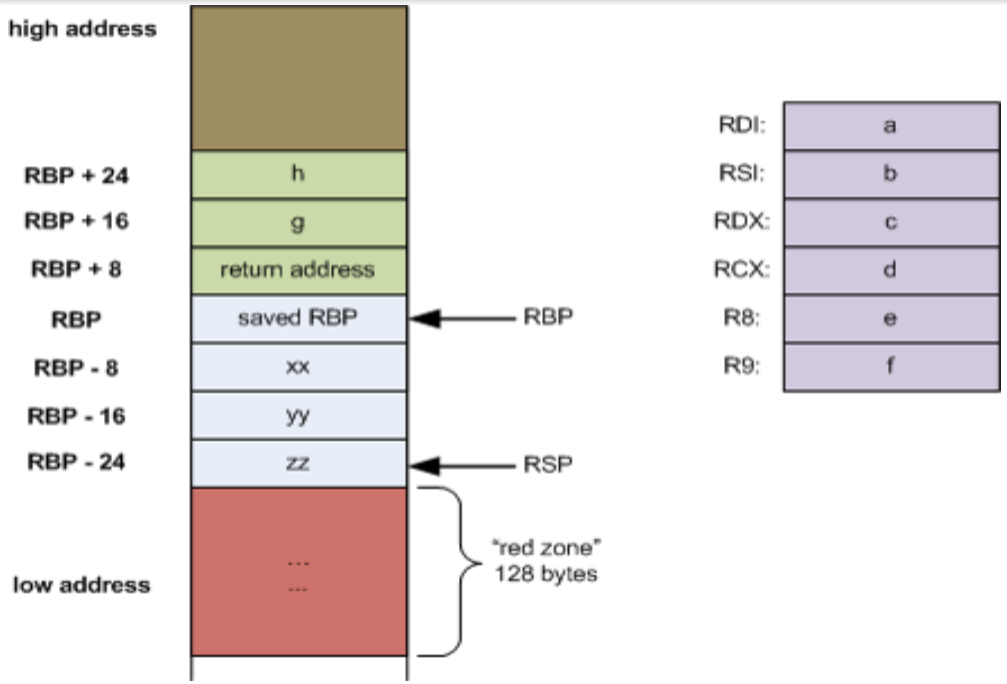
- x64的多数调用惯例更多地通过寄存器传递参数(变形的fastcall，调用者清理栈)
- Windows x64中，只有一种调用惯例用到栈，且其中前4个参数通过RCX，RDX，R8，R9 传递
- Linux x64中，System V AMD64 ABI调用惯例：前6个参数通过RDI，RSI，RDX，RCX，R8，R9传递
- 相比IA-32，减小了在栈上存储和查找值的时间，一些函数根本不需要访问栈

函数调用：参数传递(System V AMD64 ABI调用惯例)

Operand size (bits)	Argument Number 1	2	3	4	5	6
64	%rdi	%rsi	%rdx	%rcx	%r8	%r9
32	%edi	%esi	%edx	%ecx	%r8d	%r9d
16	%di	%si	%dx	%cx	%r8w	%r9w
8	%dil	%sil	%dl	%cl	%r8b	%r9b

函数调用：参数传递(System V AMD64 ABI调用惯例)

```
long myfunc(long a, long b, long c, long d,
            long e, long f, long g, long h) {
   long xx = a * b * c * d * e * f * g * h;
   long yy = a + b + c + d + e + f + g + h;
   long zz = utilfunc(xx, yy, xx % yy);
   return zz + 20;
}
```



函数调用： Red-Zone优化

Red zone

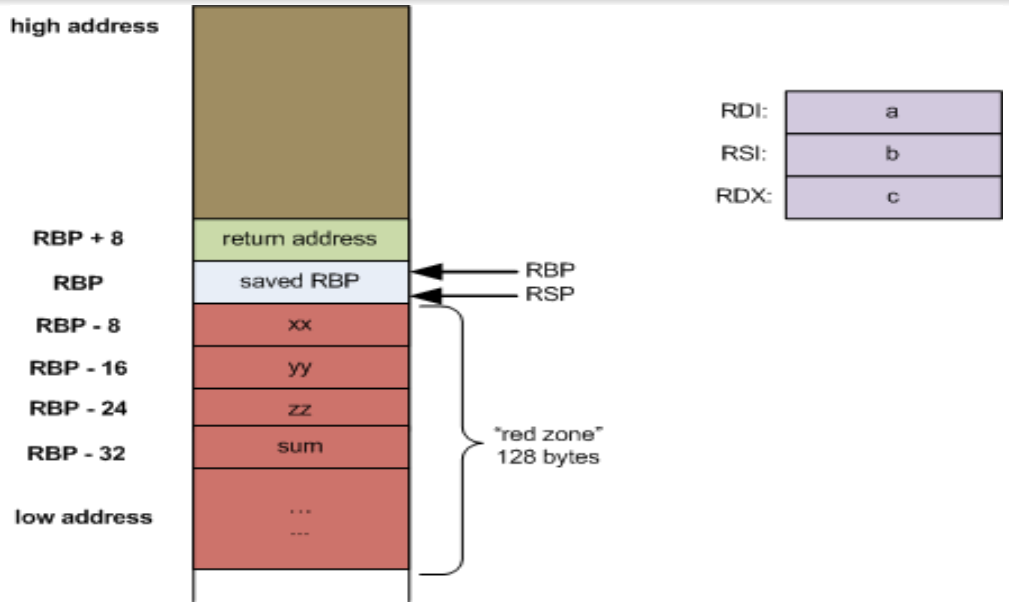
rsp指针向下(低地址)的128字节栈空间可保留为不被信号或中断处理程序更改，从而作为函数的临时数据空间，称为red zone

- 叶函数(即不调用其他函数的函数)使用red zone作为其栈帧，而不需要在其序言语句和收尾语句中修改栈指针

函数调用： Red-Zone优化

叶函数实例

```
long utilfunc(long a, long b, long c){
    long xx = a + 2;
    long yy = b + 3;
    long zz = c + 4;
    long sum = xx + yy + zz;
    return xx * yy * zz + sum;
}
```



对栈帧基址针优化

对于需要栈帧的函数，处理栈帧基址针有两种方法

- 方法1：传统方法(GCC无优化选项时)
  - 函数序言：保存caller的栈帧基址针；创建新的栈帧基址针
  - 函数体：对栈单元的引用采用相对于栈帧基址针的方式
  - 函数结尾：恢复caller的栈帧基址针
- 方法2：更快的方法(GCC优化选项时)
  - 不保存/恢复栈帧基址针；rbp作为通用寄存器使用
  - 对栈单元的访问通过相对于rsp来完成
  - 初始进行栈分配；rsp在函数执行过程中保持在一个固定位置

书面作业：第2章习题3

- 着重分析：1)调用惯例的差异；
  2)函数调用、返回时的栈状态的变化
- 完成分析报告，发送到助教邮箱，
  文档题目“[学号]姓名-调用惯例分析”
- Deadline：2020/09/27，23:59:59

参考文献

- https://software.intel.com/en-us/articles/intel-sdm
- https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64
- PSU CS597, intro to x86-64. http://www.cse.psu.edu/~gxt29/teaching/cse597s19/slides/03x86-64.pdf