汇编语言与逆向工程

北京邮电大学

2019年3月

北邮　网安学院　崔宝江

第二章 汇编语言

- 1.寄存器
- 2.x86指令集
- 3.寻址方式
- 4.字节序
- 5.栈
- 6.函数调用约定

1.寄存器

- CPU简介
- 什么是寄存器
- 寄存器分类
- 通用寄存器
- 段寄存器
- 程序状态与控制寄存器
- 指令指针寄存器

基本知识

基本知识介绍

- CPU的基本组成部件：时钟、ALU、CU、寄存器
  - 时钟是整个计算机运转的“节拍器”，机器指令的最小执行时间就是一个时钟周期
  - ALU是算数单元，顾名思义就是执行各种算数运算和逻辑运算

基本知识

- CPU的基本组成部件：时钟、ALU、CU、寄存器
  - CU是控制单元，是CPU的指挥中心，由下列组成
    - 指令寄存器（Instruction Register）
    - 指令译码器（Instruction Decoder）
    - 指令控制器（Operation Controller）
  - CU协调各个指令执行的顺序

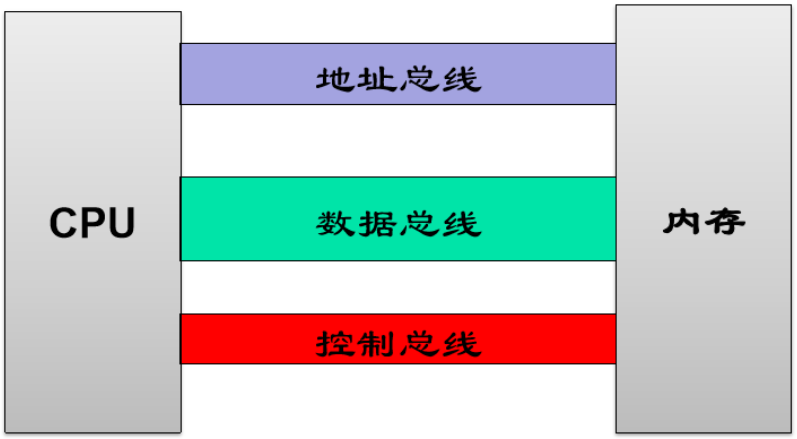
基本知识

- CPU的基本组成部件：时钟、ALU、CU、寄存器
  - 寄存器的作用是临时存储数据和地址，供cpu操作，包括
    - 8个通用寄存器（EAX,EBX,ECX,EDX,EDI,ESI,ESP,EBP）
    - 1个指令指针寄存器（EIP），EIP始终指向下一条待执行指令的地址
    - 一个CPU状态寄存器（EFLAGS）
    - 6个段寄存器

基本知识

总线

- 总线在物理上是若干根用于连接其他芯片的导线
- 在逻辑上分为地址总线、数据总线、控制总线。



基本知识

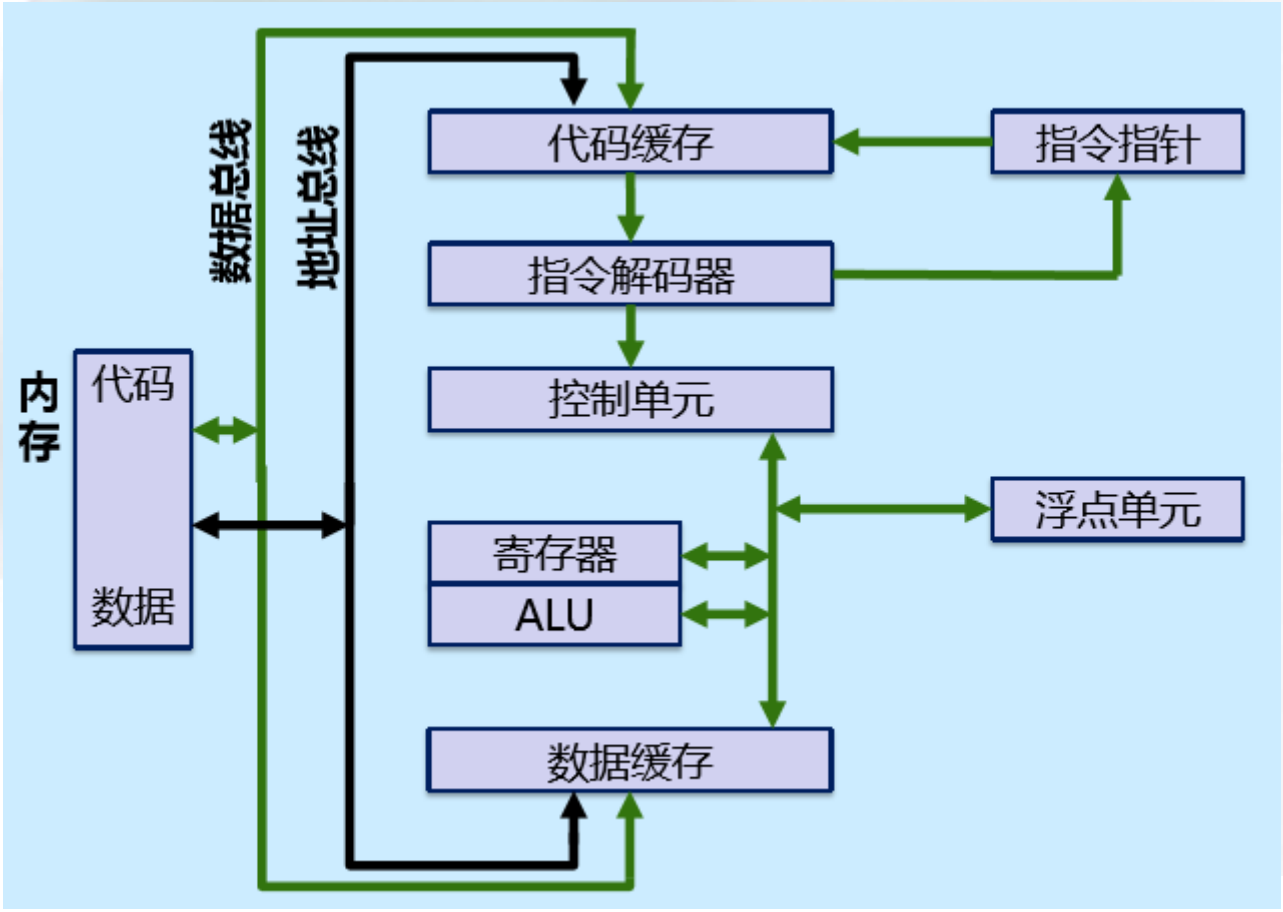
- 地址总线
  - CPU要将内存中的指令和数据取出，就必须访问内存中的地址，当CPU访问内存中的某个地址时，地址线上就保持着那个地址值。
  - 地址总线能表示多少个不同的数值，这个CPU就能够访问多少内存地址（如宽度为N的地址线，它能够寻址的空间为0-2^N）

基本知识

- 数据总线
  - 在CPU和内存中传输指令和数据
- 控制总线
  - 控制总线是不同控制线的集合，是一个总称。
  - 有多少条控制总线，就意味着CPU能够对外界提供多少种控制
  - 控制总线的宽度决定了CPU控制外界设备的能力

基本知识

CPU结构原理



1.寄存器

- 基本知识介绍
- 什么是寄存器
- 寄存器分类
- 通用寄存器
- 段寄存器
- 程序状态与控制寄存器
- 指令指针寄存器

1.寄存器

- 什么是寄存器
  - 寄存器(Register)是中央处理器(CPU，Central Processing Unit)内部的组成部分。
  - 寄存器是有限存贮容量的高速存储部件，它们可用来暂存指令、数据和地址。

1.寄存器

- 在中央处理器的控制部件中的寄存器
  - 指令寄存器(IR，Instruction Register)
- 在中央处理器的算术及逻辑部件中的寄存器
  - 累加器(ACC，accumulator)
- 寄存器是计算机系统结构下存储层次的最顶端，也是系统中操作数据的最快速路径

1.寄存器

- 基本知识介绍
- 什么是寄存器
- 寄存器分类
- 通用寄存器
- 段寄存器
- 程序状态与控制寄存器
- 指令指针寄存器

1.寄存器

- IA-32架构提供了16个基本程序执行寄存器，用于系统和应用程序编程。
  - IA-32是1985年的intel 80386到奔腾系列处理器所沿用的intel处理器架构
  - 16个寄存器可以被分成以下四类：
    - 8个通用寄存器(General-purpose registers)
      - 可用来存储操作数和指针。
    - 6个段寄存器(Segment registers)
    - 程序状态与控制寄存器(EFLAGS)
    - 指令指针寄存器(EIP register)

1.寄存器

- 基本知识介绍
- 什么是寄存器
- 寄存器分类
- 通用寄存器
- 段寄存器
- 程序状态与控制寄存器
- 指令指针寄存器

1.寄存器

- 通用寄存器
  - 32位CPU通用寄存器共有8个：EAX，EBX，ECX，EDX，ESI，EDI，EBP，ESP
  - 它们可以用于传送和暂存以下数据
    - 逻辑和算术运算的操作数；
    - 用于地址计算的操作数；
    - 内存指针

1.寄存器

- 第1- 4个寄存器
  - EAX：累加寄存器，是操作数和结果数据的累加器
  - EBX：基址寄存器，指向DS段中数据的指针
  - ECX：计数寄存器，是字符串和循环操作的计数器
  - EDX：I/O指针--数据寄存器
    - 上面4个寄存器主要用于算术运算（ADD/SUB/XOR/OR等）指令中，常用来保存常量与变量的值。

1.寄存器

- 变址寄存器（第5-6个）
  - ESI：(字符串操作源指针)源变址寄存器。
  - EDI：(字符串操作目的指针)目的变址寄存器
    - ESI和EDI与特定的串操作指令（MOVS/LODS/STOS）一起使用，在字符串操作的时候用的比较多
    - 变址寄存器存放存储单元在段内的偏移量，用它们可实现多种存储器操作数的寻址方式，为以不同的地址形式访问存储单元提供便利

1.寄存器

- 指针寄存器（第7-8个）
  - ESP：堆栈指针寄存器，用于存放当前堆栈的栈顶地址，专门用作堆栈指针，不可作为一般通用寄存器使用
  - EBP：基址指针寄存器，表示栈区域的基地址，在函数被调用时用于保存ESP的值，在函数返回时再把值返回ESP

1.寄存器

- EAX、EBX、ECX、EDX四个通用寄存器
  - 为了实现与16位CPU(8086CPU)的兼容（即对低16位数据的存取），这些低 16位寄存器分别命名为AX、BX、CX、DX
  - 对低16位数据的存取，不会影响高16位的数据

General-Purpose Registers

31 16	15 8	7 0	16-bit	32-bit
	AH	AL	AX	EAX
	BH	BL	BX	EBX
	CH	CL	CX	ECX
	DH	DL	DX	EDX
	BP			EBP
	SI			ESI
	DI			EDI
	SP			ESP

1.寄存器

- 为了兼容8086CPU的上一代CPU中8位寄存器，同样，8086CPU的AX、BX、CX、DX这四个16位寄存器又可分为两个独立使用的8位寄存器来用：
  - AX可分为AH和AL；
  - BX可分为BH和BL；
  - CX可分为CH和CL；
  - DX可分为DH和DL。

General-Purpose Registers

31 16	15 8	7 0	16-bit	32-bit
	AH	AL	AX	EAX
	BH	BL	BX	EBX
	CH	CL	CX	ECX
	DH	DL	DX	EDX
	BP			EBP
	SI			ESI
	DI			EDI
	SP			ESP

1.寄存器

- 例：EAX、AX、AH(AL)之间的关系
  - EAX为32位寄存器，AX为16位寄存器，AH(AL)为8位寄存器
  - 若想存储4个字节(DWORD，32位)的数据，就使用EAX
  - 若只想使用2个字节(WORD，16位)，使用EAX的低16位部分AX就可以
  - AX又可分为高8位的AH寄存器和低8位的AL寄存器
  - 每个寄存器都有自己的名称，可独立存取。程序员可利用数据寄存器的这种“可分可合”的特性，灵活地处理字/双字/字节的信息。

1.寄存器

- EAX、EBX、ECX、EDX四个通用寄存器

General-Purpose Registers

31 16	15 8	7 0	16-bit	32-bit
	AH	AL	AX	EAX
	BH	BL	BX	EBX
	CH	CL	CX	ECX
	DH	DL	DX	EDX
	BP			EBP
	SI			ESI
	DI			EDI
	SP			ESP

1.寄存器

- 指针寄存器EBP、ESP和变址寄存器ESI、EDI
  - 有类似的低16位寄存器BP、SP、SI、DI
  - 但是它们不可分割成8位寄存器

General-Purpose Registers

31 16	15 8	7 0	16-bit	32-bit
	AH	AL	AX	EAX
	BH	BL	BX	EBX
	CH	CL	CX	ECX
	DH	DL	DX	EDX
	BP			EBP
	SI			ESI
	DI			EDI
	SP			ESP

1.寄存器

- 基本知识介绍
- 什么是寄存器
- 寄存器分类
- 通用寄存器
- 段寄存器
- 程序状态与控制寄存器
- 指令指针寄存器

1.寄存器

- 段寄存器背景知识
  - 8086CPU有20位地址总线，可以传送20位地址，最大寻址空间为1MB()
  - 而8086CPU是16位结构的CPU
    - 也就是说在8086内部，能够一次性处理、传输、暂时存储的信息的最大长度是16位，表现出的寻址能力只有64KB()
    - 为了克服CPU寻址能力不足的缺点，Intel公司最终决定在CPU内部采用两个16位地址合成的方式，形成一个20位的物理地址

1.寄存器

- 物理地址=段基址(段地址*10H）+偏移地址
  - 8086的物理地址是20位的，而段寄存器只有16位，在合成物理地址时需要先将段寄存器中16位的段地址左移四位得到一个20位的段地址，也就是在段地址低位补四个0，相当于乘了16进制的10，这就是段地址*10H
  - 段基地址不能随意乱取，通常都是“小段的起始地址”
    - “小段”即是在物理地址中从00000H开始，每16个字节而划分的，那么整个物理地址空间就可以划分为64K个小段，且首地址的最后四位均为0（用二进制表示时），所以是16的倍数。

1.寄存器

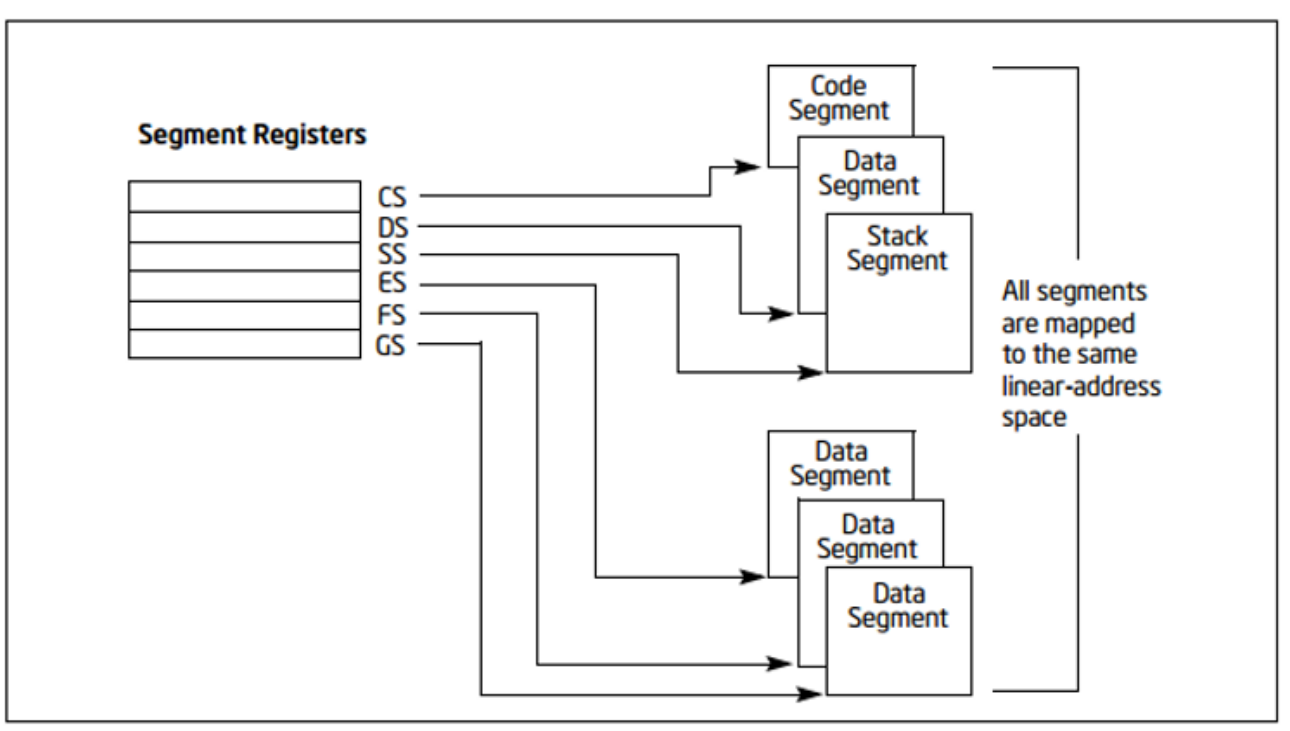
- 8086CPU使用以上方式给出内存单元的物理地址，可以用分段的方式管理内存
  - 将内存分为很多段，每一段都有一个段基址，这个段基址自然是一个20位的内存地址
  - 分段的起源和段寄存器的诞生
    - 由于8086CPU只有16位，因此当时设计了四个段寄存器(CS、DS、ES和SS)分别用于指令、数据、其它和堆栈
    - 段寄存器来存储段基址的高16位，即存储段地址，段地址左移4位()后，再加上16位的偏移地址，得到最后的物理地址。

1.寄存器

- 从80386CPU开始
  - 80x86家族的32位微处理器始于80386
  - 80386的数据总线和地址总线都达到了32根
  - 最大物理寻址空间为4GB
  - 由于寻址空间的增加，80x86为段部分提供了6个段寄存器：CS、DS、ES、SS、FS和GS

1.寄存器

- 目前80x86系统一共有6个段寄存器



1.寄存器

- CS：Code Segment，代码段寄存器
  - 存放应用程序代码所在段的段基址(这里说的段基址指的是存储在寄存器中的段基址的高16位，下同)
- DS：Data Segment，数据段寄存器
  - 用于存放数据段的段基址
- SS：Stack Segment，堆栈段寄存器
  - 用于存放栈段的段基址
- ES、FS、GS，附加数据段寄存器
  - 用于存放程序使用的附加数据段的段基址

1.寄存器

- 基本知识介绍
- 什么是寄存器
- 寄存器分类
- 通用寄存器
- 段寄存器
- 程序状态与控制寄存器
- 指令指针寄存器

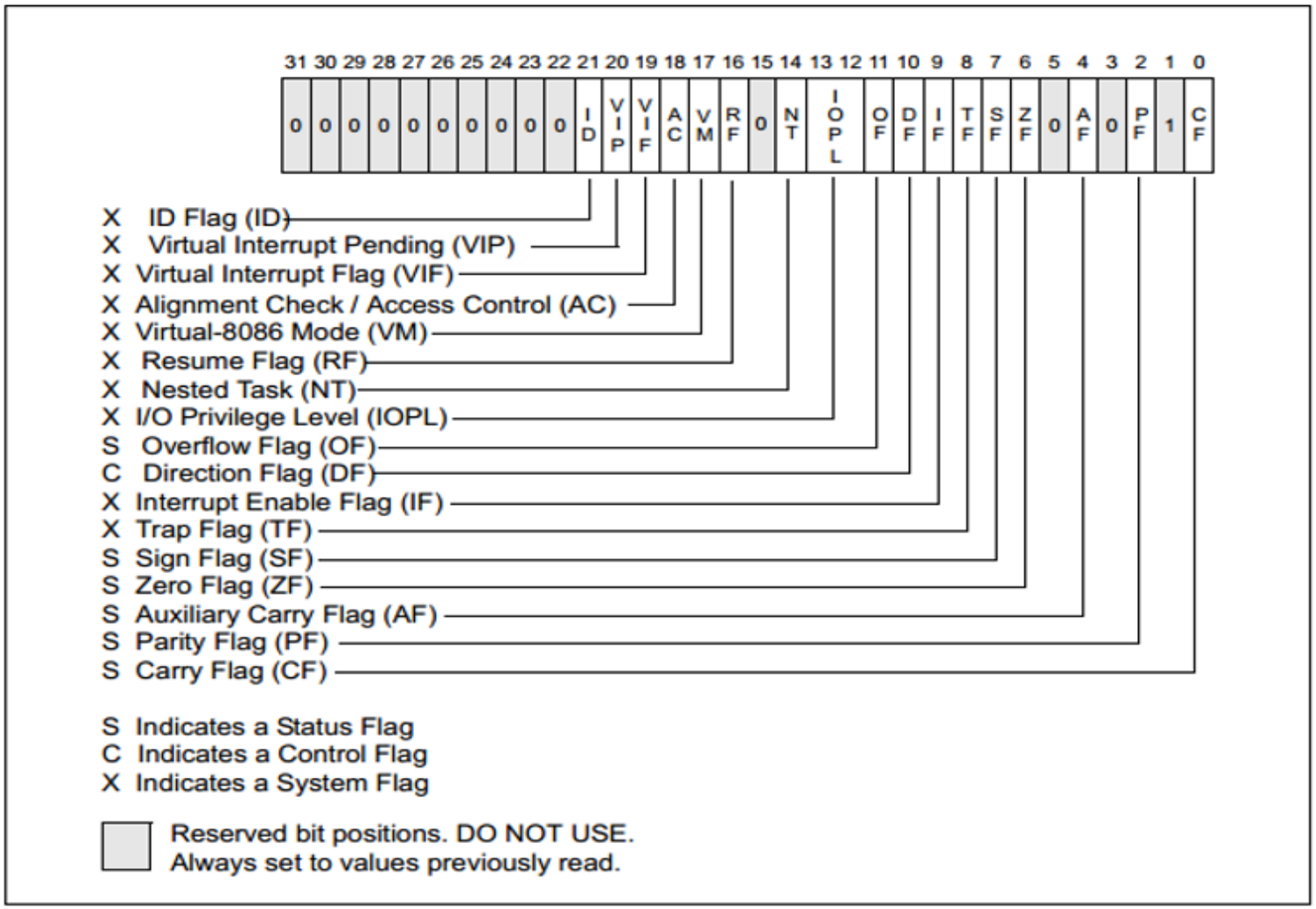
1.寄存器

- 程序状态与控制寄存器（也称为标志寄存器）
  - 3种作用：
    - 用来存储相关指令的某些执行结果；
    - 用来为CPU执行相关指令提供行为依据；
    - 用来控制CPU的相关工作方式。

1.寄存器

- 在IA-32中标志寄存器的名称为EFLAGS，其大小为4个字节(32位)
- 32位EFLAGS寄存器包含一组状态标志、一个控制标志和一组系统标志
- EFLAGS寄存器共有32个位元，每一位都有专门的含义，记录特定的信息，每位的值为1或0，代表On/Off或True/False
- 一些标志可以使用专用指令直接修改，但是没有指令可以将整个寄存器进行检查或修改

1.寄存器


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.

1.寄存器

- EFLAGS寄存器的32位标志可以分为3类：
  - 状态标志(位0，2，4，6，7和11)：表示算数指令的运算结果，如ADD、SUB、MUL和DIV指令。
  - 方向标志(DF，位10)：控制串操作指令的处理方向
    - DF=0，从低地址到高地址
    - DF=1，从高地址到低地址
  - 系统标志(位8，9，14，16，17，18，19，20，21)和IOPL(I/O Privilege Level)字段(位12，13)
    - 控制操作系统或执行操作，应用程序不能修改以上标志位。

1.寄存器

- 状态标志
  - 掌握6个与程序调试相关的状态标志(位0，2，4，6，7，和11)
    - CF(位0)
      - 进位标志位，一般情况下，在进行无符号数运算的时候，它记录了运算结果的最高有效位向更高位的进位值，或从更高位的借位值。
      - 在加法运算中，若运算结果从字或字节的最高位产生了进位，则CF=1；否则CF=0。在减法运算中，若被减数无借位，则CF=0；否则CF=1。
    - PF(位2)
      - 奇偶标志位，记录相关指令执行后，其结果的所有位中1的个数是否为偶数。如果1的个数为偶数，PF=1；如果1的个数为奇数，PF=0。

1.寄存器

- AF(位4)
  - 辅助进位标志位，在发生以下情况时，辅助进位标志AF的值被置为1，否则其值为0：
    - *在字操作时，发生低字节向高字节进位或借位时
    - *在字节操作时，发生低4位向高4位进位或借位时
- ZF(位6)
  - 零标志位，记录相关指令执行后，其结果是否为0；
  - 若运算结果为0，则其值为1，否则其值为0。

1.寄存器

- SF(位7)
  - 符号标志位，记录相关指令执行后，其结果是否为负
  - 当操作数为有符号数时，若结果为负数，SF=1；若结果为非负数，SF=0。
- OF(位11)
  - 溢出标志位，一般情况下，OF记录了有符号数运算的结果是否发生了溢出
  - 如果发生了溢出，OF=1；如果没有发生溢出，OF=0
- 注：CF和OF所表示的进位和溢出，是分别针对无符号数和有符号数运算而言的，一定要分清楚CF和OF的发生条件。

1.寄存器

- 基本知识介绍
- 什么是寄存器
- 寄存器分类
- 通用寄存器
- 段寄存器
- 程序状态与控制寄存器
- 指令指针寄存器

1.寄存器

- **EIP：Instruction Pointer，指令指针寄存器**
  - **保存着CPU下一条将要执行指令的偏移量(offset)，这个偏移量是相对于目前正在运行的代码段寄存器CS而言的**
  - **偏移量加上当前代码段的基地址，就形成了下一条指令的地址**
  - **它的大小为32位，是由原来的16位IP寄存器扩展而来**

1.寄存器

- 程序运行时，CPU会读取EIP中一条指令的地址，传送指令到指令缓冲区后，EIP的值自动增加
- CPU每次执行完一条指令，就会通过EIP寄存器读取并执行下一条指令
- 不能直接修改EIP的值，只能通过其他指令间接修改
  - 这些特定指令包括JMP、JC、CALL、RET
  - 可以通过中断或异常来修改EIP的值

第二章 基础知识

- 1.寄存器
- 2.x86指令集
- 3.寻址方式
- 4.字节序
- 5.栈
- 6.函数调用约定

2. x86指令集

- 数据传送指令
- 算数运算指令
- 逻辑运算和移位指令
- 串操作指令
- 控制转移指令
- 处理器控制指令

2. x86指令集

- 数据传送指令
  - 通用数据传送指令
    - 数据传送指令
    - 堆栈操作指令
    - 数据交换指令
  - 地址传送指令
  - 标志寄存器传送指令

2. x86指令集

- 数据传送指令
  - 数据传送指令是将数据、地址或立即数传送到寄存器或存储单元中。
  - 大部分这类指令不影响状态标志位，部分涉及标志寄存器FLAGS的指令（SAHF和POPF）例外。

2. x86指令集

- 通用数据传送指令
  - MOV：把源操作数传送到目的操作数。
  - MOVSX：带符号扩展传送。
  - MOVZX：带零扩展传送。
  - 指令格式：MOV(MOVSX/MOVZX) DEST，SRC
  - 例
    - MOV EAX，EDX；寄存器EDX->EAX的数据传送。
    - MOVSX EAX，BL；将80H扩展为FFFFFF80H后送EAX中
    - MOVZX AX，BL；将80H扩展为0080H后送AX中。

2. x86指令集

MOV传送指令

操作数类型：

- 1）OPS可以为：存储器、通用寄存器、段寄存器和立即数。
- 2）OPD可以为：存储器、通用寄存器和段寄存器（CS除外）。
- 注意：
  - 1）立即数不能送段寄存器，其余可以任意搭配；立即数送存储器的指令有时难以确定操作数的长度，需要在存储器操作数的前面加上类型说明BYTE PTR或WORD PTR。
  - 例如：MOV BYTE PTR[SI+10H]，30 ；8位立即数30送偏移地址为SI+10H的字节单元；MOV WORD PTR[BX+DI]，2 ；16位立即数2送偏移地址为BX+DI的字单元。

2. x86指令集

MOV传送指令

- 注意：
  - 2）两存储单元之间不能直接进行数据传送；两个段寄存器之间不能直接进行传送信息。但可以用CPU内部寄存器为桥梁来完成这样的传送

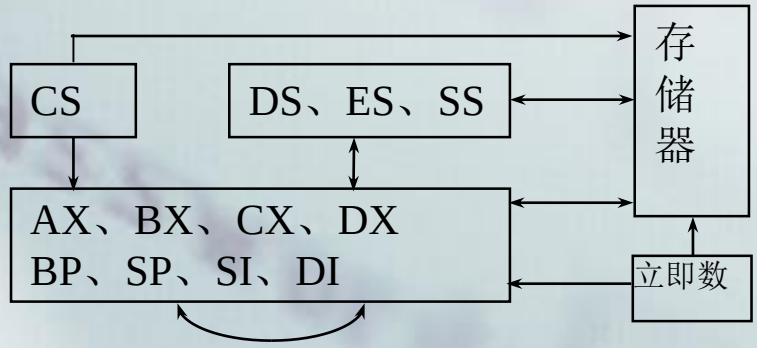
例如：MOV　AL，AREA1　　例如：　MOV　　AX，1000H

　　MOV　AREA2，AL　　MOV　　DS，　AX

  - 3)立即数、代码段寄存器CS只能作源操作数。
  - 4）IP寄存器不能作源操作数或目的操作数。

2. x86指令集

MOV传送指令



2. x86指令集

MOV传送指令

数据传送指令如下：

① 立即数送寄存器

```
MOV     AL，10H
MOV     BX，2100H
```

② 寄存器之间传送

```
MOV     DX，CX
MOV     AH，DL
MOV     DS，AX
MOV     DX，ES
```

③ 通用寄存器与存储器之间传送

```
MOV     AX，[1000H]
MOV     [BP]，DX
```

④ 段寄存器与存储器之间传送

```
MOV     [BX][DI]，ES
MOV     DS，10[BP+DI]
```

指出下列数据传送指令中的错误。

① MOV 10H，AX

立即数不能作为目的操作数

② MOV DS，2000

立即数不能送段寄存器

③ MOV CS，AX

CS不能作为目的操作数

④ MOV DS，ES

目的操作数和源操作数不能同时为段寄存器

⑤ MOV [DI]，[SI]

目的操作数和源操作数不能同时为存储器

⑥ MOV AL，BX

类型不匹配，AL为8位、BX为16位寄存器

⑦ MOV DL，300

类型不匹配，DL为8位寄存器，300超过1B

2. x86指令集

- PUSH：操作数进栈
- PUSHA：把AX，CX，DX，BX，SP，BP，SI，DI依次压入堆栈。
- PUSHAD：把EAX，ECX，EDX，EBX，ESP，EBP，ESI，EDI依次压入堆栈。
- 指令格式：PUSH (PUSHA/PUSHAD) SRC

2. x86指令集

❍PUSH：操作数进栈

1）入栈指令

格式：PUSH　OPS

操作：SP ← SP-2，[SP+1][SP] ← OPS

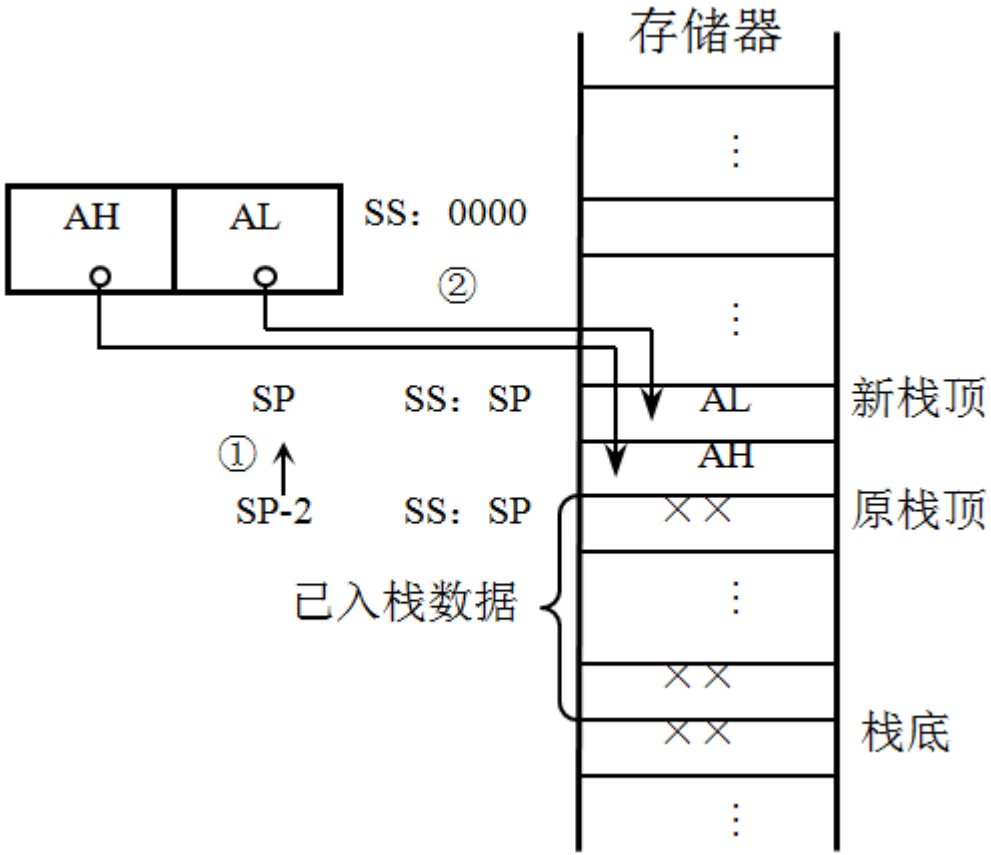
操作数类型： OPS可以是存储器、通用寄存器和段寄存器，但不能是立即数。

说明：PUSH指令先将SP的内容减2，然后再将操作数OPS的内容送入由SP指出的栈顶即偏移地址为SP和SP+1的两个连续字节中。

```
PUSH    AX          ；通用寄存器内容入栈
PUSH    CS          ；段寄存器内容入栈
PUSH    [SI]        ；字存储单元内容入栈
```


2. x86指令集

- PUSH：操作数进栈



2. x86指令集

- POP：出栈到目的操作数，把当前的SP所指向的堆栈顶部的一个字送到指定的目的操作数
- POPA：把DI，SI，BP，SP，BX，DX，CX，AX依次弹出堆栈。
- POPAD：把EDI，ESI，EBP，ESP，EBX，EDX，ECX，EAX依次弹出堆栈。
- 指令格式：POP (POPA/POPAD) DEST

2. x86指令集

- **POP**：出栈到目的操作数，把当前的**SP**所指向的堆栈顶部的字送到指定的目的操作数

格式：POP　OPD

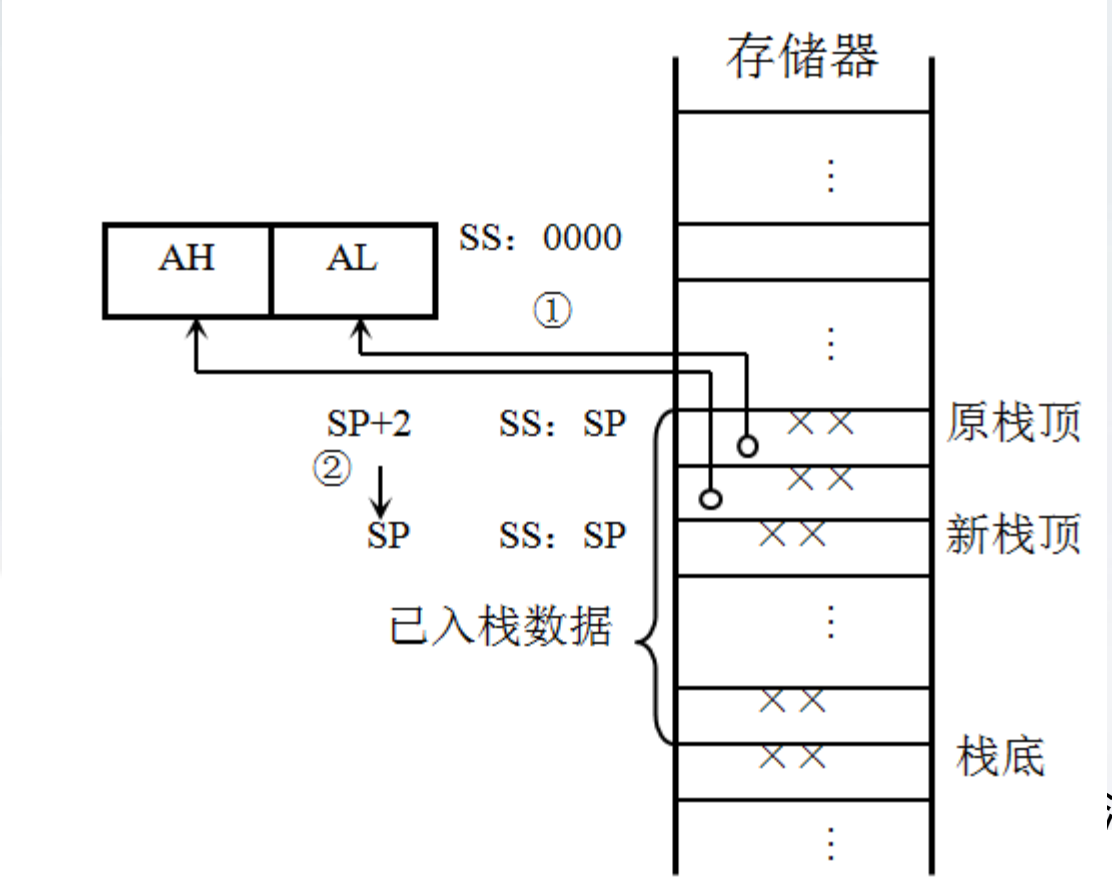
操作：OPD ← [SP+1][SP]，SP ← SP+2

操作数类型：OPD可以是存储器、通用寄存器或段寄存器（但不能是CS），同样，不能是立即数。

说明：POP指令先将堆栈指针SP所指示的栈顶存储单元的值弹出到操作数OPD中，然后再将SP的内容加2。入栈和出栈操作如图所示。

2. x86指令集

- **POP：出栈到目的操作数，把当前的SP所指向的堆栈顶部的一个字送到指定的目的操作数**



2. x86指令集

- XCHG：交换两操作数。允许通用寄存器之间，通用寄存器和存储器之间交换数据。
- 指令格式：XCHG OPR1，OPR2
- 例
  - XCHG EAX，EBX；通用寄存器之间交换数据。
  - XCHG EBX，[ESI]；通用寄存器和存储器之间交换数据
- 注：两操作数不允许同时为存储器操作数，交换指令不影响标志位。

2. x86指令集

- 数据传送指令
  - 通用数据传送指令
  - 地址传送指令
  - 标志寄存器传送指令

2. x86指令集

- 地址传送指令
  - LEA：将源操作数的有效地址传送到通用寄存器。
  - 指令格式：LEA REG，MEM
  - 例
    - LEA EAX，[EBP-4]；将SS段中EBP-4指向的存储单元送入EAX。

2. x86指令集

- 标志寄存器传送指令
  - PUSHF：16位标志寄存器进栈
  - PUSHFD：32位标志寄存器进栈
  - POPF：16位标志寄存器出栈
  - POPFD：32位标志寄存器出栈

2. x86指令集

- 数据传送指令
- 算数运算指令
- 逻辑运算和移位指令
- 串操作指令
- 控制转移指令
- 处理器控制指令

2. x86指令集

- 算数运算指令
  - 加法指令
  - 减法指令
  - 乘法指令
  - 除法指令

2. x86指令集

- 加法指令
  - ADD：将源操作数和目的操作数相加，结果送到目的操作数。
  - ADC：将源操作数与目的操作数以及CF值相加，结果传送到目的操作数。
  - 指令格式：ADD (ADC) DEST，SRC
    - 注：ADD，ADC指令影响的标志位是OF，SF，ZF，AF，PF，CF。
  - INC：目的操作数加一，结果送入目的操作数
    - 目的操作数可以为通用寄存器或存储器操作数。
  - 指令格式：INC DEST

2. x86指令集

- 减法指令
  - SUB：将目的操作数减去源操作数，结果送入目的操作数。
  - SBB：将目的操作数减去源操作数，再减去CF值，结果送入目的操作数。
  - 指令格式：SUB (SBB) DEST，SRC
    - 注：SUB，SBB指令影响的标志位是OF，SF，ZF，AF，PF，CF。
  - DEC：目的操作数减一，结果送入目的操作数
    - 目的操作数可以为通用寄存器或存储器操作数。
  - 指令格式：DEC DEST
    - 注：INC，DEC指令影响的标志位为OF，SF，ZF，AF，PF。

2. x86指令集

- 乘法指令
  - MUL：无符号数乘法指令，将源操作数和目的操作数相乘，结果送入目的操作数。
  - IMUL：有符号数乘法指令，将源操作数和累加器中的操作数相乘，结果送入目的操作数。
  - 指令格式：MUL (IMUL) SRC
  - 说明：MUL，IMUL指令的源操作数为通用寄存器或存储器操作数，目的操作数缺省存放在ACC累加器（AL，AX，EAX）中。

2. x86指令集

- 除法指令
  - DIV：无符号除法指令
  - IDIV：有符号除法指令
  - 指令格式：DIV (IDIV) SRC
  - 说明：DIV、IDIV指令的源操作数作为除数，为通用寄存器或存储器操作数，目的操作数作为被除数，目的操作数缺省存放在ACC累加器（AL，AX，EAX）中。

2. x86指令集

- 数据传送指令
- 算数运算指令
- 逻辑运算和移位指令
- 串操作指令
- 控制转移指令
- 处理器控制指令

2. x86指令集

- 逻辑运算和移位指令
  - 逻辑运算
  - 算术逻辑移位
  - 循环移位

2. x86指令集

- 逻辑运算
  - **AND**：逻辑与，目的操作数和源操作数按位进行逻辑与运算，结果存目的操作数。
  - **OR**：逻辑或，目的操作数和源操作数按位进行逻辑或运算，结果存目的操作数。
  - 源操作数可以是通用寄存器、存储器操作数或立即数，目的操作数是通用寄存器或存储器操作数。
  - **XOR**：逻辑异或，目的操作数和源操作数按位进行逻辑异或运算，结果存目的操作数。
  - 指令格式：**AND (OR/XOR/XOR) DEST，SRC**

2. x86指令集

- 逻辑运算
  - NOT：逻辑非，对目的操作数按位取反，结果存入目的操作数。
  - 指令格式：NOT DEST

2. x86指令集

- 逻辑运算
  - TEST：目的操作数与源操作数按位进行逻辑与操作，并修改标志位，结果不回送目的操作数。
  - 指令格式：TEST DEST，SRC
  - 注：TEST指令常用在检测某些条件是否满足，但又不希望改变原有操作数的情况下。紧跟在这条指令后面的往往是一条条件转移指令，根据测试结果产生分支，转向不同的处理程序。
  - 例
    - TEST ECX，ECX
    - JE Crackme.00401326；程序跳转到0x00401326处继续执行

2. x86指令集

- 算术逻辑移位
  - SAL：算术左移指令。
  - SHL：逻辑左移指令。
  - 指令格式：SAL (SHL) DEST，OPRD
  - 说明：SAL、SHL指令功能完全相同，按照操作数OPRD的规定的移位位数对目的操作数进行左移操作，每移一位，最低位补0，最高位移入标志位CF中。

2. x86指令集

- 算术逻辑移位
  - SAR：算术右移
    - 按照操作数OPRD规定的移位位数，对目的操作数进行右移操作，每移一位，最低位移入标志位CF，最高位(符号位)保持不变。相当于对有符号数进行除2操作。
  - SHR：逻辑右移
    - 按照操作数OPRD规定的移位位数，对目的操作数进行右移操作，每移一位，最低位移入标志位CF，最高位补0。
  - 指令格式：SAR (SHR) DEST，OPRD

2. x86指令集

- 算术左移和算术右移
  - 主要用来进行有符号数的倍增、减半；
- 逻辑左移和逻辑右移
  - 主要用来进行无符号数的倍增、减半

2. x86指令集

- 循环移位
  - ROL：循环左移
  - ROR：循环右移
  - 指令格式：ROL(ROR) DEST，OPRD

2. x86指令集

- 数据传送指令
- 算数运算指令
- 逻辑运算和移位指令
- 串操作指令
- 控制转移指令
- 处理器控制指令

2. x86指令集

- 串指连续存放在存储器中的一些数据字节、字或双字。
- 串操作允许程序对连续存放的数据块进行操作。
- 串操作通常以DS：ESI来寻址源串，以ES：EDI来寻址目的串

2. x86指令集

- 对于字符串的基本操作，80x86提供了五个指令

指令名称	字节/字操作	字节操作	字操作	双字操作
字符串传送	MOVS 目的串，源串	MOVSB	MOVSW	MOVSD
字符串比较	CMPS 目的串，源串	CMPSB	CMPSW	CMPSD
字符串扫描	SCAS 目的串	SCASB	SCASW	SCASD
字符串装入	LODS 源串	LODSB	LODSW	LODSD
字符串存储	STOS 目的串	STOSB	STOSW	STOSD

2. x86指令集

- MOVS 目的串，源串
  - 指令功能：字符串传送指令，把由ESI作指针的源串的字节或字，传送到由EDI作指针的目的串中。
- CMPS 目的串，源串
  - 指令功能：字符串比较指令，由DS：ESI规定的源串元素减去ES：EDI指出的目的串元素，结果不回送，仅影响标志位CF，AF，PF，OF，ZF和SF

2. x86指令集

- SCAS 目的串
  - 指令功能：串扫描指令，由EAX的内容减去ES：EDI规定的目的串元素，结果不回送，仅影响标志位CF，AF，PF，SF，OF，ZF。
- LODS 源串
  - 指令功能：串装入指令，将DS：ESI所指的源串元素装入累加器EAX中
- STOS 目的串
  - 指令功能：串存储指令，将累加器EAX中值存入ES：EDI所指的目的串存储单元中

2. x86指令集

- 数据传送指令
- 算数运算指令
- 逻辑运算和移位指令
- 串操作指令
- 控制转移指令
- 处理器控制指令

2. x86指令集

- 控制转移指令
  - 无条件转移指令
  - 条件转移指令
  - 循环控制指令

2. x86指令集

- 无条件转移指令

指令	说明
JMP	无条件转移
CALL	过程调用
RET	过程返回

2. x86指令集

- 条件转移指令
  - 条件转移指令是根据上一条指令执行后，CPU设置的状态标志作为判别测试条件，来决定是否转移。
  - 每一种条件转移指令都有它的测试条件
    - 当条件成立，便控制程序转向指令中给出的目的地址
    - 否则，程序仍按顺序执行

2. x86指令集

根据单个标志位的状态判断转移的指令

指令	转移条件	说明
JC DEST	CF = 1	有进位/借位时转移
JNC DEST	CF = 0	无进位/借位时转移
JE/JZ DEST	ZF = 1	相等/等于零时转移
JNE/JNZ DEST	ZF = 0	不相等/不等于零时转移
JS DEST	SF = 1	是负数时转移
JNS DEST	SF = 0	不是负数时转移
JO DEST	OF =1	有溢出时转移
JNO DEST	OF = 0	无溢出时转移
JP/JPE DEST	PF = 1	有偶数个“1”时转移
JNP/JPO DEST	PF = 0	有奇数个“1”时转移

2. x86指令集

- 循环控制指令
  - 这类指令用ECX计数器的内容控制循环次数，先将循环次数存放在ECX中，每循环一次ECX内容减1，直到ECX为0时循环结束。

2. x86指令集

- 数据传送指令
- 算数运算指令
- 逻辑运算和移位指令
- 串操作指令
- 控制转移指令
- 处理器控制指令

2. x86指令集

- 处理器控制指令
  - 空操作指令
    - **NOP（机器码0x90）：空操作，除了使EIP增1外，不做任何操作。**
  - 中断指令：
    - **INTn：软件中断指令，也称为软中断指令，其中n为终端类型号，其值必须在0~255的范围内。**
    - 可以在编程时安排在程序中的任何位置上，因此也被称为陷阱中断。

第二章 基础知识

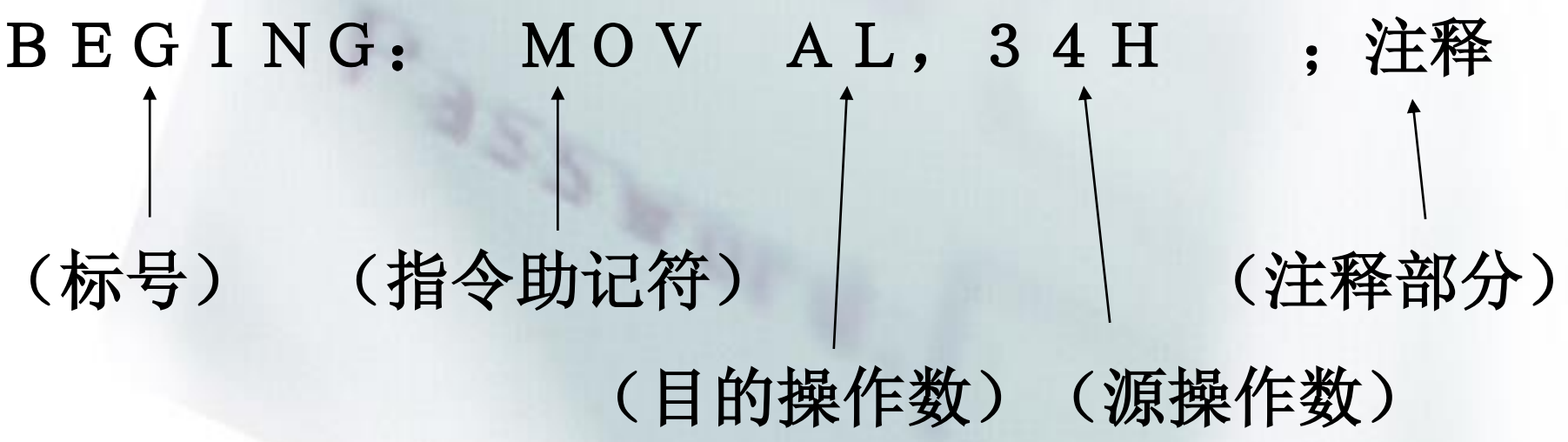
- 1.寄存器
- 2.x86指令集
- 3.寻址方式
- 4.字节序
- 5.栈
- 6.函数调用约定

3.寻址方式

- 寻址方式(Addressing Mode)
  - 确定当前指令操作数地址以及下一条要执行指令地址的方法

3.寻址方式

- 一条指令的标准格式



3.寻址方式

- 汇编代码是由两部分组成：操作码+操作数
  - 操作码在相应的机器指令体系中有相关的表示
  - 操作数的不同表示，则寻址方式不同
  - （1）直接在汇编代码中：那么这种寻址方式就是立即数寻址　**mov ax, 1234H**
  - （2）存放在寄存器中：那么这种寻址方式就是寄存器寻址　**mov ax,bx**
  - （3）存放在内存中，那么这种寻址方式就多了

3.寻址方式

- 立即寻址
- 寄存器寻址
- 直接寻址
- 寄存器间接寻址
- 寄存器相对寻址
- 基址变址寻址
- 相对基址变址寻址
- 比例变址寻址
- 基址比例变址寻址
- 相对基址比例变址寻址

3.寻址方式

- 立即寻址
  - 操作数直接放在指令中，紧跟在操作码之后，它作为指令的一部分存放在代码段中，这种操作数叫做立即数
  - 立即数寻址常用来给寄存器赋初值，不用访问寄存器、存储器，指令执行速度快。
  - 例：**MOV EAX，26H**；将一个立即数**26H**送到**EAX**寄存器中。
  - 注意：立即数只能作为源操作数，不能作为目的操作数，源操作数的长度应该和目的操作数保持一致

3.寻址方式

- 示例： **mov ax,1234H**
- **AH = ？ H， AL= ？ H**

3.寻址方式

- 示例： mov ax,1234H
- AH = 12H， AL=34H

3.寻址方式

- 寄存器寻址
  - 这种寻址方式由于操作数就在寄存器中,不需要访问存储器来取得操作数
  - 例： MOV EAX，EBX；
  - 操作数存放在寄存器中，在指令中给出具体寄存器，指令执行时会到指定寄存器中取出相应的操作数，源和目的操作数都可以是寄存器。
  - 寄存器寻址只访问寄存器，不访问存储器，速度快
  - 指令中可以引用的寄存器及其符号名称如下：
    - 8位寄存器有：AH、AL、BH、BL、CH、CL、DH和DL
    - 16位寄存器有：AX、BX、CX、DX、SI、DI、SP和BP
    - 32位寄存器有：EAX、EBX、ECX、EDX、ESI、EDI、ESP和EBP等。

3.寻址方式

- 寄存器寻址
  - 例： MOV EAX，EBX；
  - 若执行前(EAX) = 3047H，(EBX) =2378H，执行后(EAX) = ?H。

3.寻址方式

- 寄存器寻址
  - 例： MOV EAX，EBX；
  - 若执行前(EAX) = 3047H，(EBX) =2378H，执行后(EAX) = (EBX) = 2378H。

3.寻址方式

- 以上立即寻址和寄存器寻址两种寻址方式是对寄存器的寻址
- 下面介绍的寻址方式，都在除代码段以外的存储区中，通过不同的寻址方式求得操作数地址，从而取得操作数

3.寻址方式

- 在80x86系统中，内存单元的物理地址由段基址和偏移地址(又称为偏移量)组成
  - 段基址：在IA-32的保护模式下，段基址由16位的段选择符得到，这些段选择符存放在六个段寄存器(CS，SS，DS，ES，FS，GS)中。
  - 偏移地址的计算包含以下四个基本部分：
    - ①基址寄存器
    - ②变址寄存器
    - ③比例因子
    - ④位移量

3.寻址方式

- 将基址寄存器、变址寄存器、比例因子、位移量四部分，按某种计算方法组合形成的偏移地址，称为有效地址EA(Effective Address)
- 有效地址EA=基址+变址*比例因子+位移量
- 其中，基址、变址、位移量的值可正可负，比例因子只能为正。

3.寻址方式

- 当采用16位寻址方式时，有效地址四种成分的组成为：
  - 基址寄存器：BX，BP
  - 变址寄存器：SI，DI
  - 比例因子：1
  - 位移量：0，8，16位
- 当采用32位寻址方式时，有效地址四种成分的组成为：
  - 基址寄存器：所有的32位通用寄存器
  - 变址寄存器：除ESP以外的32位通用寄存器
  - 比例因子：1，2，4，8
  - 位移量：0，8，16，32位

3.寻址方式

- 各种访存类型下所对应的段的默认选择

访存类型	段寄存器	缺省选择规则
指令	CS	用于取指令
堆栈	SS	堆栈操作 任何使用ESP或EBP作为基址寄存器的访存
局部数据	DS	除堆栈和目的串操作之外的其他数据的访问
目的串	ES	串处理指令的目的串

3.寻址方式

- 段超越前缀
  - 当使用内存操作数时， 无论哪种内存操作数寻址都有默认的段寄存器， 然而至多一个内存操作数可不使用默认段寄存器时，允许在程序中自行选择段寄存器，就需要使用段超越前缀
  - 段超越前缀的格式为：

    段寄存器：指令操作数
  - 例 **MOV EAX，ES：[EBP]**

3.寻址方式

- 虽然段超越前缀，允许改变系统所指定的默认段，但是有些情况下不允许修改：
  - 串处理操作中目的串必须使用ES段，即默认为ES:EDI不可修改。
  - 压栈(push)、弹栈(pop)必须使用SS段，即默认为SS:ESP不可修改。
  - 指令必须存放在CS段中。

3.寻址方式

- 直接寻址
  - 操作数在寄存器中，指令中包含有操作数的有效地址(偏移地址)
  - 注：操作数一般存放在数据段DS，所以操作数的有效地址由DS加上指令中直接给出的16位偏移地址得到
  - 指令示例：
  - MOV AX,[1234H]
  - 假设DS = 4567H，内存中[468A4H] = 0001H，那么有效地址 = [4567H] *16+ [1234H]=[468A4H]
  - 那么寄存器AX = ?H

3.寻址方式

- 直接寻址
  - 操作数在寄存器中，指令中包含有操作数的有效地址(偏移地址)
  - 注：操作数一般存放在数据段DS，所以操作数的有效地址由DS加上指令中直接给出的16位偏移地址得到
  - 指令示例：
  - MOV AX,[1234H]
  - 假设DS = 4567H，内存中[468A4H] = 0001H，那么物理地址 = [4567H] *16+ [1234H]=[468A4H]
  - 那么寄存器AX = 0001H

3.寻址方式

- 例题
  - MOV AX,[8054H]
  - 如(DS) = 2000H,
  - 28054H里的内容为3050H
  - 则执行结果为(AX) = ?H

3.寻址方式

- 例题
  - MOV AX,[8054H]
  - 如(DS) = 2000H
  - 28054H里的内容为3050H
  - (物理地址=20000H+8054H=28054H)
  - 则执行结果为(AX) = 3050H

3.寻址方式

- 直接寻址（段超越前缀）
  - 因为默认的是DS寄存器，其实也可以指定前缀寄存器，即段超越前缀
  - MOV AX, SS:[1234H]
  - 把SS数据段中偏移地址为1234H 的字复制到寄存器AX 。

3.寻址方式

- 寄存器间接寻址
  - 操作数的有效地址在寄存器中，这种寻址方式为寄存器间接寻址
  - 如果操作数的有效地址在EAX，EBX，ECX，EDX，ESI和EDI中，以上寄存器默认使用DS作为段寄存器， 即DS段寄存器为段基值。
  - 如果操作数的有效地址在ESP，EBP，这两个寄存器默认使用SS作为段寄存器，即SS段寄存器为段基值。

3.寻址方式

- 例：MOV EAX，[EBP]；
- 把SS段中EBP指向的单元复制到EAX。
  - (EAX)= (SS)*16+ (EBP)
- 例：MOV EAX，[EDX]；
- 把DS段中EDX指向的字节复制到EAX。
  - (EAX)= (DS)*16+ (EDX)
- 例：MOV [EDX]，EBX；
- 把EBX的值复制到DS段中EDI指向的单元。
  - (DS)*16+ (EDX)= EBX

3.寻址方式

- 例题
  - MOV AX,[SI]
  - 如果(DS) = 5000H, (SI) = 1234H
  - 51234H地址中的内容为:6789H
  - 执行该指令后,(AX) = ?H

3.寻址方式

- 例题
  - MOV AX,[SI]
  - 如果(DS) = 5000H, (SI) = 1234H
  - 则物理地址 = 50000 + 1234 = 51234H
  - 51234H地址中的内容为:6789H
  - 执行该指令后,(AX) = 6789H

3.寻址方式

- 寄存器相对寻址
  - 操作数的有效地址EA为基址寄存器或变址寄存器的内容和指令中的位移量之和
    - EAX，EBX，ECX，EDX，ESI和EDI，以上寄存器默认使用DS作为段寄存器。
    - ESP，EBP，这两个寄存器默认使用SS作为段寄存器。
  - 例：MOV ECX，[EAX+24H]，也可以写成 MOV ECX，24H [EAX]；
    - 由DS段中EAX指向的内容加上位移量24,最终组成操作数的有效地址。
    - (ECX) = (DS*16+EAX+24H)

3.寻址方式

- 例题
  - MOV AX,[DI+1223H]
  - 假设，(DS) = 5000H，(DI) = 3678H
  - 则物理地址 = ？ H
  - 5589BH地址中的内容:568AH
  - 5489BH地址中的内容:55AAH
  - 5491BH地址中的内容:5B87H
  - 执行该指令后AX = ？ H

3.寻址方式

- 例题
  - MOV AX,[DI+1223H]
  - 假设，(DS) = 5000H，(DI) = 3678H
  - 则物理地址 = 50000H + 3678H + 1223H = 5489BH
  - 5489BH地址中的内容:55AAH
  - 执行该指令后AX = 55AAH

3.寻址方式

- 基址变址寻址
  - 操作数在寄存器中，操作数的有效地址由基址寄存器的内容与变址寄存器的内容之和获得
  - 通常将指令中的第二操作数的第一个寄存器作为基址寄存器，第二个寄存器作为变址寄存器
  - 其中，ESP不能作为变址寄存器

3.寻址方式

- 例：MOV EAX，[EBX][ESI]，也可写成：MOV EAX，[EBX+ESI]
  - EBX为基址寄存器，ESI为变址寄存器，该指令将DS段中地址为EBX+ESI的存储单元的4字节数据送到EAX。
- 例：MOV EAX，[EBP+ESI]
  - 将SS段中地址为EBP+ESI的存储单元的4字节数据送到EAX。
- 例：MOV EAX，ES：[EBX+ESI]
  - 将ES段中地址为EBX+ESI的存储单元的4字节数据送到EAX。

3.寻址方式

- 例题：
  - MOV AX,[BX][DI]
  - 如:(DS)=2100H,　(BX)=0158H, (DI)=10A5H
  - 如：
    - 221FDH地址中的内容:2234H
    - 212FDH地址中的内容:1224H
    - 222FDH地址中的内容:1232H
  - 执行该指令后AX = ?H

3.寻址方式

- 例题：
  - MOV AX,[BX][DI]
  - 如:(DS)=2100H,　(BX)=0158H, (DI)=10A5H
  - 则有效地址EA=0158H + 10A5H = 11FDH
  - 物理地址=21000H + 11FD H= 221FDH
  - 221FDH地址中的内容:2234H
  - 执行该指令后AX = 2234H

3.寻址方式

- 下面指令中，目的操作数采用基址加变址寻址
- **MOV DS:[BP+SI],AL**

3.寻址方式

- 相对基址变址寻址
  - 操作数的地址为基址寄存器的内容、变址寄存器的内容和指令中的位移量之和。
    - 常用于对二维数组的寻址。
  - 例：MOV EAX 10H [EBX][ESI]
  - 或MOV EAX，[EBX+ESI+10H]
  - 或MOV EAX，10H [EBX+ESI]；
  - 将DS段中地址为EBX + ESI + 10H的存储单元的4字节数据送到EAX

3.寻址方式

- 例题:
  - MOV AX,[BX+DI-2H]
  - 假设，(DS) = 5000H, (BX) = 1223H, DI = 54H, (51274) = 43H (51275) = 54H, (51276) = 76H，
  - 物理地址=？ H
  - 执行该指令后 (AX) = ？ H

3.寻址方式

- 例题:
  - MOV AX,[BX+DI-2H]
  - 假设，(DS) = 5000H, (BX) = 1223H, DI = 54H, (51275) = 54H, (51276) = 76H
  - 物理地址= 50000 + 1223 + 0054 + FFFE(-2各位取反末位加一) = 51275H
  - 执行该指令后 (AX) = 7654H

3.寻址方式

- 比例变址寻址
  - 由指令中的变址寄存器的内容乘以比例因子再加上位移量得到操作的有效地址。
  - EA＝变址寄存器×比例因子＋位移量
  - 此寻址只有32位寻址一种情况。
  - 例：MOV EAX，1000H[ESI＊4]
  - DS段中地址为ESI所指向的内容乘4再加上1000H的内容形成有效地址。
  - 例： MOV [EDI*2+100H]，ECX；
  - 把ECX的内容存储到由EDI*2+100H寻址的DS段存储单元中。

3.寻址方式

- 基址比例变址寻址
  - 由指令中的变址寄存器的内容乘以比例因子，再加上基址寄存器的内容，得到操作的有效地址。
  - EA＝变址寄存器×比例因子＋[基址寄存器]
    - 注1. 此寻址方式只有32位寻址一种情况。
    - 注2.此寻址方式主要用于数组元素大小为2、4、8字节的二维数组操作。
  - 例：MOV EAX，[EBX+ECX*4]，也可写成MOV EAX，[EBX][ECX*4]；
  - 把由EBX+4*ECX之和寻址的DS段存储单元的4字节内容装入EAX。

3.寻址方式

- 相对基址比例变址寻址
  - 操作数的有效地址EA是变址寄存器的内容乘指令中的比例因子，加上基址寄存器的内容，再加上位移量之和。
  - EA＝变址寄存器×比例因子＋[基址寄存器]+位移量
    - 注1. 此寻址方式只有32位寻址一种情况。
    - 注2.此寻址方式主要用于数组元素大小为2、4、8字节，且数组起始地址不为0的二维数组操作。
  - 例：MOV EAX，[EBP+EDI*2+2]；
  - 把由EBP+2+EDI*2寻址的SS段存储单元的4字节内容装入EAX。
  - MOV ECX,10H[EDX*8][EAX]
  - MOV AX,10H[EBX*4][ESI]

3.寻址方式

格式总结

1、立即寻址 ： MOV AL，05H

2、寄存器寻址： MOV AL，BL

3、直接寻址： MOV AL，[2000H]

4、寄存器间接寻址：MOV AL，[SI]

5、基址寻址： MOV AL，[BX+3]

6、变址寻址： MOV AL，[SI +3]

7、基址加变址寻址：MOV AL，[BX+SI]

8、带位移的基址加变址寻址：MOV AL，[BX+SI+3]

9、比例变址寻址： MOV EAX，50[ESI＊4]

10、基址加比例变址寻址： MOV EAX，[ESI＊4][ECX]

11、带位移的基址加比例变址寻址： MOV EAX，[ESI＊4][ECX＋10]

只适合32位寻址

第二章 基础知识

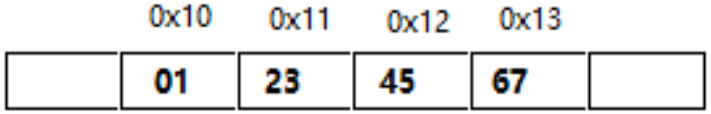
- 1.寄存器
- 2.x86指令集
- 3.寻址方式
- 4.字节序
- 5.栈
- 6.函数调用约定

4.字节序

- 在计算机存储多字节数据时，不同的CPU，字节的存放序列存在两种顺序
- 大端
  - Motorola的PowerPC系列CPU采用的大端序（big-endian）
- 小端
  - Intel的x86系列CPU使用的小端序（little-endian）
- “大端”、“小端”表示多字节数据的哪一端存储在该数据的起始地址

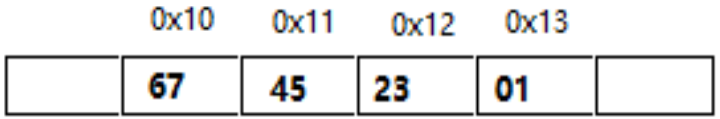
4.字节序

- 大端字节序
  - 字节数据的高位字节存放于内存的低地址端（内存的起始地址），低位字节存放于内存的高地址端，这是人们正常读写数值的方法。
  - 例：0x1234567以大端字节序存放于内存中的写法如下图



4.字节序

- 小端字节序
  - 字节数据的低位字节存放于内存的低地址端（内存的起始地址），高位字节存放于内存的高地址端
  - 例：0x1234567以小端字节序存放于内存中的写法如下图



4.字节序

- 大部分用户的操作系统（如windows, FreeBsd,Linux）是小端序的。
  - 因为数据计算是由低位开始的，计算机电路在处理计算过程中先处理低位字节会比先处理高位字节效率高，因此在计算机内部采用小端模式来保存数据。
  - 而在其他场合通常使用大端字节序，如网络传输和文件储存。

第二章 基础知识

- 1.寄存器
- 2.x86指令集
- 3.寻址方式
- 4.字节序
- 5.栈
- 6.函数调用约定

5.栈

栈

- 一种线性的数据结构
  - 先入后出，
    - 后入栈的数据会比之前入栈的数据先出栈。
  - 栈操作主要有两种——压栈和弹栈，
    - 每次压栈会把把数据从栈顶压入，栈的长度增加，栈顶位置上升；
    - 每次弹栈也是从栈顶弹出一个数据，栈的长度减少，栈顶位置下降。
  - 依次向栈中压入1、2、3三个数据，其弹出顺序为3、2、1。

5.栈

- 程序栈
- 栈的布局
- 栈的生成与销毁

5.栈

- 程序栈
  - 栈在程序中起着不可替代的作用
    - 每一个函数拥有自己的函数栈
    - 函数栈中保存着函数的局部变量、流控制结构等等
    - 每次一个函数调用后函数栈会被收回，并再次分配给其他被调用的函数
  - 程序栈是由高地址向低地址生长
    - 在程序中函数站的栈顶地址是比栈底地址低的

5.栈

程序栈

- x86框架为例，支持栈的寄存器包括ESP和EBP，以ESP和EBP作为栈的边界
- ESP是栈寄存器，指示当前栈顶的位置
  - 该寄存器在一个函数调用过程中会随着数据的压入弹出不断改变
- EBP是栈基址寄存器，这个寄存器指向栈底的位置
  - 主要作用是做栈划分
  - 在一个函数执行的过程中，该寄存器的值在调用其他函数时暂时改变为其他函数栈的栈底位置，当其他函数调用结束后会恢复为原函数栈栈底位置

5.栈

- 程序栈
  - x86指令集中，与栈操作相关的指令：push、pop、call、leave、ret
    - push和pop是栈的基本操作
    - call、leave、ret可以用来构建函数栈和销毁函数栈。

5.栈

- 栈的布局
  - 栈的生长是从高地址向低地址生长的，下图展示了栈在内存中是如何布局的。
  - 每一次函数的调用会生成一个新的栈帧，栈帧可供函数存储局部变量，或调用其他函数。
  - 当函数执行结束后，该函数栈帧会被释放，ESP、EBP会重新赋值为上一栈帧栈顶、栈底位置。
  - 栈帧在内存中会不断的分配、释放

栈的生成与销毁

- 当函数调用发生时，新的栈帧被压入栈中，而当函数返回时，栈帧将从栈中弹出。

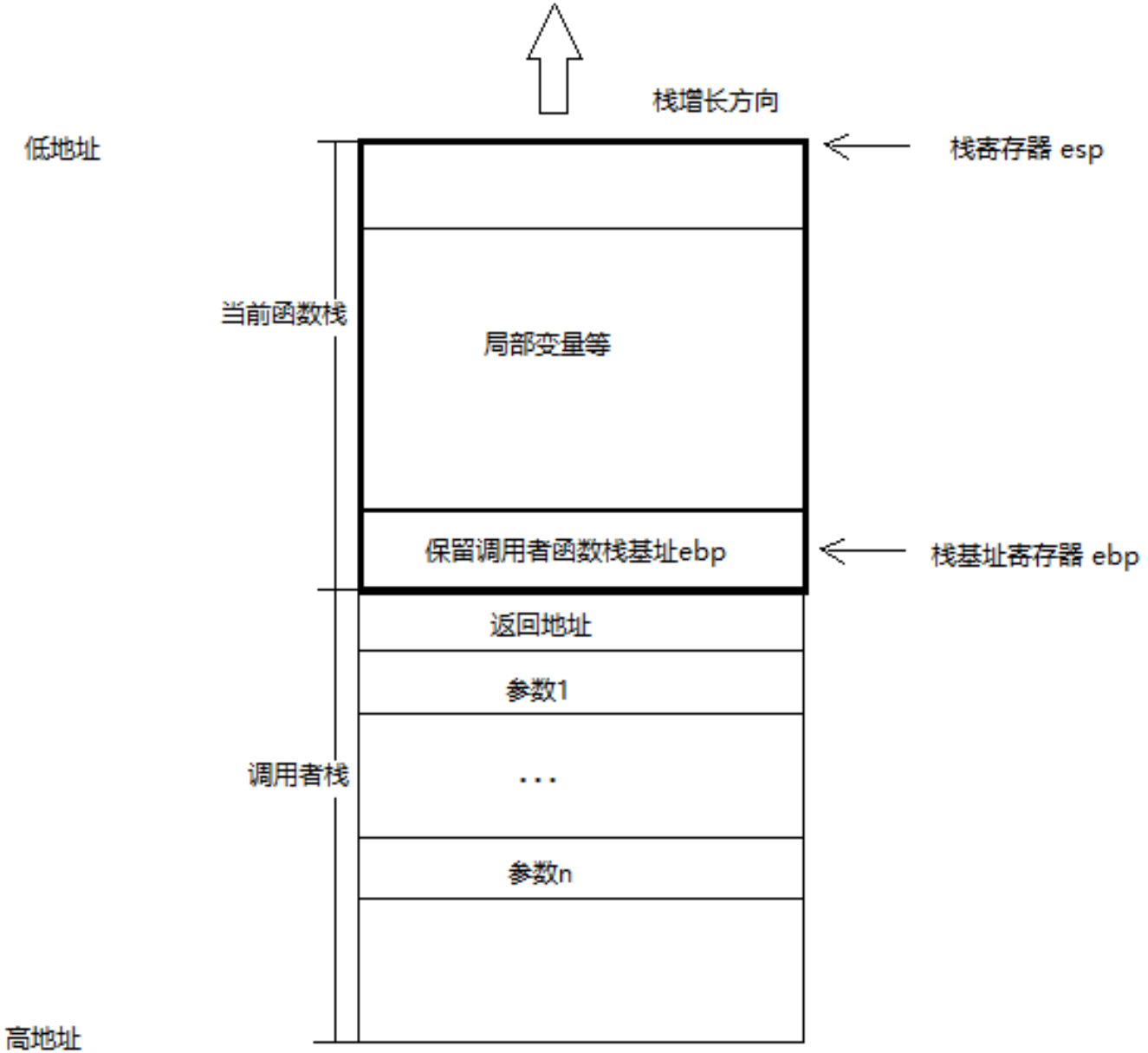
- 第一步：push 参数 参数入栈
  - 母函数调用子函数时，在母函数栈帧再往内存低地址方向的邻接区域，创建子函数的栈帧，首先把参数压入栈中；
- 第二步： call 子函数地址
  - 返回地址入栈，将指令寄存器eip中保存的下一条执行指令的地址做为返回母函数的返回地址压入栈，当子函数调用结束后返回时，程序应该按照返回地址跳转到母函数的下一条指令继续执行；
  - 代码区跳转，处理器从当前母函数的代码区跳转到子函数程序的入口，即程序的控制权转移到被调用的子函数；

栈的生成与销毁

- 第三步：push ebp ；
  - ebp中母函数栈帧基址指针入栈，子函数将基址寄存器ebp中保存的母函数帧栈的基地址指针压入栈中保存；
- 第四步：mov ebp esp ；
  - esp值装入ebp，ebp更新为新栈帧基地址。把esp中保存的最新栈顶指针拷贝到基址寄存器ebp中，这时ebp中保存的就是正在被调用子函数的基地址,即子函数的栈帧底部地址；
- 第五步：sub esp xxx ；
  - 给新栈帧分配空间，根据函数需要保存局部变量的空间大小，栈顶指针esp从子函数的基地址向内存低地址偏移，为局部变量留出一定空间

栈的生成与销毁

- 第六步：mov esp，ebp
  - 栈顶指针esp恢复到栈底位置，也就是将分配的栈空间全部释放，将分配的栈空间全部释放
- 第七步：pop ebp
  - 将ESP会指向的栈帧中母函数栈帧基址指针弹出保存在ebp
- 第八部：ret
  - 按照返回地址指向的位置，返回main函数下一步指令


栈增长方向
低地址
栈寄存器 esp
当前函数栈
局部变量等
保留调用者函数栈基址ebp
栈基址寄存器 ebp
返回地址
参数1
调用者栈
...
参数n
高地址

5.栈

- 栈的生成与销毁
  - 从一个简单的程序，从汇编语言层面观察栈帧的生成与销毁
  - 使用ida pro工具对编译好的可执行文件进行反汇编
  - 看一下关于main函数和foo函数的汇编代码

5.栈

```
#include<stdio.h>
Int foo(int a){
    printf("argv = %d",a);
    return0;
}
Int main(){
    int a;
    printf("hello stack \n");
    a=1;
    foo(a);
    return0;
}
```


5.栈

- 子函数foo的调用步骤如下：
  - 1.在.text 40109c处，执行了mov eax, [ebp+var_4]指令，ebp+var_4位置的偏移是变量a在栈帧中的位置，继续执行了push eax指令，导致a被压入栈中，ESP向低地址增长4字节。
  - 2.在.text 4010a0执行了call foo指令，call指令会将调用函数结束的返回地址，即函数中下一条指令add esp,4的地址0x4010a5压入栈中，程序进入foo函数中执行。

5.栈

- 子函数foo的调用步骤如下：
  - 3.进入foo函数后，首先执行的是push ebp指令，该指令的目的是保存上一函数栈的栈基址位置，便于函数结束后恢复上一函数栈。指令执行后，EBP中的数据被压入栈中，ESP向低地址增长4字节。
  - 4.在.text 401021执行了mov ebp，esp指令，该指令将EBP寄存器的值改写为当前ESP寄存器的值，导致栈底上移，新栈开始分配。
  - 5.在.text 401023执行了sub esp，40h的指令，该指令均导致ESP寄存器向低地址增长了64字节，也就是说栈的长度为64字节，新栈分配到此结束。

5.栈

- 子函数foo的调用步骤如下：
  - 6.从.text 401023至.text 401053为函数调用了printf函数，是函数正常的执行逻辑及编译器添加的检查函数。
  - 7.在.text 401058处开始执行了mov esp，ebp和pop ebp两条操作，在某些编译器中这两条也被整合成为leave指令。
    - 首先mov esp，ebp指令将栈顶指针esp恢复到栈底位置，也就是将分配的栈空间全部释放。
    - 再执行pop ebp指令，由之前的步骤可知，当前ESP寄存器指向空间的内容是main函数栈的栈基址位置，指令结束后，ESP会指向返回地址位置，EBP寄存器会恢复到main函数函数栈的栈底位置。

5.栈

- 子函数foo的调用步骤如下：
  - **8.在.text 40105b处执行了ret指令，该指令内容可以理解为pop eip，执行后程序跳转回main函数，并且栈帧也恢复回到main函数执行call foo之前的状态。**

5.栈

- foo函数的调用步骤
  - IDA的示例
  - OllyDbg的动态调试过程

5.栈

- foo函数的调用步骤如下：

```
.text:00401070 _main_0          proc near               ; CODE XREF: _main↑j
.text:00401070
.text:00401070 var_44           = byte ptr -44h
.text:00401070 var_4            = dword ptr -4
.text:00401070
.text:00401070                  push    ebp
.text:00401071                  mov     ebp, esp
.text:00401073                  sub     esp, 44h
.text:00401076                  push    ebx
.text:00401077                  push    esi
.text:00401078                  push    edi
.text:00401079                  lea     edi, [ebp+var_44]
.text:0040107C                  mov     ecx, 11h
.text:00401081                  mov     eax, 0CCCCCCCCh
.text:00401086                  rep stosd
.text:00401088                  push    offset aHelloStack ; "hello stack \n"
.text:0040108D                  call    _printf
.text:00401092                  add     esp, 4
.text:00401095                  mov     [ebp+var_4], 1
.text:0040109C                  mov     eax, [ebp+var_4]
.text:0040109F                  push    eax
.text:004010A0                  call    sub_401005
.text:004010A5                  add     esp, 4
.text:004010A8                  xor     eax, eax
.text:004010AA                  pop     edi
.text:004010AB                  pop     esi
.text:004010AC                  pop     ebx
.text:004010AD                  add     esp, 44h
.text:004010B0                  cmp     ebp, esp
.text:004010B2                  call    __chkesp
.text:004010B7                  mov     esp, ebp
.text:004010B9                  pop     ebp
.text:004010BA                  retn
.text:004010BA _main_0          endp
```


5.栈

- foo函数的调用步骤如下：

```
.text:00401020 ; Attributes: bp-based frame
.text:00401020
.text:00401020 sub_401020      proc near               ; CODE XREF: sub_401005↑j
.text:00401020
.text:00401020 var_40          = byte ptr -40h
.text:00401020 arg_0           = dword ptr  8
.text:00401020
.text:00401020                 push    ebp
.text:00401021                 mov     ebp, esp
.text:00401023                 sub     esp, 40h
.text:00401026                 push    ebx
.text:00401027                 push    esi
.text:00401028                 push    edi
.text:00401029                 lea     edi, [ebp+var_40]
.text:0040102C                 mov     ecx, 10h
.text:00401031                 mov     eax, 0CCCCCCCCh
.text:00401036                 rep stosd
.text:00401038                 mov     eax, [ebp+arg_0]
.text:0040103B                 push    eax
.text:0040103C                 push    offset aArgvD   ; "argv = %d"
.text:00401041                 call    _printf
.text:00401046                 add     esp, 8
.text:00401049                 xor     eax, eax
.text:0040104B                 pop     edi
.text:0040104C                 pop     esi
.text:0040104D                 pop     ebx
.text:0040104E                 add     esp, 40h
.text:00401051                 cmp     ebp, esp
.text:00401053                 call    __chkesp
.text:00401058                 mov     esp, ebp
.text:0040105A                 pop     ebp
.text:0040105B                 retn
.text:0040105B sub_401020      endp
```

5.栈

- foo函数的调用步骤如下：
  - OllyDbg的动态调试过程

第二章 基础知识

- 1.寄存器
- 2.x86指令集
- 3.寻址方式
- 4.字节序
- 5.栈
- 6.函数调用约定

6.函数调用约定

- 函数调用约定
  - 对函数调用时如何传递参数的一种约定
    - 例如：使用VC++ 6.0对源代码进行编译，在函数定义时可以选择函数的调用约定
  - 主要解决的是问题就是函数的参数是如何传递的，以及函数执行结束后，参数应该如何处理
  - 主要的函数约定由三种
    - **Cdecl**
    - **Stdcall**
    - **Fastcall**

6.函数调用约定

```
#include<stdio.h>
void __cdecl foo_cdecl(int a,int b,int c){
      printf("argv %d %d %d \n",a,b,c);
}
void __stdcall foo_stdcall(int a,int b,int c){
      printf("argv %d %d %d \n",a,b,c);
}
void __fastcall foo_fastcall(int a,int b,int c,int d){
      printf("argv %d %d %d %d \n",a,b,c,d);
}
intmain(){
      foo_cdecl(1,2,3);
      foo_stdcall(1,2,3);
      foo_fastcall(1,2,3,4);
      return0;
}
```


6.函数调用约定

- cdcel是C/C++默认方式，参数从右向左入栈

```
.text:00401150                push    ebp
.text:00401151                mov     ebp, esp
.text:00401153                sub     esp, 40h
.text:00401156                push    ebx
.text:00401157                push    esi
.text:00401158                push    edi
.text:00401159                lea     edi, [ebp+var_40]
.text:0040115C                mov     ecx, 10h
.text:00401161                mov     eax, 0CCCCCCCCh
.text:00401166                rep stosd
.text:00401168                push    3
.text:0040116A                push    2
.text:0040116C                push    1
.text:0040116E                call    j__foo_cdecl
.text:00401173                add     esp, 0Ch
.text:00401176                push    3
.text:00401178                push    2
.text:0040117A                push    1
.text:0040117C                call    j__foo_stdcall@12 ; foo_stdcall(x,x,x)
.text:00401181                push    4
.text:00401183                push    3
.text:00401185                mov     edx, 2
.text:0040118A                mov     ecx, 1
.text:0040118F                call    j_@foo_fastcall@12 ; foo_fastcall(x,x,x)
.text:00401194                xor     eax, eax
.text:00401196                pop     edi
.text:00401197                pop     esi
.text:00401198                pop     ebx
```

- 查看main函数汇编代码

6.函数调用约定

- 参数入栈顺序
  - 参数是被压入栈中传递的，并且压入栈中的顺序是从右向左，即从最后一个参数开始压栈
  - 此时相比未压栈时，**ESP指针向减小了3*4=12（0x0c）。**
- 参数清除
  - 当执行过**callfoo_cdecl**后，**ESP**指针是会恢复到压入参数后的位置，下一条执行**add esp，0x0c**就会把**ESP**指针增加**12**字节，也就恢复到了压入参数之前的位置，从而实现了参数清除

6.函数调用约定

```
.text:00401030 arg_0           = dword ptr  8
.text:00401030 arg_4           = dword ptr  0Ch
.text:00401030 arg_8           = dword ptr  10h
.text:00401030
.text:00401030                 push    ebp
.text:00401031                 mov     ebp, esp
.text:00401033                 sub     esp, 40h
.text:00401036                 push    ebx
.text:00401037                 push    esi
.text:00401038                 push    edi
.text:00401039                 lea     edi, [ebp+var_40]
.text:0040103C                 mov     ecx, 10h
.text:00401041                 mov     eax, 0CCCCCCCCh
.text:00401046                 rep stosd
.text:00401048                 mov     eax, [ebp+arg_8]
.text:0040104B                 push    eax
.text:0040104C                 mov     ecx, [ebp+arg_4]
.text:0040104F                 push    ecx
.text:00401050                 mov     edx, [ebp+arg_0]
.text:00401053                 push    edx
.text:00401054                 push    offset Format   ; "argv %d %d %d \n"
.text:00401059                 call    _printf
.text:0040105E                 add     esp, 10h
.text:00401061                 pop     edi
.text:00401062                 pop     esi
.text:00401063                 pop     ebx
.text:00401064                 add     esp, 40h
.text:00401067                 cmp     ebp, esp
.text:00401069                 call    __chkesp
.text:0040106E                 mov     esp, ebp
.text:00401070                 pop     ebp
.text:00401071                 retn
.text:00401071 _foo_cdecl      endp
```

foo_cdecl反汇编代码中

6.函数调用约定

- 在foo_cdecl反汇编代码中，可以看到函数是利用的是EBP寄存器与参数位置的相对关系顺序传递参数的
- 参数自右向左传递的目的也是保证生成汇编语言时,这些参数相对于EBP指向的栈位置的偏移量是固定的。

6.函数调用约定

- stdcall调用者未对传入参数进行清理
  - stdcall是windows API默认方式，参数从右向左入栈
  - 通过对比在main函数foo_stdcall和foo_cdecl两种的函数调用代码，发现调用foo_stdcall相比foo_cdecl，缺少了add esp , 0ch指令，也就是说调用者未对传入参数进行清理

6.函数调用约定

```
.text:00401150                push    ebp
.text:00401151                mov     ebp, esp
.text:00401153                sub     esp, 40h
.text:00401156                push    ebx
.text:00401157                push    esi
.text:00401158                push    edi
.text:00401159                lea     edi, [ebp+var_40]
.text:0040115C                mov     ecx, 10h
.text:00401161                mov     eax, 0CCCCCCCCh
.text:00401166                rep stosd
.text:00401168                push    3
.text:0040116A                push    2
.text:0040116C                push    1
.text:0040116E                call    j__foo_cdecl
.text:00401173                add     esp, 0Ch
.text:00401176                push    3
.text:00401178                push    2
.text:0040117A                push    1
.text:0040117C                call    j__foo_stdcall@12 ; foo_stdcall(x,x,x)
.text:00401181                push    4
.text:00401183                push    3
.text:00401185                mov     edx, 2
.text:0040118A                mov     ecx, 1
.text:0040118F                call    j_@foo_fastcall@12 ; foo_fastcall(x,x,x)
.text:00401194                xor     eax, eax
.text:00401196                pop     edi
.text:00401197                pop     esi
.text:00401198                pop     ebx
```

- 查看main函数汇编代码

6.函数调用约定

foo_stdcall反汇编代码

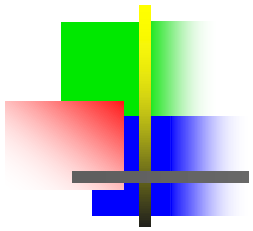
```
.text:00401090                 push    ebp
.text:00401091                 mov     ebp, esp
.text:00401093                 sub     esp, 40h
.text:00401096                 push    ebx
.text:00401097                 push    esi
.text:00401098                 push    edi
.text:00401099                 lea     edi, [ebp+var_40]
.text:0040109C                 mov     ecx, 10h
.text:004010A1                 mov     eax, 0CCCCCCCCh
.text:004010A6                 rep stosd
.text:004010A8                 mov     eax, [ebp+arg_8]
.text:004010AB                 push    eax
.text:004010AC                 mov     ecx, [ebp+arg_4]
.text:004010AF                 push    ecx
.text:004010B0                 mov     edx, [ebp+arg_0]
.text:004010B3                 push    edx
.text:004010B4                 push    offset Format   ; "argv %d %d %d \n"
.text:004010B9                 call    _printf
.text:004010BE                 add     esp, 10h
.text:004010C1                 pop     edi
.text:004010C2                 pop     esi
.text:004010C3                 pop     ebx
.text:004010C4                 add     esp, 40h
.text:004010C7                 cmp     ebp, esp
.text:004010C9                 call    __chkesp
.text:004010CE                 mov     esp, ebp
.text:004010D0                 pop     ebp
.text:004010D1                 retn    0Ch
.text:004010D1 _foo_stdcall@12 endp
```

6.函数调用约定

- foo_stdcall采用的是retn 0ch的指令，而非retn。
  - 该指令的作用是retn + pop 0x0c字节，即返回后使ESP增加12个字节，这与foo_cdecl的指令执行是一致的。
  - stdcall方式的优点在于，被调用者函数内部存在清理参数代码，与调用函数后再执行add esp，xxx相比，代码尺寸要小，是Win 32 API库使用的函数调用约定。

6.函数调用约定

- fastcall方式
  - 与stdcall方式基本类似，参数的清理也由被调用函数来负责
  - 但该方式通常会使用寄存器（而非栈内存）去传递参数。
    - 若函数需要传递多个参数，会优先使用ECX和EDX来传递后两个参数，其余的参数再从右向左由栈传入。



```
.text:00401150
.text:00401150                  push    ebp
.text:00401151                  mov     ebp, esp
.text:00401153                  sub     esp, 40h
.text:00401156                  push    ebx
.text:00401157                  push    esi
.text:00401158                  push    edi
.text:00401159                  lea     edi, [ebp+var_40]
.text:0040115C                  mov     ecx, 10h
.text:00401161                  mov     eax, 0CCCCCCCCh
.text:00401166                  rep stosd
.text:00401168                  push    3
.text:0040116A                  push    2
.text:0040116C                  push    1
.text:0040116E                  call    sub_401005
.text:00401173                  add     esp, 0Ch
.text:00401176                  push    3
.text:00401178                  push    2
.text:0040117A                  push    1
.text:0040117C                  call    sub_40100F
.text:00401181                  push    4
.text:00401183                  push    3
.text:00401185                  mov     edx, 2
.text:0040118A                  mov     ecx, 1
.text:0040118F                  call    sub_401019
.text:00401194                  xor     eax, eax
.text:00401196                  pop     edi
.text:00401197                  pop     esi
.text:00401198                  pop     ebx
.text:00401199                  add     esp, 40h
.text:0040119C                  cmp     ebp, esp
.text:0040119E                  call    sub_401240
.text:004011A3                  mov     esp, ebp
.text:004011A5                  pop     ebp
.text:004011A6                  retn
.text:004011A6 sub_401150       endp
.text:004011A6
```


6.函数调用约定

- fastcall方式是很快的
  - 由于CPU对寄存器的访问要快于对栈所在内存的访问速度，因此从函数调用本身来看，fastcall方式是很快的
  - 但有时需要额外的系统开销来管理寄存器，如在调用函数前ECX、EDX中存有重要数据，那么需要先进行备份
  - 此外，如果函数本身需要使用这两个寄存器，也需要将参数迁移到其他位置进行使用。

Q & A