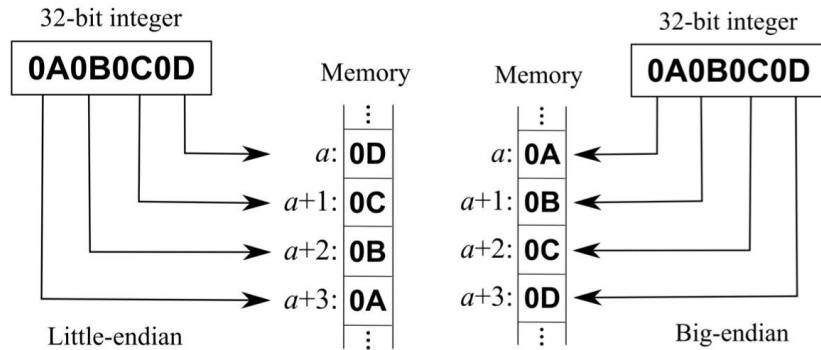


80x86是一种Intel出的微处理器架构

80x86简介 (2)

• 特点

- 采用复杂指令集
- 小端存储



1.什么是复杂指令集？什么是精简指令集？80x86采用的是哪种？

复杂指令集，也称为CISC指令集，英文名是CISC，(Complex Instruction Set Computing的缩写)。在CISC微处理器中，程序的各条指令是按顺序串行执行的，每条指令中的各个操作也是按顺序串行执行的。顺序执行的优点是控制简单，但计算机各部分的利用率不高，执行速度慢。

精简指令集具备以下特点：

第一、统一采用单周期指令，简单理解就是指令周期数长短统一。

第二、采用高效的流水线操作，让指令在流水线中实现并行处理，大幅度提高CPU处理效率。

所谓指令流水线是为了提高处理器执行指令的效率，把一条指令的操作分成多个细小的步骤，每个步骤由专门的电路完成的方式。

例如，一条指令要执行三个步骤：取指令、译码、执行指令，每个步骤都要花费一个机器时间周期。如果没有流水线技术，那这套指令完成下来就是三个机器周期。如果用流水线，那么当这条指令完成“取指令”后进入“译码”的同时，下一条指令就可以进行“取指令”了。

二者比较

比较内容	CISC	RISC
指令系统	复杂，庞大	简单，精简
指令数目	一般大于200	一般小于100
指令格式	一般大于4	一般小于4
寻址方式	一般大于4	一般小于4
指令字长	不固定	等长
可访存指令	不加限制	只有LOAD/STORE指令
各种指令使用频率	相差很大	相差不大

比较内容	CISC	RISC
各种指令执行时间	相差很大	绝大多数在一个周期内完成
优化编译实现	很难	较容易
程序源代码长度	较短	较长
控制器实现方式	绝大多数为微程序控制	绝大多数为硬布线控制
软件系统开发时间	较短	较长

2.什么是小端存储？什么是大端存储？80x86采用的是哪种？

1). 大端存储：大端模式，是指数据的高字节保存在内存的低地址中，而数据的低字节保存在内存的高地址中，这样的存储模式有点儿类似于把数据当作字符串顺序处理：地址由小向大增加，而数据从高位往低位放。

2). 小端存储：小端模式，是指数据的高字节保存在内存的高地址中，而数据的低字节保存在内存的低地址中，这种存储模式将地址的高低和数据位权有效地结合起来，高地址部分权值高，低地址部分权值低，和我们的逻辑方法一致。

小端：较高的有效字节存放在较高的存储器地址，较低的有效字节存放在较低的存储器地址。

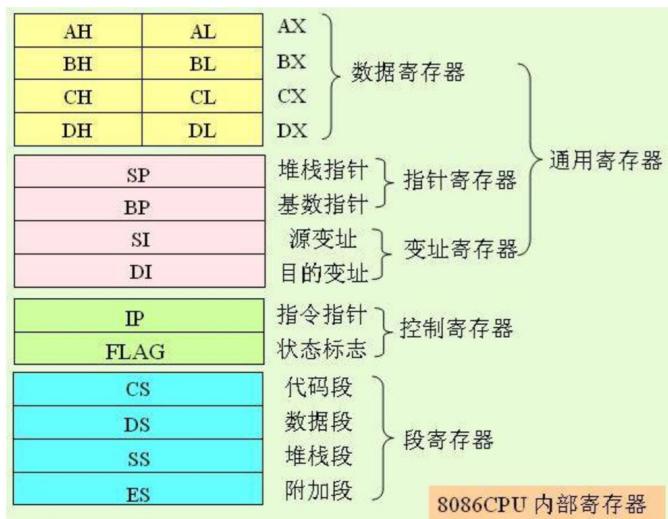
大端：较高的有效字节存放在较低的存储器地址，较低的有效字节存放在较高的存储器地址。

8086是小端模式

3.8086有哪5类寄存器？请分别举例说明其作用。

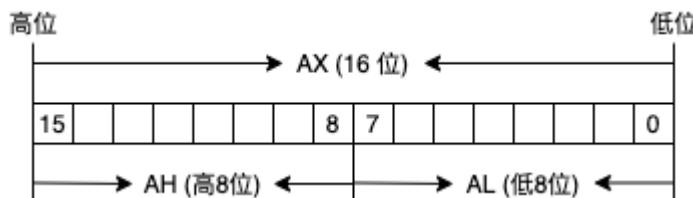
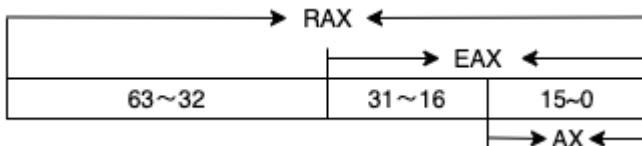
8086的寄存器

- **SP:** 堆栈指针，与SS配合使用，指向目前的堆栈位置
- **BP:** 基址指针寄存器，可用作SS的一个相对基址位置
- **SI:** 源变址寄存器，可用来存放相对于DS段的源变址指针
- **DI:** 目的变址寄存器，可用来存放相对于ES段的目的变址指针



数据寄存器、指针寄存器、变址寄存器、控制寄存器、段寄存器（前三个合称为通用寄存器）

(1) 数据寄存器



AX：累加寄存器 (Accumulator) 通常用于保存算中间值的操作数，也是与 I/O 设备交互时与外设传输数据的寄存器

```
mov eax,[length]
inc eax
mov [length],eax
```

BX：基址寄存器 (Base) 通常用于内存寻址时保存地址基址的寄存器，可以配合 DI、SI 提供更复杂的寻址模式

```
mov ebx,num  
mov eax,[i]  
mov ecx,[ebx+eax]
```

CX : 计数寄存器 (Counter) 通常用于保存循环次数(loop 指令会用到) , 也可用于保存用于算数运算、位运算的参数等(乘数、位移数等)

```
mov ecx,0  
loop:  
    cmp ecx,5  
    jge endLoop  
    .....  
    inc ecx  
  
endLoop:
```

DX : 数据寄存器 (Data) 通常就是用于存储数据的 , 偶尔在大数字的乘除运算时搭配 AX 形成一个操作数

```
mov eax,2 (a1中存的是2)  
mov edx,3 (d1中存的是3)  
mul d1 (结果存在a1中)
```

4.有哪些段寄存器 , 它们的作用是什么 ?

CS 指令段寄存器 (Code)

用于保存当前执行程序的指令段(code segment)的起始地址 , 相当于 section .text 的地址

DS 数据段寄存器 (Data)

用于保存当前执行程序的数据段(data segment)的起始地址，相当于 section .data 的地址

SS 栈寄存器 (Stack)

用于保存当前栈空间(Stack)的基址，与 SP(偏移量) 相加 -> SS:SP 可找到当前栈顶地址

ES 额外段寄存器 (Extra)

常用于字符串操作的内存寻址基址，与变址寄存器 DI 共用

FS、GS 指令段寄存器

80386 额外定义的段寄存器，提供程序员更多的段地址选择

5.什么是寻址？8086有哪些寻址方式？

寻址的目的是找到指令中操作数所在的地址

大多数汇编语言指令的要求，要被处理的操作数。提供一个操作数地址要被处理的数据被存储的位置。某些指令不需要操作数，而其他一些指令可能需要一个，两个或三个操作数。当一个指令需要两个操作数，第一个操作数是一般的目的，其中包含在一个寄存器或内存位置和第二个操作数源数据。源码包含要传送的数据（立即寻址）或（寄存器或存储器）中的数据的地址。一般来说，操作后的源数据将保持不变。（位置）

寻址的三种基本模式是：

寄存器寻址

立即寻址

存储器寻址

8086的寻址方式 (1)

- 寻址
 - 找到操作数的地址(从而能够取出操作数)
- 8086的寻址方式
 - 立即寻址、直接寻址
 - 寄存器寻址、寄存器间接寻址、寄存器相对寻址
 - 基址加变址、相对基址加变址

8086的寻址方式 (2)

- 立即寻址
 - MOV AX 1234H
 - 直接给出了操作数，事实上没有“寻址”
- 直接寻址
 - MOV AX [1234H]
 - 直接给出了地址1234H，用[]符号取数

8086的寻址方式 (3)

- 寄存器寻址
 - MOV AX BX
 - 操作数在寄存器里，给出寄存器名即可取走操作数
- 寄存器间接寻址
 - MOV AX [BX]
 - 操作数有效地址在寄存器之中(SI、DI、BX、BP)
- 寄存器相对寻址
 - MOV AX [SI+3]

8086的寻址方式 (4)

- 基址加变址
 - MOV AX [BX+DI]
 - 把一个基址寄存器(BX、BP)的内容，加上变址寄存器(SI、DI)的内容。
- 相对基址加变址
 - MOV AX [BX+DI+3]

6.什么是直接寻址？直接寻址的缺点是什么？

art.org

- 存储器寻址

寄存器寻址

在这种寻址方式中，寄存器包含操作数。根据不同的指令，寄存器可能是第一个操作数，第二个操作数或两者兼而有之。

例如，

1. MOV DX, TAX_RATE ; Register in first operand
2. MOV COUNT, CX ; Register in second operand
3. MOV EAX, EBX ; Both the operands are in registers

随着处理数据寄存器之间不涉及内存，它提供数据的处理速度是最快的。

立即寻址

立即数有一个恒定的值或表达式。当一个指令有两个操作数使用立即寻址，第一个操作数是寄存器或内存中的位置，和第二个操作数是立即数。第一个操作数定义的数据的长度。

例如，

1. BYTE_VALUE DB 150; A byte value is defined
2. WORD_VALUE DW 300; A word value is defined
3. ADD BYTE_VALUE, 65; An immediate operand 65 is added
4. MOV AX, 45H ; Immediate constant 45H is transferred to AX

直接存储器寻址

当操作数指定内存寻址模式，直接访问主存储器的数据段，通常是必需的。这种方式处理的数据的处理速度较慢的结果。为了找到确切的位置在内存中的数据，我们需要段的起始地址，这是通常出现在DS寄存器和偏移值。这个偏移值也被称为有效的地址。

在直接寻址方式，是直接指定的偏移值作为指令的一部分，通常由变量名表示。汇编程序计算的偏移值，并维护一个符号表，它存储在程序中使用的所有变量的偏移值。

在直接存储器寻址，其中一个操作数是指一个内存位置和另一个操作数引用一个寄存器。

例如，

1. ADD BYTE_VALUE, DL ; Adds the register in the memory location
2. MOV BX, WORD_VALUE ; Operand from the memory is added to register

直接偏移量寻址

这种寻址模式使用算术运算符修改一个地址。例如，看看下面的定义来定义数据表：

1. BYTE_TABLE DB 14, 15, 22, 45 ; Tables of bytes
2. WORD_TABLE DW 134, 345, 564, 123 ; Tables of words

可以进行以下操作：从存储器到寄存器中的表访问数据：

1. MOV CL, BYTE_TABLE[2] ; Gets the 3rd element of the BYTE_TABLE
2. MOV CL, BYTE_TABLE + 2 ; Gets the 3rd element of the BYTE_TABLE
3. MOV CX, WORD_TABLE[3] ; Gets the 4th element of the WORD_TABLE
4. MOV CX, WORD_TABLE + 3 ; Gets the 4th element of the WORD_TABLE

间接寻址

- CEF浏览器
- CENTO
- C语言教程
- dedecms
- Docker
- gcc命令行
- Go语言
- H264学习
- Java教程
- jQuery教程
- JSON教程
- Libuv教程
- Linux
- 技术提问/建议
- Linux驱动
- Lua教程
- Makefile
- Memcached
- MongoDB
- MySQL
- Nasm教程
- Nginx配置
- Objectivec
- OsgEAR
- Perl
- PHP教程
- Protocol
- Python
- QT学习
- Redis教程
- Ruby教程
- SQLite
- SQL教程
- UNIX/LINN
- VBScript
- VueJS教程
- Windows
- XPath教程
- 关于本站
- 批处理脚本
- 技术笔迹
- 正则表达式
- 设计模式
- 高质量代码

这种寻址模式利用计算机的能力分部：偏移寻址。一般基寄存器EBX, EBP (BX, BP) 和索引寄存器 (DI, SI) , 编码的方括号内的内存引用，用于此目的。

通常用于含有几个元素的类似，数组变量间接寻址。存储在数组的起始地址是EBX寄存器。

下面的代码片段显示了如何访问不同元素的变量。

```

1. MY_TABLE TIMES 10 DW 0 ; Allocates 10 words (2 bytes) each initialized to 0
2. MOV EBX, [MY_TABLE] ; Effective Address of MY_TABLE in EBX
3. MOV [EBX], 110 ; MY_TABLE[0] = 110
4. ADD EBX, 2 ; EBX = EBX + 2
5. MOV [EBX], 123 ; MY_TABLE[1] = 123

```

MOV指令

我们已经使用MOV指令是用于将数据从一个存储空间移动到另一个。 MOV指令需要两个操作数。

语法:

MOV指令的语法是：

```
1. MOV destination, source
```

MOV指令可以具有以下五种形式之一：

```

1. MOV register, register
2. MOV register, immediate
3. MOV memory, immediate
4. MOV register, memory
5. MOV memory, register

```

请注意：

- MOV操作操作数应该是同样大小
- 源操作数的值保持不变

MOV指令产生引起歧义次数。例如，下面语句：

```

1. MOV EBX, [MY_TABLE] ; Effective Address of MY_TABLE in EBX
2. MOV [EBX], 110 ; MY_TABLE[0] = 110

```

目前尚不清楚是否要移动相当于一个字节或字相当于数字110。在这种情况下，比较好的做法是使用类型说明符。

下表列出了一些常见的类型说明符：

Type Specifier	Bytes addressed
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10

例子:

下面的程序说明，上面讨论的一些概念。它存储的名称‘ Zara Ali’ 在数据段的内存。然后其值更改为另一个名字‘ Nuha Ali’ 编程和显示名称。

```

1. section .text
2. global start:must be declared for linker (ld)

```

在线运行

```
-->----->----->----->----->----->
3. _start:;tell linker entry bytekits
4.
5. ;writing the name 'Zara Ali'
6.     mov    edx,9    ;message length
7.     mov    ecx, name ;message to write
8.     mov    ebx,1    ;file descriptor (stdout)
9.     mov    eax,4    ;system call number (sys_write)
10.    int   0x80;call kernel
11.
12.    mov    [name], dword 'Nuha'; Changed the name to Nuha Ali
13. ;writing the name 'Nuha Ali'
14.    mov    edx,8    ;message length
15.    mov    ecx, name;message to write
16.    mov    ebx,1    ;file descriptor (stdout)
17.    mov    eax,4    ;system call number (sys_write)
18.    int   0x80;call kernel
19.    mov    eax,1    ;system call number (sys_exit)
20.    int   0x80;call kernel
21.
22. section .data
23. name db 'Zara Ali '
```

上面的代码编译和执行时，它会产生以下结果：

```
1. Zara Ali Nuha Ali
```



7. 主程序与子程序之间如何传递参数？你的实验代码中在哪里体现的？

如何进行函数传参

- 利用寄存器传递参数
 - 缺点：能传递的参数有限，因为寄存器有限
- 利用约定的地址传递参数
- 利用堆栈传递参数（常用）

一般是用寄存器和指定内存地址存放参数

下面是一个例子(打印数组的例子)

```
printArr:  
    ; 打印一个数组 ( 数组每一位是4字节 ) · 长度放在length中 · 数组起始放在arrPointer中  
    ;mov dword[length],5  
    ;mov dword[i],0  
loop:  
    ;mov eax,[length]  
    ;sub eax,[i]  
    ;cmp eax,0  
    ;jle endLoop  
    ;mov ecx,[i]  
    ;mov edx,[arrPointer]  
    ;mov eax,[edx+4*ecx]  
    ;add eax,'0'  
    ;mov [num],eax  
    ;mov ecx,num  
    ;mov eax,4  
    ;mov ebx,1  
    ;mov edx,4 ;注意 · 这里一次性要输出四字节(数组的一位))
```

```
int 80h
mov eax,[i]
inc eax
mov [i],eax ;自增1
jmp loop
endLoop:
ret
```

8.如何处理输入和输出？你的代码中在哪里体现的？

调用系统进程(int 80h)

参数：把EAX寄存器中的系统调用号；在寄存器存储的参数的系统调用 EBX, ECX等；调用相关的中断 (80h)

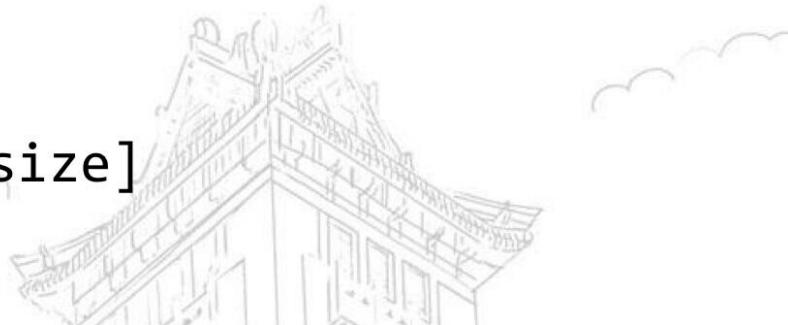
Write System Call

- mov eax, 4
 mov ebx, 1
 mov ecx, msg1
 mov edx, size1
 int 80h



Read System Call

- mov eax, 3
mov ebx, 0
mov ecx, var
mov edx, dword[size]
int 80h



下面是提示输入的打印语句和一个字节一个字节读入的代码

```
;prompt for X
mov eax,4
mov ebx,1
mov ecx,prompt
mov edx,7
int 80h

;这里要一个字符一个字符的输入·否则不知道什么时候结束 唉.....
mov dword[i],0
loopInput: ;注意！！！ 结尾一定要加一个0xa ·否则之后处理中不知道字符串什么时候结束
    mov eax,message
    add eax,[i] ;计算出下一个字符放在哪里
    mov [num],eax
    mov eax,3
    mov ebx,0
    mov ecx,[num]
    mov edx,1
    int 80h

    ;如果输入的是回车·则结束循环
    mov eax,[num] ;此时eax中是该字符的地址
    mov ebx,[eax]
    cmp ebx,0xa
    je endLoopInput

    mov eax,[i]
    inc eax
    mov [i],eax
    jmp loopInput

endLoopInput:
```

9.通过什么寄存器保存前一次的运算结果？你的代码中在哪里体现的？

(1) eax, ebx, ecx, edx

计算Y的长度部分（前面）。这里把计算的字符串总长度放在eax中了，然后再把eax暂存到[lenY]中

```
;calculate the length of Y
    mov [lenY],eax
    mov edx,[lenX]
    sub eax,edx
    dec eax
    mov [lenY],eax
```

(2) 内存段

例子在加法处理进位的部分

arrPointer中存放的是指向数组当前元素的指针

```
mov eax,resultArr
mov [arrPointer],eax ;此时arrayPointer指向的是结果数组的起始位置
mov dword[i],0
```

10.请分别简述 MOV 指令和 LEA 指令的用法和作用。

- 存储器寻址

寄存器寻址

在这种寻址方式中，寄存器包含操作数。根据不同的指令，寄存器可能是第一个操作数，第二个操作数或两者兼而有之。

例如，

1. MOV DX, TAX_RATE ; Register in first operand
2. MOV COUNT, CX ; Register in second operand
3. MOV EAX, EBX ; Both the operands are in registers

随着处理数据寄存器之间不涉及内存，它提供数据的处理速度是最快的。

立即寻址

立即数有一个恒定的值或表达式。当一个指令有两个操作数使用立即寻址，第一个操作数是寄存器或内存中的位置，和第二个操作数是立即数。第一个操作数定义的数据的长度。

例如，

1. BYTE_VALUE DB 150; A byte value is defined
2. WORD_VALUE DW 300; A word value is defined
3. ADD BYTE_VALUE, 65; An immediate operand 65 is added
4. MOV AX, 45H ; Immediate constant 45H is transferred to AX

直接存储器寻址

当操作数指定内存寻址模式，直接访问主存储器的数据段，通常是必需的。这种方式处理的数据的处理速度较慢的结果。为了找到确切的位置在内存中的数据，我们需要段的起始地址，这是通常出现在DS寄存器和偏移值。这个偏移值也被称为有效的地址。

在直接寻址方式，是直接指定的偏移值作为指令的一部分，通常由变量名表示。汇编程序计算的偏移值，并维护一个符号表，它存储在程序中使用的所有变量的偏移值。

在直接存储器寻址，其中一个操作数是指一个内存位置和另一个操作数引用一个寄存器。

例如，

1. ADD BYTE_VALUE, DL ; Adds the register in the memory location
2. MOV BX, WORD_VALUE ; Operand from the memory is added to register

直接偏移量寻址

这种寻址模式使用算术运算符修改一个地址。例如，看看下面的定义来定义数据表：

1. BYTE_TABLE DB 14, 15, 22, 45 ; Tables of bytes
2. WORD_TABLE DW 134, 345, 564, 123 ; Tables of words

可以进行以下操作：从存储器到寄存器中的表访问数据：

1. MOV CL, BYTE_TABLE[2] ; Gets the 3rd element of the BYTE_TABLE
2. MOV CL, BYTE_TABLE + 2 ; Gets the 3rd element of the BYTE_TABLE
3. MOV CX, WORD_TABLE[3] ; Gets the 4th element of the WORD_TABLE
4. MOV CX, WORD_TABLE + 3 ; Gets the 4th element of the WORD_TABLE

间接寻址

这种寻址模式利用计算机的能力分部：偏移寻址。一般基寄存器EBX, EBP (BX, BP) 和索引寄存器 (DI, SI)，编码的方括号内的内存引用，用于此目的。

CEF浏览
CENTO
C语言教
dedecn
Docker
gcc命令
Go语言
H264学
Java教科
jQuery教
JSON教
Libuv教
Linux
技术提问/建议
Linux驱动
Lua教程
Makefile
Memcached
MongoDB
MySQL
Nasm教程
Nginx配置
Objectivec
OsgEngine
Perl
PHP教程
Protocol
Python
QT学习
Redis教程
Ruby教程
SQLite
SQL教程
UNIX/LINN
VBScript
Vue.js教程
Windows
XPath教程
关于本站
批处理脚本
技术笔谈
正则表达式
设计模式
高质量代码

通常用于含有几个元素的类似，数组变量间接寻址。存储在数组的起始地址是EBX寄存器。

下面的代码片段显示了如何访问不同元素的变量。

```

1. MY_TABLE TIMES 10 DW 0 ; Allocates 10 words (2 bytes) each initialized to 0
2. MOV EBX, [MY_TABLE] ; Effective Address of MY_TABLE in EBX
3. MOV [EBX], 110 ; MY_TABLE[0] = 110
4. ADD EBX, 2 ; EBX = EBX +2
5. MOV [EBX], 123 ; MY_TABLE[1] = 123

```

MOV指令

我们已经使用MOV指令是用于将数据从一个存储空间移动到另一个。MOV指令需要两个操作数。

语法:

MOV指令的语法是：

```
1. MOV destination, source
```

MOV指令可以具有以下五种形式之一：

```

1. MOV register, register
2. MOV register, immediate
3. MOV memory, immediate
4. MOV register, memory
5. MOV memory, register

```

请注意：

- MOV操作操作数应该是同样大小
- 源操作数的值保持不变

MOV指令产生引起歧义次数。例如，下面语句：

```

1. MOV EBX, [MY_TABLE] ; Effective Address of MY_TABLE in EBX
2. MOV [EBX], 110 ; MY_TABLE[0] = 110

```

目前尚不清楚是否要移动相当于一个字节或字相当于数字110。在这种情况下，比较好的做法是使用类型说明符。

下表列出了一些常见的类型说明符：

Type Specifier	Bytes addressed
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10

例子:

下面的程序说明，上面讨论的一些概念。它存储的名称‘Zara Ali’ 在数据段的内存。然后其值更改为另一个名字‘Nuha Ali’ 编程和显示名称。

```

1. section .text
2. global _start;must be declared for linker (ld)
3. _start:;tell linker entry bytekits
4.

```

在线运行

```
5. ;writing the name 'Zara Ali'
6.     mov    edx,9    ;message length
7.     mov    ecx, name   ;message to write
8.     mov    ebx,1    ;file descriptor (stdout)
9.     mov    eax,4    ;system call number (sys_write)
10.    int   0x80;call kernel
11.
12.    mov    [name], dword 'Nuha'; Changed the name to Nuha Ali
13. ;writing the name 'Nuha Ali'
14.    mov    edx,8    ;message length
15.    mov    ecx, name;message to write
16.    mov    ebx,1    ;file descriptor (stdout)
17.    mov    eax,4    ;system call number (sys_write)
18.    int   0x80;call kernel
19.    mov    eax,1    ;system call number (sys_exit)
20.    int   0x80;call kernel
21.
22. section .data
23. name db 'Zara Ali '
```

上面的代码编译和执行时，它会产生以下结果：

```
1. Zara Ali Nuha Ali
```



```
mov destination, source
移动ebx数据到eax
mov eax, ebx
移动 ebx+8 内存地址的4个字节数据到 eax
mov eax, [ebx+8]
```

```
lea des, src
;加载有效地址，假设 ebx = 0x00403A40，那么下面操作后，eax = 0x00403A48
lea eax, [ebx+8]
```

11.解释 boot.asm 文件中 org 0700h 的作用。

org 07c00h (1)

- 为什么需要`org 07c00h`这一行代码?
 - BIOS将软盘内容放在`07c00h`位置，并不是由这行代码决定的。
- `org`是伪指令
 - 伪指令是指，不生成对应的二进制指令，只是汇编器使用的。也就是说，`boot.bin`文件里面，没有`07c00h`这个东西，BIOS不是因为这条指令才把代码放在`07c00h`的。
 - `mov`这种指令，就会生成二进制代码，可以直接告诉CPU该做什么。

org 07c00h (2)

- `org 07c00h`的作用：告诉汇编器，当前这段代码会放在`07c00h`处。所以，如果之后遇到需要绝对寻址的指令，那么绝对地址就是`07c00h`加上相对地址。
 - 绝对地址：内存的实际位置（先不考虑内存分页一类逻辑地址）。
 - 相对地址：当前指令相对第一行代码的位置。
- 在第一行加上`org 07c00h`只是让编译器从相对地址`07c00h`处开始编译第一条指令，相对地址被编译加载后就正好和绝对地址吻合。

org 07c00h (3)

- 如果去掉这一行会发生什么？
- 反编译`boot1.bin`
- `ndisasm boot1.bin -o 0`

```

00000000 8CC8          mov ax,cs
00000002 8ED8          mov ds,ax
00000004 8EC0          mov es,ax
00000006 E80200        call word 0xb
00000009 EBFE          jmp short 0x9
0000000B B81E00        mov ax,0x1e
0000000E 89C5          mov bp,ax
00000010 B91000        mov cx,0x10
00000013 B80113        mov ax,0x1301
00000016 BB0C00        mov bx,0xc
00000019 B200          mov dl,0x0
0000001B CD10          int 0x10
0000001D C3             ret
0000001E 48             dec ax

```

0000000B B81E00 mov ax,0x1e

取字符串，由之前所说，第一行可执行代码加载到内存的`0x7c00`处，故字符串地址应该为`0x7c00+0x1e`

22

org 07c00h (4)

- 修改boot1.asm

mov ax,cs	000 8CC8	mov ax,cs
mov ds,ax	00000002 8ED8	mov ds,ax
mov es,ax	00000004 8EC0	mov es,ax
call DispStr	00000006 E80200	call word 0xb
jmp \$	00000009 EBFE	jmp short 0x9
DispStr:	0000000B B81E7C	mov ax,0x7c1e
mov ax,BootMessage+07c00h	0000000E 89C5	mov bp,ax
mov bp,ax	00000010 B91000	mov cx,0x10
mov cx,16	00000013 B80113	mov ax,0x1301
mov ax,01301h	00000016 BB0C00	mov bx,0xc
mov bx,000ch	00000019 B200	mov dl,0x0
mov dl,0	0000001B CD10	int 0x10
int 10h	0000001D C3	ret
BootMessage: db "Hello, OS world!"		
times 510-(\$-\$) db 0		
dw 0xaa55		

12.解释 boot.asm 文件中 times 510-(\$-\$) db 0 的作用。

填充剩下的空间，使生成的二进制代码恰好为512bytes

13.解释 bochs 中各参数的含义。

启动虚拟机 (1)

- 启动还需要Bochs的配置文件。告诉Bochs，你希望你的虚拟机是什么样子的，如内存多大，硬盘映像和软盘映像都是哪些文件等内容。
 - display_library: Bochs使用的GUI库
 - megs: 虚拟机内存大小 (MB)
 - floppya: 虚拟机外设，软盘为a.img文件
 - boot: 虚拟机启动方式，从软盘启动

```
megs:32
display_library: sdl2
floppya: 1_44=a.img, status=inserted
boot: floppy
```

14.boot.bin 应该放在软盘的哪一个扇区？为什么？

BIOS

- 开机，从ROM运行BIOS程序，BIOS是厂家写好的。
- BIOS程序检查软盘0面0磁道1扇区，如果扇区以0xaa55结束，则认定为引导扇区，将其512字节的数据加载到内存的07c00处，然后设置PC，跳到内存07c00处开始执行代码。
- 以上的0xaa55以及07c00都是一种约定，BIOS程序就是这样做的，所以我们就需要把我们的OS放在软盘的第一个扇区，填充，并在最末尾写入0xaa55。

```
17 times 510-($-$) db 0 ; 填充剩下的空间，使生成的二进制代码恰好为512字节
18 dw 0xaa55 ; 结束标志
```

写入boot.bin

- 使用dd命令将boot.bin写入软盘中

- `dd if=boot.bin of=a.img bs=512 count=1 conv=notrunc`

- `if`: 代表输入文件
 - `of`: 代表输出设备
 - `bs`: 代表一个扇区大小
 - `count`: 代表扇区数
 - `conv`: 代表不作其它处理

15.Loader的作用有哪些？

突破512字节的限制

- 其实，除了加载内核，我们要做的事还有准备保护模式（等讲到保护模式后再来理解）等，512字节显然不够，为了突破512字节的限制，我们引入另外一个重要文件loader.asm，引导扇区只负责把loader加载入内存并把控制权交给他，这样将会灵活得多。
- 最终，由loader将内核kernel加载入内存，才开始了真正操作系统内核的运行。

Loader

- 跳入保护模式
 - 最开始的x86处理器16位，寄存器用ax, bx等表示，称为实模式。后来扩充成32位，eax, ebx等，为了向前兼容，提出了保护模式
 - 必须从实模式跳转到保护模式，才能访问1M以上的内存。
- 启动内存分页
- 从kernel.bin中读取内核，并放入内存，然后跳转到内核所在的开始地址，运行内核
 - 跟boot类似，使用汇编直接在软盘下搜索kernel.bin
 - 但是，不能把整个kernel.bin放在内存，而是要以ELF文件的格式读取并提取代码。

Kernel

- 这才是真正的操作系统
- 内存管理，进程调度，图像显示，网络访问等等，都是内核的功能。
- 内核的开发使用高级语言
 - C语言可以更高效的编写内核
 - 但我们是在操作系统层面上编写另一个操作系统，于是生成的内核可执行文件是和当前操作系统平台相关的。比如Linux下是ELF格式，有许多无关信息，于是，内核并不能像boot.bin或loader.bin那样直接放入内存中，需要loader从kernel.bin中提取出需要放入内存中的部分。

