第4章　数据传送指令

罗文坚
中国科大 计算机学院

http://staff.ustc.edu.cn/~wjluo/mcps/

1

本章内容

- MOV指令回顾
- PUSH/POP指令
- 装入有效地址
- 数据串传送
- 其他数据传送指令
- 段超越前缀
- 汇编程序详述

2

MOV指令回顾

- 机器语言
  - 操作码、MOD字段、寄存器分配、R/M存储器寻址、特殊寻址方式、32位寻址方式、立即指令、段寄存器MOV指令
- 64位模式

3

机器语言

- 机器语言：作为指令由微处理器理解和使用的二进制代码，用来控制微处理器自身的运行。
- 8086~Core 2的机器语言指令长度从1字节到13字节。
- 尽管机器语言好像很复杂，但这些微处理器的机器语言也很规则。
  - 共有100,000多种变化形式的机器语言指令。
  - 几乎不能列一个完整的指令表来包含这多么变形。
  - 在机器语言指令中，某些二进制位是已给定的，其余二进制位则由每条指令的变化形式来确定。

4

16位指令模式

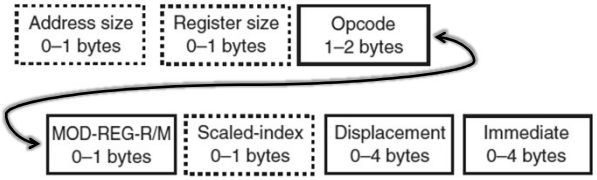
- 8086~80286的指令是16位指令模式。
- 16位指令模式与80386及更高型号微处理器工作在16位指令模式时是兼容的。
- 80386及更高型号微处理器工作在实模式时，假定所有指令都是16位模式。
- 在保护模式中，描述符的高端字节包含选择16位模式或32位模式指令的D位。

5

8086~Core 2指令的格式

- 16位指令模式

- 32位指令模式（仅用于80386~Pentium 4）



6

8086~Core 2指令的格式

- 32位指令格式的头2个字节，因为不常出现，故称为超越前缀（Override prefix）。
  - 第1个字节用来修改指令操作数的长度，称为地址长度前缀。
  - 第2个字节修改寄存器的长度，称为寄存器长度前缀。
- 32位指令格式的寄存器长度前缀：
  - 如果80386~Pentium 4按16位指令模式的机制操作（实模式或保护模式），而使用32位寄存器，则指令的前面出现寄存器长度前缀66H。
  - 如果微处理器按32位指令模式操作（只在保护模式），而且使用32位寄存器，则不存在寄存器长度前缀。
  - 如果在32位指令模式中出现16位寄存器，则要用寄存器长度前缀选择16位寄存器。

7

8086~Core 2指令的格式

- 32位指令格式的地址长度前缀（67H）的用法与寄存器长度前缀类似。
- 在带有前缀的指令中，前缀把寄存器及操作数的长度从16位转换到32位，或是从32位转换到16位。
- 注意：16位指令模式用8位及16位寄存器和寻址方式；而32位指令模式使用8位及32位寄存器和寻址方式，这是默认的用法。
  - 前缀可以超越这些默认，使得32位寄存器可以用于16位模式，而16位寄存器可以用于32位模式。

8

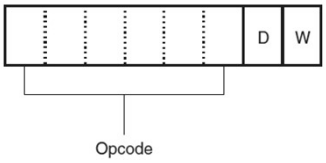
8086~Core 2指令的格式

- 操作模式（mode of operation）的选择（16位或32位）要符合应用程序的特征。
  - 如果应用程序多使用8位或32位数据，则要选择32位模式。
  - 如果大多使用8位或16位数据，则要选择16位模式。
- DOS只能按照16位模式操作，Window所可以工作于两类模式。

9

8086~Core 2指令的格式

- 操作码：选择微处理器执行的操作（加、减、传送）等。多数机器语言指令的操作码长度为1或2个字节。
- 多数机器语言指令的第1个操作码字节的一般格式。

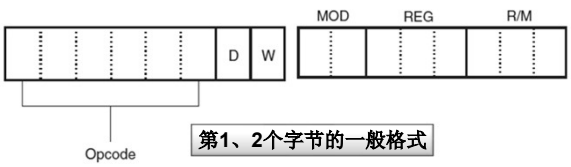


- D位：指示数据流的方向（REG→R/M或R/M→REG）。
- W位：指令模式位（W=0: 8位；W=1: 16或32位）。

10

8086~Core 2指令的格式

- D=1，数据从第2字节的R/M字段流向REG字段。
- D=0，数据从REG字段流向R/M字段。
- W=0，表示数据长度为字节。
- W=1，表示数据的长度是字或双字（由寄存器长度前缀确定）。



第1、2个字节的一般格式

11

8086~Core 2指令的格式

- MOD字段：规定指令的寻址方式，选择寻址类型及所选类型是否有位移量。

MOD	*Function*
00	No displacement
01	8-bit sign-extended displacement
10	16-bit signed displacement
11	R/M is a register

16位指令模式的MOD字段（不带操作数长度前缀）

12

8086~Core 2指令的格式

- 16位指令模式的MOD字段（不带操作数长度前缀）
  - 如果MOD=11，选择寄存器寻址方式，用R/M字段指定寄存器而不是存储单元。
  - 如果MOD=00，01，或10，R/M字段选择一种数据存储器寻址方式。
    - 00表示不带位移量，例，MOV AL, [DI]
    - 01表示包含8位有符号扩展的位移量。例，MOV AL, [DI+2]
    - 10表示包含16位有符号扩展的位移量。例，MOV AL, [DI+1000H]

13

8086~Core 2指令的格式

- 16位指令模式的MOD字段（不带操作数长度前缀）
  - ……
  - 当微处理器执行指令时，将8位的位移量符号扩展为16位的位移量。
  - 例如，80H符号扩展（sign-extended）后成为FF80H。
- 80386~Core 2微处理器中，对于16位指令模式，MOD字段的含义没有变化。

14

8086~Core 2指令的格式

- 80386~Core 2微处理器中的32位指令模式，MOD字段的含义：

MOD	*Function*
00	No displacement
01	8-bit sign-extended displacement
10	32-bit signed displacement
11	R/M is a register

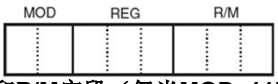
15

8086~Core 2指令的格式

- 80386~Core 2微处理器中，MOD字段的含义受地址长度超越前缀影响。
  - 当MOD=10时，16位指令模式下，16位位移量变成32位位移量。
  - 在80386以上微处理器中，工作在32位指令模式下时，如果不用地址长度超越前缀，则只允许用8位或32位位移量。
  - 当然，32位指令模式下，8位/16位的位移量将被符号扩展至32位。

16

8086~Core 2指令的格式



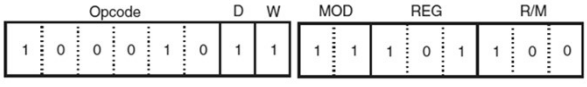
- REG字段和R/M字段（仅当MOD=11时）指示寄存器的分配情况

Code	W = 0 *(Byte)*	W = 1 *(Word)*	W = 1 *(Doubleword)*
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

17

8086~Core 2指令的格式

- 例，假设有一个2字节指令8BECH。



Opcode = MOV
D = Transfer to register (REG)
W = Word
MOD = R/M is a register
REG = BP
R/M = SP

MOV BP, SP

18

8086~Core 2指令的格式

- 例，对于80386以上微处理器，以16位指令模式操作时，出现了668BE8H指令。
  - 第1个字节是66H，寄存器长度超越前缀。
  - 操作码MOV，源操作数EAX，目的操作数EBP。
  - 这条指令是：MOV EBP, EAX
- 例，对于80386以上微处理器，以32位指令模式操作时，出现了668BE8H指令.
  - 这条指令是：MOV BP, AX
- 编写汇编程序时，可以指定16位或32位指令模式。

19

8086~Core 2指令的格式

- R/M字段的存储器寻址
  - 如果MOD=00，01或10，则R/M指示存储器寻址。

16位的R/M存储器寻址方式

R/M Code	Addressing Mode
000	DS:[BX+SI]
001	DS:[BX+DI]
010	SS:[BP+SI]
011	SS:[BP+DI]
100	DS:[SI]
101	DS:[DI]
110	SS:[BP]*
111	DS:[BX]

直接寻址如何表示？

20

8086~Core 2指令的格式

- 16位的R/M存储器寻址方式
  - 当MOD=00，R/M=110时，指示直接寻址（只用位移量寻址存储器的数据）。
  - 但是，“MOD=00，R/M=110”指示用[BP]寻址，而且没有位移量。存在冲突？
    - 实际上，机器语言中，不可以用没有位移量的[BP]寻址方式。每当指令中出现[BP]寻址方式时，汇编程序就使用一个8位的位移量00H（此时MOD=01）。
    - 或者说，[BP]被汇编成[BP+0]。

21

8086~Core 2指令的格式

- R/M选择的32位寻址方式

R/M Code	Function
000	DS:[EAX]
001	DS:[ECX]
010	DS:[EDX]
011	DS:[EBX]
100	Uses scaled-index byte ✹
101	SS:[EBP]* ✹
110	DS:[ESI]
111	DS:[EDI]

22

8086~Core 2指令的格式

- R/M选择的32位寻址方式

  - 当R/M=100时，指示“比例变址寻址”，指令中出现附加字节——比例变址字节。

ss
00 = ×1
01 = ×2
10 = ×4
11 = ×8

注意1：Index=100时，无变址寄存器。
注意2：Base=101时，使用或不用EBP？

思考：如何指示是否只包含比例变址？

23

8086~Core 2指令的格式

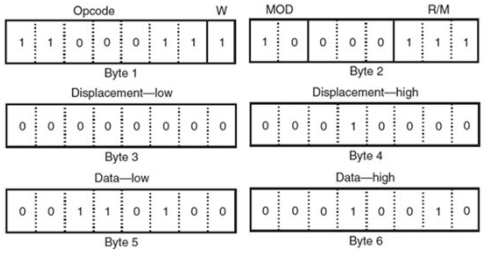
- R/M选择的32位寻址方式

  - 比例变址字节有8位，MOD有2位，REG有3位，D和W各1位，共计15位，则比例变址有2^{15}种组合。
  - 在80386~Core2中，仅MOV指令就有32000中组合。

24

8086~Core 2指令的格式

- 立即指令。例，MOV WORD PTR [BX+1000H], 1234H



Opcode = MOV (immediate)
W = Word
MOD = 16-bit displacement
REG = 000 (not used in immediate addressing)
R/M = DS:[BX]
Displacement = 1000H
Data = 1234H

25

8086~Core 2指令的格式

- 段寄存器MOV指令
  - 段寄存器的内容通过MOV、PUSH、POP指令传送，用一组专门的寄存器位（REG字段）选择段寄存器。

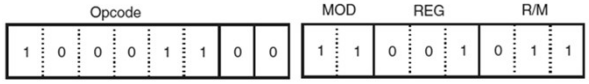
Code	*Segment Register*
000	ES
001	CS*
010	SS
011	DS
100	FS
101	GS

*Note: MOV CS,R/M and POP CS are not allowed.

26

8086~Core 2指令的格式

- 段寄存器MOV指令
  - 例，MOV [DI], DS



Opcode = MOV
MOD = R/M is a register
REG = CS
R/M = BX

注意Opcode!

27

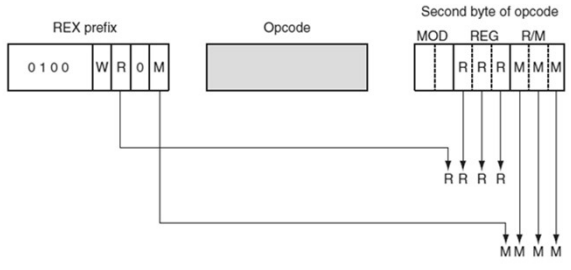
64位模式

- 64位模式增加了一个REX前缀（40H~4FH），放在在其它前缀之后，操作码之前。
- REX前缀的目的是修改指令的第二字节中的REG字段和R/M字段。
  - 寻址R8~R15寄存器时必须用REX前缀。
- 分两种情况解释：
  - 无比例变址字段
  - 有比例变址字段

28

64位模式

- REX无比例变址的应用



W = 1 (64 bits)
W = 0 (CS descriptor)

注：W=0 表示 Operand size determined by CS.D。

29

64位模式

- 64位寄存器和内存的RRRR和MMMM字段的标志符

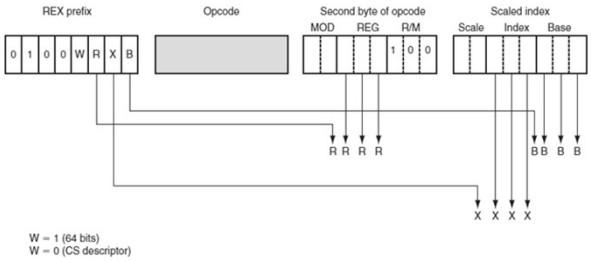
Code	*Register*	*Memory*
0000	RAX	[RAX]
0001	RCX	[RCX]
0010	RDX	[RDX]
0011	RBX	[RBX]
0100	RSP	See note
0101	RBP	[RBP]
0110	RSI	[RSI]
0111	RDI	[RDI]
1000	R8	[R8]
1001	R9	[R9]
1010	R10	[R10]
1011	R11	[R11]
1100	R12	[R12]
1101	R13	[R13]
1110	R14	[R14]
1111	R15	[R15]

Note: This addressing mode specifies the inclusion of the scaled-index byte.

30

64位模式

- 含比例变址



31

本章内容

- MOV指令回顾
- PUSH/POP指令
- 装入有效地址
- 数据串传送
- 其他数据传送指令
- 段超越前缀
- 汇编程序详述

32

PUSH/POP指令

- 6中形式的PUSH/POP指令
  - PUSH 立即数
  - PUSH / POP 寄存器
  - PUSH / POP 存储器
  - PUSH / POP 段寄存器
  - PUSH / POP 标志寄存器
  - PUSHA / POPA （全部寄存器）
- 8086/8088不支持PUSH/POP立即数，不支持PUSHA/POPA。
- Pentium4和Core2的64位模式不支持PUSHA/POPA。

33

PUSH/POP指令

- 立即数寻址（8086/8088不支持，其余支持）
  - 允许立即数压入堆栈，但不能从堆栈弹出。
- 寄存器寻址
  - 可以将任何16位通用寄存器入栈、出栈。
  - 80386以上微处理器中，可以将32位扩展寄存器和标志寄存器EFLAGS入栈、出栈。
  - 对于段寄存器寻址，所有段寄存器可以入栈，但CS不作为出栈的目的操作数。
- 采用存储器寻址时
  - 可将16位存储单元的内容入栈、出栈。
  - 80386以上CPU中，可将32位存储单元的内容入栈、出栈。

34

PUSH指令

- PUSH指令的格式

Symbolic	*Example*	*Note*
PUSH reg16	PUSH BX	16-bit register
PUSH reg32	PUSH EDX	32-bit register
PUSH mem16	PUSH WORD PTR[BX]	16-bit pointer
PUSH mem32	PUSH DWORD PTR[EBX]	32-bit pointer
PUSH mem64	PUSH QWORD PTR[RBX]	64-bit pointer (64-bit mode)
PUSH seg	PUSH DS	Segment register
PUSH imm8	PUSH 'R'	8-bit immediate
PUSH imm16	PUSH 1000H	16-bit immediate
PUSHD imm32	PUSHD 20	32-bit immediate
PUSHA	PUSHA	Save all 16-bit registers
PUSHAD	PUSHAD	Save all 32-bit registers
PUSHF	PUSHF	Save flags
PUSHFD	PUSHFD	Save EFLAGS

35

PUSH指令

- PUSHA指令：将AX、CX、DX、BX、SP、BP、SI、DI顺序入栈。
  - 压入堆栈的SP的内容是其在PUSHA指令执行前的值。
- PUSHF指令：将标志寄存器FLAGS压入堆栈。
- PUSHAD指令：压入80386~Pentium 4中全部32位寄存器。此时，需要32字节（4×8）的存储空间。
- PUSHFD指令：将标志寄存器EFLAGS压入堆栈。

36

PUSH指令

- PUSH立即数指令有两种操作码，都是将一个16位立即数压入堆栈。
  - PUSH imm8，操作码为6AH
  - PUSH imm16，操作码为68H
- PUSHD imm32：将32位立即数压入堆栈。
- 例如，PUSH 8指令，实际是将0008H入栈。

37

POP指令

- POP指令的格式

Symbolic	Example	Note
POP reg16	POP CX	16-bit register
POP reg32	POP EBP	32-bit register
POP mem16	POP WORD PTR[BX+1]	16-bit pointer
POP mem32	POP DATA3	32-bit memory address
POP mem64	POP FROG	64-bit memory address (64-bit mode)
POP seg	POP FS	Segment register
POPA	POPA	Pops all 16-bit registers
POPAD	POPAD	Pops all 32-bit registers
POPF	POPF	Pops flags
POPFD	POPFD	Pops EFLAGS

38

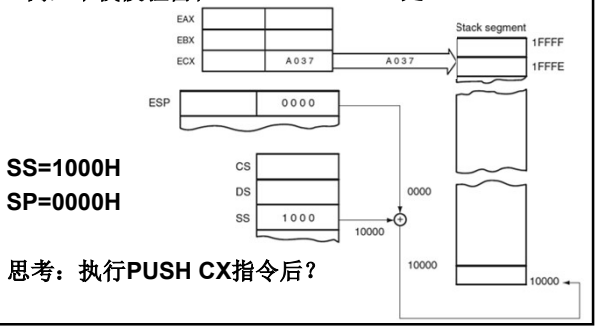
POP指令

- POP指令不能用立即寻址方式。
- POPF从堆栈弹出2字节数据，并送入FLAGS。
  - POPFD恢复EFLAGS。
- POPA从堆栈弹出16字节的数据，依次送入DI、SI、BP、SP、BX、DX、CX和AX。
  - POPAD类似，但针对32位的扩展寄存器。
  - 数据出栈时，SP/ESP被忽略。
- POP CS是不允许的。

39

初始化堆栈

- 初始化堆栈时应加载堆栈段寄存器和堆栈指针寄存器。
- 例，堆栈段驻留在10000H~1FFFFH处。



SS=1000H
SP=0000H

思考：执行PUSH CX指令后？

40

初始化堆栈

- 汇编语言中堆栈段的设置
  - 第一种方式：完整段定义

```
0000                    STACK_SEG       SEGMENT STACK
0000 0100[                              DW    100H DUP(?)
         ????
            ]
0200                    STACK_SEG       ENDS
```

- 第一条语句定义堆栈段的开始；
- 最后一条语句说明堆栈段的结束。
- 汇编和连接程序将正确的堆栈段的地址压入SS，把栈顶地址压入SP。

41

初始化堆栈

- 汇编语言中堆栈段的设置
  - 第二种方式：简化段定义
  - 适用于MASM

.MODEL SMALL；定义存储模型
.STACK 200H；设置堆栈，初始化SS和SP

42

初始化堆栈

- 如果程序没有定义堆栈段，会出现警告信息。
  “LINK: warning L4021: no stack segment”
  - 如果所需的堆栈空间比较小，该警告信息可以不必理会，因为在操作系统装入程序时会自动为其添加一个默认的堆栈段，即“缺省指定”。

43

本章内容

- MOV指令回顾
- PUSH/POP指令
- 装入有效地址
- 数据串传送
- 其他数据传送指令
- 段超越前缀
- 汇编程序详述

44

装入有效地址

- LEA
  - 把有效地址装入指定寄存器。
- LDS、LES
  - 把有效地址装入寄存器，段地址装入DS或ES。
  - 64位模式下无效和不可用（因为用了平展模式）。
- LFS、LGS、LSS
  - 有效地址装入寄存器，段地址装入FS、GS、SS。
  - 80386以上新增加。

45

LEA指令

- LEA指令：把由操作数指定的数据的偏移地址装入16位或32位寄存器。
- 例，LEA AX, NUMB
  - 将NUMB的偏移地址装入AX寄存器
  - 与MOV AX, NUMB不同
- 例，下面两条指令的区别
  - LEA BX, [DI]；将DI的内容装入BX
  - MOV BX, [DI]；将DI寻址的存储器数据装入BX

46

LEA指令

- 例，下面两条指令功能相同
  - LEA BX, LIST
  - MOV BX, OFFSET LIST；OFFSET指示取偏移地址
- 既然有OFFSET伪指令，为何要有LEA指令？
  - 因为OFFSET只能用于类似于LIST这样的简单变量，不能用于类似于[DI]、LIST[DI]这样的操作数。
- 处理器执行LEA BX, LIST指令花费的时间比执行MOV BX, OFFSET LIST花费的时间长。
  - 原因：OFFSET LIST是由汇编程序事先计算的。

47

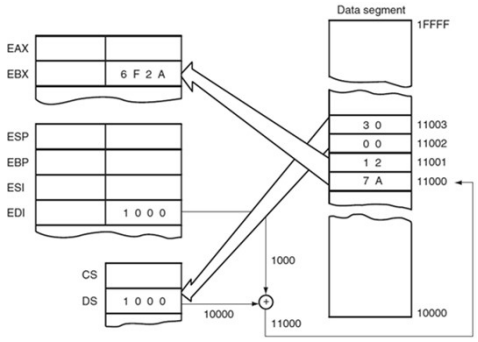
LDS、LES、LFS、LGS、LSS

- LDS、LES、LFS、LGS、LSS指令：把偏移地址装入16位或32位寄存器，并把段地址装入段寄存器DS、ES、FS、GS、SS。
- 这些指令可寻址32位（16+16）或48位（32+16）存储区。
- 偏移地址在前（低地址），段地址在后（高地址）。

48

LDS、LES、LFS、LGS、LSS

- 例，LDS BX, [DI]



49

本章内容

- MOV指令回顾
- PUSH/POP指令
- 装入有效地址
- 数据串传送
- 其他数据传送指令
- 段超越前缀
- 汇编程序详述

50

数据串传送

- 有5条数据串传送指令
  - LODS
  - STOS
  - MOVS
  - INS
  - OUTS
- 每条数据串传送指令都允许传送字节、字、或双字。
- 串指令与方向标志（D）、DI和SI寄存器密切相关。

51

方向标志

◆方向标志D只用于串操作指令。

- D=0，SI和/或DI寄存器自动递增；
- D=1，SI和/或DI寄存器自动递减。
  - 仅串操作指令实际使用的寄存器才自动递增或递减。
  - 传送字节时，SI和/或DI自动 ±1；
  - 传送字时，SI和/或DI自动 ±2；
  - 传送双字时，SI和/或DI自动 ±4；
- CLD指令清除方向标志，D=0；
- STD指令设置方向标志，D=1。

52

DI和SI

- 串操作指令执行时，对存储器的访问是通过DI和SI两个寄存器（或其中1个）实现的。
- 默认情况下，对于所有的串操作指令，DI偏移地址是用于访问附加段中的数据，SI偏移地址是用于访问数据段中的数据。
  - SI的默认段寄存器DS可以用段超越前缀改变。
  - DI的默认段寄存器ES不能改变。
- 对80386以上的CPU，工作在32位模式时，用ESI、EDI代替SI、DI。

53

LODS指令

- LODS指令：将SI寻址的数据装入AL、AX、EAX或RAX。数据装入寄存器后，SI自动±1，2, 4, 或8。
- LODS指令的格式：
  - LODSB ；装入字节数据至AL
  - LODSW ；装入字数据至AX
  - LODSD ；装入双字数据至EAX
  - LODSQ ；装入8字节数据至RAX
  - LODS LIST；如果LIST是字节变量，则装入字节
  - LODS DATA1；如果DATA1是字变量？
  - LODS DATA2；如果DATA2是双字变量？

54

STOS指令

- STOS指令：将AL、AX、EAX、RAX的内容存储到DI寄存器存执的存储单元。数据传送后，DI自动±1，2, 4, 或8。
- STOS指令的格式：
  - STOSB ； ES: [DI]=AL, DI=DI±1
  - STOSW ； ES: [DI]=AX, DI=DI±2
  - STOSD ； ES: [DI]=EAX, DI=DI±4
  - STOSQ ； ES: [DI]=RAX, DI=DI±8
  - STOS LIST ； ES: [DI]=AX, DI=DI±1，设LIST是字节变量
  - STOS DATA3 ； 如果DATA3是字变量？
  - STOS DATA4 ； 如果DATA4是双字变量？

55

带REP前缀的STOS

- 重复前缀REP（Repeat Prefix）可以加到除了LODS指令以外的任何数据串传送指令上。
  - 执行重复的LODS操作没有意义。
- REP前缀的作用：使得每次执行串指令后CX减1。CX减1后，重复执行指令，直到CX值为0时，指令终止。
  - 80386以上微处理器使用ECX。
  - Pentium 4在64位模式下，使用RCX寄存器。
- 例，如果CX=100，执行REP STOSB
  - CPU自动执行STOSB指令100次，将AL的内容存储相应的存储块。

56

带REP前缀的STOS

- 例，用内嵌汇编清除Buffer缓冲区。

```
void ClearBuffer (int count, short* buffer)
{
   _asm{
         push edi          ;save registers
         push es
         push ds
         mov  ax,0
         mov  ecx, count
         mov  edi, buffer
         pop  es           ;load ES with DS
         rep  stosw        ;clear Buffer
         pop  es           ;restore registers
         pop  edi
   }
}
```

57

MOVS指令

- MOVS指令：将数据从一个存储单元传送到另一个存储单元。
  - SI寻址源操作数，DI寻址目的操作数。
  - 传送结束后，SI、DI自动递增或递减。
  - 对SI可以使用段超越前缀，而DI不能使用段超越前缀。
- MOVS指令是唯一允许的存储器到存储器的传送指令。

58

MOVS指令

- MOVS指令的格式：
  - 使用SI或ESI或RSI，DI或EDI或RDI

Assembly Language	*Operation*
MOVSB	ES:[DI] = DS:[SI]; DI = DI ± 1; SI = SI ± 1 (byte transferred)
MOVSW	ES:[DI] = DS:[SI]; DI = DI ± 2; SI = SI ± 2 (word transferred)
MOVSD✦	ES:[DI] = DS:[SI]; DI = DI ± 4; SI = SI ± 4 (doubleword transferred)
MOVSQ✦	[RDI] = [RSI]; RDI = RDI ± 8; RSI = RSI ± 8 (64-bit mode)
MOVS BYTE1, BYTE2	ES:[DI] = DS:[SI]; DI = DI ± 1; SI = SI ± 1 (byte transferred if BYTE1 and BYTE2 are bytes)
MOVS WORD1,WORD2	ES:[DI] = DS:[SI]; DI = DI ± 2; SI = SI ± 2 (word transferred if WORD1 and WORD2 are words)
MOVS TED,FRED	ES:[DI] = DS:[SI]; DI = DI ± 4; SI = SI ± 4 (doubleword transferred if TED and FRED are doublewords)

59

MOVS指令

- 例，把BlockA复制到BlockB，用内嵌汇编实现

```
void TransferBlocks (int blockSize, int* blockA, int* blockB)
{
      _asm{
            push es                  ;save registers
            push edi
            push esi
            push ds                  ;copy DS into ES
            pop  es
            mov  esi, blockA         ;address blockA
            mov  edi, blockB         ;address blockB
            mov  ecx, blockSize      ;load count
            rep  movsd               ;move data
            pop  esi
            pop  edi
            pop  es                  ;restore registers
      }
}
```

60

MOVS指令

- 例，把BlockA复制到BlockB，用C语言实现

```
void TransferBlocks (int blockSize, int* blockA, int* blockB)
{
      for (int a = 0; a < blockSize; a++)
      {
            blockA = blockB++;
            blockA++;
      }
}
```

61

MOVS指令

- 例，把BlockA复制到BlockB，用C语言实现后，对应的汇编语言版本。

```
void TransferBlocks(int blockSize, int* blockA, int* blockB)
{
004136A0  push        ebp
004136A1  mov         ebp,esp
004136A3  sub         esp,0D8h
004136A9  push        ebx
004136AA  push        esi
004136AB  push        edi
004136AC  push        ecx
004136AD  lea         edi,[ebp-0D8h]
004136B3  mov         ecx,36h
004136B8  mov         eax,0CCCCCCCCh
004136BD  rep stos    dword ptr [edi]
004136BF  pop         ecx
004136C0  mov         dword ptr [ebp-8],ecx
```

62

MOVS指令

- 例，把BlockA复制到BlockB，用C语言实现后，对应的汇编语言版本。

```
      for( int a = 0; a < blockSize; a++ )
004136C3  mov         dword ptr [a],0
004136CA  jmp         TransferBlocks+35h (4136D5h)
004136CC  mov         eax,dword ptr [a]
004136CF  add         eax, 1
004136D2  mov         dword ptr [a],eax
004136D5  mov         eax,dword ptr [a]
004136D8  cmp         eax,dword ptr [blockSize]
004136DB  jge         TransferBlocks+57h (4136F7h)
      {
          blockA = blockB++;
004136DD  mov         eax,dword ptr [blockB]
004136E0  mov         dword ptr [blockA],eax
004136E3  mov         ecx,dword ptr [blockB]
004136E6  add         ecx,4
004136E9  mov         dword ptr [blockB],ecx
```

63

MOVS指令

- 例，把BlockA复制到BlockB，用C语言实现后，对应的汇编语言版本。

```
          blockA++;
004136EC  mov         eax,dword ptr [blockA]
004136EF  add         eax,4
004136F2  mov         dword ptr [blockA],eax
      }
004136F5  jmp         TransferBlocks+2Ch (4136CCh)
}
004136F7  pop         edi
004136F8  pop         esi
004136F9  pop         ebx
004136FA  mov         esp,ebp
004136FC  pop         ebp
004136FD  ret         0Ch
```

显然，用内嵌汇编实现的程序更快！

64

INS指令

- INS指令（Input String，串输入）：从I/O设备把字节、字、双字传送到DI寻址的存储单元。
  - I/O地址存放在DX寄存器中。
- INS指令对于将外部设备的数据块直接输入到存储器非常有用。
- 注意：
  - INS指令不能用于8086/8088 CPU。
  - 64位模式中，没有64位输入。但是，存储地址是64位，且由INS指令同RDI中定位。

65

INS指令

- INS指令的格式：
  - 使用DI、EDI或RDI。

Assembly Language	*Operation*
INSB	ES:[DI] = [DX]; DI = DI ± 1 (byte transferred)
INSW	ES:[DI] = [DX]; DI = DI ± 2 (word transferred)
INSD	ES:[DI] = [DX]; DI = DI ± 4 (doubleword transferred)
INS LIST	ES:[DI] = [DX]; DI = DI ± 1 (if LIST is a byte)
INS DATA4	ES:[DI] = [DX]; DI = DI ± 2 (if DATA4 is a word)
INS DATA5	ES:[DI] = [DX]; DI = DI ± 4 (if DATA5 is a doubleword)

66

INS指令

- 例，用REP INSB输入数据

```
MOV DI,OFFSET LISTS     ;address array
MOV DX,3ACH             ;address I/O
CLD                     ;auto-increment
MOV CX,50               ;load counter
REP INSB                ;input data
```

67

OUTS指令

- OUTS指令（Output String，串输出）：从数据段中SI寻址的字节、字、或双字传送到I/O设备。
  - I/O地址存放在DX寄存器中。
- 注意：
  - OUTS指令不能用于8086/8088 CPU。
  - 64位模式中，没有64位的输出。但是，RSI的地址是64位宽的。

68

OUTS指令

- OUTS指令格式
  - 使用DI、EDI或RDI。

Assembly Language	*Operation*
OUTSB	[DX] = DS:[SI]; SI = SI ± 1 (byte transferred)
OUTSW	[DX] = DS:[SI]; SI = SI ± 2 (word transferred)
OUTSD	[DX] = DS:[SI]; SI = SI ± 4 (doubleword transferred)
OUTS DATA7	[DX] = DS:[SI]; SI = SI ± 1 (if DATA7 is a byte)
OUTS DATA8	[DX] = DS:[SI]; SI = SI ± 2 (if DATA8 is a word)
OUTS DATA9	[DX] = DS:[SI]; SI = SI ± 4 (if DATA9 is a doubleword)

69

OUTS指令

- 例，用REP OUTSB输出数据

```
MOV SI,OFFSET ARRAY     ;address array
MOV DX,3ACH             ;address I/O
CLD                     ;auto-increment
MOV CX,100              ;load counter
REP OUTSB               ;output data
```

70

本章内容

- MOV指令回顾
- PUSH/POP指令
- 装入有效地址
- 数据串传送
- 其他数据传送指令
- 段超越前缀
- 汇编程序详述

71

其他数据传送指令

- 其他数据传送指令
  - XCHG
  - LAHF，SAHF
  - XLAT
  - IN，OUT
  - MOVSX，MOVZX
  - BSWAP
  - CMOV

72

XCHG指令

- XCHG指令（Exchange，交换指令）：将一个寄存器内容与其他寄存器或存储单元的内容交换。
- 注意事项：
  1. 可以交换字节、字、双字、8字节。
  2. 不能实现段寄存器之间的交换。
  3. 不能实现存储器和存储器之间的数据交换。
  4. 存储器操作数可以用除立即数寻址方式以外的其他任何寻址方式。

73

XCHG指令的格式举例

- XCHG AL, CL
- XCHG CX, BP
- XCHG EDX, ESI
- XCHG AL, DATA2
- XCHG RBX, RCX

74

LAHF和SAHF指令

- LAHF指令：把标志寄存器的低8位送到AH。
- SAHF指令：把AH内容送到标志寄存器的低8位。
- LAHF和SAHF已经很少使用。
- 在64位模式下，LAHF和SAHF是无效的和不起作用的。

75

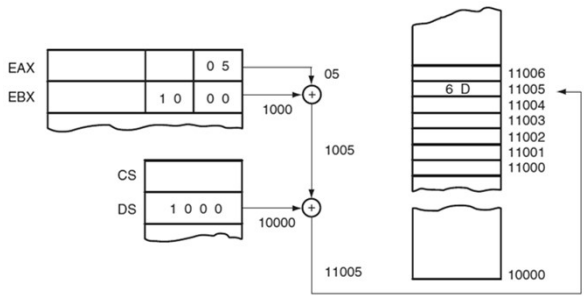
XLAT指令

- XLAT（Translate，换码）指令：把AL寄存器中内容转换成存储器中的数据。
- XLAT指令执行过程：
  1) 首先将AL与BX的内容相加，形成数据段内的偏移地址。
  2) 根据该偏移地址，将对应的数据复制到AL。
- XLAT指令是唯一一条把8位数据加到16位数字上的指令。

76

XLAT指令

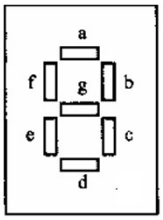
- XLAT指令示意图



77

XLAT指令

- 例，用XLAT指令把AL中的BCD码转换成LED显示器的7段码。




```
TABLE  DB  3FH, 06H, 5BH, 4FH    ;lookup table
       DB  66H, 6DH, 7DH, 27H
       DB  7FH, 6FH

LOOK:  MOV AL,5              ;load AL with 5 (a test number)
       MOV BX,OFFSET TABLE   ;address lookup table
       XLAT                  ;convert
```

78

IN和OUT指令

- 在I/O设备与微处理器之间只能传送AL、AX或EAX的内容。
  - IN指令将外部I/O设备的数据传送到AL、AX或EAX。
  - OUT指令将AL、AX或EAX的数据送到外部的I/O设备。
- 对于IN和OUT指令，I/O设备的地址端口有两种形式：
  - 固定端口：端口号跟在指令操作码后面，只能是8位的I/O端口。
  - 可变端口：端口号放在DX寄存器中。

79

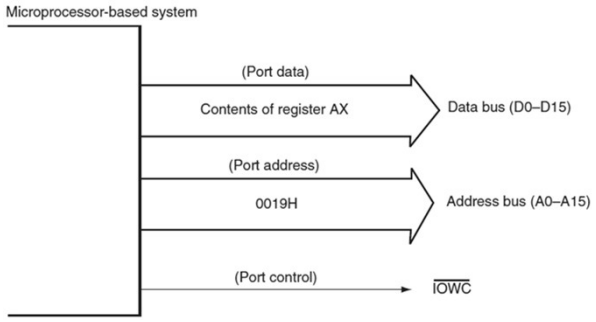
IN和OUT指令

- IN和OUT指令只使用低16位地址总线。
  - 如果是8位端口，则零扩展的形式使之成为16位端口。
  - 除低16位地址总线外，其余的地址总线是没有定义的。
- 注意：INTEL为它的某些外围设备保留了最后的16个I/O端口。

80

IN和OUT指令

- OUT 19H，AX指令的示意图



81

IN和OUT指令

- IN和OUT指令的形式

Assembly Language	*Operation*
IN AL,p8	8 bits are input to AL from I/O port p8
IN AX,p8	16 bits are input to AX from I/O port p8
IN EAX,p8	32 bits are input to EAX from I/O port p8
IN AL,DX	8 bits are input to AL from I/O port DX
IN AX,DX	16 bits are input to AX from I/O port DX
IN EAX,DX	32 bits are input to EAX from I/O port DX
OUT p8,AL	8 bits are output to I/O port p8 from AL
OUT p8,AX	16 bits are output to I/O port p8 from AX
OUT p8,EAX	32 bits are output to I/O port p8 from EAX
OUT DX,AL	8 bits are output to I/O port DX from AL
OUT DX,AX	16 bits are output to I/O port DX from AX
OUT DX,EAX	32 bits are output to I/O port DX from EAX

Note: p8 = an 8-bit I/O port number (0000H to 00FFH) and DX = the 16-bit I/O port number (0000H to FFFFH) held in register DX.

82

MOVSX和MOVZX指令

- MOVSX指令：Move and Sign-extend，传送及符号扩展。
- MOVZX指令：Move and Zero-extend，传送及零扩展
- 符号扩展：高位部分用符号位填充。
- 零扩展：高位部分用零填充。
- 这两条指令只出现在80386以上CPU中。

83

MOVSX和MOVZX指令

- 例，
  - MOVSX CX, BL
  - MOVSX ECX, AX
  - MOVSX BX, DATA1；DATA1是已定义字节变量
  - MOVSX EAX, [EDI]
  - MOVSX RAX, [RDI]
  - MOVZX DX, AL
  - MOVZX EBP, DI
  - MOVZX DX, DATA2
  - MOVZX EAX, DATA3
  - MOVZX RBX, ECX

84

BSWAP指令

- BSWAP指令：Byte Swap，字节交换指令
  - 只能用于80486~Pentium 4微处理器。
  - 该指令将32位寄存器内的第1字节和第4字节交换，第2字节和第3字节交换。
- 例，EAX=00112233H。
  - 执行BSWAP EAX指令后，EAX=33221100H。
- 该指令可以把由大到小的数据转换为由小到大的格式。反之亦然。

85

CMOV指令

- CMOV指令：Conditional move，条件传送。
  - Pentium Pro及其以上增加的指令。
- CMOV指令的含义：当条件为真时，执行数据的传送。
  - 因为条件很多，因此CMOV指令有很多种格式。
- CMOV指令的目的操作数只能是16位或32位或64位寄存器；而源操作数可以是16位、32位、64位寄存器，或者存储单元。

86

CMOV指令

Assembly Language	*Flag(s) Tested*	*Operation*
CMOVB	C = 1	Move if below
CMOVAE	C = 0	Move if above or equal
CMOVBE	Z = 1 or C = 1	Move if below or equal
CMOVA	Z = 0 and C = 0	Move of above
CMOVE or CMOVZ	Z = 1	Move if equal or move if zero
CMOVNE or CMOVNZ	Z = 0	Move if not equal or move if not zero
CMOVL	S ! = O	Move if less than
CMOVLE	Z = 1 or S ! = O	Move if less than or equal
CMOVG	Z = 0 and S = O	Move if greater than
CMOVGE	S = O	Move if greater than or equal
CMOVS	S = 1	Move if sign (negative)
CMOVNS	S = 0	Move if no sign (positive)
CMOVC	C = 1	Move if carry
CMOVNC	C = 0	Move if no carry
CMOVO	O = 1	Move if overflow
CMOVNO	O = 0	Move if no overflow
CMOVP or CMOVPE	P = 1	Move if parity or move if parity even
CMOVNP or CMOVPO	P = 0	Move if no parity or move if parity odd

87

本章内容

- MOV指令回顾
- PUSH/POP指令
- 装入有效地址
- 数据串传送
- 其他数据传送指令
- 段超越前缀
- 汇编程序详述

88

段超越前缀

- 段超越前缀允许程序设计者改变默认的段寄存器。
- 段超越前缀（Segment override prefix）可以附加到几乎任何指令的存储器寻址方式前。
  - About the only instructions that cannot be prefixed are the jump and call instructions that must use the code segment register for address generation.
- 在机器指令中，段超越前缀出现在之前的附加字节上，以便选择代替的段寄存器。

89

段超越前缀

- 例，MOV AX, [DI]指令，默认的情况是访问数据段中的数据。
  - 如果将指令改成MOV AX, ES: [DI]，则访问附加段中的数据。
- 当指令加了段超越前缀后，指令的长度就增加了1个字节，而且执行时间也增加了。

90

段超越前缀的例子

Assembly Language	*Segment Accessed*	*Default Segment*
MOV AX,DS:[BP]	Data	Stack
MOV AX,ES:[BP]	Extra	Stack
MOV AX,SS:[DI]	Stack	Data
MOV AX,CS:LIST	Code	Data
MOV ES:[SI],AX	Extra	Data
LODS ES:DATA1	Extra	Data
MOV EAX,FS:DATA2	FS	Data
MOV GS:[ECX],BL	GS	Data

91