CMSC 313 COMPUTER ORGANIZATION & ASSEMBLY LANGUAGE PROGRAMMING

LECTURE 06, FALL 2012

TOPICS TODAY

- More Arithmetic Instructions
  - NEG, MUL, IMUL, DIV
- Indexed Addressing Modes
- Some i386 String Instructions

MORE ARITHMETIC INSTRUCTIONS

More Arithmetic Instructions

- NEG: two's complement negation of operand
- MUL: unsigned multiplication
  - Multiply AL with r/m8 and store product in AX
  - Multiply AX with r/m16 and store product in DX:AX
  - Multiply EAX with r/m32 and store product in EDX:EAX
  - Immediate operands are not supported.
  - CF and OF cleared if upper half of product is zero.
- IMUL: signed multiplication
  - Use with signed operands
  - More addressing modes supported
- DIV: unsigned division

NEG—Two's Complement Negation

Opcode	Instruction	Description
F6 /3	NEG *r/m8*	Two's complement negate *r/m8*
F7 /3	NEG *r/m16*	Two's complement negate *r/m16*
F7 /3	NEG *r/m32*	Two's complement negate *r/m32*

Description

Replaces the value of operand (the destination operand) with its two's complement. (This operation is equivalent to subtracting the operand from 0.) The destination operand is located in a general-purpose register or a memory location.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

Operation

```
IF DEST    0
    THEN CF    0
    ELSE CF    1;
FI;
DEST      – (DEST)
```

Flags Affected

The CF flag cleared to 0 if the source operand is 0; otherwise it is set to 1. The OF, SF, ZF, AF, and PF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

MUL—Unsigned Multiply

Opcode	Instruction	Description
F6 /4	MUL *r/m8*	Unsigned multiply (AX AL ∗ *r/m8*)
F7 /4	MUL *r/m16*	Unsigned multiply (DX:AX AX ∗ *r/m16*)
F7 /4	MUL *r/m32*	Unsigned multiply (EDX:EAX EAX ∗ *r/m32*)

Description

Performs an unsigned multiplication of the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand is an implied operand located in register AL, AX or EAX (depending on the size of the operand); the source operand is located in a general-purpose register or a memory location. The action of this instruction and the location of the result depends on the opcode and the operand size as shown in the following table.

Operand Size	Source 1	Source 2	Destination
Byte	AL	r/m8	AX
Word	AX	r/m16	DX:AX
Doubleword	EAX	r/m32	EDX:EAX

The result is stored in register AX, register pair DX:AX, or register pair EDX:EAX (depending on the operand size), with the high-order bits of the product contained in register AH, DX, or EDX, respectively. If the high-order bits of the product are 0, the CF and OF flags are cleared; otherwise, the flags are set.

Operation

```
IF byte operation
   THEN
       AX    AL ∗ SRC
   ELSE (* word or doubleword operation *)
       IF OperandSize    16
           THEN
               DX:AX    AX ∗ SRC
           ELSE (* OperandSize    32 *)
               EDX:EAX    EAX ∗ SRC
       FI;
FI;
```

Flags Affected

The OF and CF flags are cleared to 0 if the upper half of the result is 0; otherwise, they are set to 1. The SF, ZF, AF, and PF flags are undefined.

IMUL—Signed Multiply

Opcode	Instruction	Description
F6 /5	IMUL *r/m8*	AX AL ∗ *r/m* byte
F7 /5	IMUL *r/m16*	DX:AX AX ∗ *r/m* word
F7 /5	IMUL *r/m32*	EDX:EAX EAX ∗ *r/m* doubleword
0F AF /*r*	IMUL *r16,r/m16*	word register word register ∗ *r/m* word
0F AF /*r*	IMUL *r32,r/m32*	doubleword register doubleword register ∗ *r/m* doubleword
6B /*r ib*	IMUL *r16,r/m16,imm8*	word register *r/m16* ∗ sign-extended immediate byte
6B /*r ib*	IMUL *r32,r/m32,imm8*	doubleword register *r/m32* ∗ sign-extended immediate byte
6B /*r ib*	IMUL *r16,imm8*	word register word register ∗ sign-extended immediate byte
6B /*r ib*	IMUL *r32,imm8*	doubleword register doubleword register ∗ sign-extended immediate byte
69 /*r iw*	IMUL *r16,r/m16,imm16*	word register *r/m16* ∗ immediate word
69 /*r id*	IMUL *r32,r/m32,imm32*	doubleword register *r/m32* ∗ immediate doubleword
69 /*r iw*	IMUL *r16,imm16*	word register *r/m16* ∗ immediate word
69 /*r id*	IMUL *r32,imm32*	doubleword register *r/m32* ∗ immediate doubleword

Description

Performs a signed multiplication of two operands. This instruction has three forms, depending on the number of operands.

- **One-operand form.** This form is identical to that used by the MUL instruction. Here, the source operand (in a general-purpose register or memory location) is multiplied by the value in the AL, AX, or EAX register (depending on the operand size) and the product is stored in the AX, DX:AX, or EDX:EAX registers, respectively.
- **Two-operand form.** With this form the destination operand (the first operand) is multiplied by the source operand (second operand). The destination operand is a general-purpose register and the source operand is an immediate value, a general-purpose register, or a memory location. The product is then stored in the destination operand location.
- **Three-operand form.** This form requires a destination operand (the first operand) and two source operands (the second and the third operands). Here, the first source operand (which can be a general-purpose register or a memory location) is multiplied by the second source operand (an immediate value). The product is then stored in the destination operand (a general-purpose register).

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

IMUL—Signed Multiply (Continued)

The CF and OF flags are set when significant bits are carried into the upper half of the result. The CF and OF flags are cleared when the result fits exactly in the lower half of the result.

The three forms of the IMUL instruction are similar in that the length of the product is calculated to twice the length of the operands. With the one-operand form, the product is stored exactly in the destination. With the two- and three- operand forms, however, result is truncated to the length of the destination before it is stored in the destination register. Because of this truncation, the CF or OF flag should be tested to ensure that no significant bits are lost.

The two- and three-operand forms may also be used with unsigned operands because the lower half of the product is the same regardless if the operands are signed or unsigned. The CF and OF flags, however, cannot be used to determine if the upper half of the result is non-zero.

Operation

```
IF (NumberOfOperands    1)
   THEN IF (OperandSize    8)
       THEN
           AX    AL ∗ SRC  (* signed multiplication *)
           IF ((AH    00H) OR (AH    FFH))
               THEN CF    0; OF    0;
               ELSE CF    1; OF    1;
           FI;
       ELSE IF OperandSize    16
           THEN
               DX:AX    AX ∗ SRC  (* signed multiplication *)
               IF ((DX    0000H) OR (DX    FFFFH))
                   THEN CF    0; OF    0;
                   ELSE CF    1; OF    1;
               FI;
           ELSE (* OperandSize    32 *)
               EDX:EAX    EAX ∗ SRC  (* signed multiplication *)
               IF ((EDX    00000000H) OR (EDX    FFFFFFFFH))
                   THEN CF    0; OF    0;
                   ELSE CF    1; OF    1;
               FI;
       FI;
   ELSE IF (NumberOfOperands    2)
       THEN
           temp    DEST ∗ SRC   (* signed multiplication; temp is double DEST size*)
           DEST    DEST ∗ SRC  (* signed multiplication *)
           IF temp   DEST
               THEN CF    1; OF    1;
               ELSE CF    0; OF    0;
           FI;

       ELSE (* NumberOfOperands    3 *)
```

IMUL—Signed Multiply (Continued)

```
            DEST    SRC1 ∗ SRC2  (* signed multiplication *)
            temp    SRC1 ∗ SRC2   (* signed multiplication; temp is double SRC1 size *)
            IF temp   DEST
                THEN CF    1; OF    1;
                ELSE CF    0; OF    0;
            FI;
    FI;
FI;
```

Flags Affected

For the one operand form of the instruction, the CF and OF flags are set when significant bits are carried into the upper half of the result and cleared when the result fits exactly in the lower half of the result. For the two- and three-operand forms of the instruction, the CF and OF flags are set when the result must be truncated to fit in the destination operand size and cleared when the result fits exactly in the destination operand size. The SF, ZF, AF, and PF flags are undefined.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

DIV—Unsigned Divide

Opcode	Instruction	Description
F6 /6	DIV *r/m8*	Unsigned divide AX by *r/m8*, with result stored in AL ← Quotient, AH ← Remainder.
F7 /6	DIV *r/m16*	Unsigned divide DX:AX by *r/m16*, with result stored in AX ← Quotient, DX ← Remainder.
F7 /6	DIV *r/m32*	Unsigned divide EDX:EAX by *r/m32*, with result stored in EAX ← Quotient, EDX ← Remainder.

Description

Divides (unsigned) the value in the AX, DX:AX, or EDX:EAX registers (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, or EDX:EAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size (dividend/divisor). See Table 3-19.

Table 3-19. DIV Action

Operand Size	Dividend	Divisor	Quotient	Remainder	Maximum Quotient
Word/byte	AX	r/m8	AL	AH	255
Doubleword/word	DX:AX	r/m16	AX	DX	65,535
Quadword/doubleword	EDX:EAX	r/m32	EAX	EDX	$2^{32} – 1$

Non-integral results are truncated (chopped) towards 0. The remainder is always less than the divisor in magnitude. Overflow is indicated with the #DE (divide error) exception rather than with the CF flag.

Operation

```
IF SRC = 0
   THEN #DE; (* divide error *)
FI;
IF OperandSize = 8 (* word/byte operation *)
   THEN
      temp ← AX / SRC;
      IF temp > FFH
         THEN #DE; (* divide error *) ;
         ELSE
            AL ← temp;
            AH ← AX MOD SRC;
      FI;
   ELSE
      IF OperandSize = 16 (* doubleword/word operation *)
         THEN
```

```
                temp ← DX:AX / SRC;

                IF temp > FFFFH
                    THEN #DE; (* divide error *) ;
                    ELSE
                        AX ← temp;
                        DX ← DX:AX MOD SRC;
                FI;
            ELSE (* quadword/doubleword operation *)
                temp ← EDX:EAX / SRC;
                IF temp > FFFFFFFFH
                    THEN #DE; (* divide error *) ;
                    ELSE
                        EAX ← temp;
                        EDX ← EDX:EAX MOD SRC;
                FI;
        FI;
FI;
```

Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are undefined.

Protected Mode Exceptions

#DE	If the source operand (divisor) is 0
	If the quotient is too large for the designated register.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#DE	If the source operand (divisor) is 0.
	If the quotient is too large for the designated register.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#DE	If the source operand (divisor) is 0.
	If the quotient is too large for the designated register.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

INDEXED ADDRESSING MODES

Indexed Addressing

- Operands of the form: [ESI + ECX*4 + DISP]
- ESI = Base Register
- ECX = Index Register
- 4 = Scale factor
- DISP = Displacement
- The operand is in memory
- The address of the memory location is
  ESI + ECX*4 + DISP

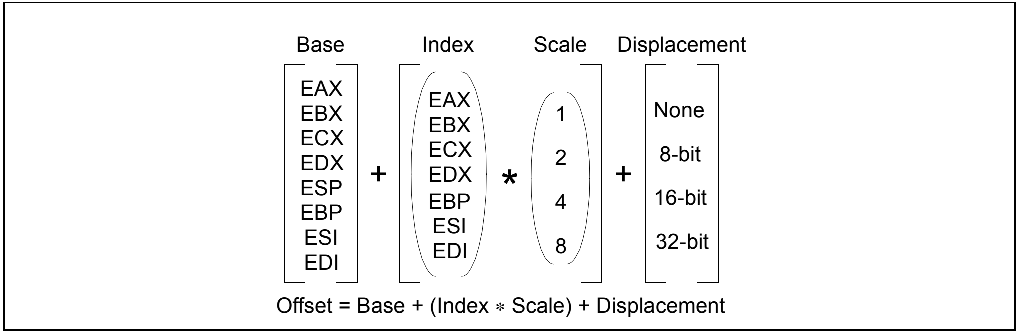


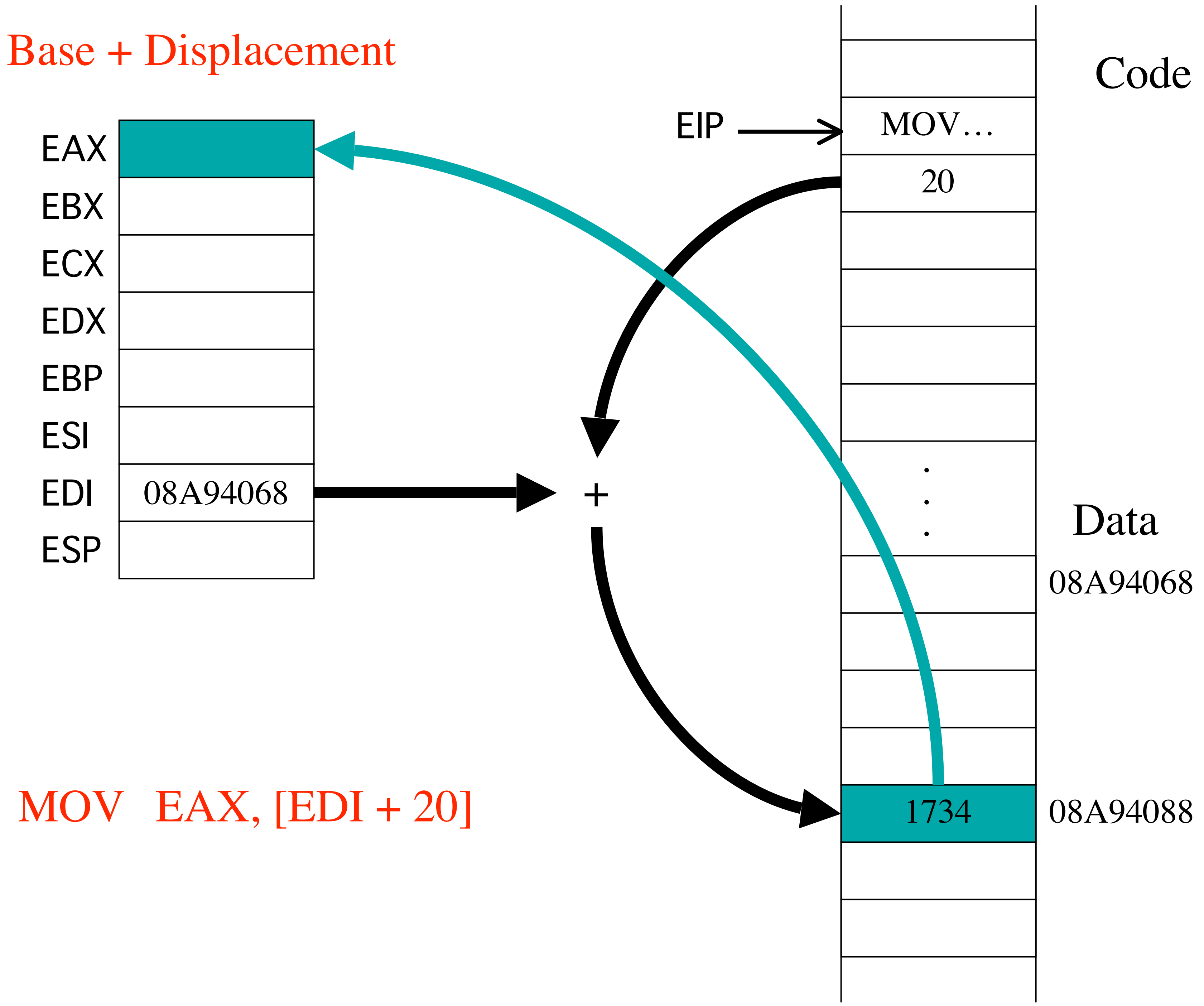
Figure 3-9. Offset (or Effective Address) Computation

The uses of general-purpose registers as base or index components are restricted in the following manner:

- The ESP register cannot be used as an index register.
- When the ESP or EBP register is used as the base, the SS segment is the default segment. In all other cases, the DS segment is the default segment.

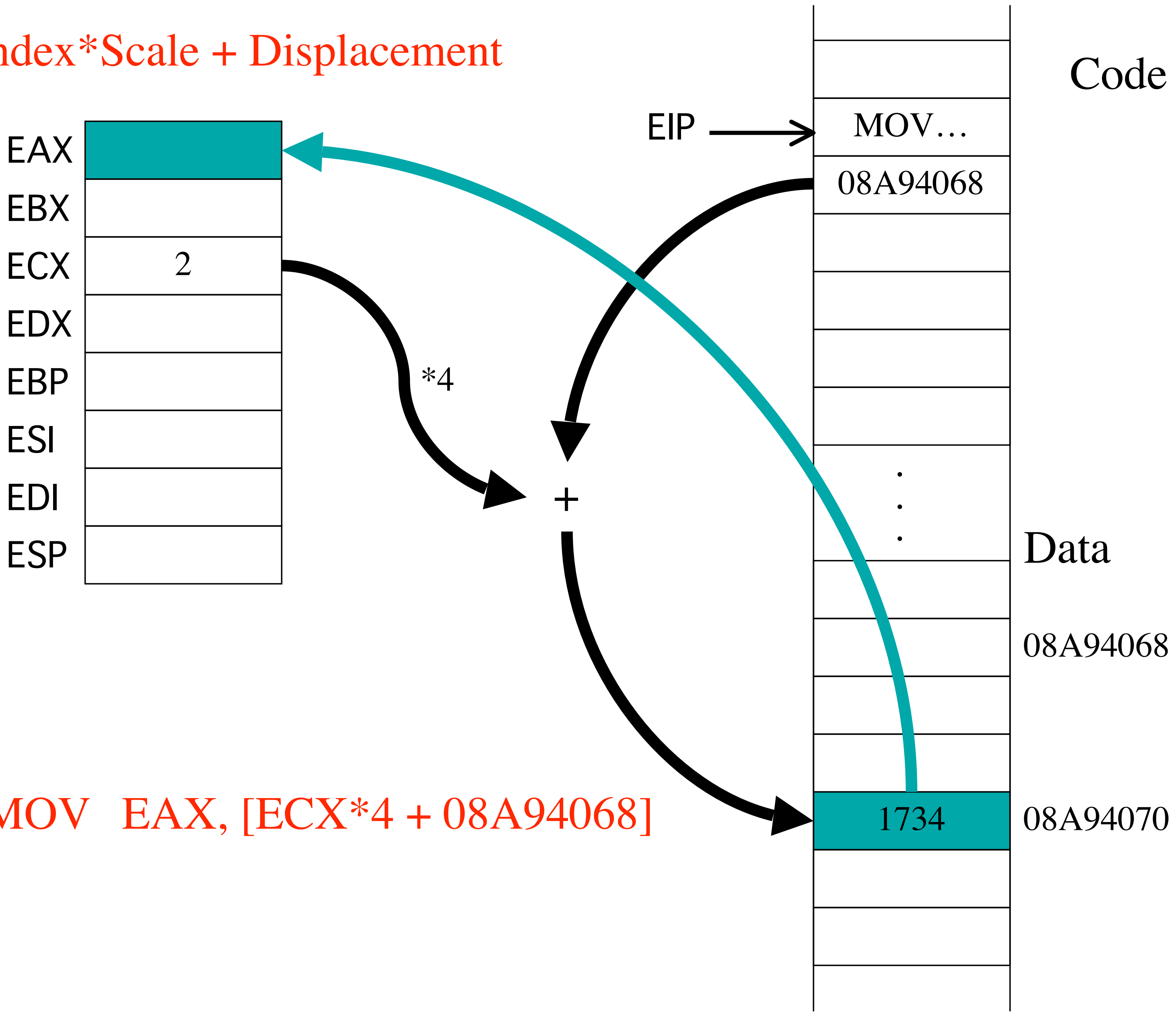
The base, index, and displacement components can be used in any combination, and any of these components can be null. A scale factor may be used only when an index also is used. Each possible combination is useful for data structures commonly used by programmers in high-level languages and assembly language. The following addressing modes suggest uses for common combinations of address components.

Base + Displacement



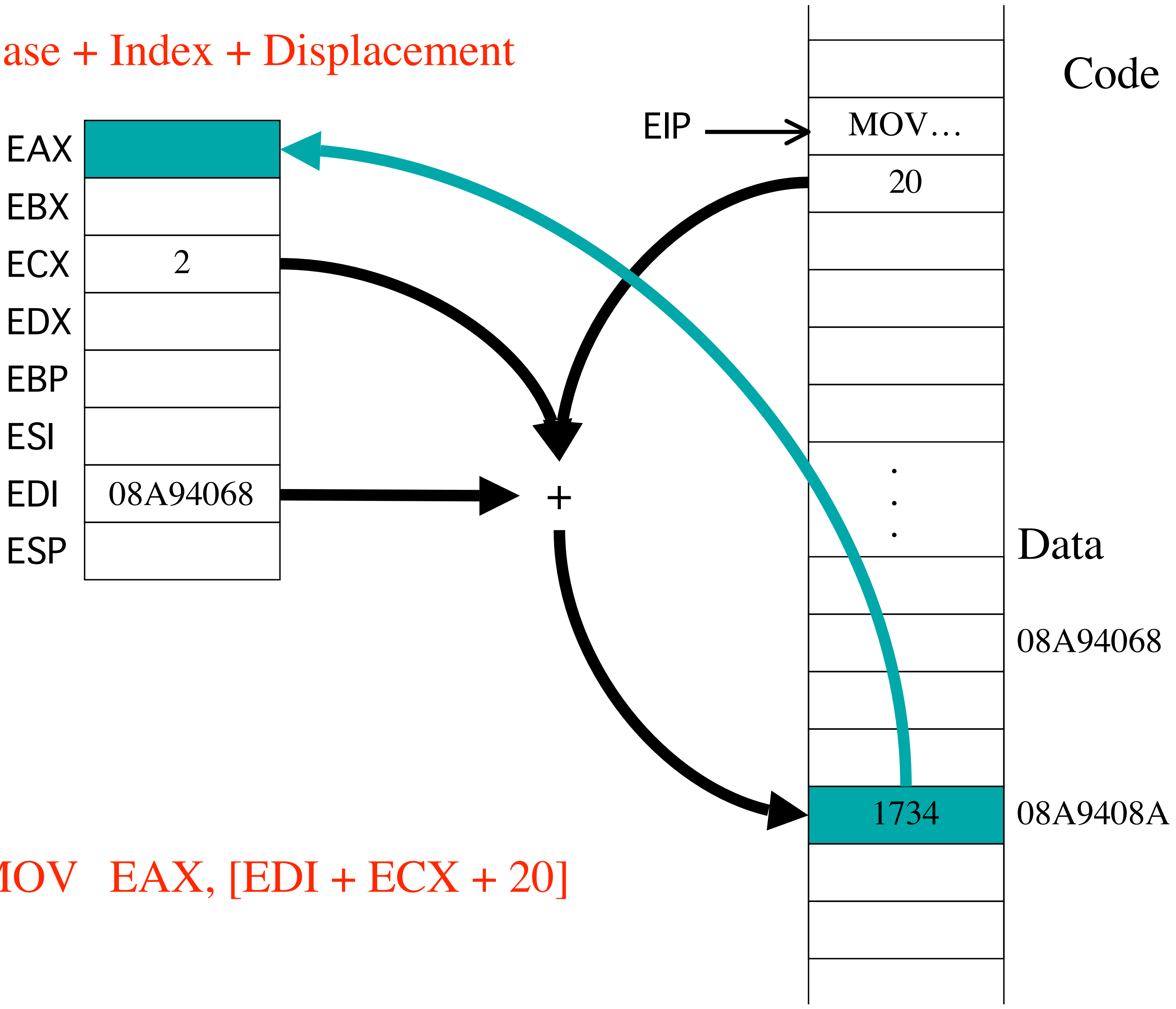
MOV   EAX, [EDI + 20]

Index*Scale + Displacement



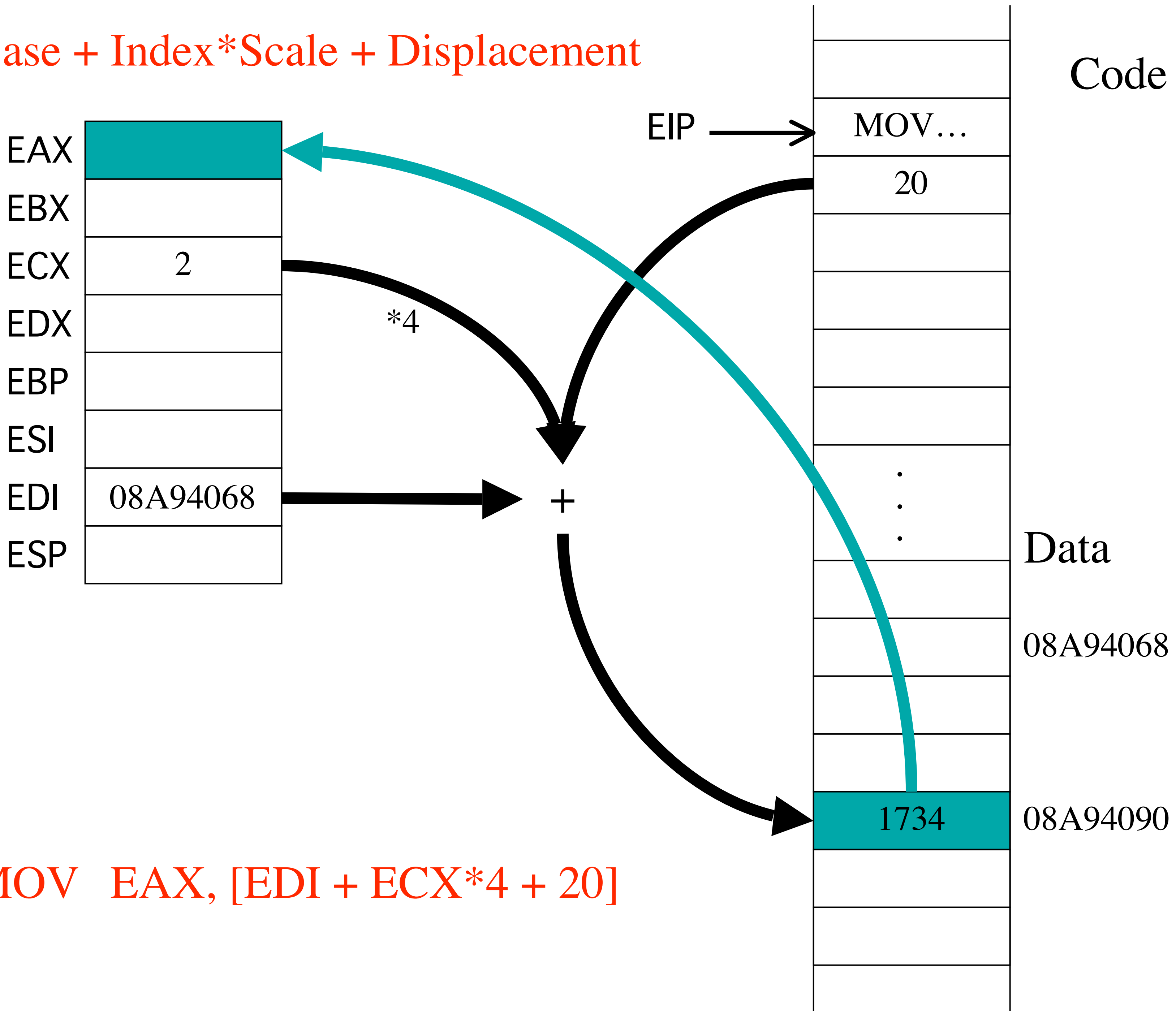
MOV   EAX, [ECX*4 + 08A94068]

Base + Index + Displacement



MOV EAX, [EDI + ECX + 20]

Base + Index*Scale + Displacement



MOV EAX, [EDI + ECX*4 + 20]

Typical Uses for Indexed Addressing

- **Base + Displacement**
  - ◇ access character in a string or field of a record
  - ◇ access a local variable in function call stack
- **Index*Scale + Displacement**
  - ◇ access items in an array where size of item is 2, 4 or 8 bytes
- **Base + Index + Displacement**
  - ◇ access two dimensional array (displacement has address of array)
  - ◇ access an array of records (displacement has offset of field in a record)
- **Base + (Index*Scale) + Displacement**
  - ◇ access two dimensional array where size of item is 2, 4 or 8 bytes

```
; File: index1.asm
;
; This program demonstrates the use of an indexed addressing mode
; to access array elements.
;
; This program has no I/O. Use the debugger to examine its effects.
;
        SECTION .data                   ; Data section

arr:    dd 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ; ten 32-bit words
base:   equ arr - 4


        SECTION .text                   ; Code section.
        global _start
_start: nop                             ; Entry point.

        ; Add 5 to each element of the array stored in arr.
        ; Simulate:
        ;
        ;  for (i = 0 ; i < 10 ; i++) {
        ;     arr[i] += 5 ;
        ;  }

init1:  mov     ecx, 0                  ; ecx simulates i
loop1:  cmp     ecx, 10                 ; i < 10 ?
        jge     done1
        add     [ecx*4+arr], dword 5    ; arr[i] += 5
        inc     ecx                     ; i++
        jmp     loop1
done1:

        ; more idiomatic for an assembly language program
init2:  mov     ecx, 9                  ; last array elt's index
loop2:  add     [ecx*4+arr], dword 5
        dec     ecx
        jge     loop2                   ; again if ecx >= 0


        ; another way
init3:  mov     edi, base               ; base computed by ld
        mov     ecx, 10                 ; for(i=10 ; i>0 ; i--)
loop3:  add     [edi+ecx*4], dword 5
        loop    loop3                   ; loop = dec ecx, jne

alldone:
        mov     ebx, 0                  ; exit code, 0=normal
        mov     eax, 1                  ; Exit.
        int     80H                     ; Call kernel.
```

```
Script started on Fri Sep 19 13:06:02 2003
linux3% nasm -f elf index1.asm
linux3% ld index1.o

linux3% gdb a.out
GNU gdb Red Hat Linux (5.2-2)
...
(gdb) break *init1
Breakpoint 1 at 0x8048081
(gdb) break *init2
Breakpoint 2 at 0x8048099
(gdb) break *init3
Breakpoint 3 at 0x80480ac
(gdb) break * alldone
Breakpoint 4 at 0x80480bf
(gdb) run
Starting program: /afs/umbc.edu/users/c/h/chang/home/asm/a.out

Breakpoint 1, 0x08048081 in init1 ()
(gdb) x/10wd &arr
0x80490cc <arr>:        0       1       2       3
0x80490dc <arr+16>:     4       5       6       7
0x80490ec <arr+32>:     8       9
(gdb) cont
Continuing.

Breakpoint 2, 0x08048099 in init2 ()
(gdb) x/10wd &arr
0x80490cc <arr>:        5       6       7       8
0x80490dc <arr+16>:     9       10      11      12
0x80490ec <arr+32>:     13      14
(gdb) cont
Continuing.

Breakpoint 3, 0x080480ac in init3 ()
(gdb) x/10wd &arr
0x80490cc <arr>:        10      11      12      13
0x80490dc <arr+16>:     14      15      16      17
0x80490ec <arr+32>:     18      19
(gdb) cont
Continuing.

Breakpoint 4, 0x080480bf in alldone ()
(gdb) x/10wd &arr
0x80490cc <arr>:        15      16      17      18
0x80490dc <arr+16>:     19      20      21      22
0x80490ec <arr+32>:     23      24
(gdb) cont
Continuing.

Program exited normally.
(gdb) quit
linux3% exit
exit

Script done on Fri Sep 19 13:07:41 2003
```

```
; File: index2.asm
;
; This program demonstrates the use of an indexed addressing mode
; to access 2 dimensional array elements.
;
; This program has no I/O. Use the debugger to examine its effects.
;
        SECTION .data                   ; Data section

        ; simulates a 2-dim array
twodim:
row1:   dd 00, 01, 02, 03, 04, 05, 06, 07, 08, 09
row2:   dd 10, 11, 12, 13, 14, 15, 16, 17, 18, 19
        dd 20, 21, 22, 23, 24, 25, 26, 27, 28, 29
        dd 30, 31, 32, 33, 34, 35, 36, 37, 38, 39
        dd 40, 41, 42, 43, 44, 45, 46, 47, 48, 49
        dd 50, 51, 52, 53, 54, 55, 56, 57, 58, 59
        dd 60, 61, 62, 63, 64, 65, 66, 67, 68, 69
        dd 70, 71, 72, 73, 74, 75, 76, 77, 78, 79
        dd 80, 81, 82, 83, 84, 85, 86, 87, 88, 89
        dd 90, 91, 92, 93, 94, 95, 96, 97, 98, 99

rowlen: equ row2 - row1

        SECTION .text                   ; Code section.
        global _start
_start: nop                             ; Entry point.

        ; Add 5 to each element of row 7. Simulate:
        ;
        ;  for (i = 0 ; i < 10 ; i++) {
        ;     towdim[7][i] += 5 ;
        ;  }

init1:  mov     ecx, 0                  ; ecx simulates i
        mov     eax, rowlen             ; offset of twodim[7][0]
        mov     edx, 7
        mul     edx                     ; eax := eax * edx
        jc      alldone                 ; 64-bit product is bad

loop1:  cmp     ecx, 10                 ; i < 10 ?
        jge     done1
        add     [eax+4*ecx+twodim], dword 5
        inc     ecx                     ; i++
        jmp     loop1
done1:

alldone:
        mov     ebx, 0                  ; exit code, 0=normal
        mov     eax, 1                  ; Exit.
        int     80H                     ; Call kernel.
```

```
Script started on Fri Sep 19 13:19:22 2003
linux3% nasm -f elf index2.asm
linux3% ld index2.o
linux3%
linux3% gdb a.out
GNU gdb Red Hat Linux (5.2-2)
...
(gdb) break *init1
Breakpoint 1 at 0x8048081
(gdb) break *alldone
Breakpoint 2 at 0x80480a7
(gdb) run
Starting program: /afs/umbc.edu/users/c/h/chang/home/asm/a.out

Breakpoint 1, 0x08048081 in init1 ()
(gdb) x/10wd &twodim
0x80490b4 <twodim>:     0       1       2       3
0x80490c4 <twodim+16>:  4       5       6       7
0x80490d4 <twodim+32>:  8       9
(gdb) x/10wd &twodim+60
0x80491a4 <row2+200>:   60      61      62      63
0x80491b4 <row2+216>:   64      65      66      67
0x80491c4 <row2+232>:   68      69
(gdb)
0x80491cc <row2+240>:   70      71      72      73
0x80491dc <row2+256>:   74      75      76      77
0x80491ec <row2+272>:   78      79
(gdb)
0x80491f4 <row2+280>:   80      81      82      83
0x8049204 <row2+296>:   84      85      86      87
0x8049214 <row2+312>:   88      89
(gdb) cont
Continuing.

Breakpoint 2, 0x080480a7 in done1 ()
(gdb) x/10wd &twodim+60
0x80491a4 <row2+200>:   60      61      62      63
0x80491b4 <row2+216>:   64      65      66      67
0x80491c4 <row2+232>:   68      69
(gdb)
0x80491cc <row2+240>:   75      76      77      78
0x80491dc <row2+256>:   79      80      81      82
0x80491ec <row2+272>:   83      84
(gdb)
0x80491f4 <row2+280>:   80      81      82      83
0x8049204 <row2+296>:   84      85      86      87
0x8049214 <row2+312>:   88      89
(gdb) cont
Continuing.

Program exited normally.
(gdb) quit
linux3% exit
exit

Script done on Fri Sep 19 13:20:35 2003
```

```c
#include <stdio.h>

int main() {
   char choice ;

   printf("Make a selection [a, b, c, d or f]: ") ;
   scanf("%c", &choice) ;

   switch (choice) {

      case 'a':
         printf("Good choice!\n") ;
         break ;
      case 'b':
         printf("b is my favorite\n") ;
         break ;
      case 'c':
         printf("c is a popular selection\n") ;
         break ;
      case 'd':
         printf("Sorry, we are fresh out of d\n") ;
         break ;
      case 'f':
         printf("Not a good day for f. Try something else.\n") ;
         break ;
      default :
         printf("Please pick one of the choices!\n") ;
   }

   return 0 ;
}
```

```
; File: switch.asm
;
; Demonstrate jump tables
;
;

%define STDIN 0
%define STDOUT 1
%define SYSCALL_EXIT  1
%define SYSCALL_READ  3
%define SYSCALL_WRITE 4
%define BUFLEN 256


        SECTION .data                   ; initialized data section

mesg1:  db "Make a selection [a, b, c, d or f]: "
len1:   equ $ - mesg1

err_mesg:       db "Read Error", 10
err_len:        equ $ - err_mesg


mesga:  db "Good choice!", 10
lena:   equ $ - mesga

mesgb:  db "b is my favorite", 10
lenb:   equ $ - mesgb

mesgc:  db "c is a popular selection", 10
lenc:   equ $ - mesgc

mesgd:  db "Sorry, we are fresh out of d", 10
lend:   equ $ - mesgd

mesgf:   db "Not a good day for f. Try something else.", 10
lenf:   equ $-mesgf

dflt_mesg: db "Please pick one of the choices!", 10
dflt_len:  equ $ - dflt_mesg




jump_table:
        dd do_a         ; dd = "define double" = 32-bit data
        dd do_b
        dd do_c
        dd do_d
        dd do_default
        dd do_f



        SECTION .bss                    ; uninitialized data section
buf:    resb BUFLEN                     ; buffer for read
```

```
        SECTION .text                   ; Code section.
        global  _start                  ; let loader see entry point

_start:

        ; prompt user for input
        ;
        mov     eax, SYSCALL_WRITE      ; write function
        mov     ebx, STDOUT             ; Arg1: file descriptor
        mov     ecx, mesg1              ; Arg2: addr of message
        mov     edx, len1               ; Arg3: length of message
        int     080h                    ; ask kernel to write

        ; read user input
        ;
        mov     eax, SYSCALL_READ       ; read function
        mov     ebx, STDIN              ; Arg 1: file descriptor
        mov     ecx, buf                ; Arg 2: address of buffer
        mov     edx, BUFLEN             ; Arg 3: buffer length
        int     080h

        ; error check
        ;
        cmp     eax, 0                  ; check if any chars read
        jg      read_OK                 ; >0 chars read = OK
        mov     eax, SYSCALL_WRITE      ; ow print error mesg
        mov     ebx, STDOUT
        mov     ecx, err_mesg
        mov     edx, err_len
        int     080h
        jmp     exit                    ; skip over rest
read_OK:

        mov     eax, 0
        mov     al, [buf]               ; get a character
        cmp     al, 'a'                 ; below 'a'?
        jb      do_default
        cmp     al, 'f'                 ; above 'f'?
        ja      do_default

        sub     eax, 'a'                ; calculate offset
        jmp     [4*eax + jump_table]


do_a:
        mov     eax, SYSCALL_WRITE      ; write message for choice a
        mov     ebx, STDOUT
        mov     ecx, mesga
        mov     edx, lena
        int     080h
        jmp     exit

do_b:
        mov     eax, SYSCALL_WRITE      ; write message for choice b
        mov     ebx, STDOUT
        mov     ecx, mesgb
        mov     edx, lenb
        int     080h
        jmp     exit
```

```
do_c:
        mov     eax, SYSCALL_WRITE      ; write message for choice c
        mov     ebx, STDOUT
        mov     ecx, mesgc
        mov     edx, lenc
        int     080h
        jmp     exit

do_d:
        mov     eax, SYSCALL_WRITE      ; write message for choice d
        mov     ebx, STDOUT
        mov     ecx, mesgd
        mov     edx, lend
        int     080h
        jmp     exit

do_f:
        mov     eax, SYSCALL_WRITE      ; write message for choice f
        mov     ebx, STDOUT
        mov     ecx, mesgf
        mov     edx, lenf
        int     080h
        jmp     exit

do_default:
        mov     eax, SYSCALL_WRITE      ; write message for default
        mov     ebx, STDOUT
        mov     ecx, dflt_mesg
        mov     edx, dflt_len
        int     080h
        jmp     exit


        ; final exit
        ;
exit:   mov     EAX, SYSCALL_EXIT       ; exit function
        mov     EBX, 0                  ; exit code, 0=normal
        int     080h                    ; ask kernel to take over
```

```
                                 ; File: switch.asm
                                 ;
                                 ; Demonstrate jump tables
                                 ;
                                 ;

                                 %define STDIN 0
                                 %define STDOUT 1
                                 %define SYSCALL_EXIT  1
                                 %define SYSCALL_READ  3
                                 %define SYSCALL_WRITE 4
                                 %define BUFLEN 256


                                         SECTION .data                   ; initialized data section

00000000 4D616B652061207365-     mesg1:  db "Make a selection [a, b, c, d or f]: "
00000009 6C656374696F6E205B-
00000012 612C20622C20632C20-
0000001B 64206F7220665D3A20
                                 len1:   equ $ - mesg1

00000024 526561642045727260F-    err_mesg:       db "Read Error", 10
0000002D 720A
                                 err_len:        equ $ - err_mesg


0000002F 476F6F642063686F69-     mesga:  db "Good choice!", 10
00000038 6365210A
                                 lena:   equ $ - mesga

0000003C 62206973206D792066-     mesgb:  db "b is my favorite", 10
00000045 61766F726974650A
                                 lenb:   equ $ - mesgb

0000004D 632069732061207060F-    mesgc:  db "c is a popular selection", 10
00000056 70756C61722073656C-
0000005F 656374696F6E0A
                                 lenc:   equ $ - mesgc

00000066 536F7272792C207765-     mesgd:  db "Sorry, we are fresh out of d", 10
0000006F 206172652066726573-
00000078 68206F7574206F6620-
00000081 640A
                                 lend:   equ $ - mesgd

00000083 4E6F74206120676F6F-     mesgf:   db "Not a good day for f. Try something else.", 10
0000008C 642064617920666F72-
00000095 20662E205472792073-
0000009E 6F6D657468696E6720-
```

```
000000A7 656C73652E0A
                                 lenf:   equ $-mesgf

000000AD 506C65617365207069-     dflt_mesg: db "Please pick one of the choices!", 10
000000B6 636B206F6E65206F66-
000000BF 207468652063686F69-
000000C8 636573210A
                                 dflt_len:  equ $ - dflt_mesg

                                 jump_table:
000000CD [74000000]                      dd do_a         ; dd = "define double" = 32-bit data
000000D1 [8F000000]                      dd do_b
000000D5 [AA000000]                      dd do_c
000000D9 [C5000000]                      dd do_d
000000DD [FB000000]                      dd do_default
000000E1 [E0000000]                      dd do_f


                                         SECTION .bss                    ; uninitialized data section
00000000 <res 00000100>          buf:    resb BUFLEN                     ; buffer for read


                                         SECTION .text                   ; Code section.
                                         global  _start                  ; let loader see entry point

                                 _start:

                                         ; prompt user for input
                                         ;
00000000 B804000000                      mov     eax, SYSCALL_WRITE      ; write function
00000005 BB01000000                      mov     ebx, STDOUT             ; Arg1: file descriptor
0000000A B9[00000000]                    mov     ecx, mesg1              ; Arg2: addr of message
0000000F BA24000000                      mov     edx, len1               ; Arg3: length of message
00000014 CD80                            int     080h                    ; ask kernel to write

                                         ; read user input
                                         ;
00000016 B803000000                      mov     eax, SYSCALL_READ       ; read function
0000001B BB00000000                      mov     ebx, STDIN              ; Arg 1: file descriptor
00000020 B9[00000000]                    mov     ecx, buf                ; Arg 2: address of buffer
00000025 BA00010000                      mov     edx, BUFLEN             ; Arg 3: buffer length
0000002A CD80                            int     080h

                                         ; error check
                                         ;
0000002C 3D00000000                      cmp     eax, 0                  ; check if any chars read
00000031 7F1B                            jg      read_OK                 ; >0 chars read = OK
00000033 B804000000                      mov     eax, SYSCALL_WRITE      ; ow print error mesg
00000038 BB01000000                      mov     ebx, STDOUT
0000003D B9[24000000]                    mov     ecx, err_mesg
```

```
00000042 BA0B000000                      mov     edx, err_len
00000047 CD80                            int     080h
00000049 E9C8000000                      jmp     exit                    ; skip over rest
                                 read_OK:

0000004E B800000000                      mov     eax, 0
00000053 A0[00000000]                    mov     al, [buf]               ; get a character
00000058 3C61                            cmp     al, 'a'                 ; below 'a'?
0000005A 0F829B000000                    jb      do_default
00000060 3C66                            cmp     al, 'f'                 ; above 'f'?
00000062 0F8793000000                    ja      do_default

00000068 2D61000000                      sub     eax, 'a'                ; calculate offset
0000006D FF2485[CD000000]                jmp     [4*eax + jump_table]


                                 do_a:
00000074 B804000000                      mov     eax, SYSCALL_WRITE      ; write message for choice a
00000079 BB01000000                      mov     ebx, STDOUT
0000007E B9[2F000000]                    mov     ecx, mesga
00000083 BA0D000000                      mov     edx, lena
00000088 CD80                            int     080h
0000008A E987000000                      jmp     exit

                                 do_b:
0000008F B804000000                      mov     eax, SYSCALL_WRITE      ; write message for choice b
00000094 BB01000000                      mov     ebx, STDOUT
00000099 B9[3C000000]                    mov     ecx, mesgb
0000009E BA11000000                      mov     edx, lenb
000000A3 CD80                            int     080h
000000A5 E96C000000                      jmp     exit

                                 do_c:
000000AA B804000000                      mov     eax, SYSCALL_WRITE      ; write message for choice c
000000AF BB01000000                      mov     ebx, STDOUT
000000B4 B9[4D000000]                    mov     ecx, mesgc
000000B9 BA19000000                      mov     edx, lenc
000000BE CD80                            int     080h
000000C0 E951000000                      jmp     exit

                                 do_d:
000000C5 B804000000                      mov     eax, SYSCALL_WRITE      ; write message for choice d
000000CA BB01000000                      mov     ebx, STDOUT
000000CF B9[66000000]                    mov     ecx, mesgd
000000D4 BA1D000000                      mov     edx, lend
000000D9 CD80                            int     080h
000000DB E936000000                      jmp     exit

                                 do_f:
000000E0 B804000000                      mov     eax, SYSCALL_WRITE      ; write message for choice f
```

```
000000E5 BB01000000                      mov     ebx, STDOUT
000000EA B9[83000000]                    mov     ecx, mesgf
000000EF BA2A000000                      mov     edx, lenf
000000F4 CD80                            int     080h
000000F6 E91B000000                      jmp     exit

                                 do_default:
000000FB B804000000                      mov     eax, SYSCALL_WRITE      ; write message for default
00000100 BB01000000                      mov     ebx, STDOUT
00000105 B9[AD000000]                    mov     ecx, dflt_mesg
0000010A BA20000000                      mov     edx, dflt_len
0000010F CD80                            int     080h
00000111 E900000000                      jmp     exit


                                         ; final exit
                                         ;
00000116 B801000000              exit:   mov     EAX, SYSCALL_EXIT       ; exit function
0000011B BB00000000                      mov     EBX, 0                  ; exit code, 0=normal
00000120 CD80                            int     080h                    ; ask kernel to take over
```

SOME
I386
STRING INSTRUCTIONS

i386 String Instructions

- Special instructions for searching & copying strings
- Assumes that AL holds the data
- Assumes that ECX holds the "count"
- Assumes that ESI and/or EDI point to the string(s)
- Some examples (there are many others):
  - LODS: loads AL with [ESI], then increments or decrements ESI
  - STOS: stores AL in [EDI], then increments or decrements EDI
  - CLD/STD: clears/sets direction flag DF. Makes LODS & STOS auto-inc/dec.
  - LOOP: decrements ECX. Jumps to label if ECX != 0 after decrement.
  - SCAS: compares AL with [EDI], sets status flags, auto-inc/dec EDI.
  - REP: Repeats a string instruction

LODS/LODSB/LODSW/LODSD—Load String

Opcode	Instruction	Description
AC	LODS m8	Load byte at address DS:(E)SI into AL
AD	LODS m16	Load word at address DS:(E)SI into AX
AD	LODS m32	Load doubleword at address DS:(E)SI into EAX
AC	LODSB	Load byte at address DS:(E)SI into AL
AD	LODSW	Load word at address DS:(E)SI into AX
AD	LODSD	Load doubleword at address DS:(E)SI into EAX

Description

Loads a byte, word, or doubleword from the source operand into the AL, AX, or EAX register, respectively. The source operand is a memory location, the address of which is read from the DS:EDI or the DS:SI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The DS segment may be overridden with a segment override prefix.

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operands" form and the "no-operands" form. The explicit-operands form (specified with the LODS mnemonic) allows the source operand to be specified explicitly. Here, the source operand should be a symbol that indicates the size and location of the source value. The destination operand is then automatically selected to match the size of the source operand (the AL register for byte operands, AX for word operands, and EAX for doubleword operands). This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the DS:(E)SI registers, which must be loaded correctly before the load string instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the LODS instructions. Here also DS:(E)SI is assumed to be the source operand and the AL, AX, or EAX register is assumed to be the destination operand. The size of the source and destination operands is selected with the mnemonic: LODSB (byte loaded into register AL), LODSW (word loaded into AX), or LODSD (doubleword loaded into EAX).

After the byte, word, or doubleword is transferred from the memory location into the AL, AX, or EAX register, the (E)SI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI register is incremented; if the DF flag is 1, the ESI register is decremented.) The (E)SI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The LODS, LODSB, LODSW, and LODSD instructions can be preceded by the REP prefix for block loads of ECX bytes, words, or doublewords. More often, however, these instructions are used within a LOOP construct because further processing of the data moved into the register is usually necessary before the next transfer can be made. See "REP/REPE/REPZ/REPNE /REPNZ—Repeat String Operation Prefix" in this chapter for a description of the REP prefix.

LODS/LODSB/LODSW/LODSD—Load String (Continued)

Operation

```
IF (byte load)
    THEN
        AL     SRC; (* byte load *)
            THEN IF DF     0
                THEN (E)SI     (E)SI + 1;
                ELSE (E)SI     (E)SI – 1;
            FI;
    ELSE IF (word load)
        THEN
            AX     SRC; (* word load *)
                THEN IF DF     0
                    THEN (E)SI     (E)SI + 2;
                    ELSE (E)SI     (E)SI – 2;
                FI;
        ELSE (* doubleword transfer *)
            EAX     SRC; (* doubleword load *)
                THEN IF DF     0
                    THEN (E)SI     (E)SI + 4;
                    ELSE (E)SI     (E)SI – 4;
                FI;
    FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

STOS/STOSB/STOSW/STOSD—Store String

Opcode	Instruction	Description
AA	STOS m8	Store AL at address ES:(E)DI
AB	STOS m16	Store AX at address ES:(E)DI
AB	STOS m32	Store EAX at address ES:(E)DI
AA	STOSB	Store AL at address ES:(E)DI
AB	STOSW	Store AX at address ES:(E)DI
AB	STOSD	Store EAX at address ES:(E)DI

Description

Stores a byte, word, or doubleword from the AL, AX, or EAX register, respectively, into the destination operand. The destination operand is a memory location, the address of which is read from either the ES:EDI or the ES:DI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The ES segment cannot be overridden with a segment override prefix.

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the STOS mnemonic) allows the destination operand to be specified explicitly. Here, the destination operand should be a symbol that indicates the size and location of the destination value. The source operand is then automatically selected to match the size of the destination operand (the AL register for byte operands, AX for word operands, and EAX for doubleword operands). This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the destination operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the ES:(E)DI registers, which must be loaded correctly before the store string instruction is executed.

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the STOS instructions. Here also ES:(E)DI is assumed to be the destination operand and the AL, AX, or EAX register is assumed to be the source operand. The size of the destination and source operands is selected with the mnemonic: STOSB (byte read from register AL), STOSW (word from AX), or STOSD (doubleword from EAX).

After the byte, word, or doubleword is transferred from the AL, AX, or EAX register to the memory location, the (E)DI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)DI register is incremented; if the DF flag is 1, the (E)DI register is decremented.) The (E)DI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

STOS/STOSB/STOSW/STOSD—Store String (Continued)

The STOS, STOSB, STOSW, and STOSD instructions can be preceded by the REP prefix for block loads of ECX bytes, words, or doublewords. More often, however, these instructions are used within a LOOP construct because data needs to be moved into the AL, AX, or EAX register before it can be stored. See “REP/REPE/REPZ/REPNE /REPNZ—Repeat String Operation Prefix” in this chapter for a description of the REP prefix.

Operation

```
IF (byte store)
   THEN
       DEST     AL;
           THEN IF DF     0
               THEN (E)DI     (E)DI + 1;
               ELSE (E)DI     (E)DI – 1;
           FI;
   ELSE IF (word store)
       THEN
           DEST     AX;
               THEN IF DF     0
                   THEN (E)DI     (E)DI + 2;
                   ELSE (E)DI     (E)DI – 2;
               FI;
       ELSE (* doubleword store *)
           DEST     EAX;
               THEN IF DF     0
                   THEN (E)DI     (E)DI + 4;
                   ELSE (E)DI     (E)DI – 4;
               FI;
   FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable segment.
	If a memory operand effective address is outside the limit of the ES segment.
	If the ES register contains a null segment selector.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

CLD—Clear Direction Flag

Opcode	Instruction	Description
FC	CLD	Clear DF flag

Description

Clears the DF flag in the EFLAGS register. When the DF flag is set to 0, string operations increment the index registers (ESI and/or EDI).

Operation

```
DF      0;
```

Flags Affected

The DF flag is cleared to 0. The CF, OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

None.

STD—Set Direction Flag

Opcode	Instruction	Description
FD	STD	Set DF flag

Description

Sets the DF flag in the EFLAGS register. When the DF flag is set to 1, string operations decrement the index registers (ESI and/or EDI).

Operation

```
DF     1;
```

Flags Affected

The DF flag is set. The CF, OF, ZF, SF, AF, and PF flags are unaffected.

Operation

```
DF     1;
```

Exceptions (All Operating Modes)

None.

LOOP/LOOP*cc*—Loop According to ECX Counter

Opcode	Instruction	Description
E2 *cb*	LOOP *rel8*	Decrement count; jump short if count 0
E1 *cb*	LOOPE *rel8*	Decrement count; jump short if count 0 and ZF=1
E1 *cb*	LOOPZ *rel8*	Decrement count; jump short if count 0 and ZF=1
E0 *cb*	LOOPNE *rel8*	Decrement count; jump short if count 0 and ZF=0
E0 *cb*	LOOPNZ *rel8*	Decrement count; jump short if count 0 and ZF=0

Description

Performs a loop operation using the ECX or CX register as a counter. Each time the LOOP instruction is executed, the count register is decremented, then checked for 0. If the count is 0, the loop is terminated and program execution continues with the instruction following the LOOP instruction. If the count is not zero, a near jump is performed to the destination (target) operand, which is presumably the instruction at the beginning of the loop. If the address-size attribute is 32 bits, the ECX register is used as the count register; otherwise the CX register is used.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register). This offset is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit immediate value, which is added to the instruction pointer. Offsets of –128 to +127 are allowed with this instruction.

Some forms of the loop instruction (LOOP*cc*) also accept the ZF flag as a condition for terminating the loop before the count reaches zero. With these forms of the instruction, a condition code (*cc*) is associated with each instruction to indicate the condition being tested for. Here, the LOOP*cc* instruction itself does not affect the state of the ZF flag; the ZF flag is changed by other instructions in the loop.

Operation

```
IF AddressSize    32
   THEN
       Count is ECX;
   ELSE (* AddressSize    16 *)
       Count is CX;
FI;
Count    Count – 1;

IF instruction is not LOOP
   THEN
       IF (instruction    LOOPE) OR (instruction    LOOPZ)
           THEN
               IF (ZF =1) AND (Count   0)
                   THEN BranchCond    1;
                   ELSE BranchCond    0;
```

LOOP/LOOP*cc*—Loop According to ECX Counter (Continued)

```
                FI;
        FI;
        IF (instruction     LOOPNE) OR (instruction     LOOPNZ)
            THEN
                IF (ZF =0 ) AND (Count    0)
                    THEN BranchCond     1;
                    ELSE BranchCond     0;
                FI;
        FI;
    ELSE (* instruction     LOOP *)
        IF (Count    0)
            THEN BranchCond     1;
            ELSE BranchCond     0;
        FI;
FI;
IF BranchCond     1
    THEN
         EIP     EIP + SignExtend(DEST);
        IF OperandSize     16
            THEN
                EIP     EIP AND 0000FFFFH;
            ELSE (* OperandSize = 32 *)
                IF EIP < CS.Base OR EIP > CS.Limit
                    #GP
        FI;
    ELSE
        Terminate loop and continue program execution at EIP;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the offset being jumped to is beyond the limits of the CS segment.

Real-Address Mode Exceptions

#GP	If the offset being jumped to is beyond the limits of the CS segment or is outside of the effective address space from 0 to FFFFH. This condition can occur if a 32-bit address size override prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

```
; File: toupper2.asm last updated 09/26/2001
;
; Convert user input to upper case.
; This version uses some special looping instructions.
;
; Assemble using NASM:  nasm -f elf toupper2.asm
; Link with ld:  ld toupper2.o
;

; [... same old, same old ...]


        ; Loop for upper case conversion
        ; assuming rlen > 0
        ;
L1_init:
        mov     ecx, [rlen]                 ; initialize count
        mov     esi, buf                    ; point to start of buffer
        mov     edi, newstr                 ; point to start of new str
        cld                                 ; clear dir. flag, inc ptrs

L1_top:
        lodsb                               ; load al w char in [esi++]
        cmp     al, 'a'                     ; less than 'a'?
        jb      L1_cont
        cmp     al, 'z'                     ; more than 'z'?
        ja      L1_cont
        and     al, 11011111b               ; convert to uppercase

L1_cont:
        stosb                               ; store al in [edi++]
        loop    L1_top                      ; loop if --ecx > 0
L1_end:
```

SCAS/SCASB/SCASW/SCASD—Scan String

Opcode	Instruction	Description
AE	SCAS m8	Compare AL with byte at ES:(E)DI and set status flags
AF	SCAS m16	Compare AX with word at ES:(E)DI and set status flags
AF	SCAS m32	Compare EAX with doubleword at ES(E)DI and set status flags
AE	SCASB	Compare AL with byte at ES:(E)DI and set status flags
AF	SCASW	Compare AX with word at ES:(E)DI and set status flags
AF	SCASD	Compare EAX with doubleword at ES:(E)DI and set status flags

Description

Compares the byte, word, or double word specified with the memory operand with the value in the AL, AX, or EAX register, and sets the status flags in the EFLAGS register according to the results. The memory operand address is read from either the ES:EDI or the ES:DI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The ES segment cannot be overridden with a segment override prefix.

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operand form (specified with the SCAS mnemonic) allows the memory operand to be specified explicitly. Here, the memory operand should be a symbol that indicates the size and location of the operand value. The register operand is then automatically selected to match the size of the memory operand (the AL register for byte comparisons, AX for word comparisons, and EAX for doubleword comparisons). This explicit-operand form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the memory operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the ES:(E)DI registers, which must be loaded correctly before the compare string instruction is executed.

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the SCAS instructions. Here also ES:(E)DI is assumed to be the memory operand and the AL, AX, or EAX register is assumed to be the register operand. The size of the two operands is selected with the mnemonic: SCASB (byte comparison), SCASW (word comparison), or SCASD (doubleword comparison).

After the comparison, the (E)DI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)DI register is incremented; if the DF flag is 1, the (E)DI register is decremented.) The (E)DI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The SCAS, SCASB, SCASW, and SCASD instructions can be preceded by the REP prefix for block comparisons of ECX bytes, words, or doublewords. More often, however, these instructions will be used in a LOOP construct that takes some action based on the setting of the status flags before the next comparison is made. See “REP/REPE/REPZ/REPNE /REPNZ—Repeat String Operation Prefix” in this chapter for a description of the REP prefix.

SCAS/SCASB/SCASW/SCASD—Scan String (Continued)

Operation

```
IF (byte cmparison)
   THEN
       temp    AL – SRC;
       SetStatusFlags(temp);
           THEN IF DF    0
               THEN (E)DI    (E)DI + 1;
               ELSE (E)DI    (E)DI – 1;
           FI;
   ELSE IF (word comparison)
       THEN
           temp    AX – SRC;
           SetStatusFlags(temp)
               THEN IF DF    0
                   THEN (E)DI    (E)DI + 2;
                   ELSE (E)DI    (E)DI – 2;
               FI;
       ELSE (* doubleword comparison *)
           temp    EAX – SRC;
           SetStatusFlags(temp)
               THEN IF DF    0
                   THEN (E)DI    (E)DI + 4;
                   ELSE (E)DI    (E)DI – 4;
               FI;
   FI;
FI;
```

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are set according to the temporary result of the comparison.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the limit of the ES segment.
	If the ES register contains a null segment selector.
	If an illegal memory operand effective address in the ES segment is given.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix

Opcode	Instruction	Description
F3 6C	REP INS *r/m8*, DX	Input (E)CX bytes from port DX into ES:[(E)DI]
F3 6D	REP INS *r/m16*, DX	Input (E)CX words from port DX into ES:[(E)DI]
F3 6D	REP INS *r/m32*, DX	Input (E)CX doublewords from port DX into ES:[(E)DI]
F3 A4	REP MOVS *m8, m8*	Move (E)CX bytes from DS:[(E)SI] to ES:[(E)DI]
F3 A5	REP MOVS *m16, m16*	Move (E)CX words from DS:[(E)SI] to ES:[(E)DI]
F3 A5	REP MOVS *m32, m32*	Move (E)CX doublewords from DS:[(E)SI] to ES:[(E)DI]
F3 6E	REP OUTS DX, *r/m8*	Output (E)CX bytes from DS:[(E)SI] to port DX
F3 6F	REP OUTS DX, *r/m16*	Output (E)CX words from DS:[(E)SI] to port DX
F3 6F	REP OUTS DX, *r/m32*	Output (E)CX doublewords from DS:[(E)SI] to port DX
F3 AC	REP LODS AL	Load (E)CX bytes from DS:[(E)SI] to AL
F3 AD	REP LODS AX	Load (E)CX words from DS:[(E)SI] to AX
F3 AD	REP LODS EAX	Load (E)CX doublewords from DS:[(E)SI] to EAX
F3 AA	REP STOS *m8*	Fill (E)CX bytes at ES:[(E)DI] with AL
F3 AB	REP STOS *m16*	Fill (E)CX words at ES:[(E)DI] with AX
F3 AB	REP STOS *m32*	Fill (E)CX doublewords at ES:[(E)DI] with EAX
F3 A6	REPE CMPS *m8, m8*	Find nonmatching bytes in ES:[(E)DI] and DS:[(E)SI]
F3 A7	REPE CMPS *m16, m16*	Find nonmatching words in ES:[(E)DI] and DS:[(E)SI]
F3 A7	REPE CMPS *m32, m32*	Find nonmatching doublewords in ES:[(E)DI] and DS:[(E)SI]
F3 AE	REPE SCAS *m8*	Find non-AL byte starting at ES:[(E)DI]
F3 AF	REPE SCAS *m16*	Find non-AX word starting at ES:[(E)DI]
F3 AF	REPE SCAS *m32*	Find non-EAX doubleword starting at ES:[(E)DI]
F2 A6	REPNE CMPS *m8, m8*	Find matching bytes in ES:[(E)DI] and DS:[(E)SI]
F2 A7	REPNE CMPS *m16, m16*	Find matching words in ES:[(E)DI] and DS:[(E)SI]
F2 A7	REPNE CMPS *m32, m32*	Find matching doublewords in ES:[(E)DI] and DS:[(E)SI]
F2 AE	REPNE SCAS *m8*	Find AL, starting at ES:[(E)DI]
F2 AF	REPNE SCAS *m16*	Find AX, starting at ES:[(E)DI]
F2 AF	REPNE SCAS *m32*	Find EAX, starting at ES:[(E)DI]

Description

Repeats a string instruction the number of times specified in the count register ((E)CX) or until the indicated condition of the ZF flag is no longer met. The REP (repeat), REPE (repeat while equal), REPNE (repeat while not equal), REPZ (repeat while zero), and REPNZ (repeat while not zero) mnemonics are prefixes that can be added to one of the string instructions. The REP prefix can be added to the INS, OUTS, MOVS, LODS, and STOS instructions, and the REPE, REPNE, REPZ, and REPNZ prefixes can be added to the CMPS and SCAS instructions. (The REPZ and REPNZ prefixes are synonymous forms of the REPE and REPNE prefixes, respectively.) The behavior of the REP prefix is undefined when used with non-string instructions.

The REP prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct.

REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix (Continued)

All of these repeat prefixes cause the associated instruction to be repeated until the count in register (E)CX is decremented to 0 (see the following table). (If the current address-size attribute is 32, register ECX is used as a counter, and if the address-size attribute is 16, the CX register is used.) The REPE, REPNE, REPZ, and REPNZ prefixes also check the state of the ZF flag after each iteration and terminate the repeat loop if the ZF flag is not in the specified state. When both termination conditions are tested, the cause of a repeat termination can be determined either by testing the (E)CX register with a JECXZ instruction or by testing the ZF flag with a JZ, JNZ, and JNE instruction.

Repeat Prefix	Termination Condition 1	Termination Condition 2
REP	ECX=0	None
REPE/REPZ	ECX=0	ZF=0
REPNE/REPNZ	ECX=0	ZF=1

When the REPE/REPZ and REPNE/REPNZ prefixes are used, the ZF flag does not require initialization because both the CMPS and SCAS instructions affect the ZF flag according to the results of the comparisons they make.

A repeating string operation can be suspended by an exception or interrupt. When this happens, the state of the registers is preserved to allow the string operation to be resumed upon a return from the exception or interrupt handler. The source and destination registers point to the next string elements to be operated on, the EIP register points to the string instruction, and the ECX register has the value it held following the last successful iteration of the instruction. This mechanism allows long string operations to proceed without affecting the interrupt response time of the system.

When a fault occurs during the execution of a CMPS or SCAS instruction that is prefixed with REPE or REPNE, the EFLAGS value is restored to the state prior to the execution of the instruction. Since the SCAS and CMPS instructions do not use EFLAGS as an input, the processor can resume the instruction after the page fault handler.

Use the REP INS and REP OUTS instructions with caution. Not all I/O ports can handle the rate at which these instructions execute.

A REP STOS instruction is the fastest way to initialize a large block of memory.

REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix (Continued)

Operation

```
IF AddressSize     16
    THEN
         use CX for CountReg;
    ELSE (* AddressSize     32 *)
         use ECX for CountReg;
FI;
WHILE CountReg    0
    DO
         service pending interrupts (if any);
         execute associated string instruction;
         CountReg     CountReg – 1;
         IF CountReg     0
              THEN exit WHILE loop
         FI;
         IF (repeat prefix is REPZ or REPE) AND (ZF=0)
         OR (repeat prefix is REPNZ or REPNE) AND (ZF=1)
              THEN exit WHILE loop
         FI;
    OD;
```

Flags Affected

None; however, the CMPS and SCAS instructions do set the status flags in the EFLAGS register.

Exceptions (All Operating Modes)

None; however, exceptions can be generated by the instruction a repeat prefix is associated with.

```
; File: rep.asm
;
; Demonstrates the use of the REP prefix with
; string instructions.
;
; This program does no I/O. Use gdb to examine its effects.
;
        SECTION .data                   ; Data section

msg:    db "Hello, world", 10           ; The string to print.
len:    equ $-msg

        SECTION .text                   ; Code section.
        global _start
_start: nop                             ; Entry point.

find:   mov     al, 'o'                 ; look for an 'o'
        mov     edi, msg                ; here
        mov     ecx, len                ; limit repetitions
        cld                             ; auto inc edi
        repne scasb                     ; while (al != [edi])
        jnz     not_found               ;
        mov     bl, [edi-1]             ; what did we find?
not_found:

erase:  mov     edi, msg                ; where?
        mov     ecx, len                ; how many bytes?
        mov     al, '?'                 ; with which char?
        cld                             ; auto inc edi
        rep stosb

alldone:
        mov     ebx, 0                  ; exit code, 0=normal
        mov     eax, 1                  ; Exit.
        int     80H                     ; Call kernel.
```

```
Script started on Fri Sep 19 14:51:13 2003
linux3% nasm -f elf rep.asm
linux3% ld rep.o
linux3%
linux3% gdb a.out
GNU gdb Red Hat Linux (5.2-2)
...

(gdb) display/i $eip
(gdb) display/x $edi
(gdb) display $ecx
(gdb) display/c $ebx
(gdb) display/c $eax

(gdb) break *find
Breakpoint 1 at 0x8048081
(gdb) break *erase
Breakpoint 2 at 0x8048095
(gdb) break *alldone
Breakpoint 3 at 0x80480a4
(gdb) run
Starting program: /afs/umbc.edu/users/c/h/chang/home/asm/a.out

Breakpoint 1, 0x08048081 in find ()
5: /c $eax = 0 '\0'
4: /c $ebx = 0 '\0'
3: $ecx = 0
2: /x $edi = 0x0
1: x/i $eip  0x8048081 <find>:  mov    al,0x6f
(gdb) x/14cb &msg
0x80490b0 <msg>:        72 'H'  101 'e' 108 'l' 108 'l' 111 'o' 44
','  32 ' '  119 'w'
0x80490b8 <msg+8>:      111 'o' 114 'r' 108 'l' 100 'd' 10 '\n' 0
'\0'
(gdb) si
0x08048083 in find ()
5: /c $eax = 111 'o'
4: /c $ebx = 0 '\0'
3: $ecx = 0
2: /x $edi = 0x0
1: x/i $eip  0x8048083 <find+2>:        mov    edi,0x80490b0
(gdb)
0x08048088 in find ()
5: /c $eax = 111 'o'
4: /c $ebx = 0 '\0'
3: $ecx = 0
2: /x $edi = 0x80490b0
1: x/i $eip  0x8048088 <find+7>:        mov    ecx,0xd
(gdb)
0x0804808d in find ()
5: /c $eax = 111 'o'
4: /c $ebx = 0 '\0'
3: $ecx = 13
2: /x $edi = 0x80490b0
1: x/i $eip  0x804808d <find+12>:       cld
```

```
(gdb)
0x0804808e in find ()
5: /c $eax = 111 'o'
4: /c $ebx = 0 '\0'
3: $ecx = 13
2: /x $edi = 0x80490b0
1: x/i $eip  0x804808e <find+13>:       repnz scas al,es:[edi]
(gdb)
0x0804808e in find ()
5: /c $eax = 111 'o'
4: /c $ebx = 0 '\0'
3: $ecx = 12
2: /x $edi = 0x80490b1
1: x/i $eip  0x804808e <find+13>:       repnz scas al,es:[edi]
(gdb)
0x0804808e in find ()
5: /c $eax = 111 'o'
4: /c $ebx = 0 '\0'
3: $ecx = 11
2: /x $edi = 0x80490b2
1: x/i $eip  0x804808e <find+13>:       repnz scas al,es:[edi]
(gdb)
0x0804808e in find ()
5: /c $eax = 111 'o'
4: /c $ebx = 0 '\0'
3: $ecx = 10
2: /x $edi = 0x80490b3
1: x/i $eip  0x804808e <find+13>:       repnz scas al,es:[edi]
(gdb)
0x0804808e in find ()
5: /c $eax = 111 'o'
4: /c $ebx = 0 '\0'
3: $ecx = 9
2: /x $edi = 0x80490b4
1: x/i $eip  0x804808e <find+13>:       repnz scas al,es:[edi]
(gdb)
0x08048090 in find ()
5: /c $eax = 111 'o'
4: /c $ebx = 0 '\0'
3: $ecx = 8
2: /x $edi = 0x80490b5
1: x/i $eip  0x8048090 <find+15>:       jne    0x8048095 <not_found
(gdb)
0x08048092 in find ()
5: /c $eax = 111 'o'
4: /c $ebx = 0 '\0'
3: $ecx = 8
2: /x $edi = 0x80490b5
1: x/i $eip  0x8048092 <find+17>:       mov    bl,BYTE PTR [edi-1]
(gdb)

Breakpoint 2, 0x08048095 in not_found ()
5: /c $eax = 111 'o'
4: /c $ebx = 111 'o'
3: $ecx = 8
2: /x $edi = 0x80490b5
1: x/i $eip  0x8048095 <not_found>:     mov    edi,0x80490b0
```

```
(gdb)
0x0804809a in not_found ()
5: /c $eax = 111 'o'
4: /c $ebx = 111 'o'
3: $ecx = 8
2: /x $edi = 0x80490b0
1: x/i $eip  0x804809a <not_found+5>:   mov    ecx,0xd
(gdb)
0x0804809f in not_found ()
5: /c $eax = 111 'o'
4: /c $ebx = 111 'o'
3: $ecx = 13
2: /x $edi = 0x80490b0
1: x/i $eip  0x804809f <not_found+10>:  mov    al,0x3f
(gdb)
0x080480a1 in not_found ()
5: /c $eax = 63 '?'
4: /c $ebx = 111 'o'
3: $ecx = 13
2: /x $edi = 0x80490b0
1: x/i $eip  0x80480a1 <not_found+12>:  cld
(gdb)
0x080480a2 in not_found ()
5: /c $eax = 63 '?'
4: /c $ebx = 111 'o'
3: $ecx = 13
2: /x $edi = 0x80490b0
1: x/i $eip  0x80480a2 <not_found+13>:  repz stos es:[edi],al
(gdb)
0x080480a2 in not_found ()
5: /c $eax = 63 '?'
4: /c $ebx = 111 'o'
3: $ecx = 12
2: /x $edi = 0x80490b1
1: x/i $eip  0x80480a2 <not_found+13>:  repz stos es:[edi],al
(gdb)
0x080480a2 in not_found ()
5: /c $eax = 63 '?'
4: /c $ebx = 111 'o'
3: $ecx = 11
2: /x $edi = 0x80490b2
1: x/i $eip  0x80480a2 <not_found+13>:  repz stos es:[edi],al
(gdb) cont
Continuing.

Breakpoint 3, 0x080480a4 in alldone ()
5: /c $eax = 63 '?'
4: /c $ebx = 111 'o'
3: $ecx = 0
2: /x $edi = 0x80490bd
1: x/i $eip  0x80480a4 <alldone>:       mov    ebx,0x0
(gdb) x/14cb &msg
0x80490b0 <msg>:        63 '?'  63 '?'  63 '?'  63 '?'  63 '?'  63
'?'  63 '?'  63 '?'
0x80490b8 <msg+8>:      63 '?'  63 '?'  63 '?'  63 '?'  63 '?'  0
'\0'
(gdb) quit
```

NEXT TIME

- a bigger example
- subroutines

References

- Some figures and diagrams from *IA-32 Intel Architecture Software Developer's Manual, Vols 1-3*

  <http://developer.intel.com/design/Pentium4/manuals/>