# CMSC 313 COMPUTER ORGANIZATION & ASSEMBLY LANGUAGE PROGRAMMING

EXTRA MATERIALS 1, SPRING 2013

# TOPICS NOT COVERED

- **Jump Tables**

- **Some i386 String Instructions**

# JUMP TABLES

```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 ;
}
```

```asm
; 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
```

```asm
        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
```

```
  1                                              ; File: switch.asm
  2                                              ;
  3                                              ; Demonstrate jump tables
  4                                              ;
  5                                              ;
  6
  7                                              %define STDIN 0
  8                                              %define STDOUT 1
  9                                              %define SYSCALL_EXIT   1
 10                                              %define SYSCALL_READ   3
 11                                              %define SYSCALL_WRITE 4
 12                                              %define BUFLEN 256
 13
 14
 15                                                      SECTION .data                        ; initialized data section
 16
 17 00000000 4D616B6520612073-65-               mesg1:  db "Make a selection [a, b, c, d or f]: "
 18 00000009 6C656374696F6E205B-
 19 00000012 612C20622C20632C20-
 20 0000001B 64206F7220665D3A20
 21                                              len1:   equ $ - mesg1
 22
 23 00000024 52656164204572726F-               err_mesg:       db "Read Error", 10
 24 0000002D 720A
 25                                              err_len:        equ $ - err_mesg
 26
 27
 28 0000002F 476F6F642063686F69-               mesga:  db "Good choice!", 10
 29 00000038 6365210A
 30                                              lena:   equ $ - mesga
 31
 32 0000003C 62206973206D792066-               mesgb:  db "b is my favorite", 10
 33 00000045 61766F726974650A
 34                                              lenb:   equ $ - mesgb
 35
 36 0000004D 63206973206120706F-               mesgc:  db "c is a popular selection", 10
 37 00000056 70756C61722073656C-
 38 0000005F 656374696F6E0A
 39                                              lenc:   equ $ - mesgc
 40
 41 00000066 536F7272792C207765-               mesgd:  db "Sorry, we are fresh out of d", 10
 42 0000006F 206172652066726573-
 43 00000078 68206F7574206F6620-
 44 00000081 640A
 45                                              lend:   equ $ - mesgd
 46
 47 00000083 4E6F74206120676F6F-               mesgf:   db "Not a good day for f. Try something else.", 10
 48 0000008C 64206461792066726F-
 49 00000095 20662E205472792073-
 50 0000009E 6F6D657468696E6720-
```

```
 51 000000A7 656C73652E0A
 52                                      lenf:    equ $-mesgf
 53
 54 000000AD 506C65617365207069-         dflt_mesg: db "Please pick one of the choices!", 10
 55 000000B6 636B206F6E65206F66-
 56 000000BF 207468652063686F69-
 57 000000C8 636573210A
 58                                      dflt_len:  equ $ - dflt_mesg
 59
 60                                      jump_table:
 61 000000CD [74000000]                          dd do_a            ; dd = "define double" = 32-bit data
 62 000000D1 [8F000000]                          dd do_b
 63 000000D5 [AA000000]                          dd do_c
 64 000000D9 [C5000000]                          dd do_d
 65 000000DD [FB000000]                          dd do_default
 66 000000E1 [E0000000]                          dd do_f
 67
 68
 69                                              SECTION .bss                ; uninitialized data section
 70 00000000 <res 00000100>         buf:         resb BUFLEN                 ; buffer for read
 71
 72
 73                                              SECTION .text               ; Code section.
 74                                              global  _start              ; let loader see entry point
 75
 76                                      _start:
 77
 78                                              ; prompt user for input
 79                                              ;
 80 00000000 B804000000                          mov     eax, SYSCALL_WRITE  ; write function
 81 00000005 BB01000000                          mov     ebx, STDOUT         ; Arg1: file descriptor
 82 0000000A B9[00000000]                         mov     ecx, mesg1          ; Arg2: addr of message
 83 0000000F BA24000000                          mov     edx, len1           ; Arg3: length of message
 84 00000014 CD80                                int     080h                ; ask kernel to write
 85
 86                                              ; read user input
 87                                              ;
 88 00000016 B803000000                          mov     eax, SYSCALL_READ   ; read function
 89 0000001B BB00000000                          mov     ebx, STDIN          ; Arg 1: file descriptor
 90 00000020 B9[00000000]                         mov     ecx, buf            ; Arg 2: address of buffer
 91 00000025 BA00010000                          mov     edx, BUFLEN         ; Arg 3: buffer length
 92 0000002A CD80                                int     080h
 93
 94                                              ; error check
 95                                              ;
 96 0000002C 3D00000000                          cmp     eax, 0              ; check if any chars read
 97 00000031 7F1B                                jg      read_OK             ; >0 chars read = OK
 98 00000033 B804000000                          mov     eax, SYSCALL_WRITE  ; ow print error mesg
 99 00000038 BB01000000                          mov     ebx, STDOUT
100 0000003D B9[24000000]                         mov     ecx, err_mesg
```

```
101 00000042 BA0B000000                              mov      edx, err_len
102 00000047 CD80                                    int      080h
103 00000049 E9C8000000                              jmp      exit                         ; skip over rest
104                              read_OK:
105
106 0000004E B800000000                              mov      eax, 0
107 00000053 A0[00000000]                            mov      al, [buf]                     ; get a character
108 00000058 3C61                                    cmp      al, 'a'                       ; below 'a'?
109 0000005A 0F829B000000                            jb       do_default
110 00000060 3C66                                    cmp      al, 'f'                       ; above 'f'?
111 00000062 0F8793000000                            ja       do_default
112
113 00000068 2D61000000                              sub      eax, 'a'                      ; calculate offset
114 0000006D FF2485[CD000000]                        jmp      [4*eax + jump_table]
115
116
117                              do_a:
118 00000074 B804000000                              mov      eax, SYSCALL_WRITE           ; write message for choice a
119 00000079 BB01000000                              mov      ebx, STDOUT
120 0000007E B9[2F000000]                            mov      ecx, mesga
121 00000083 BA0D000000                              mov      edx, lena
122 00000088 CD80                                    int      080h
123 0000008A E987000000                              jmp      exit
124
125                              do_b:
126 0000008F B804000000                              mov      eax, SYSCALL_WRITE           ; write message for choice b
127 00000094 BB01000000                              mov      ebx, STDOUT
128 00000099 B9[3C000000]                            mov      ecx, mesgb
129 0000009E BA11000000                              mov      edx, lenb
130 000000A3 CD80                                    int      080h
131 000000A5 E96C000000                              jmp      exit
132
133                              do_c:
134 000000AA B804000000                              mov      eax, SYSCALL_WRITE           ; write message for choice c
135 000000AF BB01000000                              mov      ebx, STDOUT
136 000000B4 B9[4D000000]                            mov      ecx, mesgc
137 000000B9 BA19000000                              mov      edx, lenc
138 000000BE CD80                                    int      080h
139 000000C0 E951000000                              jmp      exit
140
141                              do_d:
142 000000C5 B804000000                              mov      eax, SYSCALL_WRITE           ; write message for choice d
143 000000CA BB01000000                              mov      ebx, STDOUT
144 000000CF B9[66000000]                            mov      ecx, mesgd
145 000000D4 BA1D000000                              mov      edx, lend
146 000000D9 CD80                                    int      080h
147 000000DB E936000000                              jmp      exit
148
149                              do_f:
150 000000E0 B804000000                              mov      eax, SYSCALL_WRITE           ; write message for choice f
```

```
151 000000E5 BB01000000                            mov     ebx, STDOUT
152 000000EA B9[83000000]                          mov     ecx, mesgf
153 000000EF BA2A000000                            mov     edx, lenf
154 000000F4 CD80                                  int     080h
155 000000F6 E91B000000                            jmp     exit
156
157                             do_default:
158 000000FB B804000000                            mov     eax, SYSCALL_WRITE      ; write message for default
159 00000100 BB01000000                            mov     ebx, STDOUT
160 00000105 B9[AD000000]                          mov     ecx, dflt_mesg
161 0000010A BA20000000                            mov     edx, dflt_len
162 0000010F CD80                                  int     080h
163 00000111 E900000000                            jmp     exit
164
165
166                             ; final exit
167                             ;
168 00000116 B801000000         exit:   mov     EAX, SYSCALL_EXIT       ; exit function
169 0000011B BB00000000                 mov     EBX, 0                  ; exit code, 0=normal
170 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 over-ride 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. |

**int<sub>e</sub>l**

## 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.

**intel.**

## 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;
```

intel.

## 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

```asm
; 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.

intel.

## 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. |

**intel**®

## 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
```