CMSC 313 Lecture 08

- Announcements
- Project 2 Questions
- More Arithmetic Instructions
  - NEG, MUL, IMUL, DIV
- Indexed Addressing: [ESI + ECX*4 + DISP]
- Some i386 string instructions
Project 2: Hamming Distance

Due: Tue 09/23/03, Section 0101 (Chang) & Section 0301 (Macneil)
Wed 09/24/03, Section 0201 (Patel & Bourner)

Objective
The objective of this programming project is to practice designing your own loops and branching code in assembly language and to gain greater familiarity with the i386 instructions set.

Assignment
Write an assembly language program that prompts the user for two input strings and computes the Hamming distance between the two strings. The Hamming distance is the number of bit positions where the two strings differ. For example, the ASCII representations of the strings "foo" and "bar" in binary are:

"foo" = 0110 0110 0110 1111 0110 1111
"bar" = 0110 0010 0110 0001 0111 0010

So, the Hamming distance between "foo" and "bar" is 8.

Some details:
• Your program must return the Hamming distance of the two strings as the exit status of the program. This is the value stored in the EBX register just before the system call to exit the program.
• To see the exit status of your program, execute the program using the Unix command:
  a.out ; echo $?
  Since the exit status is a value between 0 and 255, you should restrict the user input to 31 characters.
• If the user enters two strings with different lengths, your program should return the Hamming distance up to the length of the shorter string.
• Look up the i386 instructions ADC and XOR and determine how these instructions are relevant to this programming project.
• Record some sample runs of your program using the Unix script command.

Implementation Notes
• The easiest way to examine the contents of a register bit-by-bit is to use successive SHR instruction to shift the least significant bit into the carry flag.
• When you use the gdb debugger to run your program, note that gdb reports the exit status as an octal (base 8) value. The Unix shell reports the exit status in decimal.
• The Hamming distance between the following two strings is 38:
  this is a test
  of the emergency broadcast
  You must also make your own test cases.
• Part of this project is for you to decide which registers should hold which values and whether to use 8-bit, 16-bit or 32-bit registers. A logical plan for the use of registers will make your program easier to code and easier to debug — i.e., think about this before you start coding.

Turning in your program
Use the UNIX submit command on the GL system to turn in your project. You should submit two files: 1) the modified assembly language program and 2) the typescript file of sample runs of your program. The class name for submit is cs313_0101, cs313_0201 or cs313_0301 depending on which section you attend. The name of the assignment name is proj2. The UNIX command to do this should look something like:

submit cs313_0101 proj2 hamming.asm typescript
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

UMBC, CMSC313, Richard Chang <chang@umbc.edu>
NEG—Two's Complement Negation

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F6 /3</td>
<td>NEG r/m8</td>
<td>Two's complement negate r/m8</td>
</tr>
<tr>
<td>F7 /3</td>
<td>NEG r/m16</td>
<td>Two's complement negate r/m16</td>
</tr>
<tr>
<td>F7 /3</td>
<td>NEG r/m32</td>
<td>Two's complement negate r/m32</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F6 /4</td>
<td>MUL r/m8</td>
<td>Unsigned multiply (AX ← AL + r/m8)</td>
</tr>
<tr>
<td>F7 /4</td>
<td>MUL r/m16</td>
<td>Unsigned multiply (DX:AX ← AX + r/m16)</td>
</tr>
<tr>
<td>F7 /4</td>
<td>MUL r/m32</td>
<td>Unsigned multiply (EDX:EAX ← EAX + r/m32)</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Operand Size</th>
<th>Source 1</th>
<th>Source 2</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>AL</td>
<td>r/m8</td>
<td>AX</td>
</tr>
<tr>
<td>Word</td>
<td>AX</td>
<td>r/m16</td>
<td>DX:AX</td>
</tr>
<tr>
<td>Doubleword</td>
<td>EAX</td>
<td>r/m32</td>
<td>EDX:EAX</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F6 /5</td>
<td>IMUL r/m8</td>
<td>AX ← AL * r/m byte</td>
</tr>
<tr>
<td>F7 /5</td>
<td>IMUL r/m16</td>
<td>DX:AX ← AX * r/m word</td>
</tr>
<tr>
<td>F7 /5</td>
<td>IMUL r/m32</td>
<td>EDX:EAX ← EAX * r/m doubleword</td>
</tr>
<tr>
<td>0F AF lr</td>
<td>IMUL r16,r/m16</td>
<td>word register ← word register * r/m word</td>
</tr>
<tr>
<td>0F AF lr</td>
<td>IMUL r32,r/m32</td>
<td>doubleword register ← doubleword register * r/m doubleword</td>
</tr>
<tr>
<td>6B lr ib</td>
<td>IMUL r16,r/m16,imm8</td>
<td>word register ← r/m16 * sign-extended immediate byte</td>
</tr>
<tr>
<td>6B lr ib</td>
<td>IMUL r32,r/m32,imm8</td>
<td>doubleword register ← r/m32 * sign-extended immediate byte</td>
</tr>
<tr>
<td>6B lr ib</td>
<td>IMUL r16,imm8</td>
<td>word register ← word register * sign-extended immediate byte</td>
</tr>
<tr>
<td>6B lr ib</td>
<td>IMUL r32,imm8</td>
<td>doubleword register ← doubleword register * sign-extended immediate byte</td>
</tr>
<tr>
<td>69 lr iw</td>
<td>IMUL r16,r/ m16,imm16</td>
<td>word register ← r/m16 * immediate word</td>
</tr>
<tr>
<td>69 lr id</td>
<td>IMUL r32,r/ m32,imm32</td>
<td>doubleword register ← r/m32 * immediate doubleword</td>
</tr>
<tr>
<td>69 lr iw</td>
<td>IMUL r16,imm16</td>
<td>word register ← r/m16 * immediate word</td>
</tr>
<tr>
<td>69 lr id</td>
<td>IMUL r32,imm32</td>
<td>doubleword register ← r/m32 * immediate doubleword</td>
</tr>
</tbody>
</table>

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 *)

3-322
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;

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.
IMUL—Signed Multiply

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F6 /5</td>
<td>IMUL r/m8</td>
<td>AX ← AL * r/m byte</td>
</tr>
<tr>
<td>F7 /5</td>
<td>IMUL r/m16</td>
<td>DX:AX ← AX * r/m word</td>
</tr>
<tr>
<td>F7 /5</td>
<td>IMUL r/m32</td>
<td>EDX:EAX ← EAX * r/m doubleword</td>
</tr>
<tr>
<td>0F AF /r</td>
<td>IMUL r16,/r/m16</td>
<td>word register ← word register * r/m word</td>
</tr>
<tr>
<td>0F AF /r</td>
<td>IMUL r32,/r/m32</td>
<td>doubleword register ← doubleword register * r/m doubleword</td>
</tr>
<tr>
<td>6B /r ib</td>
<td>IMUL r16,/r/m16,imm8</td>
<td>word register ← r/m16 + sign-extended immediate byte</td>
</tr>
<tr>
<td>6B /r ib</td>
<td>IMUL r32,/r/m32,imm8</td>
<td>doubleword register ← r/m32 + sign-extended immediate byte</td>
</tr>
<tr>
<td>6B /r ib</td>
<td>IMUL r16,imm8</td>
<td>word register ← word register + sign-extended immediate byte</td>
</tr>
<tr>
<td>6B /r ib</td>
<td>IMUL r32,imm8</td>
<td>doubleword register ← doubleword register + sign-extended immediate byte</td>
</tr>
<tr>
<td>69 /r iw</td>
<td>IMUL r16,/r/m16,imm16</td>
<td>word register ← r/m16 + immediate word</td>
</tr>
<tr>
<td>69 /r id</td>
<td>IMUL r32,/r/m32,imm32</td>
<td>doubleword register ← r/m32 + immediate doubleword</td>
</tr>
<tr>
<td>69 /r iw</td>
<td>IMUL r16,imm16</td>
<td>word register ← r/m16 + immediate word</td>
</tr>
<tr>
<td>69 /r id</td>
<td>IMUL r32,imm32</td>
<td>doubleword register ← r/m32 + immediate doubleword</td>
</tr>
</tbody>
</table>

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;
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;

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.
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
The offset which results from adding these components is called an **effective address**. Each of these components can have either a positive or negative (2's complement) value, with the exception of the scaling factor. Figure 3-9 shows all the possible ways that these components can be combined to create an effective address in the selected segment.

<table>
<thead>
<tr>
<th>Base</th>
<th>Index</th>
<th>Scale</th>
<th>Displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>EAX</td>
<td>1</td>
<td>None</td>
</tr>
<tr>
<td>EBX</td>
<td>EBX</td>
<td>2</td>
<td>8-bit</td>
</tr>
<tr>
<td>ECX</td>
<td>ECX</td>
<td>3</td>
<td>16-bit</td>
</tr>
<tr>
<td>EDX</td>
<td>EDX</td>
<td>4</td>
<td>32-bit</td>
</tr>
<tr>
<td>ESP</td>
<td>ESP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EBP</td>
<td>EBP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESI</td>
<td>ESI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDI</td>
<td>EDI</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Offset = Base + (Index * Scale) + Displacement

**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.

**Displacement**

A displacement alone represents a direct (uncomputed) offset to the operand. Because the displacement is encoded in the instruction, this form of an address is sometimes called an absolute or static address. It is commonly used to access a statically allocated scalar operand.

**Base**

A base alone represents an indirect offset to the operand. Since the value in the base register can change, it can be used for dynamic storage of variables and data structures.
MOV EAX, [EDI + 20]
MOV EAX, [ECX*4 + 08A94068]
MOV EAX, [EDI + ECX + 20]
MOV EAX, [EDI + ECX*4 + 20]

Base + Index*Scale + Displacement
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
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++) {
;   twodim[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>:   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.
Program exited normally.
(gdb) quit
linux3% exit
exit
Script done on Fri Sep 19 13:20:35 2003
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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>LODS m8</td>
<td>Load byte at address DS:(E)SI into AL</td>
</tr>
<tr>
<td>AD</td>
<td>LODS m16</td>
<td>Load word at address DS:(E)SI into AX</td>
</tr>
<tr>
<td>AD</td>
<td>LODS m32</td>
<td>Load doubleword at address DS:(E)SI into EAX</td>
</tr>
<tr>
<td>AC</td>
<td>LODSB</td>
<td>Load byte at address DS:(E)SI into AL</td>
</tr>
<tr>
<td>AD</td>
<td>LODSW</td>
<td>Load word at address DS:(E)SI into AX</td>
</tr>
<tr>
<td>AD</td>
<td>LODSD</td>
<td>Load doubleword at address DS:(E)SI into EAX</td>
</tr>
</tbody>
</table>

**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/REPNZ—Repeat String Operation Prefix” in this chapter for a description of the REP prefix.
INSTRUCTION SET REFERENCE

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>STOS m8</td>
<td>Store AL at address ES:(E)DI</td>
</tr>
<tr>
<td>AB</td>
<td>STOS m16</td>
<td>Store AX at address ES:(E)DI</td>
</tr>
<tr>
<td>AB</td>
<td>STOS m32</td>
<td>Store EAX at address ES:(E)DI</td>
</tr>
<tr>
<td>AA</td>
<td>STOSB</td>
<td>Store AL at address ES:(E)DI</td>
</tr>
<tr>
<td>AB</td>
<td>STOSW</td>
<td>Store AX at address ES:(E)DI</td>
</tr>
<tr>
<td>AB</td>
<td>STOSD</td>
<td>Store EAX at address ES:(E)DI</td>
</tr>
</tbody>
</table>

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

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**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FD</td>
<td>STD</td>
<td>Set DF flag</td>
</tr>
</tbody>
</table>

**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/LOOPcc—Loop According to ECX Counter

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E2 cb</td>
<td>LOOP rel8</td>
<td>Decrement count; jump short if count ≠ 0</td>
</tr>
<tr>
<td>E1 cb</td>
<td>LOOPE rel8</td>
<td>Decrement count; jump short if count ≠ 0 and ZF=1</td>
</tr>
<tr>
<td>E1 cb</td>
<td>LOOPZ rel8</td>
<td>Decrement count; jump short if count ≠ 0 and ZF=1</td>
</tr>
<tr>
<td>E0 cb</td>
<td>LOOPNE rel8</td>
<td>Decrement count; jump short if count ≠ 0 and ZF=0</td>
</tr>
<tr>
<td>E0 cb</td>
<td>LOOPNZ rel8</td>
<td>Decrement count; jump short if count ≠ 0 and ZF=0</td>
</tr>
</tbody>
</table>

**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 (LOOPcc) 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 LOOPcc 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;  

3-394
LOOP/LOOPcc—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
; 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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AE</td>
<td>SCAS m8</td>
<td>Compare AL with byte at ES:(E)DI and set status flags</td>
</tr>
<tr>
<td>AF</td>
<td>SCAS m16</td>
<td>Compare AX with word at ES:(E)DI and set status flags</td>
</tr>
<tr>
<td>AF</td>
<td>SCAS m32</td>
<td>Compare EAX with doubleword at ES:(E)DI and set status flags</td>
</tr>
<tr>
<td>AE</td>
<td>SCASB</td>
<td>Compare AL with byte at ES:(E)DI and set status flags</td>
</tr>
<tr>
<td>AF</td>
<td>SCASW</td>
<td>Compare AX with word at ES:(E)DI and set status flags</td>
</tr>
<tr>
<td>AF</td>
<td>SCASD</td>
<td>Compare EAX with doubleword at ES:(E)DI and set status flags</td>
</tr>
</tbody>
</table>

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:DI 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/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 comparison)
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

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

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, MOVBS, 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 (ECX) 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.

<table>
<thead>
<tr>
<th>Repeat Prefix</th>
<th>Termination Condition 1</th>
<th>Termination Condition 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>REP</td>
<td>ECX=0</td>
<td>None</td>
</tr>
<tr>
<td>REPE/REPZ</td>
<td>ECX=0</td>
<td>ZF=0</td>
</tr>
<tr>
<td>REPNE/REPNZ</td>
<td>ECX=0</td>
<td>ZF=1</td>
</tr>
</tbody>
</table>

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.
Demonstrates the use of the REP prefix with string instructions.

This program does no I/O. Use gdb to examine its effects.

```
; Section .data

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
        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?
        cld
        mov   al, '?' ; with which char?
        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
0x8049b0 <msg>:
    72 'H' 101 'e' 108 'l' 108 'l' 111 'o' 44 ',' 32 ' ' 119 'w'
0x8049b8 <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,0x8049b0
(gdb) 0x08048088 in find ()
5: /c $eax = 111 'o'
4: /c $ebx = 0 '\0'
3: $ecx = 0
2: /x $edi = 0x8049b0
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 = 0x8049b0
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)
0x0804808e in find ()
5: /c $eax = 111 'o'
4: /c $ebx = 0 '\0'
3: $ecx = 8
2: /x $edi = 0x80490b5
1: x/i $eip 0x804808e <find+13>: 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)
0x080480a9 in not_found ()
5: /c $eax = 111 'o'
4: /c $ebx = 111 'o'
3: $ecx = 8
2: /x $edi = 0x80490b0
1: x/i $eip 0x80480a9 <not_found+5>: mov ecx,0xd

(gdb)
0x080480af in not_found ()
5: /c $eax = 111 'o'
4: /c $ebx = 111 'o'
3: $ecx = 13
2: /x $edi = 0x80490b0
1: x/i $eip 0x80480af <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 = 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 '?' 63 'o'
0x80490b8 <msg+8>: 63 '?' 63 '?' 63 '?' 63 '?' 63 '?' 63 '?'

(gdb) quit
Next Time

• A Bigger Example: Escape Sequence Project
• Machine Language
• Project 3
References

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

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