CMSC 313 Lecture 06

- Project 1 Questions
- More on Conditional Jump Instructions
- Short Jumps vs Near Jumps
- Using Jump Instructions
- Logical (bit manipulation) Instructions
  - AND, OR, NOT, SHL, SHR, SAL, SAR,ROL, ROR, RCL, RCR
- More Arithmetic Instructions
  - NEG, MUL, IMUL, DIV
- Project 2
Recap Conditional Jumps

- **Uses flags to determine whether to jump**
  - Example: JAE (jump above or equal) jumps when the Carry Flag = 0
    - CMP EAX, 1492
    - JAE OceanBlue

- **Unsigned vs signed jumps**
  - Example: use JAE for unsigned data JGE (greater than or equal) for signed data
    - CMP EAX, 1492
    - JAE OceanBlue
    - CMP EAX, -42
    - JGE Somewhere
continues with the instruction following the Jcc instruction. As with the JMP instruction, the
transfer is one-way; that is, a return address is not saved.

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>Condition (Flag States)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unsigned Conditional Jumps</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JA/JNBE</td>
<td>(CF or ZF)=0</td>
<td>Above/not below or equal</td>
</tr>
<tr>
<td>JAE/JNB</td>
<td>CF=0</td>
<td>Above or equal/not below</td>
</tr>
<tr>
<td>JB/JNAE</td>
<td>CF=1</td>
<td>Below/not above or equal</td>
</tr>
<tr>
<td>JBE/JNA</td>
<td>(CF or ZF)=1</td>
<td>Below or equal/not above</td>
</tr>
<tr>
<td>JC</td>
<td>CF=1</td>
<td>Carry</td>
</tr>
<tr>
<td>JE/JZ</td>
<td>ZF=1</td>
<td>Equal/zero</td>
</tr>
<tr>
<td>JNC</td>
<td>CF=0</td>
<td>Not carry</td>
</tr>
<tr>
<td>JNE/JNZ</td>
<td>ZF=0</td>
<td>Not equal/not zero</td>
</tr>
<tr>
<td>JNP/JPO</td>
<td>PF=0</td>
<td>Not parity/parity odd</td>
</tr>
<tr>
<td>JP/JPE</td>
<td>PF=1</td>
<td>Parity/parity even</td>
</tr>
<tr>
<td>JCXZ</td>
<td>CX=0</td>
<td>Register CX is zero</td>
</tr>
<tr>
<td>JECXZ</td>
<td>ECX=0</td>
<td>Register ECX is zero</td>
</tr>
<tr>
<td><strong>Signed Conditional Jumps</strong></td>
<td>(SF xor OF) or ZF) =0</td>
<td>Greater/not less or equal</td>
</tr>
<tr>
<td>JG/JNLE</td>
<td>(SF xor OF)=0</td>
<td>Greater or equal/not less</td>
</tr>
<tr>
<td>JGE/JNL</td>
<td>(SF xor OF)=1</td>
<td>Less/not greater or equal</td>
</tr>
<tr>
<td>JL/JNGE</td>
<td>(SF xor OF)=1</td>
<td>Less or equal/not greater</td>
</tr>
<tr>
<td>JNO</td>
<td>OF=0</td>
<td>Not overflow</td>
</tr>
<tr>
<td>JNS</td>
<td>SF=0</td>
<td>Not sign (non-negative)</td>
</tr>
<tr>
<td>JO</td>
<td>OF=1</td>
<td>Overflow</td>
</tr>
<tr>
<td>JS</td>
<td>SF=1</td>
<td>Sign (negative)</td>
</tr>
</tbody>
</table>

The destination operand specifies a relative address (a signed offset with respect to the address
in the EIP register) that points to an instruction in the current code segment. The Jcc instructions
do not support far transfers; however, far transfers can be accomplished with a combination of
a Jcc and a JMP instruction (see “Jcc—Jump if Condition Is Met” in Chapter 3 of the Intel Archi-

Table 7-4 shows the mnemonics for the Jcc instructions and the conditions being tested for each
instruction. The condition code mnemonics are appended to the letter “J” to form the mnemonic
for a Jcc instruction. The instructions are divided into two groups: unsigned and signed condi-
tional jumps. These groups correspond to the results of operations performed on unsigned and
signed integers, respectively. Those instructions listed as pairs (for example, JA/JNBE) are alter-

7-19
### Jcc—Jump if Condition Is Met

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>77 cb</td>
<td>JA rel8</td>
<td>Jump short if above (CF=0 and ZF=0)</td>
</tr>
<tr>
<td>73 cb</td>
<td>JAE rel8</td>
<td>Jump short if above or equal (CF=0)</td>
</tr>
<tr>
<td>72 cb</td>
<td>JB rel8</td>
<td>Jump short if below (CF=1)</td>
</tr>
<tr>
<td>76 cb</td>
<td>JBE rel8</td>
<td>Jump short if below or equal (CF=1 or ZF=1)</td>
</tr>
<tr>
<td>72 cb</td>
<td>JC rel8</td>
<td>Jump short if carry (CF=1)</td>
</tr>
<tr>
<td>E3 cb</td>
<td>JCXZ rel8</td>
<td>Jump short if CX register is 0</td>
</tr>
<tr>
<td>E3 cb</td>
<td>JE rel8</td>
<td>Jump short if ECX register is 0</td>
</tr>
<tr>
<td>74 cb</td>
<td>JE rel8</td>
<td>Jump short if equal (ZF=1)</td>
</tr>
<tr>
<td>7F cb</td>
<td>JG rel8</td>
<td>Jump short if greater (ZF=0 and SF=OF)</td>
</tr>
<tr>
<td>7D cb</td>
<td>JGE rel8</td>
<td>Jump short if greater or equal (SF=OF)</td>
</tr>
<tr>
<td>7C cb</td>
<td>JL rel8</td>
<td>Jump short if less (SF&lt;&gt;OF)</td>
</tr>
<tr>
<td>7E cb</td>
<td>JLE rel8</td>
<td>Jump short if less or equal (ZF=1 or SF&lt;&gt;OF)</td>
</tr>
<tr>
<td>76 cb</td>
<td>JNA rel8</td>
<td>Jump short if not above (CF=1 or ZF=1)</td>
</tr>
<tr>
<td>72 cb</td>
<td>JNAE rel8</td>
<td>Jump short if not above or equal (CF=1)</td>
</tr>
<tr>
<td>73 cb</td>
<td>JNB rel8</td>
<td>Jump short if not below (CF=0)</td>
</tr>
<tr>
<td>77 cb</td>
<td>JNBE rel8</td>
<td>Jump short if not below or equal (CF=0 and ZF=0)</td>
</tr>
<tr>
<td>73 cb</td>
<td>JNC rel8</td>
<td>Jump short if not carry (CF=0)</td>
</tr>
<tr>
<td>75 cb</td>
<td>JNE rel8</td>
<td>Jump short if not equal (ZF=0)</td>
</tr>
<tr>
<td>7E cb</td>
<td>JNG rel8</td>
<td>Jump short if not greater (ZF=1 or SF&lt;&gt;OF)</td>
</tr>
<tr>
<td>7C cb</td>
<td>JNGE rel8</td>
<td>Jump short if not greater or equal (SF&lt;&gt;OF)</td>
</tr>
<tr>
<td>7D cb</td>
<td>JNL rel8</td>
<td>Jump short if not less (SF=OF)</td>
</tr>
<tr>
<td>7F cb</td>
<td>JNLE rel8</td>
<td>Jump short if not less or equal (ZF=0 and SF=OF)</td>
</tr>
<tr>
<td>71 cb</td>
<td>JNO rel8</td>
<td>Jump short if not overflow (OF=0)</td>
</tr>
<tr>
<td>7B cb</td>
<td>JNP rel8</td>
<td>Jump short if not parity (PF=0)</td>
</tr>
<tr>
<td>79 cb</td>
<td>JNS rel8</td>
<td>Jump short if not sign (SF=0)</td>
</tr>
<tr>
<td>75 cb</td>
<td>JNZ rel8</td>
<td>Jump short if not zero (ZF=0)</td>
</tr>
<tr>
<td>70 cb</td>
<td>JO rel8</td>
<td>Jump short if overflow (OF=1)</td>
</tr>
<tr>
<td>7A cb</td>
<td>JP rel8</td>
<td>Jump short if parity (PF=1)</td>
</tr>
<tr>
<td>7A cb</td>
<td>JPE rel8</td>
<td>Jump short if parity even (PF=1)</td>
</tr>
<tr>
<td>7B cb</td>
<td>JPO rel8</td>
<td>Jump short if parity odd (PF=0)</td>
</tr>
<tr>
<td>78 cb</td>
<td>JS rel8</td>
<td>Jump short if sign (SF=1)</td>
</tr>
<tr>
<td>74 cb</td>
<td>JZ rel8</td>
<td>Jump short if zero (ZF=1)</td>
</tr>
<tr>
<td>0F 87 cw/cd</td>
<td>JA rel16/32</td>
<td>Jump near if above (CF=0 and ZF=0)</td>
</tr>
<tr>
<td>0F 83 cw/cd</td>
<td>JAE rel16/32</td>
<td>Jump near if above or equal (CF=0)</td>
</tr>
<tr>
<td>0F 82 cw/cd</td>
<td>JB rel16/32</td>
<td>Jump near if below (CF=1)</td>
</tr>
<tr>
<td>0F 86 cw/cd</td>
<td>JBE rel16/32</td>
<td>Jump near if below or equal (CF=1 or ZF=1)</td>
</tr>
<tr>
<td>0F 82 cw/cd</td>
<td>JC rel16/32</td>
<td>Jump near if carry (CF=1)</td>
</tr>
<tr>
<td>0F 84 cw/cd</td>
<td>JE rel16/32</td>
<td>Jump near if equal (ZF=1)</td>
</tr>
<tr>
<td>0F 84 cw/cd</td>
<td>JZ rel16/32</td>
<td>Jump near if 0 (ZF=1)</td>
</tr>
<tr>
<td>0F 8F cw/cd</td>
<td>JG rel16/32</td>
<td>Jump near if greater (ZF=0 and SF=OF)</td>
</tr>
</tbody>
</table>
Jcc—Jump if Condition Is Met (Continued)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 8D cw/cd</td>
<td>JGE rel16/32</td>
<td>Jump near if greater or equal (SF=OF)</td>
</tr>
<tr>
<td>0F 8C cw/cd</td>
<td>JL rel16/32</td>
<td>Jump near if less (SF&lt;&gt;OF)</td>
</tr>
<tr>
<td>0F 8E cw/cd</td>
<td>JLE rel16/32</td>
<td>Jump near if less or equal (ZF=1 or SF&lt;&gt;OF)</td>
</tr>
<tr>
<td>0F 86 cw/cd</td>
<td>JNA rel16/32</td>
<td>Jump near if not above (CF=1 or ZF=1)</td>
</tr>
<tr>
<td>0F 82 cw/cd</td>
<td>JNAE rel16/32</td>
<td>Jump near if not above or equal (CF=1)</td>
</tr>
<tr>
<td>0F 83 cw/cd</td>
<td>JNB rel16/32</td>
<td>Jump near if not below (CF=0)</td>
</tr>
<tr>
<td>0F 87 cw/cd</td>
<td>JNBE rel16/32</td>
<td>Jump near if not below or equal (CF=0 and ZF=0)</td>
</tr>
<tr>
<td>0F 93 cw/cd</td>
<td>JNC rel16/32</td>
<td>Jump near if not carry (CF=0)</td>
</tr>
<tr>
<td>0F 85 cw/cd</td>
<td>JNE rel16/32</td>
<td>Jump near if not equal (ZF=0)</td>
</tr>
<tr>
<td>0F 8E cw/cd</td>
<td>JNG rel16/32</td>
<td>Jump near if not greater (ZF=1 or SF&lt;&gt;OF)</td>
</tr>
<tr>
<td>0F 8C cw/cd</td>
<td>JNGE rel16/32</td>
<td>Jump near if not greater or equal (SF&lt;&gt;OF)</td>
</tr>
<tr>
<td>0F 8D cw/cd</td>
<td>JNL rel16/32</td>
<td>Jump near if not less (SF=OF)</td>
</tr>
<tr>
<td>0F 8F cw/cd</td>
<td>JNLE rel16/32</td>
<td>Jump near if not less or equal (ZF=0 and SF=OF)</td>
</tr>
<tr>
<td>0F 81 cw/cd</td>
<td>JNO rel16/32</td>
<td>Jump near if not overflow (OF=0)</td>
</tr>
<tr>
<td>0F 8B cw/cd</td>
<td>JNP rel16/32</td>
<td>Jump near if not parity (PF=0)</td>
</tr>
<tr>
<td>0F 89 cw/cd</td>
<td>JNS rel16/32</td>
<td>Jump near if not sign (SF=0)</td>
</tr>
<tr>
<td>0F 85 cw/cd</td>
<td>JNZ rel16/32</td>
<td>Jump near if not zero (ZF=0)</td>
</tr>
<tr>
<td>0F 80 cw/cd</td>
<td>JO rel16/32</td>
<td>Jump near if overflow (OF=1)</td>
</tr>
<tr>
<td>0F 8A cw/cd</td>
<td>JP rel16/32</td>
<td>Jump near if parity (PF=1)</td>
</tr>
<tr>
<td>0F 8A cw/cd</td>
<td>JPE rel16/32</td>
<td>Jump near if parity even (PF=1)</td>
</tr>
<tr>
<td>0F 8B cw/cd</td>
<td>JPO rel16/32</td>
<td>Jump near if parity odd (PF=0)</td>
</tr>
<tr>
<td>0F 88 cw/cd</td>
<td>JS rel16/32</td>
<td>Jump near if sign (SF=1)</td>
</tr>
<tr>
<td>0F 84 cw/cd</td>
<td>JZ rel16/32</td>
<td>Jump near if 0 (ZF=1)</td>
</tr>
</tbody>
</table>

Description

Checks the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and, if the flags are in the specified state (condition), performs a jump to the target instruction specified by the destination operand. A condition code (cc) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, the jump is not performed and execution continues with the instruction following the Jcc instruction.

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). A relative offset (rel8, rel16, or rel32) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit or 32-bit immediate value, which is added to the instruction pointer. Instruction coding is most efficient for offsets of −128 to +127. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared to 0s, resulting in a maximum instruction pointer size of 16 bits.
Jcc—Jump if Condition Is Met (Continued)

The conditions for each Jcc mnemonic are given in the “Description” column of the table on the preceding page. The terms “less” and “greater” are used for comparisons of signed integers and the terms “above” and “below” are used for unsigned integers.

Because a particular state of the status flags can sometimes be interpreted in two ways, two mnemonics are defined for some opcodes. For example, the JA (jump if above) instruction and the JNBE (jump if not below or equal) instruction are alternate mnemonics for the opcode 77H.

The Jcc instruction does not support far jumps (jumps to other code segments). When the target for the conditional jump is in a different segment, use the opposite condition from the condition being tested for the Jcc instruction, and then access the target with an unconditional far jump (JMP instruction) to the other segment. For example, the following conditional far jump is illegal:

JZ FARLABEL;

To accomplish this far jump, use the following two instructions:

JNZ BEYOND;
JMP FARLABEL;
BEYOND:

The JECXZ and JCXZ instructions differ from the other Jcc instructions because they do not check the status flags. Instead they check the contents of the ECX and CX registers, respectively, for 0. Either the CX or ECX register is chosen according to the address-size attribute. These instructions are useful at the beginning of a conditional loop that terminates with a conditional loop instruction (such as LOOPNE). They prevent entering the loop when the ECX or CX register is equal to 0, which would cause the loop to execute $2^{12}$ or 64K times, respectively, instead of zero times.

All conditional jumps are converted to code fetches of one or two cache lines, regardless of jump address or cacheability.

Operation

IF condition
THEN
  EIP ← EIP + SignExtend(DEST);
  IF OperandSize ← 16
  THEN
    EIP ← EIP AND 0000FFFFH;
  FI;
  ELSE (* OperandSize = 32 *)
    IF EIP < CS.Base OR EIP > CS.Limit
      #GP
  FI;
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
Closer look at JGE

• JGE jumps if and only if SF = OF

Examples using 8-bit registers. Which of these result in a jump?

1. MOV AL, 96  
   CMP AL, 80  
   JGE Somewhere

2. MOV AL, -64  
   CMP AL, 80  
   JGE Somewhere

3. MOV AL, 64  
   CMP AL, -80  
   JGE Somewhere

4. MOV AL, 64  
   CMP AL, 80  
   JGE Somewhere

• if OF=0, then use SF to check whether A-B >= 0.

• if OF=1, then do opposite of SF.

• JGE works after a CMP instruction, even when subtracting the operands result in an overflow!
Short Jumps vs Near Jumps

• Jumps use relative addressing
  ◦ Assembler computes an “offset” from address of current instruction
  ◦ Code produced is “relocatable”

• Short jumps use 8-bit offsets
  ◦ Target label must be -128 bytes to +127 bytes away
  ◦ Conditional jumps use short jumps by default. To use a near jump:
    JGE NEAR Somewhere

• Near jumps use 32-bit offsets
  ◦ Target label must be $-2^{32}$ to $+2^{32}-1$ bytes away (4 gigabyte range)
  ◦ Unconditional jumps use near jumps by default. To use a short jump:
    JMP SHORT Somewhere
; File: jmp.asm
;
; Demonstrating near and short jumps
;
    section .text
    global _start

_start:    nop

    ; initialize

start:    mov    eax, 17       ; eax := 17
            cmp    eax, 42       ; 17 - 42 is ...
            jge    exit         ; exit if 17 >= 42
            jge    short exit
            jge    near exit
            jmp    exit
            jmp    short exit
            jmp    near exit

exit:     mov    ebx, 0        ; exit code, 0=normal
            mov    eax, 1        ; Exit.
            int    080H         ; Call kernel.
; File: jmp.asm

; Demonstrating near and short jumps

section .text

global _start

_start: nop

; initialize

start:    mov    eax, 17  ; eax := 17
cmp       eax, 42     ; 17 - 42 is ...

jge       exit        ; exit if 17 >= 42

jge       short exit

jge       near exit

jmp       exit

jmp       short exit

jmp       near exit

exit:     mov    ebx, 0  ; exit code, 0=normal
mov        eax, 1      ; Exit.

int        080H        ; Call kernel.
Converting an if Statement

```c
if (x < y) {
    statement block 1 ;
} else {
    statement block 2 ;
}
```

```asm
MOV    EAX,[x]
CMP    EAX,[y]
JGE    ElsePart
       ; if part
       ; statement block 1
.
JMP    Done     ; skip over else part

ElsePart:
       ; else part
       ; statement block 2
.

Done:
```
Converting a while Loop

```c
while(i > 0) {
    statement 1;
    statement 2;
    ...  
}
```

**WhileTop:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV EAX,[i]</td>
<td></td>
</tr>
<tr>
<td>CMP EAX, 0</td>
<td></td>
</tr>
<tr>
<td>JLE Done</td>
<td></td>
</tr>
<tr>
<td></td>
<td>; statement 1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>; statement 2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>JMP WhileTop</td>
<td></td>
</tr>
</tbody>
</table>

**Done:**
Logical (bit manipulation) Instructions

• **AND**: used to clear bits (store 0 in the bits):
  
  ◦ To clear the lower 4 bits of the AL register:

  \[
  \begin{array}{c}
  \text{AND} \\
  \text{AL, F0h} \\
  \hline
  1101 \ 0110 \\
  1111 \ 0000 \\
  1101 \ 0000
  \end{array}
  \]

• **OR**: used to set bits (store 1 in the bits):
  
  ◦ To set the lower 4 bits of the AL register:

  \[
  \begin{array}{c}
  \text{OR} \\
  \text{AL, 0Fh} \\
  \hline
  1101 \ 0110 \\
  0000 \ 1111 \\
  1101 \ 1111
  \end{array}
  \]

• **NOT**: flip all the bits

• **Shift and Rotate instructions** move bits around
AND—Logical AND

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>24 ib</td>
<td>AND AL, imm8</td>
<td>AL AND imm8</td>
</tr>
<tr>
<td>25 iw</td>
<td>AND AX, imm16</td>
<td>AX AND imm16</td>
</tr>
<tr>
<td>25 id</td>
<td>AND EAX, imm32</td>
<td>EAX AND imm32</td>
</tr>
<tr>
<td>80 i4 ib</td>
<td>AND r/m8, imm8</td>
<td>r/m8 AND imm8</td>
</tr>
<tr>
<td>81 i4 iw</td>
<td>AND r/m16, imm16</td>
<td>r/m16 AND imm16</td>
</tr>
<tr>
<td>81 i4 id</td>
<td>AND r/m32, imm32</td>
<td>r/m32 AND imm32</td>
</tr>
<tr>
<td>83 i4 ib</td>
<td>AND r/m16, imm8</td>
<td>r/m16 AND imm8 (sign-extended)</td>
</tr>
<tr>
<td>83 i4 ib</td>
<td>AND r/m32, imm8</td>
<td>r/m32 AND imm8 (sign-extended)</td>
</tr>
<tr>
<td>20 /r</td>
<td>AND r/m8, r8</td>
<td>r/m8 AND r8</td>
</tr>
<tr>
<td>21 lr</td>
<td>AND r/m16, r16</td>
<td>r/m16 AND r16</td>
</tr>
<tr>
<td>21 lr</td>
<td>AND r/m32, r32</td>
<td>r/m32 AND r32</td>
</tr>
<tr>
<td>22 lr</td>
<td>AND r8, r/m8</td>
<td>r8 AND r/m8</td>
</tr>
<tr>
<td>23 lr</td>
<td>AND r16, r/m16</td>
<td>r16 AND r/m16</td>
</tr>
<tr>
<td>23 lr</td>
<td>AND r32, r/m32</td>
<td>r32 AND r/m32</td>
</tr>
</tbody>
</table>

Description

Performs a bitwise AND operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is set to 1 if both corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

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

Operation

DEST ← DEST AND SRC;

Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

Protected Mode Exceptions

#GP(0)  If the destination operand points to a non-writable 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.
OR—Logical Inclusive OR

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0C ib</td>
<td>OR AL,imm8</td>
<td>AL OR imm8</td>
</tr>
<tr>
<td>0D iw</td>
<td>OR AX,imm16</td>
<td>AX OR imm16</td>
</tr>
<tr>
<td>0D id</td>
<td>OR EAX,imm32</td>
<td>EAX OR imm32</td>
</tr>
<tr>
<td>80 /1 ib</td>
<td>OR r/m8,imm8</td>
<td>r/m8 OR imm8</td>
</tr>
<tr>
<td>81 /1 iw</td>
<td>OR r/m16,imm16</td>
<td>r/m16 OR imm16</td>
</tr>
<tr>
<td>81 /1 id</td>
<td>OR r/m32,imm32</td>
<td>r/m32 OR imm32</td>
</tr>
<tr>
<td>83 /1 ib</td>
<td>OR r/m16,imm8</td>
<td>r/m16 OR imm8 (sign-extended)</td>
</tr>
<tr>
<td>83 /1 ib</td>
<td>OR r/m32,imm8</td>
<td>r/m32 OR imm8 (sign-extended)</td>
</tr>
<tr>
<td>08 /r</td>
<td>OR r/m8, r8</td>
<td>r/m8 OR r8</td>
</tr>
<tr>
<td>09 /r</td>
<td>OR r/m16, r16</td>
<td>r/m16 OR r16</td>
</tr>
<tr>
<td>09 /r</td>
<td>OR r/m32, r32</td>
<td>r/m32 OR r32</td>
</tr>
<tr>
<td>0A /r</td>
<td>OR r8, r/m8</td>
<td>r8 OR r/m8</td>
</tr>
<tr>
<td>0B /r</td>
<td>OR r16, r/m16</td>
<td>r16 OR r/m16</td>
</tr>
<tr>
<td>0B /r</td>
<td>OR r32, r/m32</td>
<td>r32 OR r/m32</td>
</tr>
</tbody>
</table>

Description

Performs a bitwise inclusive OR operation between the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result of the OR instruction is set to 0 if both corresponding bits of the first and second operands are 0; otherwise, each bit is set to 1.

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

Operation

DEST ← DEST OR SRC;

Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

Protected Mode Exceptions

#GP(0) If the destination operand points to 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.
NOT—One's Complement Negation

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F6 /2</td>
<td>NOT r/m8</td>
<td>Reverse each bit of r/m8</td>
</tr>
<tr>
<td>F7 /2</td>
<td>NOT r/m16</td>
<td>Reverse each bit of r/m16</td>
</tr>
<tr>
<td>F7 /2</td>
<td>NOT r/m32</td>
<td>Reverse each bit of r/m32</td>
</tr>
</tbody>
</table>

Description

Performs a bitwise NOT operation (each 1 is cleared to 0, and each 0 is set to 1) on the destination operand and stores the result in the destination operand location. The destination operand can be a register or a memory location.

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

Operation

DEST ← NOT DEST;

Flags Affected

None.

Protected Mode Exceptions

#GP(0) If the destination operand points to 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.

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.
## INSTRUCTION SET REFERENCE

### SAL/SAR/SHL/SHR—Shift

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0 /4</td>
<td>SAL r/m8,1</td>
<td>Multiply r/m8 by 2, once</td>
</tr>
<tr>
<td>D2 /4</td>
<td>SAL r/m8,CL</td>
<td>Multiply r/m8 by 2, CL times</td>
</tr>
<tr>
<td>C0 /4 ib</td>
<td>SAL r/m8,imm8</td>
<td>Multiply r/m8 by 2, imm8 times</td>
</tr>
<tr>
<td>D1 /4</td>
<td>SAL r/m16,1</td>
<td>Multiply r/m16 by 2, once</td>
</tr>
<tr>
<td>D3 /4</td>
<td>SAL r/m16,CL</td>
<td>Multiply r/m16 by 2, CL times</td>
</tr>
<tr>
<td>C1 /4 ib</td>
<td>SAL r/m16,imm8</td>
<td>Multiply r/m16 by 2, imm8 times</td>
</tr>
<tr>
<td>D1 /4</td>
<td>SAL r/m32,1</td>
<td>Multiply r/m32 by 2, once</td>
</tr>
<tr>
<td>D3 /4</td>
<td>SAL r/m32,CL</td>
<td>Multiply r/m32 by 2, CL times</td>
</tr>
<tr>
<td>C1 /4 ib</td>
<td>SAL r/m32,imm8</td>
<td>Multiply r/m32 by 2, imm8 times</td>
</tr>
<tr>
<td>D0 /7</td>
<td>SAR r/m8,1</td>
<td>Signed divide* r/m8 by 2, once</td>
</tr>
<tr>
<td>D2 /7</td>
<td>SAR r/m8,CL</td>
<td>Signed divide* r/m8 by 2, CL times</td>
</tr>
<tr>
<td>C0 /7 ib</td>
<td>SAR r/m8,imm8</td>
<td>Signed divide* r/m8 by 2, imm8 times</td>
</tr>
<tr>
<td>D1 /7</td>
<td>SAR r/m16,1</td>
<td>Signed divide* r/m16 by 2, once</td>
</tr>
<tr>
<td>D3 /7</td>
<td>SAR r/m16,CL</td>
<td>Signed divide* r/m16 by 2, CL times</td>
</tr>
<tr>
<td>C1 /7 ib</td>
<td>SAR r/m16,imm8</td>
<td>Signed divide* r/m16 by 2, imm8 times</td>
</tr>
<tr>
<td>D1 /7</td>
<td>SAR r/m32,1</td>
<td>Signed divide* r/m32 by 2, once</td>
</tr>
<tr>
<td>D3 /7</td>
<td>SAR r/m32,CL</td>
<td>Signed divide* r/m32 by 2, CL times</td>
</tr>
<tr>
<td>C1 /7 ib</td>
<td>SAR r/m32,imm8</td>
<td>Signed divide* r/m32 by 2, imm8 times</td>
</tr>
<tr>
<td>D0 /4</td>
<td>SHL r/m8,1</td>
<td>Multiply r/m8 by 2, once</td>
</tr>
<tr>
<td>D2 /4</td>
<td>SHL r/m8,CL</td>
<td>Multiply r/m8 by 2, CL times</td>
</tr>
<tr>
<td>C0 /4 ib</td>
<td>SHL r/m8,imm8</td>
<td>Multiply r/m8 by 2, imm8 times</td>
</tr>
<tr>
<td>D1 /4</td>
<td>SHL r/m16,1</td>
<td>Multiply r/m16 by 2, once</td>
</tr>
<tr>
<td>D3 /4</td>
<td>SHL r/m16,CL</td>
<td>Multiply r/m16 by 2, CL times</td>
</tr>
<tr>
<td>C1 /4 ib</td>
<td>SHL r/m16,imm8</td>
<td>Multiply r/m16 by 2, imm8 times</td>
</tr>
<tr>
<td>D1 /4</td>
<td>SHL r/m32,1</td>
<td>Multiply r/m32 by 2, once</td>
</tr>
<tr>
<td>D3 /4</td>
<td>SHL r/m32,CL</td>
<td>Multiply r/m32 by 2, CL times</td>
</tr>
<tr>
<td>C1 /4 ib</td>
<td>SHL r/m32,imm8</td>
<td>Multiply r/m32 by 2, imm8 times</td>
</tr>
<tr>
<td>D0 /5</td>
<td>SHR r/m8,1</td>
<td>Unsigned divide r/m8 by 2, once</td>
</tr>
<tr>
<td>D2 /5</td>
<td>SHR r/m8,CL</td>
<td>Unsigned divide r/m8 by 2, CL times</td>
</tr>
<tr>
<td>C0 /5 ib</td>
<td>SHR r/m8,imm8</td>
<td>Unsigned divide r/m8 by 2, imm8 times</td>
</tr>
<tr>
<td>D1 /5</td>
<td>SHR r/m16,1</td>
<td>Unsigned divide r/m16 by 2, once</td>
</tr>
<tr>
<td>D3 /5</td>
<td>SHR r/m16,CL</td>
<td>Unsigned divide r/m16 by 2, CL times</td>
</tr>
<tr>
<td>C1 /5 ib</td>
<td>SHR r/m16,imm8</td>
<td>Unsigned divide r/m16 by 2, imm8 times</td>
</tr>
<tr>
<td>D1 /5</td>
<td>SHR r/m32,1</td>
<td>Unsigned divide r/m32 by 2, once</td>
</tr>
<tr>
<td>D3 /5</td>
<td>SHR r/m32,CL</td>
<td>Unsigned divide r/m32 by 2, CL times</td>
</tr>
<tr>
<td>C1 /5 ib</td>
<td>SHR r/m32,imm8</td>
<td>Unsigned divide r/m32 by 2, imm8 times</td>
</tr>
</tbody>
</table>

**NOTE:**

* Not the same form of division as IDIV; rounding is toward negative infinity.
SAL/SAR/SHL/SHR—Shift (Continued)

**Description**
Shifts the bits in the first operand (destination operand) to the left or right by the number of bits specified in the second operand (count operand). Bits shifted beyond the destination operand boundary are first shifted into the CF flag, then discarded. At the end of the shift operation, the CF flag contains the last bit shifted out of the destination operand.

The destination operand can be a register or a memory location. The count operand can be an immediate value or register CL. The count is masked to 5 bits, which limits the count range to 0 to 31. A special opcode encoding is provided for a count of 1.

The shift arithmetic left (SAL) and shift logical left (SHL) instructions perform the same operation; they shift the bits in the destination operand to the left (toward more significant bit locations). For each shift count, the most significant bit of the destination operand is shifted into the CF flag, and the least significant bit is cleared (see Figure 7-7 in the *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*).

The shift arithmetic right (SAR) and shift logical right (SHR) instructions shift the bits of the destination operand to the right (toward less significant bit locations). For each shift count, the least significant bit of the destination operand is shifted into the CF flag, and the most significant bit is either set or cleared depending on the instruction type. The SHR instruction clears the most significant bit (see Figure 7-8 in the *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*); the SAR instruction sets or clears the most significant bit to correspond to the sign (most significant bit) of the original value in the destination operand. In effect, the SAR instruction fills the empty bit position’s shifted value with the sign of the unshifted value (see Figure 7-9 in the *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*).

The SAR and SHR instructions can be used to perform signed or unsigned division, respectively, of the destination operand by powers of 2. For example, using the SAR instruction to shift a signed integer 1 bit to the right divides the value by 2.

Using the SAR instruction to perform a division operation does not produce the same result as the IDIV instruction. The quotient from the IDIV instruction is rounded toward zero, whereas the “quotient” of the SAR instruction is rounded toward negative infinity. This difference is apparent only for negative numbers. For example, when the IDIV instruction is used to divide -9 by 4, the result is -2 with a remainder of -1. If the SAR instruction is used to shift -9 right by two bits, the result is -3 and the “remainder” is +3; however, the SAR instruction stores only the most significant bit of the remainder (in the CF flag).

The OF flag is affected only on 1-bit shifts. For left shifts, the OF flag is cleared to 0 if the most-significant bit of the result is the same as the CF flag (that is, the top two bits of the original operand were the same); otherwise, it is set to 1. For the SAR instruction, the OF flag is cleared for all 1-bit shifts. For the SHR instruction, the OF flag is set to the most-significant bit of the original operand.
INSTRUCTION SET REFERENCE

SAL/SAR/SHL/SHR—Shift (Continued)

IA-32 Architecture Compatibility

The 8086 does not mask the shift count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the shift count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

Operation

tempCOUNT ← (COUNT AND 1FH);
tempDEST ← DEST;
WHILE (tempCOUNT ≠ 0)
DO
  IF instruction is SAL or SHL
    THEN
      CF ← MSB(DEST);
    ELSE (* instruction is SAR or SHR *)
      CF ← LSB(DEST);
  FI;
  IF instruction is SAL or SHL
    THEN
      DEST ← DEST * 2;
    ELSE
      IF instruction is SAR
        THEN
          DEST ← DEST / 2 (*Signed divide, rounding toward negative infinity*);
        ELSE (* instruction is SHR *)
          DEST ← DEST / 2 ; (* Unsigned divide *);
      FI;
  FI;
  tempCOUNT ← tempCOUNT – 1;
OD;
(* Determine overflow for the various instructions *)
IF COUNT ← 1
  THEN
    IF instruction is SAL or SHL
      THEN
        OF ← MSB(DEST) XOR CF;
      ELSE
        IF instruction is SAR
          THEN
            OF ← 0;
        ELSE (* instruction is SHR *)
          OF ← MSB(tempDEST);
      FI;
  FI;
SAL/SAR/SHL/SHR—Shift (Continued)

ELSE IF COUNT ← 0
    THEN
        All flags remain unchanged;
    ELSE (* COUNT neither 1 or 0 *)
        OF ← undefined;
    FI;
FI;

Flags Affected

The CF flag contains the value of the last bit shifted out of the destination operand; it is undefined for SHL and SHR instructions where the count is greater than or equal to the size (in bits) of the destination operand. The OF flag is affected only for 1-bit shifts (see “Description” above); otherwise, it is undefined. The SF, ZF, and PF flags are set according to the result. If the count is 0, the flags are not affected. For a non-zero count, the AF flag is undefined.

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.

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.
7.2.4. Logical Instructions

The logical instructions AND, OR, XOR (exclusive or), and NOT perform the standard Boolean operations for which they are named. The AND, OR, and XOR instructions require two operands; the NOT instruction operates on a single operand.

7.2.5. Shift and Rotate Instructions

The shift and rotate instructions rearrange the bits within an operand. These instructions fall into the following classes:

- Shift.
- Double shift.
- Rotate.

7.2.5.1. SHIFT INSTRUCTIONS

The SAL (shift arithmetic left), SHL (shift logical left), SAR (shift arithmetic right), SHR (shift logical right) instructions perform an arithmetic or logical shift of the bits in a byte, word, or doubleword.

The SAL and SHL instructions perform the same operation (see Figure 7-7). They shift the source operand left by from 1 to 31 bit positions. Empty bit positions are cleared. The CF flag is loaded with the last bit shifted out of the operand.

<table>
<thead>
<tr>
<th>Initial State</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 1 1 1</td>
</tr>
</tbody>
</table>

After 1-bit SHL/SAL Instruction

| 1  | 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 1 1 1 |

After 10-bit SHL/SAL Instruction

| 0  | 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 |

Figure 7-7. SHL/SAL Instruction Operation

The SHR instruction shifts the source operand right by from 1 to 31 bit positions (see Figure 7-8). As with the SHL/SAL instruction, the empty bit positions are cleared and the CF flag is loaded with the last bit shifted out of the operand.
The SAR instruction shifts the source operand right by from 1 to 31 bit positions (see Figure 7-9). This instruction differs from the SHR instruction in that it preserves the sign of the source operand by clearing empty bit positions if the operand is positive or setting the empty bits if the operand is negative. Again, the CF flag is loaded with the last bit shifted out of the operand.

The SAR and SHR instructions can also be used to perform division by powers of 2 (see “SAL/SAR/SHL/SHR—Shift Instructions” in Chapter 3 of the Intel Architecture Software Developer’s Manual, Volume 2).
PROGRAMMING WITH THE GENERAL-PURPOSE INSTRUCTIONS

The SAR instruction shifts the source operand right by from 1 to 31 bit positions (see Figure 7-9). This instruction differs from the SHR instruction in that it preserves the sign of the source operand by clearing empty bit positions if the operand is positive or setting the empty bits if the operand is negative. Again, the CF flag is loaded with the last bit shifted out of the operand.

The SAR and SHR instructions can also be used to perform division by powers of 2 (see “SAL/SAR/SHL/SHR—Shift Instructions” in Chapter 3 of the Intel Architecture Software Developer’s Manual, Volume 2).

Figure 7-8. SHR Instruction Operation

Figure 7-9. SAR Instruction Operation
### RCL/RCR/ROL/ROR—Rotate

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0 /2</td>
<td>RCL r/m8, 1</td>
<td>Rotate 9 bits (CF, r/m8) left once</td>
</tr>
<tr>
<td>D2 /2</td>
<td>RCL r/m8, CL</td>
<td>Rotate 9 bits (CF, r/m8) left CL times</td>
</tr>
<tr>
<td>C0 /2 ib</td>
<td>RCL r/m8, imm8</td>
<td>Rotate 9 bits (CF, r/m8) left imm8 times</td>
</tr>
<tr>
<td>D1 /2</td>
<td>RCL r/m16, 1</td>
<td>Rotate 17 bits (CF, r/m16) left once</td>
</tr>
<tr>
<td>D3 /2</td>
<td>RCL r/m16, CL</td>
<td>Rotate 17 bits (CF, r/m16) left CL times</td>
</tr>
<tr>
<td>C1 /2 ib</td>
<td>RCL r/m16, imm8</td>
<td>Rotate 17 bits (CF, r/m16) left imm8 times</td>
</tr>
<tr>
<td>D1 /3</td>
<td>RCL r/m32, 1</td>
<td>Rotate 33 bits (CF, r/m32) left once</td>
</tr>
<tr>
<td>D3 /3</td>
<td>RCL r/m32, CL</td>
<td>Rotate 33 bits (CF, r/m32) left CL times</td>
</tr>
<tr>
<td>C1 /3 ib</td>
<td>RCL r/m32, imm8</td>
<td>Rotate 33 bits (CF, r/m32) left imm8 times</td>
</tr>
<tr>
<td>D0 /0</td>
<td>RCR r/m8, 1</td>
<td>Rotate 9 bits (CF, r/m8) right once</td>
</tr>
<tr>
<td>D2 /0</td>
<td>RCR r/m8, CL</td>
<td>Rotate 9 bits (CF, r/m8) right CL times</td>
</tr>
<tr>
<td>C0 /0 ib</td>
<td>RCR r/m8, imm8</td>
<td>Rotate 9 bits (CF, r/m8) right imm8 times</td>
</tr>
<tr>
<td>D1 /0</td>
<td>RCR r/m16, 1</td>
<td>Rotate 17 bits (CF, r/m16) right once</td>
</tr>
<tr>
<td>D3 /0</td>
<td>RCR r/m16, CL</td>
<td>Rotate 17 bits (CF, r/m16) right CL times</td>
</tr>
<tr>
<td>C1 /0 ib</td>
<td>RCR r/m16, imm8</td>
<td>Rotate 17 bits (CF, r/m16) right imm8 times</td>
</tr>
<tr>
<td>D1 /0</td>
<td>RCR r/m32, 1</td>
<td>Rotate 33 bits (CF, r/m32) right once</td>
</tr>
<tr>
<td>D3 /0</td>
<td>RCR r/m32, CL</td>
<td>Rotate 33 bits (CF, r/m32) right CL times</td>
</tr>
<tr>
<td>C1 /0 ib</td>
<td>RCR r/m32, imm8</td>
<td>Rotate 33 bits (CF, r/m32) right imm8 times</td>
</tr>
<tr>
<td>D0 /1</td>
<td>ROR r/m8, 1</td>
<td>Rotate 8 bits r/m8 right once</td>
</tr>
<tr>
<td>D2 /1</td>
<td>ROR r/m8, CL</td>
<td>Rotate 8 bits r/m8 right CL times</td>
</tr>
<tr>
<td>C0 /1 ib</td>
<td>ROR r/m8, imm8</td>
<td>Rotate 8 bits r/m8 right imm8 times</td>
</tr>
<tr>
<td>D1 /1</td>
<td>ROR r/m16, 1</td>
<td>Rotate 16 bits r/m16 right once</td>
</tr>
<tr>
<td>D3 /1</td>
<td>ROR r/m16, CL</td>
<td>Rotate 16 bits r/m16 right CL times</td>
</tr>
<tr>
<td>C1 /1 ib</td>
<td>ROR r/m16, imm8</td>
<td>Rotate 16 bits r/m16 right imm8 times</td>
</tr>
<tr>
<td>D1 /1</td>
<td>ROR r/m32, 1</td>
<td>Rotate 32 bits r/m32 right once</td>
</tr>
<tr>
<td>D3 /1</td>
<td>ROR r/m32, CL</td>
<td>Rotate 32 bits r/m32 right CL times</td>
</tr>
<tr>
<td>C1 /1 ib</td>
<td>ROR r/m32, imm8</td>
<td>Rotate 32 bits r/m32 right imm8 times</td>
</tr>
</tbody>
</table>
RCL/RCR/ROL/ROR—Rotate (Continued)

Description

Shifts (rotates) the bits of the first operand (destination operand) the number of bit positions specified in the second operand (count operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the count operand is an unsigned integer that can be an immediate or a value in the CL register. The processor restricts the count to a number between 0 and 31 by masking all the bits in the count operand except the 5 least-significant bits.

The rotate left (ROL) and rotate through carry left (RCL) instructions shift all the bits toward more-significant bit positions, except for the most-significant bit, which is rotated to the least-significant bit location (see Figure 7-11 in the IA-32 Intel Architecture Software Developer’s Manual, Volume I). The rotate right (ROR) and rotate through carry right (RCR) instructions shift all the bits toward less significant bit positions, except for the least-significant bit, which is rotated to the most-significant bit location (see Figure 7-11 in the IA-32 Intel Architecture Software Developer’s Manual, Volume I).

The RCL and RCR instructions include the CF flag in the rotation. The RCL instruction shifts the CF flag into the least-significant bit and shifts the most-significant bit into the CF flag (see Figure 7-11 in the IA-32 Intel Architecture Software Developer’s Manual, Volume I). The RCR instruction shifts the CF flag into the most-significant bit and shifts the least-significant bit into the CF flag (see Figure 7-11 in the IA-32 Intel Architecture Software Developer’s Manual, Volume I). For the ROL and ROR instructions, the original value of the CF flag is not a part of the result, but the CF flag receives a copy of the bit that was shifted from one end to the other.

The OF flag is defined only for the 1-bit rotates; it is undefined in all other cases (except that a zero-bit rotate does nothing, that is affects no flags). For left rotates, the OF flag is set to the exclusive OR of the CF bit (after the rotate) and the most-significant bit of the result. For right rotates, the OF flag is set to the exclusive OR of the two most-significant bits of the result.

IA-32 Architecture Compatibility

The 8086 does not mask the rotation count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the rotation count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

Operation

(* RCL and RCR instructions *)
SIZE ← OperandSize
CASE (determine count) OF
    SIZE ← 8:  tempCOUNT ← (COUNT AND 1FH) MOD 9;
    SIZE ← 16: tempCOUNT ← (COUNT AND 1FH) MOD 17;
    SIZE ← 32: tempCOUNT ← COUNT AND 1FH;
ESAC;
INSTRUCTION SET REFERENCE

RCL/RCR/ROL/ROR—Rotate (Continued)

(* RCL instruction operation *)
WHILE (tempCOUNT ≠ 0)
  DO
    tempCF ← MSB(DEST);
    DEST ← (DEST * 2) + CF;
    CF ← tempCF;
    tempCOUNT ← tempCOUNT – 1;
  OD;
  ELIHW;
  IF COUNT ← 1
    THEN OF ← MSB(DEST) XOR CF;
    ELSE OF is undefined;
  FI;
(* RCR instruction operation *)
IF COUNT ← 1
  THEN OF ← MSB(DEST) XOR CF;
  ELSE OF is undefined;
FI;
WHILE (tempCOUNT ≠ 0)
  DO
    tempCF ← LSB(SRC);
    DEST ← (DEST / 2) + (CF * 2^SIZE);
    CF ← tempCF;
    tempCOUNT ← tempCOUNT – 1;
  OD;
(* ROL and ROR instructions *)
SIZE ← OperandSize
CASE (determine count) OF
  SIZE ← 8:  tempCOUNT ← COUNT MOD 8;
  SIZE ← 16: tempCOUNT ← COUNT MOD 16;
  SIZE ← 32: tempCOUNT ← COUNT MOD 32;
ESAC;
(* ROL instruction operation *)
WHILE (tempCOUNT ≠ 0)
  DO
    tempCF ← MSB(DEST);
    DEST ← (DEST * 2) + tempCF;
    tempCOUNT ← tempCOUNT – 1;
  OD;
  ELIHW;
  CF ← LSB(DEST);
  IF COUNT ← 1
    THEN OF ← MSB(DEST) XOR CF;
    ELSE OF is undefined;
  FI;
RCL/RCR/ROL/ROR—Rotate (Continued)

(* ROR instruction operation *)

WHILE (tempCOUNT ≠ 0)
DO
    tempCF ← LSB(SRC);
    DEST ← (DEST / 2) + (tempCF ≠ 2^SIZE);
    tempCOUNT ← tempCOUNT – 1;
OD;
ELIHW;
CF ← MSB(Dest);
IF COUNT ← 1
    THEN OF ← MSB(Dest) XOR MSB – 1(Dest);
    ELSE OF is undefined;
FI;

Flags Affected

The CF flag contains the value of the bit shifted into it. The OF flag is affected only for single-bit rotates (see “Description” above); it is undefined for multi-bit rotates. The SF, ZF, AF, and PF flags are not affected.

Protected Mode Exceptions

#GP(0) If the source operand 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.

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.
The ROL instruction rotates the bits in the operand to the left (toward more significant bit locations). The ROR instruction rotates the operand right (toward less significant bit locations).

The RCL instruction rotates the bits in the operand to the left, through the CF flag). This instruction treats the CF flag as a one-bit extension on the upper end of the operand. Each bit which exits from the most significant bit location of the operand moves into the CF flag. At the same time, the bit in the CF flag enters the least significant bit location of the operand.

The RCR instruction rotates the bits in the operand to the right through the CF flag.

For all the rotate instructions, the CF flag always contains the value of the last bit rotated out of the operand, even if the instruction does not use the CF flag as an extension of the operand. The value of this flag can then be tested by a conditional jump instruction (JC or JNC).

### 7.2.6. Bit and Byte Instructions

The bit and byte instructions operate on bit or byte strings. They are divided into four groups:

- Bit test and modify instructions.
Example using AND, OR & SHL

- **Copy bits 4-7 of BX to bits 8-11 of AX**

  \[ AX = 0110 \text{ 1011 1001 0110} \]
  \[ BX = 1101 \text{ 0011 1100 0001} \]

1. Clear bits 8-11 of AX & all but bits 4-7 of BX using AND instructions

  \[ AX = 0110 \text{ 0000 1001 0110} \quad \text{AND} \quad AX, \text{ F0FFh} \]
  \[ BX = 0000 \text{ 0000 1100 0000} \quad \text{AND} \quad BX, \text{ 00F0h} \]

2. Shift bits 4-7 of BX to the desired position using a SHL instruction

  \[ AX = 0110 \text{ 0000 1001 0110} \]
  \[ BX = 0000 \text{ 1100 0000 0000} \quad \text{SHL} \quad BX, \text{ 4} \]

3. “Copy” bits of 4-7 of BX to AX using an OR instruction

  \[ AX = 0110 \text{ 1100 1001 0110} \quad \text{OR} \quad AX, \text{ BX} \]
  \[ BX = 0000 \text{ 1100 0000 0000} \]
More Arithmetic Instructions

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

<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;
F1;
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 /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 bytes</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.
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.
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
Next Time

• Indexed addressing: [ESI + 4*ECX + 1024]

• Example: a complex i386 instruction

• More NASM assembler directives
References

• Some figures and diagrams from IA-32 Intel Architecture Software Developer's Manual, Vols 1-3
  <http://developer.intel.com/design/Pentium4/manuals/>