x86 Assembly Language IV

CMSC 313
Sections 01, 02

Short vs. Near Jumps

Short Jumps vs. Near Jumps

- Jumps use relative addressing
  - assembler computes an *offset* from address of current instruction.
  - produces *relocatable* code
- SHORT jumps use 8-bit offsets
  - target label within -128 bytes to +127 bytes
- NEAR jumps use 32-bit offsets
  - target label within -2^{32} bytes to $+2^{32}-1$ bytes

SHORT JUMPS VS NEAR JUMPS

- **Some assemblers determine SHORT vs NEAR jumps automatically, but *some do not.***
- **explicitly specify SHORT jumps**
  - jmp SHORT somewhere
- **explicitly specify NEAR jumps**
  - jge NEAR somewhere

4

Short Jumps vs. Near Jumps

- Some assemblers determine SHORT vs NEAR jumps automatically, but *some do not*.
- explicitly specify SHORT jumps

```
jmp  SHORT somewhere
```

- explicitly specify NEAR jumps

```
jge  NEAR somewhere
```

5

```
; 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.
```

```
1                                  ; File: jmp.asm
2                                  ;
3                                  ; Demonstrating near and short jumps
4                                  ;
5
6                                          section .text
7                                          global _start
8
9  00000000 90                     _start: nop
10
11                                         ; initialize
12
13 00000001 B811000000             start:  mov     eax, 17         ; eax := 17
14 00000006 3D2A000000                     cmp     eax, 42         ; 17 - 42 is ...
15
16 0000000B 7D14                           jge     exit            ; exit if 17 >= 42
17 0000000D 7D12                           jge     short exit
18 0000000F 0F8D0C000000                   jge     near exit
19
20 00000015 E907000000                     jmp     exit
21 0000001A EB05                           jmp     short exit
22 0000001C E900000000                     jmp     near exit
23
24 00000021 BB00000000             exit:   mov     ebx, 0          ; exit code, 0=normal
25 00000026 B801000000                     mov     eax, 1          ; Exit.
26 0000002B CD80                           int     080H            ; Call kernel.
```

Bit Manipulation

8

Logical (Bit Manipulation) Instructions

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

```
AND AL, F0h          1101 0110
                     1111 0000
                     1101 0000
```

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

```
OR AL, 0Fh           1101 0110
                     0000 1111
                     1101 1111
```

- NOT: flip all the bits
- Shift and Rotate instructions move bits around

9

AND—Logical AND

Opcode	Instruction	Description
24 *ib*	AND AL,*imm8*	AL AND *imm8*
25 *iw*	AND AX,*imm16*	AX AND *imm16*
25 *id*	AND EAX,*imm32*	EAX AND *imm32*
80 /4 *ib*	AND *rm8*,*imm8*	*rm8* AND *imm8*
81 /4 *iw*	AND *rm16*,*imm16*	*rm16* AND *imm16*
81 /4 *id*	AND *rm32*,*imm32*	*rm32* AND *imm32*
83 /4 *ib*	AND *rm16*,*imm8*	*rm16* AND *imm8* (sign-extended)
83 /4 *ib*	AND *rm32*,*imm8*	*rm32* AND *imm8* (sign-extended)
20 /r	AND *rm8*,*r8*	*rm8* AND *r8*
21 /r	AND *rm16*,*r16*	*rm16* AND *r16*
21 /r	AND *rm32*,*r32*	*rm32* AND *r32*
22 /r	AND *r8*,*rm8*	*r8* AND *rm8*
23 /r	AND *r16*,*rm16*	*r16* AND *rm16*
23 /r	AND *r32*,*rm32*	*r32* AND *rm32*

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 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.

10

OR—Logical Inclusive OR

Opcode	Instruction	Description
0C *ib*	OR AL,*imm8*	AL OR *imm8*
0D *iw*	OR AX,*imm16*	AX OR *imm16*
0D *id*	OR EAX,*imm32*	EAX OR *imm32*
80 /1 *ib*	OR *rm8*,*imm8*	*rm8* OR *imm8*
81 /1 *iw*	OR *rm16*,*imm16*	*rm16* OR *imm16*
81 /1 *id*	OR *rm32*,*imm32*	*rm32* OR *imm32*
83 /1 *ib*	OR *rm16*,*imm8*	*rm16* OR *imm8* (sign-extended)
83 /1 *ib*	OR *rm32*,*imm8*	*rm32* OR *imm8* (sign-extended)
08 /r	OR *rm8*,*r8*	*rm8* OR *r8*
09 /r	OR *rm16*,*r16*	*rm16* OR *r16*
09 /r	OR *rm32*,*r32*	*rm32* OR *r32*
0A /r	OR *r8*,*rm8*	*r8* OR *rm8*
0B /r	OR *r16*,*rm16*	*r16* OR *rm16*
0B /r	OR *r32*,*rm32*	*r32* OR *rm32*

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.

11

NOT—One's Complement Negation

Opcode	Instruction	Description
F6 /2	NOT *rm8*	Reverse each bit of *rm8*
F7 /2	NOT *rm16*	Reverse each bit of *rm16*
F7 /2	NOT *rm32*	Reverse each bit of *rm32*

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.

12

intel

SAL/SAR/SHL/SHR—Shift

Opcode	Instruction	Description
D0 /4	SAL rm8,1	Multiply rm8 by 2, once
D2 /4	SAL rm8,CL	Multiply rm8 by 2, CL times
C0 /4 ib	SAL rm8,imm8	Multiply rm8 by 2, imm8 times
D1 /4	SAL rm16,1	Multiply rm16 by 2, once
D3 /4	SAL rm16,CL	Multiply rm16 by 2, CL times
C1 /4 ib	SAL rm16,imm8	Multiply rm16 by 2, imm8 times
D1 /4	SAL rm32,1	Multiply rm32 by 2, once
D3 /4	SAL rm32,CL	Multiply rm32 by 2, CL times
C1 /4 ib	SAL rm32,imm8	Multiply rm32 by 2, imm8 times
D0 /7	SAR rm8,1	Signed divide* rm8 by 2, once
D2 /7	SAR rm8,CL	Signed divide* rm8 by 2, CL times
C0 /7 ib	SAR rm8,imm8	Signed divide* rm8 by 2, imm8 times
D1 /7	SAR rm16,1	Signed divide* rm16 by 2, once
D3 /7	SAR rm16,CL	Signed divide* rm16 by 2, CL times
C1 /7 ib	SAR rm16,imm8	Signed divide* rm16 by 2, imm8 times
D1 /7	SAR rm32,1	Signed divide* rm32 by 2, once
D3 /7	SAR rm32,CL	Signed divide* rm32 by 2, CL times
C1 /7 ib	SAR rm32,imm8	Signed divide* rm32 by 2, imm8 times
D0 /4	SHL rm8,1	Multiply rm8 by 2, once
D2 /4	SHL rm8,CL	Multiply rm8 by 2, CL times
C0 /4 ib	SHL rm8,imm8	Multiply rm8 by 2, imm8 times
D1 /4	SHL rm16,1	Multiply rm16 by 2, once
D3 /4	SHL rm16,CL	Multiply rm16 by 2, CL times
C1 /4 ib	SHL rm16,imm8	Multiply rm16 by 2, imm8 times
D1 /4	SHL rm32,1	Multiply rm32 by 2, once
D3 /4	SHL rm32,CL	Multiply rm32 by 2, CL times
C1 /4 ib	SHL rm32,imm8	Multiply rm32 by 2, imm8 times
D0 /5	SHR rm8,1	Unsigned divide rm8 by 2, once
D2 /5	SHR rm8,CL	Unsigned divide rm8 by 2, CL times
C0 /5 ib	SHR rm8,imm8	Unsigned divide rm8 by 2, imm8 times
D1 /5	SHR rm16,1	Unsigned divide rm16 by 2, once
D3 /5	SHR rm16,CL	Unsigned divide rm16 by 2, CL times
C1 /5 ib	SHR rm16,imm8	Unsigned divide rm16 by 2, imm8 times
D1 /5	SHR rm32,1	Unsigned divide rm32 by 2, once
D3 /5	SHR rm32,CL	Unsigned divide rm32 by 2, CL times
C1 /5 ib	SHR rm32,imm8	Unsigned divide rm32 by 2, imm8 times

NOTE:
* Not the same form of division as IDIV; rounding is toward negative infinity.

13

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.

14

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;
```

15

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.

16

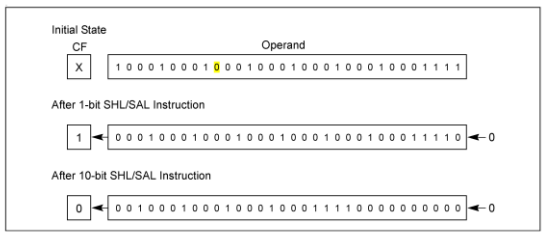


Figure 7-7. SHL/SAL Instruction Operation

17

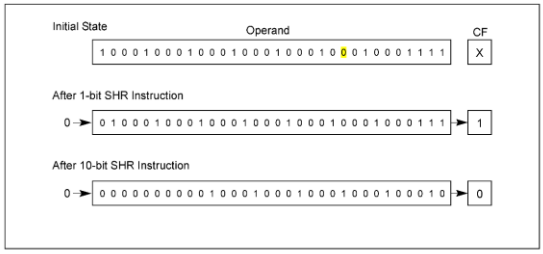


Figure 7-8. SHR Instruction Operation

18

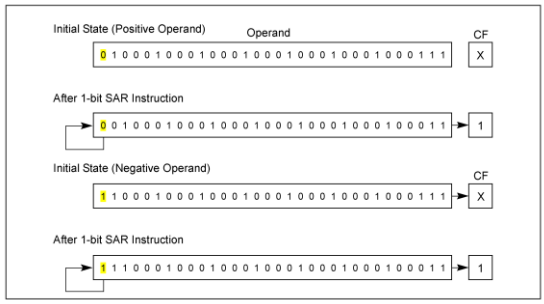


Figure 7-9. SAR Instruction Operation

19

RCL/RCR/ROL/ROR—Rotate

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

20

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 1*). 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 1*).

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 1*). 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 1*). 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;
```

21

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;
```


22

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.

23

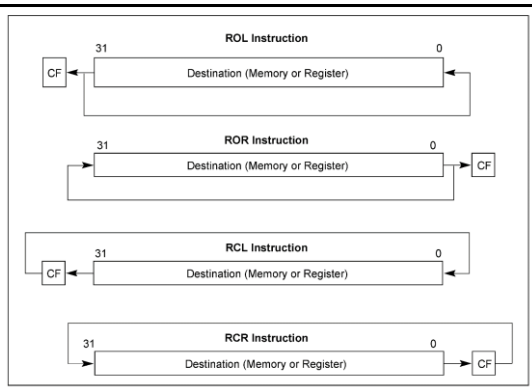


Figure 7-11. ROL, ROR, RCL, and RCR Instruction Operations

Example Using AND, OR, & SHL

- Copy bits 4-7 of BX to bits 8-11 of AX
  - AX = 0110 1011 1001 0110
  - BX = 1101 0011 1100 0001

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

```
AX = 0110 0000 1001 0110      AND  AX, F0FFh
BX = 0000 0000 1100 0000      AND  BX, 00F0h
```

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

```
AX = 0110 0000 1001 0110
BX = 0000 1100 0000 0000      SHL  BX, 4
```

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

```
AX = 0110 1100 1001 0110      OR  AX, BX
BX = 0000 1100 0000 0000
```

25

More Arithmetic Instructions

26

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

27

NEG—Two's Complement Negation

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

Description

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

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

Operation

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

Flags Affected

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

Protected Mode Exceptions

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

28

MUL—Unsigned Multiply

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

Description

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

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

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

Operation

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

Flags Affected

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

29

IMUL—Signed Multiply

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

Description

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

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

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

30

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    1; OF    1;
                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    1; OF    1;
                    ELSE CF    1; OF    1;
                FI;
            ELSE (* OperandSize    32 *)
                EDX:EAX    EAX * SRC (* signed multiplication *)
                IF ((EDX    00000000H) OR (EDX    FFFFFFFFH))
                    THEN CF    1; OF    1;
                    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 *)
```


31

IMUL—Signed Multiply (Continued)

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

Flags Affected

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

32

DIV—Unsigned Divide

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

Description

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

Table 3-19. DIV Action

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

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

Operation

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


33

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

Flags Affected

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

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

34

Virtual-8086 Mode Exceptions

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

35

Indexed Addressing Modes

36

Indexed Addressing Modes

- 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

37

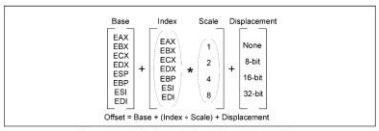


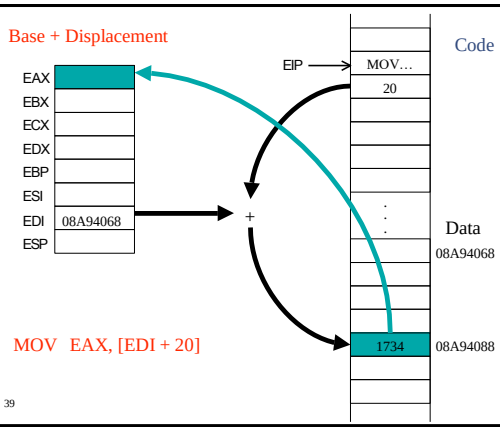
Figure 3-9. Offset (or Effective Address) Computation

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

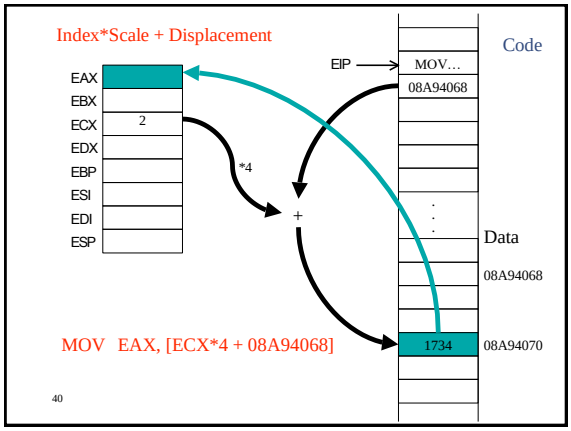
- The ESP register cannot be used as an index register.
- When the ESP or EBP register is used as the base, the SS segment is the default segment. In all other cases, the DS segment is the default segment.

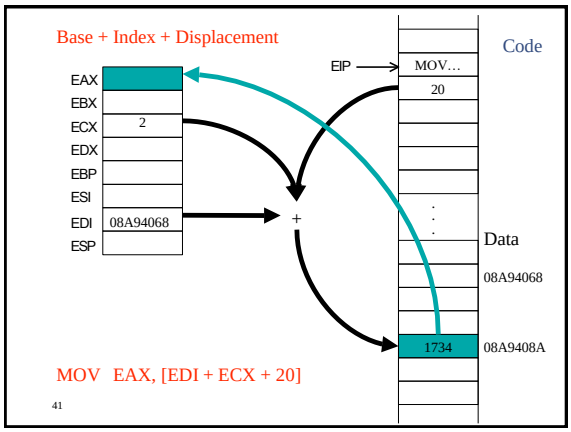
The base, index, and displacement components can be used in any combination, and any of these components can be null. A scale factor may be used only when an index also is used. Each possible combination is useful for data structures commonly used by programmers in high-level languages and assembly language.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

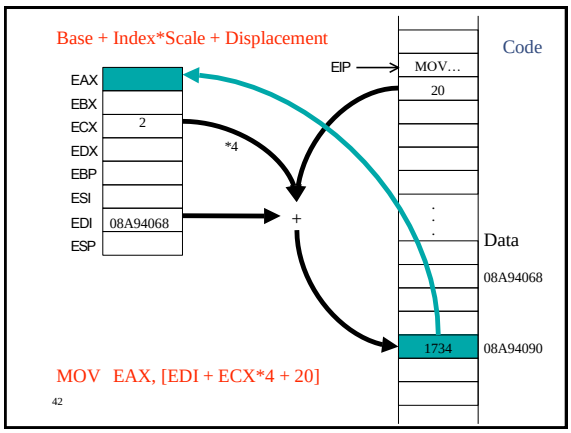
38



39


Index*Scale + Displacement
Code
EIP
MOV…
08A94068
EAX
EBX
ECX
EDX
EBP
ESI
EDI
ESP
2
*4
+
Data
08A94068
1734
08A94070
MOV EAX, [ECX*4 + 08A94068]
40


Base + Index + Displacement
Code
EIP
MOV…
20
EAX
EBX
ECX
EDX
EBP
ESI
EDI
ESP
2
08A94068
+
Data
08A94068
1734
08A9408A
MOV EAX, [EDI + ECX + 20]
41


Base + Index*Scale + Displacement
Code
EIP
MOV…
20
EAX
EBX
ECX
EDX
EBP
ESI
EDI
ESP
2
*4
08A94068
+
Data
08A94068
1734
08A94090
MOV EAX, [EDI + ECX*4 + 20]
42

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

43

```
; 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     0x80                    ; Call kernel.
```

44

```
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
0x80490c0 <arr>:        0       1       2       3
0x80490d0 <arr+16>:     4       5       6       7
0x80490e0 <arr+32>:     8       9
(gdb) cont
Continuing.

Breakpoint 2, 0x08048099 in init2 ()
(gdb) x/10wd &arr
0x80490c0 <arr>:        5       6       7       8
0x80490d0 <arr+16>:     9       10      11      12
0x80490e0 <arr+32>:     13      14
(gdb) cont
Continuing.

Breakpoint 3, 0x080480ac in init3 ()
(gdb) x/10wd &arr
0x80490c0 <arr>:        10      11      12      13
0x80490d0 <arr+16>:     14      15      16      17
0x80490e0 <arr+32>:     18      19
(gdb) cont
Continuing.

Breakpoint 4, 0x080480bf in alldone ()
(gdb) x/10wd &arr
0x80490c0 <arr>:        15      16      17      18
0x80490d0 <arr+16>:     19      20      21      22
0x80490e0 <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
```

45

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

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

rowlen: equ row2 - row1

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

        ; Add 5 to each element of row 7. Simulate:
        ;
        ;  for (i = 0 ; i < 10 ; i++) {
        ;      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
        jo      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
linux36 nasm -f elf index2.asm
linux36 ld index2.o
linux36
linux36 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, 0x8048081 in init1 ()
(gdb) x/10w &twodim
0x8049cb4 <twodim>:     0       1       2       3
0x8049cc4 <twodim+16>:  4       5       6       7
0x8049cd4 <twodim+32>:  8       9
(gdb) x/10w &twodim+60
0x8049da4 <row2+200>:   60      61      62      63
0x8049db4 <row2+216>:   64      65      66      67
0x8049dc4 <row2+232>:   68      69
(gdb)
0x8049dcc <row2+240>:   70      71      72      73
0x8049ddc <row2+256>:   74      75      76      77
0x8049dec <row2+272>:   78      79
(gdb)
0x8049df4 <row2+280>:   80      81      82      83
0x8049e04 <row2+296>:   84      85      86      87
0x8049e14 <row2+312>:   88      89
(gdb) cont
Continuing.

Breakpoint 2, 0x80480a7 in done1 ()
(gdb) x/10w &twodim+60
0x8049da4 <row2+200>:   60      61      62      63
0x8049db4 <row2+216>:   64      65      66      67
0x8049dc4 <row2+232>:   68      69
(gdb)
0x8049dcc <row2+240>:   75      76      77      78
0x8049ddc <row2+256>:   79      80      81      82
0x8049dec <row2+272>:   83      84
(gdb)
0x8049df4 <row2+280>:   80      81      82      83
0x8049e04 <row2+296>:   84      85      86      87
0x8049e14 <row2+312>:   88      89
(gdb) cont
Continuing.

Program exited normally.
(gdb) quit
linux36 exit
exit

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


References

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

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