9

Multiplication and Division Instructions

Overview

In Chapter 7, we saw how to do multiplication and division by shifting the bits in a byte or word. Left and right shifts can be used for multiplying and dividing by powers of 2. In this chapter, we introduce instructions for multiplying and dividing any numbers.

The process of multiplication and division is different for signed and unsigned numbers, so there are different instructions for signed and unsigned multiplication and division. Also, these instructions have byte and word forms. Sections 9.1 through 9.4 cover the details.

One of the most useful applications of multiplication and division is to implement decimal input and output. In section 9.5, we write procedures to carry out these operations. This application greatly extends our program's I/O capability.

9.1 MUL and IMUL

Signed Versus Unsigned Multiplication

In binary multiplication, signed and unsigned numbers must be treated differently. For example, suppose we want to multiply the eight-bit numbers 10000000 and 11111111. Interpreted as unsigned numbers, they represent 128 and 255, respectively. The product is 32,640 = 0111111110000000b. However, taken as signed numbers, they represent –128 and –1, respectively, and the product is 128 = 0000000010000000b.

Because signed and unsigned multiplication lead to different results, there are two multiplication instructions: **MUL** (multiply) for unsigned

multiplication and **IMUL** (integer multiply) for signed multiplication. These instructions multiply bytes or words. If two bytes are multiplied, the product is a word (16 bits). If two words are multiplied, the product is a doubleword (32 bits). The syntax of these instructions is

```
MUL   source
```

and

```
IMUL  source
```

Byte Form

For byte multiplication, one number is contained in the source and the other is assumed to be in AL. The 16-bit product will be in AX. The source may be a byte register or memory byte, but not a constant.

Word Form

For word multiplication, one number is contained in the source and the other is assumed to be in AX. The most significant 16 bits of the doubleword product will be in DX, and the least significant 16 bits will be in AX (we sometimes write this as DX:AX). The source may be a 16-bit register or memory word, but not a constant.

For multiplication of positive numbers (0 in the most significant bit), MUL and IMUL give the same result.

Effect of MUL/IMUL on the status flags

SF, ZF, AF, PF:	undefined
CF/OF:	
After MUL, CF/OF	= 0 if the upper half of the result is zero.
	= 1 otherwise.
After IMUL, CF/OF	= 0 if the upper half of the result is the sign extension of the lower half (this means that the bits of the upper half are the same as the sign bit of the lower half).
	= 1 otherwise.

For both MUL and IMUL, CF/OF = 1 means that the product is too big to fit in the lower half of the destination (AL for byte multiplication, AX for word multiplication).

Examples

To illustrate MUL and IMUL, we will do several examples. Because hex multiplication is usually difficult to do, we'll predict the product by converting the hex values of multiplier and multiplicand to decimal, doing decimal multiplication, and converting the product back to hex.

Example 9.1 Suppose AX contains 1 and BX contains FFFFh:

Instruction	*Decimal product*	*Hex product*	*DX*	*AX*	*CF/OF*
MUL BX	65535	0000FFFF	0000	FFFF	0
IMUL BX	−1	FFFFFFFF	FFFF	FFFF	0

For MUL, DX = 0, so CF/OF = 0.

For IMUL, the signed interpretation of BX is –1, and the product is also –1. In 32 bits, this is FFFFFFFFh. CF/OF = 0 because DX is the sign extension of AX.

Example 9.2 Suppose AX contains FFFFh and BX contains FFFFh:

Instruction	*Decimal product*	*Hex product*	*DX*	*AX*	*CF/OF*
MUL BX	4294836225	FFFE0001	FFFE	0001	1
IMUL BX	1	00000001	0000	0001	0

For MUL, CF/OF = 1 because DX is not 0. This reflects the fact that the product FFFE0001h is too big to fit in AX.

For IMUL, AX and BX both contain –1, so the product is 1. DX has the sign extension of AX, so CF/OF = 0.

Example 9.3 Suppose AX contains 0FFFh:

Instruction	*Decimal product*	*Hex product*	*DX*	*AX*	*CF/OF*
MUL AX	16769025	00FFE001	00FF	E001	1
IMUL AX	16769025	00FFE001	00FF	E001	1

Because the msb of AX is 0, both MUL and IMUL give the same product. Because the product is too big to fit in AX, CF/OF = 1.

Example 9.4 Suppose AX contains 0100h and CX contains FFFFh:

Instruction	*Decimal product*	*Hex product*	*DX*	*AX*	*CF/OF*
MUL CX	16776960	00FFFF00	00FF	FF00	1
IMUL CX	–256	FFFFFF00	FFFF	FF00	0

For MUL, the product FFFF00 is obtained by attaching two zeros to the source value FFFFh. Because the product is too big to fit in AX, CF/OF = 1.

For IMUL, AX contains 256 and CX contains –1, so the product is –256, which may be expressed as FF00h in 16 bits. DX has the sign extension of AX, so CF/OF = 0.

Example 9.5 Suppose AL contains 80h and BL contains FFh:

Instruction	*Decimal product*	*Hex product*	*AH*	*AL*	*CF/OF*
MUL BL	128	7F80	7F	80	1
IMUL BL	128	0080	00	80	1

For byte multiplication, the 16-bit product is contained in AX.

For MUL, the product is 7F80. Because the high eight bits are not 0, CF/OF = 1.

For IMUL, we have a curious situation. 80h = –128, FFh = –1, so the product is 128 = 0080h. AH does not have the sign extension of AL, so CF/OF = 1. This reflects the fact that AL does not contain the correct answer in a signed sense, because the signed decimal interpretation of 80h is –128.

9.2 Simple Applications of MUL and IMUL

To get used to programming with MUL and IMUL, we'll show how some simple operations can be carried out with these instructions.

Example 9.6 Translate the high-level language assignment statement A = 5 × A – 12 × B into assembly code. Let A and B be word variables, and suppose there is no overflow. Use IMUL for multiplication.

Solution:

```
MOV   AX,5          ;AX = 5
IMUL  A             ;AX = 5 x A
MOV   A,AX          ;A = 5 x A
MOV   AX,12         ;AX = 12
IMUL  B             ;AX = 12 x B
SUB   A,AX          ;A = 5 x A - 12 x B
```

Example 9.7 Write a procedure FACTORIAL that will compute $N!$ for a positive integer N. The procedure should receive N in CX and return $N!$ in AX. Suppose that overflow does not occur.

Solution: The definition of $N!$ is

$$N! = 1 \text{ if } N = 1$$
$$= N \times (N-1) \times (N-2) \times \ldots \times 1 \text{ if } N > 1$$

Here is an algorithm:

```
product = 1
term = N
FOR N times DO
 product = product x term
 term = term - 1
ENDFOR
```

It can be coded as follows:

```
FACTORIAL       PROC
;computes N!
;input: CX = N
;output: AX = N!
                MOV   AX,1      ;AX holds product
TOP:
                MUL   CX        ;product = product x term
                LOOP  TOP
                RET
FACTORIAL       ENDP
```

Here CX is both loop counter and term; the LOOP instruction automatically decrements it on each iteration through the loop. We assume the product does not overflow 16 bits.

9.3 DIV and IDIV

When division is performed, we obtain two results, the quotient and the remainder. As with multiplication, there are separate instructions for unsigned and signed division; **DIV** (divide) is used for unsigned division and **IDIV** (integer divide) for signed division. The syntax is

```
DIV divisor
```

and

```
IDIV divisor
```

These instructions divide 8 (or 16) bits into 16 (or 32) bits. The quotient and remainder have the same size as the divisor.

Byte Form

In this form, the divisor is an 8-bit register or memory byte. The 16-bit dividend is assumed to be in AX. After division, the 8-bit quotient is in AL and the 8-bit remainder is in AH. The divisor may not be a constant.

Word Form

Here the divisor is a 16-bit register or memory word. The 32-bit dividend is assumed to be in DX:AX. After division, the 16-bit quotient is in AX and the 16-bit remainder is in DX. The divisor may not be a constant.

For signed division, the remainder has the same sign as the dividend. If both dividend and divisor are positive, DIV and IDIV give the same result.

The effect of DIV/IDIV on the flags is that all status flags are undefined.

Divide Overflow

It is possible that the quotient will be too big to fit in the specified destination (AL or AX). This can happen if the divisor is much smaller than the dividend. When this happens, the program terminates (as shown later) and the system displays the message "*Divide Overflow*".

Example 9.8 Suppose DX contains 0000h, AX contains 0005h, and BX contains 0002h.

Instruction	*Decimal quotient*	*Decimal remainder*	*AX*	*DX*
DIV BX	2	1	0002	0001
IDIV BX	2	1	0002	0001

Dividing 5 by 2 yields a quotient of 2 and a remainder of 1. Because both dividend and divisor are positive, DIV and IDIV give the same results.

Example 9.9 Suppose DX contains 0000h, AX contains 0005h, and BX contains FFFEh.

Instruction	*Decimal quotient*	*Decimal remainder*	*AX*	*DX*
DIV BX	0	5	0000	0005
IDIV BX	-2	1	FFFE	0001

For DIV, the dividend is 5 and the divisor is FFFEh = 65534; 5 divided by 65534 yields a quotient of 0 and a remainder of 5.

For IDIV, the dividend is 5 and the divisor is FFFEh = -2; 5 divided by -2 gives a quotient of -2 and a remainder of 1.

Example 9.10 Suppose DX contains FFFFh, AX contains FFFBh, and BX contains 0002.

Instruction	*Decimal quotient*	*Decimal remainder*	*AX*	*DX*
IDIV BX	-2	-1	FFFE	FFFF
DIV BX	DIVIDE OVERFLOW			

For IDIV, DX:AX = FFFFFFFBh = -5, BX = 2. -5 divided by 2 gives a quotient of -2 = FFFEh and a remainder of -1 = FFFFh.

For DIV, the dividend DX:AX = FFFFFFFBh = 4294967291 and the divisor = 2. The actual quotient is 2147483646 = 7FFFFFFEh. This is too big to fit in AX, so the computer prints DIVIDE OVERFLOW and the program terminates. This shows what can happen if the divisor is a lot smaller than the dividend.

Example 9.11 Suppose AX contains 00FBh and BL contains FFh.

Instruction	*Decimal quotient*	*Decimal remainder*	*AX*	*AL*
DIV BL	0	251	F8	00
IDIV BL	DIVIDE OVERFLOW			

For byte division, the dividend is in AX; the quotient is in AL and the remainder in AH.

For DIV, the dividend is 00FBh = 251 and the divisor is FFh = 256. Dividing 251 by 256 yields a quotient of 0 and a remainder of 251 = FBh.

For IDIV, the dividend is 00FBh = 251 and the divisor is FFh = -1. Dividing 251 by -1 yields a quotient of -251, which is too big to fit in AL, so the message DIVIDE OVERFLOW is printed.

9.4 Sign Extension of the Dividend

Word Division

In word division, the dividend is in DX:AX even if the actual dividend will fit in AX. In this case DX should be prepared as follows:

1. For DIV, DX should be cleared.
2. For IDIV, DX should be made the sign extension of AX. The instruction **CWD** (convert word to doubleword) will do the extension.

Example 9.12 Divide –1250 by 7.

Solution:

```
MOV  AX,-1250        ;AX gets dividend
CWD                  ;Extend sign to DX
MOV  BX,7            ;BX has divisor
IDIV BX              ;AX gets quotient, DX has remainder
```

Byte Division

In byte division, the dividend is in AX. If the actual dividend is a byte, then AH should be prepared as follows:

1. For DIV, AH should be cleared.
2. For IDIV, AH should the sign extension of AL. The instruction **CBW** (convert byte to word) will do the extension.

Example 9.13 Divide the signed value of the byte variable XBYTE by –7.

Solution:

```
MOV  AL,XBYTE        ;AL has dividend
CBW                  ;Extend sign to AH
MOV  BL,-7           ;BL has divisor
IDIV BL              ;AL has quotient, AH has remainder
```

There is no effect of CBW and CWD on the flags.

9.5 Decimal Input and Output Procedures

Even though the computer represents everything in binary, it's more convenient for the user to see input and output expressed in decimal. In this section, we write procedures for handling decimal I/O.

On input, if we type 21543, for example, then we are actually typing a character string, which must be converted internally to the binary equivalent of the decimal integer 21543. Conversely on output, the binary contents of a register or memory location must be converted to a character string representing a decimal integer before being printed.

Decimal Output

We will write a procedure OUTDEC to print the contents of AX as a signed decimal integer. If AX >= 0, OUTDEC will print the contents in decimal; if AX < 0, OUTDEC will print a minus sign, replace AX by –AX (so that AX now contains a positive number), and print the contents in decimal. Thus in either case, the problem comes down to printing the decimal equivalent of a positive binary number. Here is the algorithm:

Algorithm for Decimal Output

1. IF AX < 0 /* AX holds output value */
2. THEN

```
3.   print a minus sign
4.   replace AX by its two's complement
5. END_IF
6. Get the digits in AX's decimal representation
7. Convert these digits to characters and print them
```

To see what line 6 entails, suppose the content of AX, expressed in decimal, is 24168. To get the digits in the decimal representation, we can proceed as follows:

Divide 24618 by 10. Quotient = 2461, remainder = 8
Divide 2461 by 10. Quotient = 246, remainder = 1
Divide 246 by 10. Quotient = 24, remainder = 6
Divide 24 by 10. Quotient = 2, remainder = 4
Divide 2 by 10. Quotient = 0, remainder = 2

Thus, the digits we want appear as remainders after repeated division by 10. However, they appear in reverse order; to turn them around, we can save them on the stack. Here's how line 6 breaks down:

Line 6

```
count = 0  /* will count decimal digits */
REPEAT
   divide quotient by 10
   push remainder on the stack
   count = count + 1
UNTIL quotient = 0
```

where the initial value of quotient is the original contents of AX.

Once the digits are on the stack, all we have to do is pop them off, convert them to characters, and print them. Line 7 may be expressed as follows:

Line 7

```
FOR count times DO
   pop a digit from the stack
   convert it to a character
   output the character
END_FOR
```

Now we can code the procedure as follows:

Program Listing PGM9_1.ASM

```
1:  OUTDEC  PROC
2:  ;prints AX as a signed decimal integer
3:  ;input: AX
4:  ;output: none
5:          PUSH  AX          ;save registers
6:          PUSH  BX
7:          PUSH  CX
8:          PUSH  DX
9:  ;if AX < 0
10:         OR    AX,AX       ;AX < 0?
11:         JGE   @END_IF1    ;NO, > 0
12: ;then
```

```
13:           PUSH   AX              ;save number
14:           MOV    DL,'-'          ;get '-'
15:           MOV    AH,2            ;print char function
16:           INT    21H             ;print '-'
17:           POP    AX              ;get AX back
18:           NEG    AX              ;AX = -AX
19:  @END_IF1:
20:  ;get decimal digits
21:           XOR    CX,CX           ;CX counts digits
22:           MOV    BX,10D          ;BX has divisor
23:  @REPEAT1:
24:           XOR    DX,DX           ;prepare high word of dividend
25:           DIV    BX              ;AX = quotient, DX = remainder
26:           PUSH   DX              ;save remainder on stack
27:           INC    CX              ;count = count + 1
28:  ;until
29:           OR     AX,AX           ;quotient = 0?
30:           JNE    @REPEAT1        ;no, keep going
31:  ;convert digits to characters and print
32:           MOV    AH,2            ;print char function
33:  ;for count times do
34:  @PRINT_LOOP:
35:           POP    DX              ;digit in DL
36:           OR     DL,30H          ;convert to character
37:           INT    21H             ;print digit
38:           LOOP   @PRINT_LOOP     ;loop until done
39:  ;end_for
40:           POP    DX              ;restore registers
41:           POP    CX
42:           POP    BX
43:           POP    AX
44:           RET
45:  OUTDEC   ENDP
```

After saving the registers, at line 10 the sign of AX is examined by ORing AX with itself. If AX >= 0, the program jumps to line 19; if AX < 0, a minus sign is printed and AX is replaced by its two's complement. In either case, at line 19, AX will contain a positive number.

At line 21, OUTDEC prepares for division. Because division by a constant is illegal, we must put the divisor 10 in a register.

The REPEAT loop in lines 23–30 will get the digits and put them on the stack. Because we'll be doing unsigned division, DX is cleared. After division, the quotient will be in AX and the remainder in DX (actually it is in DL, because the remainder is between 0 and 9). At line 29, AX is tested for 0 by ORing it with itself; repeated division by 10 guarantees a zero quotient eventually.

The FOR loop in lines 34–38 gets the digits from the stack and prints them. Before a digit is printed, it must first be converted to an ASCII character (line 36).

The INCLUDE Pseudo-op

We can verify OUTDEC by placing it inside a short program and running the program inside DEBUG. To insert OUTDEC into the program without having to type it in, we use the **INCLUDE** pseudo-op. It has the form

```
INCLUDE     filespec
```

where filespec identifies a file (with optional drive and path). For example the file containing OUTDEC is PGM9_1.ASM. We could use

```
INCLUDE A:PGM9_1.ASM
```

When MASM encounters this line during assembly, it retrieves file PGM9_1.ASM from the disk in drive A and inserts it into the program at the position of the INCLUDE directive. This file is on the Student Data Disk that comes with this book.

Here is the testing program:

Program Listing PGM9_2.ASM

```
TITLE PGM9_2: DECIMAL OUTPUT
.MODEL  SMALL
.STACK  100H
.CODE
MAIN   PROC
       CALL   OUTDEC
       MOV    AH,4CH
       INT    21H          ;DOS exit
MAIN   ENDP
INCLUDE A:PGM9_1.ASM
       END    MAIN
```

To test the program, we'll enter DEBUG and run the program twice, first for AX = -25487 = 9C71h and then for AX = 654 = 28Eh:

```
C>DEBUG PGM9_2.EXE
-RAX
AX 0000
:9C71
-G
-25487                    (first output)
Program terminated normally.
-RAX
AX 9C71
:28E
-G
654                       (second output)
```

Note that after the first run, DEBUG automatically resets IP to the beginning of the program.

Decimal Input

To do decimal input, we need to convert a string of ASCII digits to the binary representation of a decimal integer. We will write a procedure INDEC to do this.

In procedure OUTDEC, to output the contents of AX in decimal we repeatedly divided AX by 10. For INDEC we need repeated multiplication by 10. The basic idea is the following:

Decimal Input Algorithm (first version)

```
total = 0
read an ASCII digit
REPEAT
 convert character to a binary value
 total = 10 x total + value
 read a character
UNTIL character is a carriage return
```

For example, an input of 123 is processed as follows:

```
total = 0
read '1'
convert '1' to 1
total = 10 x 0 + 1 = 1
read '2'
convert '2' to 2
total = 10 x 1 + 2 = 12
read '3'
convert '3' to 3
total = 10 x 12 + 3 = 123
```

We will design INDEC so that it can handle signed decimal integers in the range -32768 to 32767. The program prints a question mark, and lets the user enter an optional sign, followed by a string of digits, followed by a carriage return. If the user enters a character outside the range "0" . . . "9", the procedure goes to a new line and starts over. With these added requirements, the preceding algorithm becomes the following:

Decimal Input Algorithm (second version)

```
Print a question mark
total = 0
negative = false
Read a character
CASE character OF
 '-': negative = true
      read a character
 '+': read a character
END_CASE
REPEAT
 IF character is not between '0' and '9'
  THEN
   go to beginning
  ELSE
   convert character to a binary value
    total = 10 x total + value
  END_IF
  read a character
UNTIL character is a carriage return
IF negative = true
 THEN
  total = -total
ENDIF
```

Note: A jump like this is not really "structured programming." Sometimes it's necessary to violate structure rules for the sake of efficiency, for example, when error conditions occur.

The algorithm can be coded as follows:

Program Listing PGM9_3.ASM

```
1:  INDEC   PROC
2:  ;reads a number in range -32768 to 32767
3:  ;input: none
4:  ;output: AX = binary equivalent of number
5:          PUSH  BX            ;save registers used
6:          PUSH  CX
7:          PUSH  DX
8:  ;print prompt
9:  @BEGIN:
10:         MOV   AH,2
11:         MOV   DL,'?'
12:         INT   21H           ;print '?'
13: ;total = 0
14:         XOR   BX,BX         ;BX holds total
15: ;negative = false
16:         XOR   CX,CX         ;CX holds sign
17: ;read a character
18:         MOV   AH,1
19:         INT   21H           ;character in AL
20: ;case character of
21:         CMP   AL,'-'        ;minus sign?
22:         JE    @MINUS        ;yes, set sign
23:         CMP   AL,'+'        ;plus sign
24:         JE    @PLUS         ;yes, get another character
25:         JMP   @REPEAT2      ;start processing characters
26: @MINUS:
27:         MOV   CX,1          ;negative = true
28: @PLUS:
29:         INT   21H           ;read a character
30: ;end_case
31: @REPEAT2:
32: ;if character is between '0' and '9'
33:         CMP   AL,'0'        ;character >= '0'?
34:         JNGE  @NOT_DIGIT    ;illegal character
35:         CMP   AL,'9'        ;character <= '9'?
36:         JNLE  @NOT_DIGIT    ;no, illegal character
37: ;then convert character to a digit
38:         AND   AX,000FH      ;convert to digit
39:         PUSH  AX            ;save on stack
40: ;total = total x 10 + digit
41:         MOV   AX,10         ;get 10
42:         MUL   BX            ;AX = total x 10
43:         POP   BX            ;retrieve digit
44:         ADD   BX,AX         ;total = total x 10 + digit
45: ;read a character
46:         MOV   AH,1
47:         INT   21H
48:         CMP   AL,0DH        ;carriage return?
49:         JNE   @REPEAT2      ;no, keep going
50: ;until CR
51:         MOV   AX,BX         ;store number in AX
52: ;if negative
53:         OR    CX,CX         ;negative number
```

```
54:             JE      @EXIT           ;no, exit
55: ;then
56:             NEG     AX              ;yes, negate
57: ;end_if
58: @EXIT:
59:             POP     DX              ;restore registers
60:             POP     CX
61:             POP     BX
62:             RET                     ;and return
63: ;here if illegal character entered
64: @NOT_DIGIT:
65:             MOV     AH,2            ;move cursor to a new line
66:             MOV     DL,0DH
67:             INT     21H
68:             MOV     DL,0AH
69:             INT     21H
70:             JMP     @BEGIN          ;go to beginning
71: INDEC       ENDP
```

The procedure begins by saving the registers and printing a "?". BX holds the total; in line 14, it is cleared.

CX is used to keep track of the sign; 0 means a positive number and 1 means negative. We initially assume the number is positive, so CX is cleared at line 16.

The first character is read at lines 18 and 19. It could be "+", "-" or a digit. If it's a sign, CX is adjusted if necessary and another character is read (line 29). Presumably this next character will be a digit.

At line 31, INDEC enters the REPEAT loop, which processes the current character and reads another one, until a carriage return is typed.

At lines 33–36, INDEC checks to see if the current character is in fact a digit. If not, the procedure jumps to label @NOT_DIGIT (line 64), moves the cursor to a new line, and jumps to @BEGIN. This means that the user can't escape from the procedure without entering a legitimate number.

If the current character in AL is a decimal digit, it is converted to a binary value (line 38). Then the value is saved on the stack (line 39), because AX is used when the total is multiplied by 10.

In lines 41 and 42, the total in BX is multiplied by 10. The product will be in DX:AX; however, DX will contain 0 unless the number is out of range (more about this later). At line 43, the value saved is popped from the stack and 10 times total is added to it.

At line 51, INDEC exits the REPEAT loop with the number in BX. After moving it to AX, INDEC checks the sign in CX; if CX contains 1, AX is negated before the procedure exits.

Testing INDEC

We can test INDEC by creating a program that uses INDEC for input and OUTDEC for output.

Program Listing PGM9_4.ASM

```
TITLE PGM9_4: DECIMAL I/O
.MODEL SMALL
.STACK
.CODE
MAIN PROC
```

```
;input a number
       CALL   INDEC          ;number in AX
       PUSH   AX             ;save number
;move cursor to a new line
       MOV    AH,2
       MOV    DL,0DH
       INT    21H
       MOV    DL,0AH
       INT    21H
;output the number
       POP    AX             ;retrieve number
       CALL   OUTDEC
;dos exit
       MOV    AH,4CH
       INT    21H
MAIN   ENDP
INCLUDE A:PGM9_1.ASM      ;include OUTDEC
INCLUDE A:PGM9_3.ASM      ;include INDEC
       END    MAIN
```

Sample execution:

```
C>PGM9_4
?21345
21345Overflow
```

Overflow

Procedure INDEC can handle input that contains illegal characters, but it cannot handle input that is outside the range -32768 to 32767. We call this *input overflow*.

Overflow can occur in two places in INDEC: (1) when total is multiplied by 10, and (2) when a value is added to total. As an example of the first overflow, the user could enter 99999; overflow occurs when the total = 9999 is multiplied by 10. As an example of the second overflow, if the user types 32769, then when the total = 32760, overflow occurs when 9 is added. The algorithm can be made to perform overflow checks as follows:

Decimal Input Algorithm (third version)

```
Print a question mark
total = 0
negative = false
Read a character
CASE character OF
  '-': negative = true
       read a character
  '+': read a character
END_CASE
REPEAT
 IF character is not between '0' and '9'
```

```
  THEN
  go to beginning
  ELSE
   convert character to a value
   total = 10 x total
   IF overflow
    THEN
     go to beginning
    ELSE
     total = total + value
     IF overflow
      THEN
       go to beginning
     END_IF
    END_IF
   END_IF
   read a character
UNTIL character is a carriage return
IF negative = true
 THEN
  total = -total
END_IF
```

The implementation of this algorithm is left to the student as an exercise.

Summary

- The multiplication instructions are MUL for unsigned multiplication and IMUL for signed multiplication.
- For byte multiplication, AL holds one number, and the other is in an 8-bit register or memory byte. For word multiplication, AX holds one number, and the other is in an 16-bit register or memory word.
- For byte multiplication, the 16-bit product is in AX. For word multiplication, the 32-bit product is in DX:AX.
- The division instructions are DIV for unsigned division and IDIV for signed division.
- The divisor may be a memory or register, byte or word. For division by a byte, the dividend is in AX; for division by a word, the dividend is in DX:AX.
- After byte division, AL has the quotient and AH the remainder. After word division, AX has the quotient and DX the remainder.
- For signed word division, if AX contains the dividend, then CWD can be used to extend the sign into DX. Similarly, for byte division, CBW extends the sign of AL into AH. For unsigned word division, if AX contains the dividend, then DX should be cleared. For unsigned byte division, if AL contains the dividend then AH should be cleared.
- Multiply and divide instructions are useful in doing decimal I/O.
- The INCLUDE pseudo-op provides a way to insert text from an external file into a program.