Arithmetic Flags and Operations

Contents

6.1 Using flags to monitor the outcome of arithmetic operations
6.2 Addition and subtraction
6.3 Multiplication
6.4 Division
6.5 A complete program example using the arithmetic instructions

Summary
Self-check quiz

In this chapter the 8086-family instructions for the addition, subtraction, multiplication and division of signed and unsigned numbers and certain related instructions will be described. We shall also see how to resolve ambiguity as to whether the object of certain operations (not just arithmetic) is to be a word or a byte. We start by showing how the flags can be used to monitor the outcome of arithmetic operations and finish with a complete example program which shows several of the arithmetic instructions in a typical application. A complete summary of all the 8086-family arithmetic instructions discussed is given at the end of the chapter.
6.1 Using flags to monitor the outcome of arithmetic operations

Recall that, altogether, five of the bits in the flags register (Figure 6.1) are used to indicate the results of arithmetic and related operations: the O-flag (overflow), the S-flag (sign), the Z-flag (zero), the A-flag (auxiliary carry) and the C-flag (carry).

![Figure 6.1](image)
The flags register.

Each time an arithmetic instruction is executed, certain flags will be set either to 0 or to 1 to indicate the outcome of that arithmetic operation. Full details of which instructions affect which flags can be found in Appendix V but it is important to note that not all instructions affect the flags register. Indeed, different instructions affect different flags. Some instructions do not affect any of the flags; others affect all the flags we have mentioned. Remembering which instructions affect which flags comes with practice since, at first, there seems to be no obvious pattern: **ADD** and **SUB** affect all the arithmetic flags but **INC** affects all of them except the carry flag; **MOV** affects none of the flags.

We shall now consider each of the arithmetic flags in turn and describe which aspect of an arithmetic operation each of them monitors. In the course of this discussion, whether we are working with signed or unsigned numbers is crucial (see Section 1.3).

6.1.1 The carry flag

During arithmetic operations with unsigned numbers, the **carry flag** records whether certain instructions produced an unsigned number which was too big (or too small) to be held in the specified register or memory location. For example, executing:

```
MOV AL,0FFH
ADD AL,4
```

will set the carry flag to 1 because **0FFH + 4** produces a carry as the result,
103H (= 100000011B), is too large to be correctly represented in the 8-bit register AL. It is important to note, however, that AL will in fact contain 03H after execution – the correct answer without the carry bit:

\[
\begin{array}{c}
1111 \ 1111 \ (= \text{0FFH}) \\
0000 \ 0100 \ (= \text{4H}) \\
\hline
(1) \ 0000 \ 0011 \ (= \text{103H})
\end{array}
\]

Similarly, executing

\[
\begin{array}{l}
\text{MOV DH, 2} \\
\text{SUB DH, 0FFH}
\end{array}
\]

will set the carry flag to 1 because 2 – 0FFH produces a borrow since 0FFH is bigger than 2. But in this case after execution DH will contain 03H, not –0FDH which cannot all be represented using only eight bits. This is because SUB performs

\[
x - y
\]

as

\[
x + (-y)
\]

where (−y) is represented in two’s complement form. Thus, to calculate

\[
2 - 0FFH
\]

SUB has the microprocessor work out the two’s complement of 0FFH, namely 00H + 1H = 1H (see Section 1.3), and then adds that to 2, resulting in 3H:

\[
2 - 0FFH \rightarrow 2 + (\text{two’s complement of 0FFH}) \rightarrow 2 + 1 \rightarrow 3
\]

It follows that care is needed when working with unsigned numbers to ensure that the results of arithmetic operations are kept within range to maintain accuracy. Since the carry flag tells us when things have gone wrong, this is one of its most important uses.

### 6.1.2 The overflow flag

One of the main uses of the overflow flag is to ensure that the results of operations with signed numbers are within range. First note that the largest signed number which can be held in a byte is +127 in decimal or 7FH. After execution of:

\[
\begin{array}{l}
\text{MOV BL, 4H} \\
\text{ADD BL, 7FH}
\end{array}
\]

the overflow flag will be set to 1 to indicate that the result is out of the range of byte-sized signed numbers. (In fact, after execution, BL will contain 83H – the correct result for an unsigned addition, but representing −7DH as a signed number.)

Similarly, the smallest negative number representable in 8-bit signed
form is $-128$ in decimal or $80\text{H}$. Given that $-2$ represented as an 8-bit signed number is $0\text{FEH}$ and $+127$ in decimal is $7\text{FH}$ as an 8-bit signed number, then after execution of:

```
MOV BL, 0\text{FEH}
SUB BL, 0\text{7FH}
```

the overflow flag will be set to 1 indicating that the result is out of range for 8-bit signed numbers. After execution, BL will contain $7F$, the correct answer for an unsigned subtraction. But this does not represent the correct answer when $0\text{FFH}$ and $80\text{H}$ are interpreted as signed numbers: in that case the result should be $-2 - 127 = -129$.

### 6.1.3 The sign flag

The **sign flag** is only useful when operating with *signed* numbers. It indicates whether the result of an operation on signed numbers is positive (in which case the sign flag is set to 0) or negative (in which case it is set to 1). Thus, executing the program fragment

```
MOV AL, 2
SUB AL, 5
```

would set the S-flag to 1 indicating a negative result (and leave AL containing $0\text{FDH}$, the signed 8-bit form of $-3$ in decimal), whereas:

```
MOV AL, 3
SUB AL, 0\text{FFH}
```

would set the S-flag to 0 indicating a positive result (for $0\text{FFH}$ is the 8-bit signed representation of $-1$) and leave AL containing 4.

### 6.1.4 The zero flag

The **zero flag** records whether an operation produced a zero result. Thus, executing the program fragment

```
MOV AX, 2
SUB AX, 2
```

would set the zero flag to 1 indicating that the result of $\text{SUB AX, 2}$ was 0 whereas

```
MOV AX, 3
SUB AX, 2
```

would set the zero flag to 0 indicating that the result of $\text{SUB AX, 2}$ was not 0. Remember:

- If the Z-flag = 0 it means that the result of the last instruction which affected the Z-flag was not 0.
- If the Z-flag = 1 it means that the result of the last instruction which affected the Z-flag was 0.
It follows that testing if two registers, for example CX and DX, both contained the same number would involve subtracting the contents of one register from the other and checking to see if the result was 0, this latter check being carried out by inspecting the Z-flag.

### 6.1.5 The parity flag

The **parity flag** simply records whether the result of an operation contains an even number of 1s (in which case the parity flag is set to 1) or an odd number of 1s (in which case the parity flag is set to 0). Thus, after executing

```assembly
    MOV AL, 3
    ADD AL, 7
```

the parity flag will be set to 1 because the 8-bit binary form for 10 decimal is 00001010 which contains an even number of 1s. Similarly, after

```assembly
    MOV AH, 8
    SUB AH, 1
```

the parity flag will be set to 0 because the 8-bit binary form for 7 decimal is 00000111 which contains an odd number of 1s.

For ease of reference, the purposes of the flags which are most important for simple programming tasks are summarized in Table 6.1.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carry Flag</td>
<td>CF is set to 1 if an arithmetic operation on <em>unsigned</em> numbers gives a result which is out of range:</td>
</tr>
<tr>
<td></td>
<td>if an addition has produced a carry then CF is set to 1, otherwise it is set to 0;</td>
</tr>
<tr>
<td></td>
<td>if a subtraction has produced a borrow then CF is set to 1, otherwise it is set to 0;</td>
</tr>
<tr>
<td></td>
<td>CF also indicates:</td>
</tr>
<tr>
<td></td>
<td>the result of a CoMPare operation (see Section 6.2.4);</td>
</tr>
<tr>
<td></td>
<td>the bit which has been shifted or rotated out of a register or memory location (see Chapter 9); and</td>
</tr>
<tr>
<td></td>
<td>in conjunction with OF, the result of a multiplication.</td>
</tr>
<tr>
<td>Overflow Flag</td>
<td>OF is set to 1 if an arithmetic operation on <em>signed</em> numbers gives a result which is out of range. Thus OF:</td>
</tr>
<tr>
<td>(OF)</td>
<td>is set to 1 if adding two like-signed numbers or subtracting two opposite-signed numbers gives a result which requires more bits for an accurate representation than the operands themselves, otherwise it is set to 0;</td>
</tr>
<tr>
<td></td>
<td>is set to 1 if the sign bit of an operand changes during a shift operation (see Chapter 9), otherwise it is set to 0.</td>
</tr>
</tbody>
</table>
Table 6.1  (cont.)

<table>
<thead>
<tr>
<th>Flag</th>
<th>Purpose</th>
</tr>
</thead>
</table>
| In combination with CF, OF indicates the size of the result of a multiplication:  
  both CF and OF are set to 1 if the upper half of a product is non zero, otherwise both CF and OF are set to 0.  
  Is set to 1 if a division operation produces a quotient that is too big for the register which is to contain the result. |                                                                                                                                       |
| Sign Flag (SF)      | Indicates the outcome – positive or negative – of operations on signed numbers. SF takes the same value as the most significant (sign) bit of the result. |                                                                                                                                 |
| Zero Flag (ZF)      | If the result of an operation is zero, ZF is set to 1, otherwise it is set to 0.                                                                 |                                                                                                                                 |
| Parity Flag (PF)    | If the result of an operation has an even number of bits then PF is set to 1, otherwise it is set to 0. It is mainly useful when transmitting data between devices (see Chapter 20). |                                                                                                                                 |
| Direction Flag (DF) | If DF is set to 0, the 8086-family special string manipulation instructions (see Chapter 11) increment the appropriate index registers and so progress forward through memory, whereas if DF is set to 1 the index registers are decremented and so progress is backward through memory. |                                                                                                                                 |
| Auxiliary Carry Flag (AF) | Similar to CF except that it indicates the presence or absence of a carry or borrow based on a 4-bit numeric representation in bits 0, 1, 2, and 3. It is useful for operations on 'packed decimal' numbers (see Chapter 15). |                                                                                                                                 |
| Trap Flag (TF)      | By setting TF to 1 the 8086-family microprocessors can be forced to operate in single step mode (see Chapter 21).                                                                                                                                 |
| Interrupt Enable Flag (IF) | Allows interruption to normal 8086-family operation to be initiated from peripherals and other sources (see Chapter 21).                                                                                                                                 |

6.1.6  Many instructions affect several flags simultaneously

The preceding examples may have left the impression that a given instruction only affects one of the flags. In fact, as mentioned earlier, a particular instruction may affect a number flags simultaneously – or none at all. For example, both ADD and SUB affect the carry, overflow, sign, zero and parity flags. INC and DEC affect all of these except the carry flag, and MOV affects none of the flags at all. Which flags are affected by which arithmetic instructions is summarized in Table 6.2. Some of the instructions listed there we have yet to
Table 6.2  Summary of the effect on the flags of the arithmetic instructions (see also Appendix V).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Z-flag</th>
<th>C-flag</th>
<th>S-flag</th>
<th>O-flag</th>
<th>A-flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ADC</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SUB</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SBB</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>INC</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DEC</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>NEG</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CMP</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MUL</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>IMUL</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>DIV</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>IDIV</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>CBW</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>CWD</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

meet, but by the end of the chapter we shall have encountered all of them. Appendix V contains complete information about the flags affected by all 8086-family instructions.

To see what actually happens, let us trace the state of the flags through the execution of a short program sequence involving arithmetic operations. Following our notational convention established earlier, a ? denotes a flag value which we do not know.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Carry</th>
<th>Overflow</th>
<th>Sign</th>
<th>Zero</th>
<th>Parity</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV BL,3</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>MOV does not affect the flags</td>
</tr>
<tr>
<td>ADD BL,2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Decimal 5 = 00000101 in binary</td>
</tr>
<tr>
<td>SUB BL,5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Since 3 + 2 - 5 = 0</td>
</tr>
<tr>
<td>SUB BL,2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Result (-2) is now negative</td>
</tr>
<tr>
<td>Instruction</td>
<td>Carry</td>
<td>Overflow</td>
<td>Sign</td>
<td>Zero</td>
<td>Parity</td>
<td>Comment</td>
</tr>
<tr>
<td>-------------</td>
<td>-------</td>
<td>----------</td>
<td>------</td>
<td>------</td>
<td>--------</td>
<td>---------</td>
</tr>
<tr>
<td>MOV DH,2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>MOV does not affect the flags</td>
</tr>
<tr>
<td>ADD DH,OFFH</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Carry because $2 + 0FFH = 101H$</td>
</tr>
<tr>
<td>MOV AL,4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>MOV does not affect the flags</td>
</tr>
<tr>
<td>ADD AL,7FH</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Signed number overflow</td>
</tr>
</tbody>
</table>

**EXERCISES**

6.1 For each of the following program sequences, make a similar trace of the values taken by the flags as the program is executed. Verify your answers by using a debugger.

(a) MOV DH,3
    ADD DH,2
    ADD DH,1
    SUB DH,73H
    ADD DH,79H
    ADD DH,2
    ADD DH,0FAH
    ADD DH,3

(b) MOV AX,0FEH
    INC AX
    MOV BX,0
    ADD BX,AX
    DEC BX
    SUB AX,BX
    INC AX
    INC BX
    SUB AX,BX
    DEC AX
    ADD BX,BX
    SUB AX,BX
    INC BX
    INC AX

(c) MOV CL,1
    SUB CL,65H
    SUB CL,21H
    SUB CL,86H
    SUB CL,33H
    MOV CL,4
    SUB AL,CL
    MOV CH,2
    ADD CX,AX

(d) MOV AX,2
    SUB AX,OFFFH
    MOV BX,AX
    ADD BX,AX
    DIV BX
    ADD AX,AX
    MOV AX,OF24EH

(e) MOV AX,OFFFH
    ADD AX,AX
    MUL AX
    ADD CX,OE14CH
    ADD CX,OE00EH

(f) MOV AX,0110H
(f) MOV AX,1000H
(g) MOV AX,0F24FH
(g) MOV BX,0A3F5H
(h) MOV AX,9000H
(h) MOV AX,0F2FFH
(h) MOV AX,0F750H
(h) MOV AX,0AABBH
(h) MOV AX,0EEB3H
(h) MOV AX,144CH
(h) MOV BX,2H
(h) MOV BX,0EFEH

6.2 For a certain application, a programmer wanted to regard BL,AX as containing a 24-bit signed number and BL,CX as containing another signed 24-bit number with BL and DL respectively containing the higher order bits. Explain how the programmer could add these signed numbers using the ADD instruction and how it would be possible to check whether the result was zero, or involved a carry or overflow.
6.3 Design some program fragments to carry out arithmetic on signed 10D-bit numbers held in the general-purpose 8086-family registers. Explain how you would detect overflow, record any carry from an arithmetic operation and detect if the outcome of an arithmetic operation were zero.

6.2 Addition and Subtraction

The 8086-family instructions for the addition and subtraction of signed and unsigned numbers comprise INC, ADD, ADC (ADd with Carry), DEC, SUB, SBB (SubTract with Borrow), CMP (CoMPare) and NEG (NEGate).

6.2.1 INC and DEC

INC and DEC we have already met. They both affect all of the arithmetic flags except the carry flag. INC increments its operand by one; DEC decrements its operand by one. INC and DEC take the following general forms:

INC register
INC memory
DEC register
DEC memory

where register denotes any non-segment register and memory any valid byte or word memory pointer.

Thus, if AX contains 03ABH before execution of:

INC AX

then afterwards AX will contain 03ACH. If AX contains 0FFFH before execution of INC AX, then afterwards it will contain 0000H.

We can also increment bytes or words stored in memory. But to do so it is necessary to make it clear to the assembler whether a memory reference, such as that in the incorrect instruction

INC [BX] ; incorrect instruction

is to a word or a byte. This is done by means of the pseudo-ops BYTE PTR and WORD PTR respectively (PTR is short for PoinTeR). Thus:

INC BYTE PTR [BX]

will increment the byte stored at DS:BX by 1, whereas:

INC WORD PTR [BX]

will increment the word stored at DS:BX by 1.

Suppose that we have defined a word in memory in the data segment of a program (following our layout convention established in Chapter 4) as follows:
KEEP_COUNT DW ?

Then if the word referred to by KEEP_COUNT contains 0FABCH before execution of:

DEC KEEP_COUNT

then, afterwards, that word will contain 0FABBH. (This latter instruction is unambiguous because the assembler knows from the definition of KEEP_COUNT that it refers to a word).

6.2.2 ADD and ADC

Both ADD and ADC affect all of the arithmetic flags, but the effect they have is defined only for the carry and auxiliary carry flags. The ADD and ADC instructions may take the forms:

ADD register,number        ADC register,number
ADD memory,number          ADC memory,number
ADD register,register      ADC register,register
ADD register,memory        ADC register,memory
ADD memory,register        ADC memory,register

Here, a format such as:

ADD register,register

is really an abuse of notation for we actually mean

ADD register1,register2

where register1 and register2 both denote any one of the execution unit registers including AX, BX, CX, DX and their 8-bit counterparts. Thus, typical examples of instructions of the form

ADD register,register

are ADD AL, BL, ADD CX, DX, and ADD BH, BH.

number stands for any signed or un-signed number of an appropriate size, either 8 or 16 bits long, or any assembly language expression which results in a number. Typical examples of the format

ADD register,number

are therefore ADD AL, 27H, ADD DH, 00011011B, and ADD CX, 0ABC FH. Moreover, thanks to assembly language, we can write expressions such as:

ADD AL, 'N'

and the assembler will replace 'N' by the ASCII code for the letter N (4EH) and so convert this to the machine code equivalent of

ADD AL, 4EH

memory stands for any allowed specification of an address in memory.
The range of permissible specifications will grow in the course of later chapters. With that available now, typical examples of the formats

\[ \text{ADD register, memory} \]
\[ \text{ADD memory, register} \]

are ADD AX, [BX], ADD AL, MYBYTE, and ADD MYBYTE, CL.

The instruction

\[ \text{ADD DX, CX} \]

adds the contents of register DX to the contents of register CX and leaves the result in DX. Thus, if before execution DX contained 1234H and CX contained 6AB1H then after execution of

\[ \text{ADD DX, CX} \]

DX would contain 1234H + 6AB1H = 7CE5H and CX would contain 6AB1H.

If MYBYTE corresponds to an offset of 300H relative to DS then

\[ \text{ADD AL, MYBYTE} \]

adds the contents of AL and location 300H (relative to the contents of DS). ADC is similar to ADD except that the current value of the carry flag is included in any addition. For example:

\[ \text{ADC AX, [BX]} \]

adds the contents of the word in memory pointed to by register BX to the contents of register AX, and then adds the current value of the carry flag to that and leaves the result in register AX. Thus if AX contains 0AEFFH, BX contains 1023H, the word at DS:1023H in memory contains 4BBCH and the carry flag is set to 1, then after execution of

\[ \text{ADC AX, [BX]} \]

AX will contain 0AEFFH + 4BBCH + 1 = 0FABCH, BX will contain 1023H, the word at DS:1023H in memory will contain 4BBCH and the carry flag will be set to 0.

In the latter case it was clear to the assembler that the word pointed to by BX was required since we wanted something to add to AX and we always add like quantities. However, as with INC and DEC, it is necessary to resolve the ambiguity in an incorrect instruction like:

\[ \text{ADD [BX], 3 ; incorrect instruction} \]

by means of BYTE PTR and WORD PTR. Thus, in:

\[ \text{ADD WORD PTR [BX], 3} \]

3 will be added to the word at DS:BX whereas in:

\[ \text{ADD BYTE PTR [BX], 3} \]

3 will be added to the byte at DS:BX.
### 6.2.3 SUB and SBB

The SUB and SBB instructions have the same general formats as the ADD and ADC instruction described above, namely:

```
SUB register, number     SBB register, number
SUB memory, number       SBB memory, number
SUB register, register   SBB register, register
SUB register, memory     SBB register, memory
SUB memory, register     SBB memory, register
```

where `register` denotes any non-segment register and `memory` any valid byte or word memory pointer. Both affect all the arithmetic flags and set them according to the outcome of the arithmetic operation except that the effect of SUB on the auxiliary carry flag is indeterminate.

An instruction

```
SUB TOTAL, AL
```

is unambiguous because AL determines that 8-bit arithmetic will be performed (and because of the definition of TOTAL, which must be as a byte). If, before execution, the value stored at TOTAL is 37H and AL contains 21H then after execution the value stored at TOTAL will be 16H and AL will contain 21H.

An instruction of the form

```
SBB DX, BX
```

adds the current contents of the carry flag (0 or 1) to the contents of register BX and subtracts the resulting total from the contents of register DX, leaving the result in DX.

The other formats for SBB follow this pattern. Thus, if DS:BX contains 1FH, AL contains 23H and the carry flag is set to 1, then after execution of

```
SBB AL, [BX]
```

AL will contain 3H, DS:BX will contain 1FH and the carry flag will be set to 0.

### 6.2.4 CMP

As we shall see in Chapter 7, the main use of CMP (CoMPare) is in decision making in conjunction with conditional jumps. It takes one of the following forms (where we assume that comparisons are always made between numbers of the same size):

```
CMP register, number
CMP register, register
CMP memory, number
CMP register, memory
CMP memory, register
```

The CMP instruction does not change the value of any of the 8086-family
registers but simply sets the flags according to the outcome of subtracting its second operand from the first. Thus if AL contained 38H before execution of

```
CMP AL, 38H
```

then, after execution, AL would contain 38H still but the zero flag would be set to 1 indicating that 38H subtracted from the contents of AL gave zero. Similarly, if AL contained 1AH before execution of

```
CMP AL, 41H
```

then AL would still contain 41H afterwards, but the carry and sign flags would be set to 1 indicating that the result was negative. (In fact the flags are always set exactly as they would be by a SUB instruction but CMP does not change the value of the register or memory location(s) involved.)

### 6.2.5 NEG

NEG turns positive 8-bit and 16-bit signed numbers into their negative two’s complement equivalent and vice versa (NEG is short for NEGate). It takes the general forms

```
NEG register
NEG memory
```

where register denotes any non-segment register and memory any valid byte or word memory pointer. It affects all the arithmetic flags and sets the carry flag to 1 unless the value in register or memory is zero in which case it sets the carry flag to 0.

Thus, if CX contains 00000001 00011001B (the 16-bit signed representation of +25) to begin with, then after the instruction

```
NEG CX
```

has been executed, CX contains 11111111 11100111B (the 16-bit two’s complement representation of −25) and the carry flag will be set to 1. Similarly, if DS:BX contains 11111111 11110001B (the 16-bit two’s complement representation of −15), after

```
NEG WORD PTR [BX]
```

it will contain 00000001 00001111B (the 16-bit signed representation of +15) and the carry flag will be set to 1.

---

**EXERCISES**

6.4 Add the necessary assembler pseudo-ops, assemble, link and run under debugger control the following simple program:
and hence complete the following table of flag values after the execution of each instruction in this program. Try to account for what you find.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>AF</th>
<th>SF</th>
<th>OF</th>
<th>ZF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV AL,3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC AL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV BL,0FEH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC BL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD AL, BL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB AL, BL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBB AL, BL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEG AL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.5 Hand trace the execution of the following program fragments showing what happens to the flags in each case.

(a) MOV DH,3   (c) MOV CL,1
  NEG DH,2     (d) MOV AX,OF24FH
  SUB DH,1     MOV BX,0AF5H
  SUB DH,73H   ADD PX,AX
  ADD DH,79H   (e) MOV AX,9000H
  ADD DH,2     MOV AX,OFFFH
  CMP DH,0FAH   (f) MOV CX,1000H
  ADD DH,3     MOV CX,014CH

(b) MOV AX,0FEH
  INC AX       ADD CX,0EFEH
  MOV BX,0     INC AX
  ADD BX,AX    NEG BX
  NEG BX       SUB AX,BX
  SUB AX,AX    NEG AX
  NEG BX       SBB AX,BX
  SBB AX,BX    DEC AX
  DEC AX       ADC BX,BX
  ADC BX,BX    SUB AX,BX
  SUB AX,BX    NEG BX
  NEG AX
6.6 Can you write a sequence of 8086-family instructions which is *exactly* equivalent to `SBB AL, BL` (of course, you should not use the `SBB` instruction to do so)?

6.7 Repeat Exercise 6.6 for the `SUB` instruction:
(a) without using `SUB`; and
(b) without using `SUB` or `SBB`.

6.8 Can you write a sequence of 8086-family instructions which is *exactly* equivalent to `ADC AL, BL` (of course, you should not use the `ADC` instruction to do so)?

6.9 Repeat Exercise 6.8 for the `ADD` instruction:
(a) without using `ADD`; and
(b) without using `ADD` or `ADC`.

### 6.3 Multiplication

While the `ADD` and `SUB` instructions will work with both signed and unsigned numbers, the 8086 family provides separate instructions for the multiplication and division of signed and unsigned numbers. Thus for signed multiplication and division we have `IMUL` (Integer MULtiplication) and `IDIV` (Integer DIVision), and for unsigned multiplication and division we have `MUL` and `DIV`.

#### 6.3.1 Multiplication of signed numbers

The `IMUL` instruction takes one of the forms:

```
IMUL register
IMUL memory
```

where `register` denotes any non-segment register and `memory` any valid byte or word memory pointer.

Either two 8-bit numbers are multiplied together, or two 16-bit numbers. If two 8-bit numbers are to be multiplied together, one of them must be in the AL register and the other is specified by the `register` or `memory` operand in the instruction. The 16-bit result is then left in AX. For example, if AL contains 61H and CL contains 4H then after execution of

```
IMUL CL
```

AX will contain 0184H and CL will contain 4H:
\[
\begin{align*}
AL & \quad 01100001 \; (= \; 61H) \\
CL & \quad 00000100 \; (= \; 4H) \\
\hline
AX & \quad 0000000110000100 \; (= \; 0184H)
\end{align*}
\]

If two 16-bit numbers are to be multiplied together, one of them must be in the AX register and the other is specified by the register or memory operand in the instruction. The 32-bit result is then left in DX and AX with DX containing the higher order bits. Thus

**IMUL CX**

will multiply the signed 16-bit number in register AX by the signed 16-bit number in register CX and leave the signed 32-bit result in registers AX and DX, DX containing the higher order bits. If AX contains 2003H (8195 in decimal) and CX contains 0011H (the signed equivalent of 17 in decimal), then after execution of

**IMUL CX**

DX will contain 0002H and AX will contain 2033H so that DX,AX contains 00022033H (the 32-bit signed equivalent of 8195 * 17 = 139315 in decimal) and CX will be unchanged:

\[
\begin{align*}
AX & \quad 001000000000011B \\
CX & \quad 000000000010001B \\
\hline
0010000000000110000B \\
001000000000011B
\end{align*}
\]

Similarly:

**IMUL WORD PTR LIMIT**

would multiply the signed 16-bit number in register AX by the signed 16-bit number stored at the word referred to by LIMIT and leave the signed 32-bit result in DX,AX with the contents of the word at LIMIT being unchanged.

### 6.3.2 Multiplication of unsigned numbers

In this case the formats are:

**MUL register**

**MUL memory**

where *register* denotes any non-segment register and *memory* any valid byte or word memory pointer. Both of these work in exactly the same way as the corresponding **MUL** instruction but the result is unsigned.
EXERCISES

6.10 Write fragments of 8086-family assembly language which will:
(a) Leave in DX, AX the value of \((a \times (b - c) \times d)\) where \(a, b, c, d\) denote the signed 8-bit contents of AL, BL, CL and DL respectively. (Assume that the result can be accurately represented as a signed 32-bit number.)

(b) Leave \((a + b) \times (a + b) \times (c - d)\) in DX, AX where \(a, b, c\) and \(d\) are as in (a) above.

6.11 Write an assembly language program which will accept as input from its user a digit in the range 0, 1, ..., 9 inclusive and then print a new line on the display screen, print the value of double that digit, followed by a new line and then return to the operating system. Thus, if the user types 6, then the program should display 12 (= 2 * 6).

6.12 Write an assembly language program which allows its user to type in two digits in the range 0, 1, ..., 9 inclusive and then prints out their product followed by a new line and then returns to the operating system. Thus, the program should be capable of the following dialogue with its user:

First digit: 3
Second digit: 7
Product is: 21

6.4 Division

As for multiplication, there are separate instructions for signed and unsigned division – IDIV and DIV respectively. Moreover, both IDIV and DIV allow the alternatives of dividing an 8-bit number into a 16-bit number or of dividing a 16-bit number into a 32-bit one. Both do elementary school ‘long division with whole numbers’ so that dividing 7D into 36D would give a quotient of 5D and a remainder of 1D.

Both IDIV and DIV operate according to whichever of the following schemes the programmer chooses:

1. **Division into a 32-bit number**
The 32-bit number must be placed in registers DX and AX (DX containing the higher order bits). It will then be divided by the contents of the specified 16-bit register or the corresponding memory word. The result will be left in registers AX and DX: quotient in AX, remainder in DX (see Figure 6.2).
Division into a 16-bit number

The 16-bit number in register AX is divided by the contents of the specified 8-bit register or memory location and the result left in registers AL and AH: quotient in AL, remainder in AH (see Figure 6.3).

6.4.1 Division of signed numbers

DIV takes one of the forms:

\[
\text{DIV register}
\]
\[
\text{DIV memory}
\]

where register denotes any non-segment register and memory any valid byte or word memory pointer.

Thus, if AX contains 0056H and CL contains 11H then after execution of

\[
\text{DIV CL}
\]

AL will contain 5 and AH will contain 1 because 56H = (5 * 11H) + 1 that is, dividing 11H into 56H gives 5 remainder 1. (CL will be unchanged.)

Similarly, if DX,AX contains 000244ACH and the word pointed to by TOTAL contains 3H, then after execution of

\[
\text{DIV TOTAL}
\]

AX will contain 0C18BH and DX will contain 1H because 244ACH divided by 3H is 0C18BH remainder 1. (The contents of the word pointed to by TOTAL will be unchanged.)

Note that DIV always arranges that the sign of the remainder will be the same as that of the number into which you are dividing. Thus if we divide decimal −26 by decimal +7 using DIV the result will be −3 remainder −5.

6.4.2 Division of unsigned numbers

DIV takes one of the forms:

\[
\text{DIV register}
\]
\[
\text{DIV memory}
\]
where register denotes any non-segment register and memory any valid byte or word memory pointer. Both of these work in exactly the same way as the corresponding \texttt{DIV} instruction but the results are unsigned.

**Warning – division by zero**

Any attempt to divide by zero will cause your computer to resort to all manner of strange behaviour – exactly what will depend on the particular make – and you will probably have to switch it off and start running your program all over again. Consequently, great care must be taken within a program to ensure that division by zero is never attempted. (To be completely honest, the disastrous results of dividing by zero can be brought under programmer control – see Chapter 21).

### 6.4.3 CBW and CWD

The 8086-family multiplication and division instructions were designed so that the double-length result of a multiplication could be used in a division later in the same program. But facilities are also provided to enable numbers which did not arise from a former multiplication to be used easily in a division. For example, if we want to divide the 8-bit unsigned equivalent of decimal 35 (00100011) by 7 (00000111) then the \texttt{DIV} instruction expects the number we want to divide into (35) to be in 16-bit form in AX. We cannot just put 35 into AL, therefore, or whatever was in AH will form part of the number we shall divide into. What is required is an instruction to convert the 8-bit signed number in AL into a 16-bit signed equivalent in AX. Enter \texttt{CBW}.

The instruction:

\texttt{CBW}

tests the leftmost bit of AL. If it is a 1 (so that the content of AL regarded as a signed number is negative), then AH is set to 0FFH so that AX now contains the 16-bit signed equivalent of what is in AL. If it is a 0 (so that the contents of AL regarded as a signed number is positive), then AH is set to zero so that AX now contains the 16-bit signed equivalent of what was in AL.

For example, if AL contains 0F1H (the signed 8-bit form of \(-15\) in decimal) before execution of

\texttt{CBW}

then after execution, AX contains 0FFF1H which is the signed 16-bit equivalent of \(-000000000001110B + 1) = -(000000000001111B) = \(-15\) in decimal.

For a 32-bit division which did not arise as the result of a multiplication we require an instruction which will similarly convert the signed 16-bit number in AX to a signed 32-bit equivalent in DX,AX. Enter \texttt{CWD}.

The instruction

\texttt{CWD}
stores $0FFFFH$ in $DX$ if the highest order bit of $AX$ (that is, the sign bit in a signed 16-bit representation) is 1 and stores $0000H$ in $DX$ otherwise. This allows us to regard $DX$ and $AX$ as containing the signed 32-bit form of what was in $AX$, with $DX$ containing the higher order bits.

EXERCISES

6.13 Write fragments of 8086-family assembly language which will:

(a) Leave in $AX$ the value of $(a - b) / (c - d)$ where $a$, $b$, $c$, $d$ denote the signed 16-bit contents of $AX$, $BX$, $CX$ and $DX$ respectively and where we assume that $c - d$ is not zero.

(b) Leave the average value $(a + b + c) / 3$ in register $DX$ where $a$, $b$, $c$ are as in (a) above.

(c) Leave the value of $(b * c) / a$ in register $DX$ where $a$, $b$, $c$ are as in (b) above and it is assumed that the content of $AX$ is not zero.

6.14 Write a program to find the average of a collection of at most 10D digits 0,1,2, . . . ,9 typed in at the keyboard. The program first asks the user to type the number of digits to be averaged:

Type number of digits:

and waits for this to be entered and for the ENTER key to be pressed. The program then prints a new line on the display screen and requests each digit in the form

Please enter digit 1:

and waits for that digit to be typed followed by the ENTER key. When all of the digits have been typed, a new line is printed on the display screen and then the average of the digits is displayed to the nearest whole number. It then prints another new line and returns to the operating system.

For example, your program should be capable of generating the following dialogue:

Type number of digits:3
Please enter digit 1:4
Please enter digit 2:5
Please enter digit 3:7
Average is:5

6.15 As part of an investigation into the properties of the first 12 prime numbers (2,3,5,7,11,13,17,19,23,29,31 and 37) a student came up with the hypothesis that if you multiply a prime number by itself and then
subtract 1, the result is always divisible by 2 or 3. Thus:

\[(2 \times 2) - 1 = 3; (3 \times 3) - 1 = 8; (4 \times 4) - 1 = 15;\]

Write an assembly language program which checks this hypothesis out for the first 12 prime numbers and prints YES on the display screen if it is true for all of them and NO if it is not, followed by a new line and a return to the operating system.

---

6.5 A complete program example using the arithmetic instructions

We shall write a program which will find all decimal 3-digit numbers which are equal to the sum of the cubes of their digits. Thus:

\[153 = 1^3 + 5^3 + 3^3\]

The numbers to be considered are those in the range 100D to 999D. We shall store each of the three digits in a separate word in memory and initialize the three digits to the first number in the range, 100D:

```assembly
FIRST_DIGIT DW 1
SECOND_DIGIT DW 0
THIRD_DIGIT DW 0
```

The main sequence of operations in the program can be described informally by mimicking the repeat . . . until loop in Pascal:

```pascal
repeat
  cube each of the digits and add up the cubes in BX;
  put the binary version of the number the digits represent in AX;
  compare AX and BX;
  if they are the same: display the digits by first converting each digit to ASCII and then display them using INT 21H, and then display a new line;
  think of the stored digits as a decimal number and increment that number;
until 999D has been tried
```

To this we need to add instructions to establish data segment addressability and to return to DOS control when we have found all the desired three digit numbers. The complete program is given in Figure 6.4. As usual, the manner in which it has been implemented has been determined by the desire to illustrate concepts in as straightforward a way as possible – there are more beautiful ways of doing the same thing. But nevertheless, if you take the trouble to run the program, the output is quite a surprise.
At the very end of the program, note the use of

```
NEWNUMBER:  .
  .
  .
JZ FINISH
JMP NEWNUMBER
;return to dos
FINISH:MOV AX,4COOH
```

rather than the more obvious construction:

```
NEWNUMBER:  .
  .
  .
JNZ NEWNUMBER
;return to dos
MOV AX,4COOH
```

This is necessary because in the second version the target for the JNZ (the instruction labeled NEWNUMBER) is more than 128 bytes away from the JNZ instruction itself, and so would be out of range. The interested reader may replace the first version in our example program with the second to discover how MASM reports such an error.

```plaintext
DATA SEGMENT
    FIRST_DIGIT  DW 1
    SECOND_DIGIT DW 0
    THIRD_DIGIT  DW 0
DATA ENDS

WORKING_STORE SEGMENT STACK
    DW 100H DUP(?)
WORKING_STORE ENDS

CODE SEGMENT
ASSUME DS:DATA, SS:WORKING_STORE, CS:CODE

;establish data segment addressability
START:MOV AX,DATA
    MOV DS,AX
;cube digits and add into BX
NEWNUMBER:MOV AX,FIRST_DIGIT
    MOV CX,AX
    MUL CX
    MUL CX
    MOV BX,AX
    MOV AX,SECOND_DIGIT
    MOV CX,AX
    MUL CX
    MUL CX
    ADD BX,AX
    MOV AX,THIRD_DIGIT
    MOV CX,AX
    MUL CX
```

**Figure 6.4**
Finding decimal 3-digit numbers which are the sum of the cubes of their digits.
MUL CX
ADD BX,AX
;binary version of the number the digits represent in AX
MOV AX,FIRST_DIGIT
MOV CX,10D
MUL CX
ADD AX,SECOND_DIGIT
MUL CX
ADD AX,THIRD_DIGIT
;compare
CMP AX,BX
;if they are the same then display the digits
JNZ NEWDIGITS
;convert each digit to ASCII and display
MOV DX,FIRST_DIGIT
ADD DL,30H
MOV AH,2H
INT 21H
MOV DX,SECOND_DIGIT
ADD DL,30H
INT 21H
MOV DX,THIRD_DIGIT
ADD DL,30H
INT 21H
;display a new line
MOV DL,0DH
INT 21H
MOV DL,0AH
INT 21H
;next three digits
NEWDIGITS: INC THIRD_DIGIT
CMP THIRD_DIGIT,10D
JNZ NEWNUMBER
MOV THIRD_DIGIT,0
INC SECOND_DIGIT
CMP SECOND_DIGIT,10D
JNZ NEWNUMBER
MOV SECOND_DIGIT,0
INC FIRST_DIGIT
CMP FIRST_DIGIT,10D
JZ FINISH
JMP NEWNUMBER
;return to dos
FINISH: MOV AX,4C00H
INT 21H

CODE ENDS
END START
EXERCISE

6.16 Write 8086-family assembly language programs to find:

(a) All possible sets of three decimal digits in which the digits make up a Pythagorean triple. Thus, \( \{3, 4, 5\} \) is a Pythagorean triple because \( 3^2 + 4^2 = 5^2 \)

(b) All 3-digit decimal numbers for which the sum of the digits is an exact divisor of the product of its digits. For example, this property holds for 862 because:
\[
8 + 6 + 2 = 16 \quad \text{and} \quad 8 \times 6 \times 2 = 96
\]
and \( 96 / 16 = 6 \).

(c) All 2-digit decimal numbers which are the sum of the squares of two of the decimal numbers in the set \( \{2, 3, 5, 7\} \). Thus,
\[
34 = 5^2 + 3^2
\]

(d) All decimal numbers which are the product of three of the numbers in the set \( \{2, 3, 5, 7\} \). Thus,
\[
42 = 2 \times 3 \times 7
\]

(e) The number of values of the expression \( x^2 + x + 41 \) which are prime numbers (that is, they are larger than 1 and exactly divisible only by themselves and 1) when \( x \) is chosen from the range 1, 2, \ldots, 20. For example, if \( x \) is 1 then the value of the expression is \( 1 + 1 + 41 = 43 \) which is a prime number.

SUMMARY

This chapter has featured the 8086-family arithmetic instructions (which are summarized in Table 6.3) and the use of some of them in a complete program example. Chapter 10 will provide an illustration of the use of almost all of the 8086-family arithmetic instructions in a fairly large assembly language program. Table 6.2 summarized the effect on the flags of the arithmetic instructions (see also Appendix V).
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Addition</strong></td>
<td>op1 := op1 + op2. Allowed formats:</td>
</tr>
<tr>
<td>ADD op1, op2</td>
<td>op1 = op2</td>
</tr>
<tr>
<td></td>
<td>register number</td>
</tr>
<tr>
<td></td>
<td>memory number</td>
</tr>
<tr>
<td></td>
<td>register register</td>
</tr>
<tr>
<td></td>
<td>memory register</td>
</tr>
<tr>
<td></td>
<td>register memory</td>
</tr>
<tr>
<td>ADC op1, op2</td>
<td>ADC with Carry: if CF = 0, then ADD as above. If CF = 1, then op1 := op1 + op2 + 1.</td>
</tr>
<tr>
<td>INC op</td>
<td>INCrement: op := op + 1. The operand may be in either a register or in memory.</td>
</tr>
<tr>
<td><strong>Subtraction</strong></td>
<td>op1 := op1 - op2. Allowed formats as in ADD above.</td>
</tr>
<tr>
<td>SUB op1, op2</td>
<td></td>
</tr>
<tr>
<td>SBB op1, op2</td>
<td>Subtract with Borrow: if CF = 0, then SUB as above. If CF = 1 then op1 := op1 - op2 - 1.</td>
</tr>
<tr>
<td>DEC op</td>
<td>DECrement: op := op - 1. The operand may be in either a register or in memory.</td>
</tr>
<tr>
<td>NEG op</td>
<td>NEGate: op := 0 - op. Operand as in DEC.</td>
</tr>
<tr>
<td>CMP op1, op2</td>
<td>CoMPare: set flags according to the result of op1 - op2 but do not change either operand.</td>
</tr>
<tr>
<td><strong>Multiplication</strong></td>
<td></td>
</tr>
<tr>
<td>MUL op</td>
<td>MULtiplication (unsigned): if op is a byte, then AH, AL := op * AL. If op is a word, then DX, AX := op * AX. If the upper half of the result is not zero then CF and OF are set to 1. op can be either in a register or in memory.</td>
</tr>
<tr>
<td>IMUL op</td>
<td>Integer MULtiplication (signed): as with MUL except that signed multiplication takes place.</td>
</tr>
<tr>
<td><strong>Division</strong></td>
<td></td>
</tr>
<tr>
<td>DIV op</td>
<td>DIVision (unsigned): if op is a byte, then (AH, AL) / op = AL remainder AH. If op is a word, then (DX, AX) / op = AX remainder DX. If the quotient exceeds the capacity of AL or AX, then both quotient and remainder are undefined (and a type 0 interrupt is generated - see Chapter 21). op can be as in MUL above.</td>
</tr>
<tr>
<td>IDIV op</td>
<td>Integer DIVision (signed): as with DIV above except that a signed division takes place.</td>
</tr>
<tr>
<td><strong>Sign extension</strong></td>
<td></td>
</tr>
<tr>
<td>CBW</td>
<td>Convert Byte to Word: the 8-bit signed number in AL is converted to 16-bit signed equivalent in AX.</td>
</tr>
<tr>
<td>CWD</td>
<td>The 16-bit signed number in AX is converted to a 32-bit signed equivalent in DX, AX with DX containing the higher order bits.</td>
</tr>
</tbody>
</table>
**SELF-CHECK QUIZ**

1. If AX contains 00BBH and BX contains 00BCH and then SUB AX,BX is executed, which of the following would be possible contents of the flags register:

   (a) 0211H    (b) F221H    (c) 0221H

2. Where possible, give short assembly language sequences using arithmetic instructions which perform an arithmetic operation and, as a result, will *simultaneously* set the named flags to the required values. If the required settings are not simultaneously possible, give a reason why not.

   (a) Z and C flags to 0
   (b) Z and C flags to 1
   (c) C and S flags to 0
   (d) C and S flags to 1

3. If AL contains a signed number in the range 20D to 30D, what is the largest number that can be added to it without causing overflow?

4. If a CMP instruction is used to compare two values, explain how the values of the S-flag and the O-flag can tell you if one operand were *less* than the other. Which combination of flag values will tell you whether one operand is *less than or equal* to another after a CMP instruction has been executed?

5. To double a value \( n \), we can calculate either \( n + n \), or \( 2 \times n \), or \( n - (\neg n) \). Write three sequences of assembly language instructions each of which doubles the contents of AX using each of these expressions respectively.

6. Because of overflow, arithmetic with signed numbers can fail the mathematical law which says that \( (a + (b + c)) = ((a + b) + c) \). Can you find an example of this for 8-bit signed numbers? Can you find a corresponding example involving unsigned 8-bit numbers?

7. Repeat exercise 6 but for the mathematical law which says that \( ((a * b) * c) = (a * (b * c)) \).

8. Write an assembly language sequence which will change the signed number in AL which represents a student's score out of 20D, into a whole-number percentage in AL.

9. Explain in detail the effect of the instruction sequence:

   ```
   CBW
   CWD
   ```

   on AL.

10. Does the instruction sequence:

    ```
    ADC AX,BX
    SBB AX,BX
    ```

    always leave AX unchanged? Explain your answer.