Getting at Individual Bits

Contents

9.1 The family of 'logical' instructions
9.2 The shift family
9.3 The rotate family
9.4 Analyzing beliefs – a complete program example
9.5 Efficiency
Summary
Self-check quiz

Microprocessors are not just used as the foundation stone of microcomputer architecture. They are also used to make control devices for a range of equipment including large-scale manufacturing production machinery on the one hand and traffic light controllers on the other. All of these require the ability to manipulate individual bits since, for example, a given bit can be set to 1 to turn a traffic light on, or to 0 to turn it off.

Consequently, the designer of a microprocessor must include in its instruction set an appropriate repertoire for these types of microprocessor applications. This chapter discusses a subset of those available for this purpose in the 8086 family. In fact, we shall discuss three different families of instructions: the logical family, the shift family and the rotate family.

Certain members of the rotate family of instructions can use the carry flag to save some information which would otherwise be lost. Usually, the values of flags are only changed indirectly by instructions. But
to make the fullest possible use of these instructions without recourse to tricks, the 8086 family provides instructions to change the value of the carry flag directly and these are also described in this chapter.

Finally, we briefly consider the execution efficiency of programs. Years ago, this issue was the mainstay of assembly language programming technique. But with today's high labor costs, producing programs as quickly as possible and in a form which can be maintained cheaply have replaced it as the dominant consideration.

9.1 The family of 'logical' instructions

We have seen that the contents of a register like AX or AL can be thought of as a signed or unsigned number and as the ASCII code for some character. There is a third important way of viewing the binary digits in a register in which each bit gives certain information depending on its position in the register. For example, in a 16-bit register the 16 bits could represent the result of an electronic quality control test on 16 product items – a 1 in a particular position indicating that the corresponding item was satisfactory, a 0 indicating failure. Thus, if the contents of AX was

0000100100000010

then items 1, 8 and 11 were not faulty (numbering from zero from the right-hand side in the usual way). Accessing individual bits could also be useful when a microcomputer based on some member of the 8086 family is being used to control an external device – a set of traffic lights, for example. In that case the setting of a given bit might determine whether a light is turned on or off.

The 8086 family has several instructions which enable one to process information of this kind including AND, OR, NOT, XOR and TEST. These are known as the logical family of instructions because they operate bit by bit on the contents of either one or two registers and the outcome reflects the interpretation of 1 as representing the logical value TRUE and 0 as representing the logical value FALSE.

An instruction

AND AX,BX

changes the contents of register AX in the following way. The contents of registers AX and BX are combined bit by bit. If, in a given position, both AX and BX contain a 1 then the result of AND AX,BX will have a 1 in that position, otherwise there will be a 0. For example:

<table>
<thead>
<tr>
<th>Contents of AX</th>
<th>0000101011100011</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents of BX</td>
<td>1001100000100001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Contents of AX after AND AX,BX</th>
<th>0000100000100001</th>
</tr>
</thead>
</table>
(the result has a 1 only in those positions in which both AX and BX contain a 1)

The AND instruction affects all the arithmetic flags and can be used to check if the bit in a particular position is 0 or 1. For example, to check if the bit in position 7 of register AX is a 1 we would first put 0000000010000000B into register BX and then execute an AND AX, BX instruction. Because of all the 0s in register BX the result of the AND can have at most one non-zero entry – in position 7. If AX contains a 1 in that position, so will the result which will then be 0000000010000000B, otherwise the result will be 0.

Thus, if AX does have bit 7 set to 1:

\[
\begin{array}{ll}
AX & 100101011001110 \\
BX & 0000000010000000 \\
\hline
\text{result of} & 0000000010000000 \\
\text{AND AX, BX} & \\
\end{array}
\]

and if AX does not have bit 7 set to 1:

\[
\begin{array}{ll}
AX & 0101010110101011 \\
BX & 0000000010000000 \\
\hline
\text{result of} & 0000000000000000 \\
\text{AND AX, BX} & \\
\end{array}
\]

Hence, with the given setting of BX, if after AND AX, BX register AX is 0 (in which case the Z-flag will be set to 1) then there was a 0 in position 7 of register AX initially and if not then there was a 1 in that position.

This is an instance of a general technique called masking. As another example, let us consider the problem of recouping the numeric value from the ASCII code for a digit. Consider Table 9.1 and notice the correlation between the last two columns.

### Table 9.1 Recouping a numeric value from an ASCII code.

<table>
<thead>
<tr>
<th>Decimal digit</th>
<th>ASCII code</th>
<th>8-bit unsigned number with same value as the decimal digit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hex</td>
<td>Binary</td>
</tr>
<tr>
<td>0</td>
<td>30H = 0011 0000</td>
<td>00000000</td>
</tr>
<tr>
<td>1</td>
<td>31H = 0011 0001</td>
<td>00000001</td>
</tr>
<tr>
<td>2</td>
<td>32H = 0011 0010</td>
<td>00000010</td>
</tr>
<tr>
<td>3</td>
<td>33H = 0011 0011</td>
<td>00000011</td>
</tr>
<tr>
<td>4</td>
<td>34H = 0011 0100</td>
<td>00000100</td>
</tr>
<tr>
<td>5</td>
<td>35H = 0011 0101</td>
<td>00000101</td>
</tr>
<tr>
<td>6</td>
<td>36H = 0011 0110</td>
<td>00000110</td>
</tr>
<tr>
<td>7</td>
<td>37H = 0011 0111</td>
<td>00000111</td>
</tr>
<tr>
<td>8</td>
<td>38H = 0011 1000</td>
<td>00001000</td>
</tr>
<tr>
<td>9</td>
<td>39H = 0011 1001</td>
<td>00001001</td>
</tr>
</tbody>
</table>
Whenever a digit is typed in at the computer keyboard and accepted via the DOS function 1 of interrupt number 21H, the ASCII code for that digit will end up in AL. Before the value of that digit can be used in a calculation it must be transformed from ASCII into a numeric format. One method of effecting this transformation would be to enter 00001111B into register BL and execute an AND AL,BL which would leave the numerical value of the digit in register AL. For example, if register AL contains the ASCII code for the digit 7:

<table>
<thead>
<tr>
<th>Contents of AL</th>
<th>0011 0111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents of BL</td>
<td>0000 1111</td>
</tr>
</tbody>
</table>

Contents of AL 0000 0111 (an unsigned binary number equivalent to decimal 7)

As mentioned earlier, there are four other members of the logical family of bit manipulating instructions: OR, NOT, XOR and TEST. An instruction OR AX,BX changes the contents of register AX in the following way. The contents of registers AX and BX are compared bit by bit. If, in a given position, either AX or BX contains a 1 then the result of OR AX, BX will have a 1 in that position, otherwise a 0. For example

<table>
<thead>
<tr>
<th>Contents of AX</th>
<th>0011001101011010</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents of BX</td>
<td>0101010100000111</td>
</tr>
</tbody>
</table>

Contents of AX 0111011101011111 after OR AX,BX

While the AND and OR instructions operate with two registers, NOT requires only one. NOT BX, for example, simply changes every bit in BX which is a 1 to a 0 and every 0 to a 1:

Initial contents of BX 0101111100001001

<table>
<thead>
<tr>
<th>Contents of BX</th>
<th>1010000011110110</th>
</tr>
</thead>
<tbody>
<tr>
<td>after NOT BX</td>
<td></td>
</tr>
</tbody>
</table>

The format of the XOR instruction is exactly the same as that of the OR. An instruction

XOR AX,BX

changes the contents of register AX in the following way. The contents of registers AX and BX are combined bit by bit. If, in a given position, both AX and BX contain a 1 or both AX and BX contain a 0 then the result of XOR AX,BX will have a 0 in that position, otherwise there will be a 1. In other words, after execution of XOR AX,BX, register AX will have a 1 in a given position if either AX or BX has a 1 in that position, but not both (XOR stands for eXclusive OR).
For example:

<table>
<thead>
<tr>
<th>Contents of AX</th>
<th>0110001001001110</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents of BX</td>
<td>1000111000110000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Contents of AX after XOR AX,BX</th>
<th>1110110001111110</th>
</tr>
</thead>
<tbody>
<tr>
<td>both AX and BX have a 0 in this position</td>
<td></td>
</tr>
<tr>
<td>both AX and BX have a 1 in this position</td>
<td></td>
</tr>
<tr>
<td>only one of AX and BX have a 1 in these two positions</td>
<td></td>
</tr>
</tbody>
</table>

The TEST instruction is very similar to the AND instruction in that they both have the same format and both perform the same operation between their operands, but TEST does not affect the main registers and only changes the flags. Thus,

<table>
<thead>
<tr>
<th>AH</th>
<th>DH</th>
<th>Z-flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before</td>
<td>01010011</td>
<td>10001100</td>
</tr>
<tr>
<td>After AND DL, AH</td>
<td>01010011</td>
<td>00000000</td>
</tr>
<tr>
<td>After TEST DL, AH</td>
<td>01010011</td>
<td>10001100</td>
</tr>
</tbody>
</table>

Figure 9.1 gives a trace of a program fragment involving members of the logical family of instructions and Table 9.2 summarizes the uses of the instructions.
Table 9.2 Logical family instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>Setting specified bit position of the first operand to 0, the other operand specifying the bit positions. Thus, \texttt{AND BH,00011100B} will set bits 0, 1, 5, 6 and 7 of BH to 0 leaving the others unchanged.</td>
</tr>
<tr>
<td>OR</td>
<td>Setting specified bit positions of the first operand to 1, the other operand specifying the bit positions. Thus, \texttt{OR CL,11000000B} will set bits 6 and 7 of CL to 1 leaving the others unchanged.</td>
</tr>
<tr>
<td>NOT</td>
<td>Changing every bit in its operand to the opposite of what it was initially.</td>
</tr>
<tr>
<td>XOR</td>
<td>Changing specified bits of its first operand to the opposite of their current settings, the other operand specifying the bit positions. Thus, \texttt{XOR AH,00000011B} will change bits 0 and 1 of AH to the opposite of their previous setting and leave the others unchanged.</td>
</tr>
<tr>
<td>TEST</td>
<td>Testing whether specified bit positions of the first operand are set to 0, the other operand specifying the bit positions. Thus \texttt{TEST DL,01010101B} will set the Z-flag to 1 only if bits 0, 2, 4 and 6 of DL are all 0.</td>
</tr>
</tbody>
</table>

9.1.1 General forms of the logical family instructions

This is the first occasion on which some indication of the general form of an instruction has been offered and we shall use some fairly informal notation. Should any ambiguities or other confusion arise from our notation, the precise definitions of all the 8086-family instructions are given in Appendix V.

The \texttt{AND}, \texttt{XOR}, \texttt{OR} and \texttt{TEST} instructions may all be used in the same forms which will be illustrated with the \texttt{AND} instruction. These are:

\begin{verbatim}
AND register, number
AND memory, number
AND register1, register2
AND register, memory
AND memory, register
\end{verbatim}

The \texttt{NOT} instruction may be used in the formats:

\begin{verbatim}
NOT register
NOT memory
\end{verbatim}

EXERCISES

9.1 Make traces of the execution of each of the following program fragments:
9.2 Write program fragments which *use registers only, not memory* to carry out each of the following tasks:

(a) Change every other bit of AL to its opposite value, starting with bit 7. Thus, if AL contained 01101010B before execution of the program fragment it should contain 11000000B afterwards.

(b) Swap the two nibbles in AL around. Thus if AL contained 01101010B before execution it should contain 10100110B afterwards.

(c) Change bit 3 of AL according to the values of bits 0,1 and 2; set bit 7 according to the values of bits 4,5 and 6; and leave all the other bits alone. If all three preceding bits have the same value, bit 3 or 7 (respectively) should be set to 1; otherwise to 0. Thus if AL contained 01111010B before execution, it should contain 11100101B afterwards.
9.2 The shift family

Because microprocessors in the 8086 family work with binary numbers, they can easily provide facilities for multiplying and dividing a signed or unsigned number by 2. As we shall see in the final section of this chapter – which deals with issues of efficiency – it turns out that these special facilities work much faster than using the ordinary multiply and divide instructions.

The principle used is an easy one. Consider Table 9.3. In the binary column, the first three entries differ only in that entry 2 is the first entry shifted left one bit and entry 3 is the second entry shifted left one bit. From the decimal column we see that the decimal equivalents of these binary entries double after each left shift. On the other hand, binary column entries 4 to 8 (inclusive) differ only by a shift right of one bit and the corresponding decimal values are halved after each right shift.

There are two different sets of 8086-family shift instructions, one set for doubling and halving unsigned binary numbers, the other for doubling and halving signed binary numbers. The instructions which perform the doubling and halving of unsigned numbers are SHL (Shift Left) and SHR (Shift Right) and those which do the same for signed numbers are SAL (Shift Arithmetic Left) and SAR (Shift Arithmetic Right).

9.2.1 SHL and SHR

SHL doubles an unsigned number by shifting all bits one position to the left and filling in the vacated right-most bit with a 0. To test if the doubling has resulted in a number which is too big to be represented in the number of bits available in the particular register or memory location being operated on, the bit which is shifted out of the left-hand end is moved into the carry flag.

The general forms of the SHL instruction are:

```
SHL register, 1
SHL memory, 1
```

<table>
<thead>
<tr>
<th>Entry</th>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00011111</td>
<td>31</td>
</tr>
<tr>
<td>2</td>
<td>00111110</td>
<td>62</td>
</tr>
<tr>
<td>3</td>
<td>01111100</td>
<td>124</td>
</tr>
<tr>
<td>4</td>
<td>00110000</td>
<td>48</td>
</tr>
<tr>
<td>5</td>
<td>00011000</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>00001100</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>00000110</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>00000011</td>
<td>3</td>
</tr>
</tbody>
</table>
Suppose that register AL contains 01001111B and that we execute the instruction:

```
SHL AL, 1
```

After execution the contents of AL will be 10011110B and the carry flag will be set to 0. If register CX contains 1010101011111111B, then after execution of

```
SHL CX, 1
```

CX will contain 0101010111111110B and the carry flag will be set to 1.

Halving unsigned numbers can be carried out via the `SHR` instruction which has the general forms:

```
SHR register, 1
SHR memory, 1
```

and operates by shifting all bits one position to the right, filling the then vacated left-most position with a 0 and placing the bit that was shifted off the right-hand end into the carry flag. In this case, the carry flag being set to 1 indicates that the number just halved was not even (and therefore that the ‘halving’ is only approximate).

If register DX contains 101010101111111B before execution of

```
SHR DX, 1
```

then after execution DX will contain 010101011111111B and the carry flag will be set to 0. With register BL containing 01100111B beforehand, the execution of

```
SHR BL, 1
```

will leave 00110011B in BL and the carry flag will be set to 1 (01100111 in binary is equivalent to decimal 207).

### 9.2.2 `SAL` and `SAR`

Doubling and halving signed numbers involves the slight complication of ensuring that the sign of the number is not changed. Thus, doubling +8 (00001000) should result in +16 (00010000) and doubling –20 (11101100) should result in –40 (11011000). Likewise, halving +6 (00000110) should give +3 (00000011) and halving –120 (10001000) should give –60 (11000100).

`SAL` (Shift Arithmetic Left) doubles signed numbers and `SAR` (Shift Arithmetic Right) halves them. These instructions have the general forms:

```
SAL register, 1
SAL memory, 1
SAR register, 1
SAR memory, 1
```

`SAL` operates in precisely the same way as `SHL`, since the sign bit of a positive signed number is 0. It is up to the programmer to ensure that the result
Table 9.4 Effect of SAR.

<table>
<thead>
<tr>
<th>Contents of AL before execution of SAR AL, 1</th>
<th>Contents of AL after execution of SAR AL, 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>Decimal</td>
</tr>
<tr>
<td>--------</td>
<td>---------</td>
</tr>
<tr>
<td>01101100</td>
<td>+108</td>
</tr>
<tr>
<td>00101011</td>
<td>+43</td>
</tr>
<tr>
<td>10000010</td>
<td>−126</td>
</tr>
<tr>
<td>10101111</td>
<td>−81</td>
</tr>
<tr>
<td>11000010</td>
<td>−62</td>
</tr>
</tbody>
</table>

of such doubling is in range (for otherwise, doubling a positive signed number like 01101100 may well result in a negative signed number, namely 11011000). In general, it is better if the programmer plans in advance to avoid such an eventuality, but on occasions planning ahead is impossible. In these cases the sign flag enables detection of the phenomenon, for when the SAL or SAR instruction is executed the sign flag is set to 0 or 1 corresponding to the sign of the result.

SAR halves signed binary numbers by shifting all bits one position to the right but at the same time leaving the sign bit unchanged (see Table 9.4), that is, a copy is made of the original sign bit and after everything has been shifted to the right that copy is used to fill the vacated left-most bit. Suppose, for example, that the contents of AL is 11011011. When the contents of AL is viewed as an 8-bit signed number, the sign bit is 1. Shifting the contents of AL one bit to the right gives:

?1101101

If we now use the sign bit to fill the vacated bit:

11101101

it follows that the result of SAR AL, 1 in this case leaves AL containing 11101101. To check that this does indeed give the right answer note that 11011011 is −37 in decimal and 11101101 is −19 in decimal. (Notice that halving a signed number always results in a signed number which is less than or equal to half the given number. Hence, when an odd number like +43 is halved the result is +21, and similarly halving −81 gives −41).

EXERCISE

9.3 Make execution traces of the contents of each of the registers involved in the following program fragments:
9.2.3 Quadrupling, octupling and all that

All the shift family instructions have another form which allows shifting up to eight places left or right at a time. Shifting left two bits corresponds to multiplying by 4D and shifting left five bits to multiplying by 32D. Shifting right two bits corresponds to dividing by 4D and shifting right six bits to dividing by 64D. In each case, the only change to the instruction format is the addition of a register to hold the (unsigned) number of places to shift. This register must be CL:

\[
\text{SHL register/memory,CL}
\]
\[
\text{SHR register/memory,CL}
\]
\[
\text{SAL register/memory,CL}
\]
\[
\text{SAR register/memory,CL}
\]

Thus, if AL contains \(10110111B\) before execution of:

\[
\text{MOV CL},3
\]
\[
\text{SHR AL},\text{CL}
\]

then after execution AL will contain \(00010110\). Similarly, if BX contains \(010110110000000000B\) before execution of:

\[
\text{MOV CL},4
\]
\[
\text{SAR BX},\text{CL}
\]

then after execution BX will contain \(111101101110000\).
EXERCISES

9.4 Write and test an 8086-family assembly language program to multiply the unsigned number in the AL register by 10D using shift instructions. Assume that the result can be represented as an 8-bit unsigned number. (*Hint:* calculate eight times the contents of AL and twice the contents of AL and add.)

9.5 Expand the program you developed for Exercise 9.4 so that the result is left as an unsigned 16-bit number in AX and the program works for any unsigned number in AL. (*Hint:* shift one bit at a time and use the carry flag to make decisions about the contents of AH after each shift.)

9.6 Repeat Exercises 9.4 and 9.5 for division by 8D.

9.3 The rotate family

When a 4-bit shift left is performed, the 4 left-most bits ‘fall off the end’ and are lost forever (one of them may possibly end up in the carry flag). There are occasions when we do not wish to throw away information with such abandon. The rotate family of instructions provides the ability to rearrange bits without losing information. Two types of rotation are possible: one which simply affects the register or memory location nominated in the instruction, and one which involves the carry flag in an essential way.

9.3.1 Rotate

The rotate instructions take the following general forms:

- `ROL memory/register,1`
- `ROL memory/register,CL`
- `ROR memory/register,1`
- `ROR memory/register,CL`

`ROL` (R0tate Left) and `ROR` (R0tate Right) permit left or right rotation of the bits respectively and any bits which fall off one end are rotated around to fill the vacated positions at the other end. Thus, if AL contains 01000111B then execution of `ROR AL,1` will leave 10100011 in AL:

- begin with AL 01000111
- rotate one place to the right AL 70100011 and 1 'falls' off the right-hand end
1 rotates round AL 10100011
to the other end

Similarly, if AL contains 0100011B execution of ROL AL,1 will leave 10001110 in
AL:

```
begin with AL 01000111
rotate one place AL 1000111? and 0 ‘falls’ off the
      to the left
left-hand end
0 rotates round AL 10001110
      to the other end
```

In diagrammatic form, ROR can therefore be represented:

```
[Diagram showing ROR]
```

and ROL:

```
[Diagram showing ROL]
```

### 9.3.2 Rotate through the carry flag

Because the flags are not involved in ROL and ROR it is difficult to keep track of
the value of a particular bit and so there are alternative forms of the rotate
instructions involving the carry flag. Thus, RCL (Rotate through Carry Left) and
RCR (Rotate through Carry Right) include the carry flag in the rotation: the bit
that falls off one end goes into the carry flag, the bit that was in the carry flag
goes into the vacated bit (Figure 9.2).

```
Figure 9.2
The (a) RCR and (b) RCL instructions.
```
The rotate family instructions take the following general forms:

\[
\begin{align*}
\text{RCL} & \text{ memory/register,1} \\
\text{RCL} & \text{ memory/register,CL} \\
\text{RCR} & \text{ memory/register,1} \\
\text{RCR} & \text{ memory/register,CL}
\end{align*}
\]

Hence, if the carry flag is 0 and BL contains 01101100B before execution of \text{RCL BL,1} then after execution BL contains 11011000 and the carry flag will be set to 0. If the carry flag is 0 and AL contains 10101101 before execution of \text{RCR BL,1}, then afterwards AL will contain 11010110 and the carry flag will be set to 1.

### 9.3.3 Setting and clearing the carry flag

For the fullest possible control over rotation through the carry flag we need instructions which allow us to set and clear the carry flag directly. These are provided in the form of \text{STC} (SeT Carry flag) and \text{CLC} (CLear Carry flag) respectively. Thus, if AL contains 10101110B before execution of:

\[
\begin{align*}
\text{STC} \\
\text{RCR AL,1}
\end{align*}
\]

then after execution AL will contain 11010111 and the carry flag will be set to 0. Similarly, if AL contains 00110011B before execution of:

\[
\begin{align*}
\text{CLC} \\
\text{RCR AL,1}
\end{align*}
\]

then after execution AL will contain 00011001 and the carry flag will be set to 1.

---

**EXERCISES**

### 9.7 Make execution traces of the contents of each of the registers involved in the following program fragments:

(a) \[
\begin{align*}
\text{MOV CL,4} \\
\text{MOV BX,10110111B} \\
\text{ROR BX,1} \\
\text{ROR BX,CL}
\end{align*}
\]

(b) \[
\begin{align*}
\text{MOV DX,1101101101010011B} \\
\text{MOV CL,7} \\
\text{STC} \\
\text{RCR DX,1} \\
\text{RCR DX,CL} \\
\text{SHR DX,1} \\
\text{SHL DX,CL}
\end{align*}
\]
SAL DX,1
CLC
RCL DX,CL

(c) MOV CX,0303H
SHL CH,CL
STC
ROL CX,1
RCL CH,CL
SAR CX,1
RCR CX,CH

9.8 Using only SHL, RCL and MOV instructions devise routines to:

(a) multiply the 32-bit unsigned number stored in registers DX and AX by 4 (register DX contains the higher order bits so that

00001111100000000111111101010101B

would be stored with DX containing 00001111100000000B and AX containing 1111111101010101B). Assume that the result can be accurately represented as an unsigned 32-bit number;

(b) repeat (a) but for division by 4; and

(c) repeat (a) and (b) for the 48-bit unsigned number in DX, AX and BX.

9.4 Analyzing beliefs – a complete program example

This section illustrates some of the bit manipulation instructions introduced in this chapter in a complete example program. Perhaps the best illustration would have been a program to control a hardware device such as a set of traffic lights. But such an example would restrict the ability to run the program to those with access to the relevant device. Consequently, we shall illustrate instead with an example program, which could be used in the course of undertaking some market research about the general public’s perception of products and for which it is natural to store the users’ input in bit form.

In fact, the program will implement a Kelly grid system for analyzing how similar an individual regards four different makes of car – Chevrolet, Volkswagen, Volvo and Porsche – with regard to five pairs of attributes in which each pair represents two opposite extremes:

- luxury car – family car
- expensive – good value for money
- exclusive – common
- aggressive image – sedate image
- for the young – for the old
Jumps and Loops

Contents

7.1 Conditional jumps
7.2 Implementing loops
7.3 The loop family of instructions
7.4 Decisions
7.5 A complete example program involving a loop and decisions
Summary
Self-check quiz

Normally, assembly language instructions are executed in the order in which they are given to the microprocessor (or, what often amounts to the same thing, in the order that they occur in the program). The programmer can use jump instructions to alter this mechanism. Conditional jump instructions cause execution to continue not from the next instruction in sequence, but from some remote instruction provided some condition is satisfied. In this chapter we shall introduce all 18 of the different conditional jumps which the 8086-family members can execute and describe the actions of many of them.

The most important ways in which conditional jumps can be used to change the flow of control in a program are in the implementation of loops similar to for, repeat and while loops in Pascal and in taking decisions. Sections 7.2 and 7.3 discuss how to program loops in 8086-family assembly language using both conditional jumps and the instructions loop, loopz and loopwnz which 8086-family assembly language provides especially for this purpose. It is necessary to consider both of
these approaches since the simplicity and efficiency gains from using LOOP, LOOPZ and LOOPNZ may only be obtained when there are a fixed maximum number of repetitions of a loop and a single condition can be used to determine whether the end of a loop has been reached.

The use of conditional jumps to implement decision-taking in programs is discussed in the penultimate section and this is followed by a complete example program which illustrates several of the ideas in this chapter.

7.1 Conditional jumps

The execution of conditional jumps depends on the 8086-family flags, since whether or not a certain condition is met is dependent on the current setting of a particular flag. Thus, the precise meaning of

\[ \text{JZ } <\text{label}> \]

is ‘if the Z-flag is currently set to 1, then jump to the instruction labeled \(<\text{label}>\) and continue execution from there’. Another, slightly less formal interpretation, is ‘jump to \(<\text{label}>\) if the result of the last instruction to affect the Z-flag was zero’. For

\[ \text{JNZ } <\text{label}> \]

the precise meaning is similarly, ‘if the Z-flag is currently set to 0, then jump to the instruction labeled \(<\text{label}>\) and continue execution from there’. Slightly less formally, ‘jump to \(<\text{label}>\) if the result of the last instruction to affect the Z-flag was not zero’.

Instructions such as JZ allow the programmer to change the order of instruction execution depending on whether the result of the last instruction to affect the flags was zero or non-zero or, equivalently, whether when the contents of two registers were compared, it turned out that they contained the same thing. Provision has also been made for jumps conditional upon comparing signed numbers and unsigned numbers, and for jumps conditional upon the settings of particular flags and conditional upon the value of the CX register.

Altogether, the 8086 family has 18 different kinds of conditional jump instructions allowing the execution sequence to be altered and these are summarized in Table 7.1. In the table, CF stands for the Carry Flag, ZF for the Zero Flag and so on. Under ‘flags tested’ an entry of the form

\[ (\text{CF or ZF}) = 0 \]

means that the jump is made if either one of CF and ZF is currently set to 0.
Table 7.1 8086-family conditional jumps and the flags tested by them.

<table>
<thead>
<tr>
<th>Name</th>
<th>Jump if</th>
<th>Flags tested</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Testing for zero</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JZ</td>
<td>Zero</td>
<td>ZF = 1</td>
</tr>
<tr>
<td>JNZ</td>
<td>Not Zero</td>
<td>ZF = 0</td>
</tr>
<tr>
<td><strong>Comparing unsigned numbers</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JA</td>
<td>Above</td>
<td>(CF and ZF) = 0</td>
</tr>
<tr>
<td>JB</td>
<td>Below</td>
<td>CF = 1</td>
</tr>
<tr>
<td>JAE</td>
<td>Above or equal</td>
<td>CF = 0</td>
</tr>
<tr>
<td>JBE</td>
<td>Below or equal</td>
<td>(CF or ZF) = 1</td>
</tr>
<tr>
<td>JNC</td>
<td>No carry</td>
<td>CF = 0</td>
</tr>
<tr>
<td><strong>Comparing signed numbers</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JG</td>
<td>Greater</td>
<td>ZF = 0 and SF = OF</td>
</tr>
<tr>
<td>JL</td>
<td>Less</td>
<td>SF&lt;&gt;OF</td>
</tr>
<tr>
<td>JGE</td>
<td>Greater or equal</td>
<td>SF = OF</td>
</tr>
<tr>
<td>JLE</td>
<td>Less or equal</td>
<td>(ZF = 1) or (SF&lt;&gt;OF)</td>
</tr>
<tr>
<td><strong>Testing for overflow</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JO</td>
<td>Overflow</td>
<td>OF = 1</td>
</tr>
<tr>
<td>JNO</td>
<td>Not overflow</td>
<td>OF = 0</td>
</tr>
<tr>
<td><strong>Testing signs</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JS</td>
<td>Sign</td>
<td>SF = 1</td>
</tr>
<tr>
<td>JNS</td>
<td>No sign</td>
<td>SF = 0</td>
</tr>
<tr>
<td><strong>Testing parity</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JPO</td>
<td>Parity odd</td>
<td>PF = 0</td>
</tr>
<tr>
<td>JPE</td>
<td>Parity even</td>
<td>PF = 1</td>
</tr>
<tr>
<td><strong>Checking CX for zero without inspecting the flags</strong></td>
<td></td>
<td>none</td>
</tr>
<tr>
<td>JCXZ</td>
<td>CX is equal to zero</td>
<td></td>
</tr>
</tbody>
</table>

### 7.1.1 Aliases for conditional jumps

There is one further complication which need not detain us very much at this point in the text. This arises because possibilities for the outcome of comparisons other than those listed in Table 7.1 may come to mind. For example, the table mentions a ‘jump if above or equal’, but what about a corresponding ‘jump if not above or equal’? In fact, the latter would be exactly equivalent to ‘jump if below’. Instead of arbitrarily fixing on one of these expressions rather than another, the designers of 8086-family assembly language decided to allow several names for conditional jumps as alternatives to those given in Table 7.1. These are listed in Table 7.2.

### 7.1.2 A jump ‘survival kit’

While there may seem a daunting number of jump instructions to remember, this is because jumps are provided to deal with conditions arising from signed
Table 7.2 Aliases for conditional jumps.

<table>
<thead>
<tr>
<th>Name</th>
<th>Alias(es)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing for zero</td>
<td></td>
</tr>
<tr>
<td>J2</td>
<td>JE</td>
</tr>
<tr>
<td>JNZ</td>
<td>JNE</td>
</tr>
<tr>
<td>Comparing unsigned numbers</td>
<td></td>
</tr>
<tr>
<td>JA</td>
<td>JNBE</td>
</tr>
<tr>
<td>JB</td>
<td>JNAE</td>
</tr>
<tr>
<td>JAE</td>
<td>JC</td>
</tr>
<tr>
<td>JBE</td>
<td>JNB</td>
</tr>
<tr>
<td>JNC</td>
<td>no alias</td>
</tr>
<tr>
<td>Comparing signed numbers</td>
<td></td>
</tr>
<tr>
<td>JG</td>
<td>JNLE</td>
</tr>
<tr>
<td>JL</td>
<td>JNGE</td>
</tr>
<tr>
<td>JGE</td>
<td>JNL</td>
</tr>
<tr>
<td>JLE</td>
<td>JNG</td>
</tr>
<tr>
<td>Testing for overflow</td>
<td></td>
</tr>
<tr>
<td>JO</td>
<td>no alias</td>
</tr>
<tr>
<td>JNO</td>
<td>no alias</td>
</tr>
<tr>
<td>Testing signs</td>
<td></td>
</tr>
<tr>
<td>JS</td>
<td>no alias</td>
</tr>
<tr>
<td>JNS</td>
<td>no alias</td>
</tr>
<tr>
<td>Testing parity</td>
<td></td>
</tr>
<tr>
<td>JPO</td>
<td>JNP</td>
</tr>
<tr>
<td>JPE</td>
<td>JP</td>
</tr>
<tr>
<td>Checking CX for zero without inspecting the flags</td>
<td></td>
</tr>
<tr>
<td>JCXZ</td>
<td>no alias</td>
</tr>
</tbody>
</table>

and unsigned arithmetic, from the settings of individual flags such as the parity flag, and from other considerations.

The main reason for there being so many arithmetic conditional jump instructions is that we want to be able to jump depending on whether one number is bigger than another as well as whether two numbers are equal. But this poses a problem, for the 8-bit binary number 11111111 is bigger than the 8-bit binary number 00000000 provided the former is interpreted as an unsigned number. If, however, 11111111 is interpreted as an 8-bit signed number then it is less than 00000000 (for its decimal value is then −1). So ‘jump if bigger’ and ‘jump if smaller’ need to be interpreted differently for signed and unsigned numbers.

To make the necessary distinction, the designers of the 8086 family of microprocessors decided to use the terms above and below when comparing unsigned numbers and to use the terms greater than and less than when comparing signed numbers. Thus, JA and JB are jumps which are made as a
result of comparisons of unsigned numbers whereas JG and JL are jumps depending on signed number comparisons.

To save memorizing the whole of Table 7.1 at once, it is sufficient for now to work with a ‘survival-kit’ of jump instructions – just seven of them. If we first remind ourselves of two points, we can state these very simply. The two points are:

1. decisions whether to jump are usually made upon the settings of the flags when the jump instruction is executed, and
2. this decision therefore depends on the outcome of the last instruction executed prior to the jump instruction itself which affected the flags.

The seven ‘survival-kit’ jump instructions are:

- **JMP** Jump (no matter what)
- **JZ** Jump if the result was Zero
- **JNZ** Jump if the result was Not Zero
- **JA** Jump if the result was Above zero ;unsigned
- **JB** Jump if the result Below zero ;arithmetic
- **JC** Jump if the result was Greater than zero ;signed
- **JL** Jump if the result was Less than zero ;arithmetic

and these should enable the reader to cope until Table 7.1 has been fully absorbed.

---

**EXERCISES**

7.1 Three numbers held in AX, BX and CX respectively are to be put into ascending numerical order with AX containing the smallest, BX the next largest and CX the largest of all. Write two different assembly language fragments to do this. In the first, assume that AX, BX and CX contain unsigned 16-bit numbers and in the second assume that they contain signed 16-bit numbers.

7.2 Write an assembly language program fragment which will add two signed 8-bit numbers. If the result is too large to represent as an 8-bit signed number then the message:

```
overflow during signed addition
```

should be displayed on the screen.

7.3 Write an assembly language program which will add together the two signed 8-bit numbers stored in locations referred to by **FIRST_NO** and