Addresses and Instructions Which Affect Memory

Contents

3.1 Jumps
3.2 Storing and retrieving data from memory
3.3 Programs which affect memory – some examples
3.4 The code and data segment registers
3.5 Defining segments in an assembly language program

Summary
Self-check quiz

This chapter begins by examining just three of the 8086-family's rich supply of jump instructions. These are the mechanism by which loops can be created in assembly language programs – just as for, repeat and while make this possible in Pascal.

However, since the great majority of assembly language programs require the storage and retrieval of data items in memory, the main focus of this chapter is on a version of the mov instruction which makes this possible, and on the actual mechanism used for storing and retrieving the contents of registers. Register BX has a special role to play in this: this chapter describes how it allows us to access a group of memory locations which are only determined during program execution rather than in advance by the programmer. Because this material is so fundamental to understanding the contents of later chapters several detailed examples (including traces) of programs which access memory are given.

The 8086-family microprocessors use segmented memory, so all
addresses must be given in segment:offset form. Four special 16-bit registers are used to specify the segment involved in the storage or retrieval of instructions and data, namely the segment registers CS, DS, SS and ES; the offset of an instruction or data item is then given either explicitly or in an appropriate register. The penultimate section describes how actual memory locations are determined in this way.

Reflecting the fact that an instruction stored in memory has its segment specified by the Code Segment (CS) register, and a data item has its segment specified by the Data Segment (DS) register, an assembly language program must make it quite clear what is to be stored relative to each of these segment registers, and the others (SS and ES). Consequently, the final section in this chapter describes how to do this by adopting a conventional layout for assembly language programs. This skeleton program structure will be used throughout the book except where it is necessary to depart from it in order to describe particular features of 8086-family assembly language.

## 3.1 Jumps

**Jump** instructions allow members of the 8086 family to take decisions according to information provided by the flags register: for example, if registers AX and BX contain the ASCII code for the same letter then do one thing, if not then do another.

Generally speaking, instructions in an assembly language program are obeyed one after another. Jumps can enable execution to continue from an instruction further on in the program. Thus, in the following example, the

```
JZ MISSLOTS
```

instruction can be interpreted to mean, ‘Jump if execution of the previous instruction resulted in Zero (JZ) to the instruction labeled MISSLOTS and continue execution in the normal way from there’:

```
MOV AX,3
SUB BX,AX
JZ MISSLOTS
INC DX ;obey this and the following
;instructions only if subtracting
;the contents of AX from BX did
;not give zero
MISSLOTS: MOV CX,DX
```

Also, jumps can enable execution to continue from an earlier instruction in a similar fashion:

```
...
BACKAGAIN: MOV BX,DX
...
SUB DX,AX
JZ BACKAGAIN
INC DX
...
```

As we shall see in Chapter 7, 8086-family processors have a rich supply of jump instructions. For the time being just three of them will suffice:

```
JZ <label>
JNZ <label>
JMP <label>
```

The precise meaning of these instructions will be discussed in Chapter 7. For now we can simply interpret them as follows:

- **JZ**: Jump to `<label>` if the result of the last arithmetic instruction was zero;
- **JNZ**: Jump to `<label>` if the result of the last arithmetic instruction was non-zero; and
- **JMP**: Jump to `<label>`

The `<label>` is a luxury facility available thanks to assembly language. In machine code jump instructions you must either specify the precise address in memory where the instruction you want to jump to can be found or indicate how many instructions before or after the current one you want to jump. During the conversion process from assembly language to machine code all the `<label>`s in jump instructions are replaced by the appropriate numbers.

To make this conversion possible, there are certain rules governing the way in which `<label>`s are made up. A `<label>` must contain only alphabetic characters (A,B,...,Z,a,b,...,z), digits (0,1,2,3,4,...,9) or underline characters (_) and begin with an alphabetic character. `<label>`s can be as long as you like though the assembler cannot differentiate between two labels both beginning with the same 31 characters. Thus, `process_gr`, `remove_12`, `step_19`, `list56`...
are allowable \(<\text{label}>\)s but

\begin{verbatim}
3rdmonth
tues+weds
mylabel\ which\ contains;\ a\ semicolon
\end{verbatim}

are not.

### 3.1.1 An example program using jump instructions

Let us write an assembly language program for the 8086 family to compare the contents of registers AX and BX. If they both contain the same number we shall put a 1 in register DX; otherwise we shall put a 0 in DX. Further, suppose that the contents of both AX and BX are to be unaltered at the end of the program. The following satisfies our design constraints. (It is certainly not the best program possible but has been chosen to illustrate the use of jump instructions.)

\begin{verbatim}
MOV CX,AX ; take a copy of the contents of AX as the next
 ; instruction will destroy what is in AX
SUB AX,BX
JZ MAKE1 ; if AX and BX contained the same number this
 ; instruction will cause execution to continue
 ; from the instruction labeled MAKE1
MOV DX,0 ; otherwise execution continues sequentially,
 ; so set DX to 0 indicating that the contents
 ; of AX and BX were different
JMP RESET ; jump unconditionally to the instruction
 ; labelled RESET (this is to avoid obeying
 ; the following instruction automatically)
MAKE1: MOV DX,1 ; we only obey this instruction if the
 ; contents of AX and BX were the same
RESET: MOV AX,CX ; restore the original contents of AX from CX
\end{verbatim}

### EXERCISES

3.1 Write an assembly language program fragment which puts the ASCII code for the letter Y in register DX if the sum of the contents of registers AX and BX is the same as the current contents of register CX. If not, put the ASCII code for N in register DX. Lastly, put 0 in registers AX, BX and CX.

3.2 Given that on entry register AX contains 3H and register BX contains 4H, what will be the contents of registers AX, BX and CX after execution of the following program fragment:

\begin{verbatim}
MOV CX,0
AGAIN: ADD CX,AX
SUB BX,1
\end{verbatim}
If registers AX and BX contained 2H and 5H respectively, can you say what registers AX, BX and CX would contain after execution of the above program fragment without doing any more work?

3.3 Write an assembly language program which will examine the contents of register AX and leave 0 in register DX if AX contains 47H and leave 0FFH in register DX otherwise. The program should preserve the contents of register AX, that is, after the program has been run the contents of AX should be the same as at the beginning.

3.2 Storing and retrieving data from memory

Until now our programs have only used some of the 8086-family registers and flags. Any ‘real’ program will necessitate the use of memory to store (possibly large amounts of) data. Data can be transferred from memory into a register one or two bytes at a time. The instruction

```
MOV AL, [10H]
```

will transfer the contents of location 10H into register AL whereas the instruction

```
MOV BX, [20H]
```

will transfer the contents of locations 20H and 21H into BX. The symbol [20H] is read as ‘the contents of location 20H’ to distinguish between

```
MOV AL, 20H  (put the hexadecimal number 20H in AL)
```

and

```
MOV AL, [20H] (put the contents of location 20H in AL)
```

Storing data in memory can also be done one or two bytes at a time. MOV [20H], CL would copy the contents of register CL into location 20H; MOV [20H], DX would copy the contents of register DX into locations 20H and 21H.

If register AX contained 2A8BH and the instruction

```
MOV [20H], AX
```

were executed, location 20H would contain 8BH and location 21H would contain 2AH. It is rarely the case that the programmer needs to be aware of
this as moving data the other way – from memory to register – reverses this ‘swap’, as in Figure 3.1.

### 3.2.1 Changing addresses

Varying an address while a program is running involves specifying the location concerned in a register. The following program adds 1 to each of the 8-bit numbers stored in locations 200H–202H inclusive. At the core of the program is the instruction

\[
\text{MOV AL}, [BX]
\]

which moves into AL the contents of the location whose address is given in register BX, and its counterpart

\[
\text{MOV [BX]}, AL
\]

which saves what is in register AL as the contents of the location whose address is given in BX.

The program will work by repeating the same basic group of instructions three times – once each for locations 200H, 201H and 202H. In order to count the number of repeats, we shall use CX to record the number of repetitions left to be done. At the beginning of the program, CX will be initialized to 3 therefore. After execution of the basic group of instructions, we shall arrange to have CX reduced by 1 by using the

\[
\text{DEC CX}
\]

instruction. DEC can be used in a similar form with any of the 16-bit general-purpose registers AX, BX, CX or DX and their 8-bit constituent registers. On
<table>
<thead>
<tr>
<th>AL</th>
<th>BX</th>
<th>CX</th>
<th>200H</th>
<th>201H</th>
<th>202H</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>200</td>
<td>?</td>
<td>2A</td>
<td>B4</td>
<td>51</td>
</tr>
<tr>
<td>?</td>
<td>200</td>
<td>3</td>
<td>2A</td>
<td>B4</td>
<td>51</td>
</tr>
<tr>
<td>2A</td>
<td>200</td>
<td>3</td>
<td>2A</td>
<td>B4</td>
<td>51</td>
</tr>
<tr>
<td>2B</td>
<td>200</td>
<td>3</td>
<td>2B</td>
<td>B4</td>
<td>51</td>
</tr>
<tr>
<td>2B</td>
<td>201</td>
<td>3</td>
<td>2B</td>
<td>B4</td>
<td>51</td>
</tr>
<tr>
<td>2B</td>
<td>201</td>
<td>2</td>
<td>2B</td>
<td>B4</td>
<td>51</td>
</tr>
<tr>
<td>B4</td>
<td>201</td>
<td>2</td>
<td>2B</td>
<td>B4</td>
<td>51</td>
</tr>
<tr>
<td>B5</td>
<td>201</td>
<td>2</td>
<td>2B</td>
<td>B5</td>
<td>51</td>
</tr>
<tr>
<td>B5</td>
<td>201</td>
<td>2</td>
<td>2B</td>
<td>B5</td>
<td>51</td>
</tr>
<tr>
<td>B5</td>
<td>202</td>
<td>2</td>
<td>2B</td>
<td>B5</td>
<td>51</td>
</tr>
<tr>
<td>51</td>
<td>202</td>
<td>1</td>
<td>2B</td>
<td>B5</td>
<td>51</td>
</tr>
<tr>
<td>52</td>
<td>202</td>
<td>1</td>
<td>2B</td>
<td>B5</td>
<td>51</td>
</tr>
<tr>
<td>52</td>
<td>202</td>
<td>1</td>
<td>2B</td>
<td>B5</td>
<td>52</td>
</tr>
<tr>
<td>52</td>
<td>203</td>
<td>1</td>
<td>2B</td>
<td>B5</td>
<td>52</td>
</tr>
<tr>
<td>52</td>
<td>203</td>
<td>0</td>
<td>2B</td>
<td>B5</td>
<td>52</td>
</tr>
</tbody>
</table>

**Figure 3.2**
Trace of program to increment the contents of locations 200H–202H by 1.

execution, the contents of the named register are decremented by 1.

The program now follows. Once again this is not necessarily the best or easiest way of going about the task – the choice of instructions has been dominated by the desire to explain.

```
MOV BX,200H ;BX will specify the location to be worked on
MOV CX,3H ;CX will maintain a count of how many more locations to be done

ALTER_NEXT: MOV AL,[BX]
INC AL
MOV [BX],AL ;the preceding three instructions add 1 to the contents of the location whose address is given in BX
INC BX ;advance BX to the next location
DEC CX ;one less location to be done now
JNZ ALTER_NEXT ;do some more unless none left to do
```

Assuming that before this program is run locations 200H–202H have the contents

<table>
<thead>
<tr>
<th>Location</th>
<th>200H</th>
<th>201H</th>
<th>202H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>2A</td>
<td>B4</td>
<td>51</td>
</tr>
</tbody>
</table>

then a trace of the execution of the program will look like Figure 3.2.
There are corresponding instructions whereby two bytes are moved instead of one. For example

```
MOV AX,[BX]
```

transfers two bytes, one from the location whose address is given in register BX and one from the location after that.

**BX is different from AX, CX and DX**

There is no equivalent of the `[BX]` part of instructions like

```
MOV CX,[BX]
```

for registers AX, CX and DX. For example,

```
MOV CX,[DX]
```

is illegal.

is not allowed because of the `[DX]`. Only register BX may be used in this way.

---

**EXERCISE**

3.4 What will be the contents of locations 200H, 201H, 202H and 203H after execution of each of the following program fragments (use ? to describe a value which is unknown):

(a) MOV AL,6
    MOV AH,5
    MOV BX,0ABDCH
    MOV [200H],AX
    MOV [202H],BX
(b) MOV AX,43BCH
    MOV BX,0FE01H
    MOV [200H],AX
    MOV [202H],BX
(c) MOV CX,34H
    MOV [201H],CX
    MOV DX,808H
    MOV [202H],DX
(d) MOV BX,200H
    MOV AX,2A3BH
    MOV [201H],AX
    MOV [BX],AX

---

**3.3 Programs which affect memory – some examples**

**Example 1**

A program to swap the contents of locations 20H and 21H is:
Example 2

A program to store 0 in locations 200H–300H inclusive. (This is part of a technique used to test if the memory chips in a microcomputer are working properly. Every location is filled with 0 and then after a fraction of a second pause the contents of each location are checked to verify they are still set at 0. If not, then one of the chips is malfunctioning.)

Since locations 200H–300H inclusive are to be filled, there are 101H locations to fill. (Compare this with being asked to deliver mail to addresses 0 to 7 – if you do, you’ll visit eight houses in total.)

```
MOV BX,200H ;BX records the address of the next location to be done
MOV CX,101H ;CX will keep a count of the number of locations left to do
DQ,ANOTHER: MOV [BX],0H ;fill the location
INC BX ;move on to the next address
DEC CX ;one less to do now
JNZ DQ,ANOTHER
```

Example 3

Count the number of occurrences of a given 16-bit number and leave this count in register DX. If locations 200H–24FH inclusive contain 28H 16-bit numbers (that is, 28H numbers having 4 hexadecimal digits), the program in Figure 3.3 will count the number of them which are equal to 0127H.

In order to give part of a trace for the program we will assume that the contents of locations 200H–205H are as follows:

<table>
<thead>
<tr>
<th>Location</th>
<th>200</th>
<th>201</th>
<th>202</th>
<th>203</th>
<th>204</th>
<th>205</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>2A</td>
<td>B5</td>
<td>27</td>
<td>01</td>
<td>6C</td>
<td>1B</td>
</tr>
</tbody>
</table>

Thus, as far as the 8086-family microprocessor is concerned, the 16-bit number in locations 200H and 201H is 0B52AH; that in 202H and 203H is 0127H; and that in 204H and 205H is 1B6CH.

Example 4

Making a total of 16-bit numbers stored in memory. If locations 500H to 54FH contain 28H 16-bit numbers, the program below will add up these numbers and leave the total in register AX.

The program makes use of the instruction

```
ADD AX,[BX]
```

which takes the two bytes stored starting at the address given in register BX as
a 16-bit number (stored with its least significant byte in the lower address, the most significant byte in the higher address) and then adds that number to the contents of register AX leaving the answer in AX.

\[
\begin{array}{cccc}
\text{Before:} & \text{Location} & \text{Location} \\
\text{AX} & \text{BX} & 2011 & 2012 \\
0115 & 2011 & 3A & 1B \\
\text{After:} & \text{Location} & \text{Location} \\
\text{AX} & \text{BX} & 2011 & 2012 \\
1C4F & 2011 & 3A & 1B \\
\end{array}
\]

(since 0115H + 1B3AH = 1C4FH)

In order that a partial trace of the execution of the totalling program (Figure 3.4) can be given we assume that the contents of locations 500H–505H before execution are as follows:

\[
\begin{array}{cccccccc}
\text{Location} & 500 & 501 & 502 & 503 & 504 & 505 \\
\text{Contents} & 00 & 01 & 00 & 01 & E2 & 00 \\
\end{array}
\]

---

**Figure 3.3**
Program and trace to count occurrences of 16-bit numbers.
Programs which affect memory – some examples

<table>
<thead>
<tr>
<th></th>
<th>AX</th>
<th>BX</th>
<th>CX</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV AX,0</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>MOV BX,500H</td>
<td>0</td>
<td>500</td>
<td>?</td>
</tr>
<tr>
<td>MOV CX,28H</td>
<td>0</td>
<td>500</td>
<td>28</td>
</tr>
<tr>
<td>ADD_IN_NEXT: ADD AX,[BX]</td>
<td>100</td>
<td>500</td>
<td>28</td>
</tr>
<tr>
<td>INC BX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC BX</td>
<td>100</td>
<td>501</td>
<td>28</td>
</tr>
<tr>
<td>DEC CX</td>
<td>100</td>
<td>502</td>
<td>28</td>
</tr>
<tr>
<td>JNZ ADD_IN_NEXT</td>
<td>100</td>
<td>502</td>
<td>27</td>
</tr>
<tr>
<td>200</td>
<td>502</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>503</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>504</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>504</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>2E2</td>
<td>504</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>2E2</td>
<td>505</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>2E2</td>
<td>506</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>2E2</td>
<td>506</td>
<td>25</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3.4** Program (and trace) to total 16-bit numbers stored in memory.

---

**Memory locations**

<table>
<thead>
<tr>
<th>BX</th>
<th>AX</th>
<th>CX</th>
<th>200</th>
<th>201</th>
<th>202</th>
<th>203</th>
<th>204</th>
<th>205</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV AX,0H</td>
<td>200</td>
<td>0</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>MOV CX,101H</td>
<td>200</td>
<td>0</td>
<td>100</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>NEXT_LOC: MOV [BX],AX</td>
<td>200</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>INC BX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC BX</td>
<td>201</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>INC AX</td>
<td>202</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>INC AX</td>
<td>202</td>
<td>1</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>DEC CX</td>
<td>202</td>
<td>2</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>JNZ NEXT_LOC</td>
<td>202</td>
<td>2</td>
<td>OFF</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>202</td>
<td>2</td>
<td>OFF</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>203</td>
<td>2</td>
<td>OFF</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>204</td>
<td>2</td>
<td>OFF</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>204</td>
<td>3</td>
<td>OFF</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>204</td>
<td>4</td>
<td>OFF</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>204</td>
<td>4</td>
<td>0FF</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>204</td>
<td>4</td>
<td>0FE</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>204</td>
<td>4</td>
<td>0FE</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 3.5** Program and trace storing 16-bit numbers in memory.
Example 5

The next example is rather academic but illustrates how 16-bit numbers are stored in memory. The program will store 0H as a 16-bit number in locations 200H,201H; 2H in locations 202H,203H; 4H in locations 204H,205H; and so on, up until 200H has been stored in 400H and 401H (Figure 3.5).

EXERCISE

3.5 Write 8086-family assembly language program fragments to achieve each of the following tasks.

(a) Fill locations 200H, 202H, 204H, . . . , 300H inclusive with the 8-bit unsigned number 0FFH.

(b) Swap the contents of location 200H with those of location 300H, those of 201H with 2FFH, and so on. Thus:

<table>
<thead>
<tr>
<th>Location</th>
<th>Contents</th>
<th>Location</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>2A</td>
<td>200</td>
<td>3F</td>
</tr>
<tr>
<td>201</td>
<td>56</td>
<td>201</td>
<td>AB</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>Before</td>
<td></td>
<td>After</td>
<td></td>
</tr>
<tr>
<td>2FF</td>
<td>AB</td>
<td>2FF</td>
<td>56</td>
</tr>
<tr>
<td>300</td>
<td>3F</td>
<td>300</td>
<td>2A</td>
</tr>
</tbody>
</table>

(c) Count the number of locations in the range 200H–300H inclusive which contain the ASCII code for the letter X.

(d) Locations 200H–03FFH inclusive contain 100H 16-bit numbers. Leave in register AX the number of them which are equal to 1984H.

(e) Store

500H in locations 200H,201H
4FEH in locations 202H,203H
4FCH in locations 204H,205H
.
.
.
400H in locations 300H,301H
3.4 The code and data segment registers

Recall from Section 2.2 that the machine code version of a program is stored in a segment in memory which we call the code segment. Usually, the operating system decides exactly where in memory this code segment should be. While we can override the operating system's decision and specify precisely where in memory our program is to be stored, for our purposes we shall be very glad of the operating system's help. This is also the case regarding the storage of any data which a machine code program needs, for (provided we don't object) the operating system will also choose where our data segment should be.

By leaving the location of our programs and data to it, we can be sure that we still have easy access to all the utilities that the operating system provides. These include several easily exploitable ready-made routines, for example, to read a character from the keyboard and print a character on the display screen (see Chapter 4). However, we do need to understand the mechanism by which members of the 8086 family access instructions and data from wherever they have been stored in memory.

Members of the 8086 family have four 16-bit registers called **segment registers** which form an essential part of this mechanism. These are the **Code Segment** (CS), **Data Segment** (DS), **Stack Segment** (SS) and **Extra Segment** (ES) registers. Their purposes can be roughly summarized as follows:

- **CS** used to access the memory segment containing program **instructions**.
- **DS** used to access the memory segment containing **data items** knowing the exact whereabouts of which is crucial to the program's design.
- **SS** used to access the memory segment containing **working memory** in which the precise location of a temporary data item (perhaps used as an intermediary step in some complicated calculation) is not crucial to the design of the program.
- **ES** used during the **manipulation of sequences of characters** by special 8086-family instructions.

Later chapters will explain more about the SS and ES registers: for now we concentrate on CS and DS.

3.4.1 Offsets within segments referred to by segment registers

The segment registers are thus titled because they specify the segment to which a particular object belongs. It follows that an offset within that segment must also be given in order to specify the complete memory address of that object (see Section 2.2).
Table 3.1 Specifying offsets in relation to segment registers.

<table>
<thead>
<tr>
<th>Object</th>
<th>Segment given by</th>
<th>Offset given by</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction</td>
<td>CS</td>
<td>IP</td>
</tr>
<tr>
<td>program data item</td>
<td>DS</td>
<td>explicitly or in BX, SI or DI</td>
</tr>
<tr>
<td>working storage item</td>
<td>SS</td>
<td>SP or BP</td>
</tr>
<tr>
<td>member of character sequence</td>
<td>ES</td>
<td>DI</td>
</tr>
</tbody>
</table>

For instructions the offset is given by a 16-bit register called the Instruction Pointer (IP) register. At any instant in time, the complete address of the next instruction to be executed is specified by the contents of CS (its segment) and the contents of IP (its offset within that segment).

The offset of data items may be specified by values given explicitly in the program, by the contents of BX, or by the contents of two other 16-bit registers: SI and DI (see Sections 8.2, 10.6 and 11.1). Among other things, SI and DI are used to index the elements of arrays and for string manipulation.

Items for which the segment is specified by SS can have their offset specified by two 16-bit registers, SP and BP (see Sections 8.3 and 8.7), and in other ways. When ES is used to give the segment address of sequences of characters, DI is used for the offset address. Table 3.1 summarizes this information.

During normal execution of a program, instructions are automatically fetched from memory so, as programmers, we generally do not have to worry about the particular values of CS and IP, or indeed, about the actual values of any of the segment registers. And, for the time being, the precise purpose of SI, DI, SP and BP will not concern us: these registers will feature large in later chapters.

3.4.2 Arrangement of registers inside the execution and bus interface units

Apart from the 8086-family registers we have already mentioned, the Execution Unit (EU) contains a set of circuits which actually carry out the arithmetic: the Arithmetic and Logic Unit (the ALU). Consequently registers which are frequently involved in arithmetic operations, such as AX, BX, CX and DX, are located there. As was pointed out in Chapter 2, it is the job of the Bus Interface Unit (BIU) to determine instruction and data memory addresses. It follows that the segment registers are located there, as shown in Figure 3.6.

3.4.3 Accessing the next instruction in the code segment

Even though the actual order in which the instructions in a program are executed is arranged automatically by the microprocessor, it will be important later for the reader to have an understanding of the mechanism used. As was
mentioned above, the address of any program instruction is specified via CS (segment) and IP (offset). Given:

Contents of CS: 3B12H        Contents of IP: 12EFH

we can determine the precise location in memory of the next instruction to be executed from the addition:

\[
\begin{align*}
3B120H & \\
12EFH & \\
\hline
3C40FH &
\end{align*}
\]

which indicates that it begins at location 3C40FH.

Different instructions may vary in length. For example, \texttt{INC AX} consists of just one byte whereas \texttt{MOV AX, [100H]} consists of three bytes. To be explicit, let us consider execution of the following program fragment by the 8086 microprocessor:

```
.*
.*
INC DX
MOV AX, [100H]
INC AX
.*
```


and assume that somehow we have discovered that it is stored in memory as follows:

```
1023:20AA INC DX
1023:20AB MOV AX,[100H]
1023:20AE INC AX
```

Now let us focus attention on what happens just before execution of the MOV AX,[100H] instruction. Once INC DX has been moved into the instruction queue within the BIU, CS and IP will have been set to 1023H and 20ABH respectively. During execution of INC DX (or, possibly, even earlier) the three bytes of the MOV instruction will be moved into the instruction queue within the BIU and IP advanced to point to the next instruction in sequence, INC AX at 1023:20AEH. IP will therefore be set to 20AEH and as soon as it is possible, the INC AX instruction will be fetched from memory and put into the instruction queue in the BIU.

Once execution of INC DX is complete, execution of MOV AX,[100H] will commence. Since the MOV cannot be executed until the contents of location 100H is fetched from memory, fetching this data item will take priority over any further instruction fetches necessary to keep the instruction queue filled.

### 3.4.4 Addressing data items in memory

At this point we have to admit to telling a little white lie in respect of instructions which address memory; or at least not telling the whole truth. In fact, the actual address from which data is fetched when an instruction like

```
MOV AL,[200H]
```

is obeyed is not location 200H but *location 200H relative to the contents of the data segment register*. Any explicit address specified in an 8086-family instruction is, in fact, taken relative to the DS register’s contents.

To see how this works out, suppose the DS register contains 0500H. The actual whereabouts of the byte which will be moved into AL on execution of the instruction:

```
MOV AL,[200H]
```

can be calculated as follows:

1. Add a zero to the right-hand side of the contents of the DS register.  
   
   
   \[
   \begin{array}{c}
   200H \\
   \hline
   \end{array}
   \]

   \[
   \begin{array}{c}
   5000H \\
   \hline
   \end{array}
   \]

2. Add to that the address given.

   \[
   \begin{array}{c}
   5000H \\
   \hline
   \end{array}
   \]

   \[
   \begin{array}{c}
   200H \\
   \hline
   \end{array}
   \]

   \[
   \begin{array}{c}
   5200H \\
   \hline
   \end{array}
   \]

   Total 5200H
Thus, the actual location accessed by the `MOV AL,[200H]` instruction is 5200H in this case.

Loading the DS register (or any of the segment registers) cannot be done in the same way as loading registers like AX, BX, CX or DX. *DS must be loaded indirectly.* Thus,

```
MOV BX,500H
MOV DS,BX
```

would set the DS register to 500H.

In the following program fragment, the right-hand column shows in parentheses the actual location(s) addressed by each instruction in hexadecimal.

```
MOV BX,1275H
MOV DS,BX
MOV AX,5H
MOV DX,17H
MOV BH,[100H]  (12750 + 100  = 12850)
MOV BL,[30H]    (12750 +  30  = 12780)
MOV CX,[BX]     (12750 +  17 = 12767 and
                 12750 +  18 = 12768)
ADD BX,CX
ADD AX,BX
MOV [75H],AX   (12750 +  75 = 127C5 and
                12750 +  76 = 127C6)
MOV [204H],BL  (12750 + 204 = 12954)
MOV [31A5H],BH (12750 + 31A5 = 158F5)
```

However, usually the computer's operating system sets the DS register to a suitable value before a program is run. In absolute terms, it (the operating system!) decides where in memory to locate both our programs and data. By setting the DS register with a `MOV` instruction like the one above it is possible to insist on using a collection of specific locations for data but it is rare that one wants to do this. On the contrary, with our knowledge of the 8086 family in its present state we are only too grateful for all the help the operating system can provide.

### EXERCISE

3.6 In the following program fragments, some of the instructions transfer data between registers or move a number into a register whereas others involve the transfer of data to and from memory. For each instruction involving a transfer of data to or from memory, make a list of the actual absolute 20-bit memory address(es) used for each data transfer.
(a) MOV CX,23ABH  
    MOV DS,CX
    MOV AX,[31AH]
    MOV BX,[34BH]
    ADD AX,BC
    ADD AX,CX
    MOV [33BH],AX

(b) MOV DX,0ABCDH  
    MOV DS,DX
    MOV AL,[2DEH]
    MOV BL,[2FFH]
    DEC BL
    DEC AL
    ADD AL,BL
    MOV [3DFH],AL
    MOV AH,AL
    MOV [222H],AX

3.7 Given the following program fragment stored in the locations specified:

    BC1A:0000 ADD AX,3
    BC1A:0003 INC AX
    BC1A:0004 MOV DX,AX
    BC1A:0006 SUB DX,3
    BC1A:0009 MUL DX
    BC1A:000B ADD AX,DX

and assuming that prior to its execution by the 8086 microprocessor (in which the BIU instruction queue has length six bytes) the operating system has set the contents of CS to 0BC1AH and IP to 0000H, make a trace of the execution of the program in which the sequence of values of CS and IP are carefully recorded and show the contents of the BIU instruction queue at each stage.

3.5 Defining segments in an assembly language program

The most common assembler for 8086-family assembly language is called MASM. It requires that every program should have at least three segments: a data segment, a stack segment to provide working storage, and a code segment.

When any computer program is executed, both the program and any necessary data must be in memory. MASM gives the programmer absolute control over the memory so that exactly where both program and data are stored is the programmer’s choice. Normally one would have the program and data stored in completely separate areas of memory so that there is less chance of the program interfering with the data. There are occasions, however, where program and data can be mixed to advantage and it is to allow for this possibility that MASM insists that code and data segments in an assembly language program be clearly labeled.
3.5.1 Stacks: providing working storage

Most large programs require temporary working storage and for members of the 8086 family this is provided by means of a stack. A stack is a storage structure in which data items are stored in memory on a last-in, first-out basis, rather like the storage of plates in a cafeteria plate dispenser (see Figure 3.7). Items can be stored on the stack and retrieved from it during program execution without reference to any particular addresses.

![Figure 3.7](image)

The operation of a cafeteria plate dispenser models that of a stack: the plates are removed in the reverse order from that in which they were inserted.

Selection of stack size depends on the particular program. All the programs in this book will be adequately served by a 100H word stack which we shall conventionally allocate in a stack segment.

3.5.2 Telling the assembler which segments are which

Data segments, stack segments and code segments are distinguished by including assembler *pseudo-ops* in the assembly language program. These are not 8086-family instructions. Rather they are instructions to the assembler itself to assist it in its task of converting assembly language to machine code. Thus the pseudo-ops SEGMENT and ENDS tell MASM that everything which they enclose belongs to the same segment. In addition, each segment must be given a name:

```
<my_segment_name> SEGMENT
.
.
<my_segment_name> ENDS
```

Thus, for example,

```
MYPROGRAM SEGMENT
  MOV AX,4H
```
could be the bare bones of a code segment with name MYPROGRAM.

The programmer is free to make up <my_segment_name> subject to similar restrictions to those on jump label names in Section 3.1. In this book we shall adopt a convention that our data segment will be called DATA, our stack segment WORKING STORAGE, and the code segment CODE. Thus, the instructions in our programs will be written between CODE SEGMENT and CODE ENDS pseudo-ops.

We shall call the stack segment in our programs the WORKING STORAGE segment. The stack segment in a program must be clearly marked for the assembler by means of the STACK pseudo-op which must be appended to the WORKING STORAGE SEGMENT pseudo-op. Thus, the working store in our programs will be achieved by:

```
WORKING STORAGE SEGMENT STACK
  DW 100H DUP(?)
WORKING STORAGE ENDS
```

The stack segment of our programs will have addresses relative to the SS register because 8086-family instructions which operate on the stack (see Chapter 8) use SS to calculate the corresponding actual addresses (which, as we have observed, the programmer does not need to know about during normal execution).

MASM must also be told to which segment register a named segment belongs. This is achieved via the ASSUME pseudo-op. Hence, if our program had segments called MYDATA, MYSTACK and MYPROGRAM then:

```
ASSUME DS:MYDATA
ASSUME SS:MYSTACK
ASSUME CS:MYPROGRAM
```

would inform MASM that the DS register must be used for address calculation in the MYDATA segment (that is, MYDATA is a data segment), that the SS register must be used for address calculation in the MYSTACK segment (that is, MYSTACK is a stack segment) and that the CS register must be used to calculate all addresses in the MYPROGRAM segment (that is, MYPROGRAM is a code segment). The alternative form:

```
ASSUME DS:MYDATA, SS:MYSTACK, CS:MYPROGRAM
```

has the same effect.

Following our naming convention, we want DS associated with DATA, SS with WORKING STORAGE and CS with CODE. This is arranged by using the ASSUME pseudo-op as follows:

```
ASSUME DS:DATA, SS:WORKING STORAGE, CS:CODE
```

which is included as the first line of the code segment:
Throughout this book, we will adopt a conventional format for our assembly language programs which incorporates this code segment proforma. Thus, our programs will conventionally take the shape shown in Figure 3.8.

Figure 3.8
The conventional format to be used for an assembly language program in this book.

**SUMMARY**

Jump instructions allow us to alter the flow of control during program execution. The chapter began with an informal introduction to three of the 8086-family jump instructions: JZ, JNZ and JMP. Chapter 7 contains a more detailed consideration of jump instructions.

We also saw that data can be transferred between registers and memory by means of versions of the MOV instruction which, among others, can take one of the forms:

(a) `MOV register_name, [n]`
(b) `MOV [n], register_name`
where \( n \) is a 16-bit unsigned number specifying an offset relative to the DS segment register and thus determining a unique location in memory. If the `register_name` refers to one of the 8-bit registers, the contents of that location is copied into the named register (format (a)), or the contents of the named register copied into that location (format (b)).

If a 16-bit `register_name` is employed then, in format (a), the 16-bit data item to be moved into the register is constructed from the contents of the uniquely determined memory location (let us call this \( m \)) and the following one (\( m + 1 \)). The contents of location \( m \) goes into the eight lower order bits of the 16-bit register, and the contents of \( m + 1 \) into the eight higher order bits. With format (b) the lower order eight bits of the register go into location \( m \) and the higher order eight bits go into location \( m + 1 \).

The `mov` instruction formats:

(c) \texttt{mov register\_name, [bx]}

(d) \texttt{mov [bx], register\_name}

can be used to access memory in a similar way by setting the contents of BX to the offset of the required address. Thus, if BX contains the 16-bit offset \( n \), instructions in the above two formats will be executed in an equivalent way to:

\texttt{mov register\_name, [n]}
\texttt{mov [n], register\_name}

respectively. BX is special in this respect, however, for there are no equivalents of (c) and (d) involving AX, CX or DX.

The segment registers CS, DS, SS and ES determine the segment to which an address used in one of the above instructions corresponds. Segment registers are given values automatically by DOS, but we can override this if we wish. However, segment registers cannot be loaded with a value directly.

In order that the segment registers can be correctly associated with instructions and data in our assembly language programs, the assembly language programs in this book will usually conform to a conventional layout which was described in Section 3.4. It is possible to use other layouts – indeed the MASM manual recommends a much simpler looking format – but we have chosen ours as the best compromise between ease of use and scope for illustration of 8086-family assembly language features.
SELF-CHECK QUIZ

1. Which of the following are valid label names and which are not? For those which are not, give a reason.
   (a) MUL
   (b) TESTING&CHECKING
   (c) ZERO CHECK
   (d) 12_STEPS_LEFT_TO_GO
   (e) PRICES_IN_£
   (f) PLLLLLLPLLP

2. What would be the contents of locations 200H, 201H, ..., 210H after execution of the following program fragment:
   MOV CX,200
   MOV AX,32ABH
   NEXT: MOV BX,CX
   MOV [BX],AX
   INC AX
   INC BX
   MOV [BX],AX
   MOV CX,BX
   SUB BX,210H
   JNZ NEXT

3. Given that the contents of locations 200H and 201H are 4AH and 8F8H respectively, what will be the contents of the specified register or memory location(s) after execution of the following instructions:
   (a) AX, after MOV AX,[200H]
   (b) AH after MOV AX,10AH
   (c) ADD [200H],AX
   (d) MOV AX,[200H]
   (e) MOV BL,AL
   (f) MOV AL,AH
   (g) MOV AH,BL
   (h) MOV [200H],AX

4. Why are segment registers necessary in an 8086-family microprocessor? Could segment registers be dispensed with entirely or, if not, how would we provide the facilities they provide without using segment registers?

5. What would be the advantages of having more segment registers? What would be the disadvantages?

6. Write an 8086-family assembly language fragment that would fill a complete segment of memory (the data segment?) with zeros.

7. Given that owing to previous instructions, locations 300H to 600H inclusive contain either 0 or 1, write an assembly language fragment which will set AL to 0 if the number of locations containing a 1 is even or to 0FFH if it is odd.

8. If DS contains 3AB5H, which absolute location(s) in memory will be addressed by each of the following:
   (a) MOV BX,2AB1H
   (b) MOV BX,1A9CH
   (c) ADD AX,[BX]

9. How would you arrange for a 250H byte stack to be included in an assembly language program?

10. Explain the positioning of registers within the execution unit and bus interface unit. Why do you think the 8086-family designers arranged things the way they are?