This chapter examines the different ways in which an 8086-family microprocessor instruction can specify the data items on which an instruction is to operate. These different ways are referred to as addressing modes. We shall study 8086-family addressing modes in relation to the problem of implementing the machine code generation phase of a compiler for a high-level language such as Pascal. The 8086-family machine code instruction formats are highly dependent on these addressing modes and so in turn is the way in which 8086-family assembly language instructions are encoded into binary. Section 13.2 describes the encoding process itself, since an understanding of it is vital to someone who has to write an 8086-family assembler or debugger.

13.1 Addressing modes

The designers of the 8086 family of microprocessors had the convenient implementation of high-level language data types, control constructs and sub-program mechanisms as one of their major objectives. This is reflected in the
### General-purpose 16-bit registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td></td>
</tr>
<tr>
<td>CX</td>
<td></td>
</tr>
<tr>
<td>DX</td>
<td></td>
</tr>
<tr>
<td>SI</td>
<td></td>
</tr>
<tr>
<td>DI</td>
<td></td>
</tr>
<tr>
<td>SP</td>
<td></td>
</tr>
<tr>
<td>BP</td>
<td></td>
</tr>
</tbody>
</table>

### General purpose 8-bit registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH</td>
<td></td>
</tr>
<tr>
<td>BH</td>
<td></td>
</tr>
<tr>
<td>CH</td>
<td></td>
</tr>
<tr>
<td>DH</td>
<td></td>
</tr>
</tbody>
</table>

### Segment registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td></td>
</tr>
<tr>
<td>DS</td>
<td></td>
</tr>
<tr>
<td>SS</td>
<td></td>
</tr>
<tr>
<td>ES</td>
<td></td>
</tr>
</tbody>
</table>

### Other 16-bit registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP</td>
<td></td>
</tr>
</tbody>
</table>

### Segment registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td></td>
</tr>
<tr>
<td>DS</td>
<td></td>
</tr>
<tr>
<td>SS</td>
<td></td>
</tr>
<tr>
<td>ES</td>
<td></td>
</tr>
</tbody>
</table>

### Other 16-bit registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td></td>
</tr>
<tr>
<td>DS</td>
<td></td>
</tr>
<tr>
<td>SS</td>
<td></td>
</tr>
<tr>
<td>ES</td>
<td></td>
</tr>
</tbody>
</table>

Figure 13.1
A summary of registers common to all members of the 8086 family.

The choice of addressing modes which the designers made available, namely: register, immediate, direct, register indirect, direct indexed, base relative and base indexed addressing modes. In this section each of them will be considered in turn.

#### 13.1.1 Register addressing

Microprocessors in the 8086 family have eight general-purpose 16-bit registers, eight general-purpose 8-bit registers related to four of the 16-bit registers, at least four segment registers, the flags register and the IP register (which the programmer cannot access directly) as summarized in Figure 13.1.

If every instruction operand is fetched from or moved into a register then an instruction is said to employ register addressing mode. Thus:

```
MOV AL, BL
INC BX
DEC AL
SUB DX, CX
```

are all examples of instructions using register addressing.

Instructions which affect only registers execute faster than instructions involving memory access, as no time is spent fetching operands. Consequently, a good Pascal compiler will endeavour to ensure that data items required in repeatedly executed sections of a program are held in registers whenever possible.

Thus, if \(a\) and \(b\) are identifiers corresponding to integer variables, then by the time it comes to executing:
\[ a := 4; \ b := 3; \]
\[ \text{for } i := 1 \text{ to } 10000 \text{ do} \begin{align*}
\text{begin} \\
\text{code}[i] & := \text{code}[a] + \text{code}[b]; \\
a & := a + 2; \\
b & := b + 1
\end{align*} \text{end} \]
a Pascal compiler would have already assigned memory locations to \( i, a \) and \( b \), but would try to arrange that the values of \( i, a, \) and \( b \) are moved into registers for the duration of the \text{for} loop execution.

### 13.1.2 Immediate addressing

In **immediate addressing** mode the data item on which the instruction is to operate is stored as part of the instruction in memory. Thus, the instruction

\[ \text{MOV CL,61H} \]

uses immediate addressing mode because the data item on which it operates, \( 61H \), is stored as part of the machine code version of the instruction which, in hexadecimal, is:

\[ B1 \ 61 \]

\( 0B1H \) being the \text{opcode}, the binary code which corresponds to assembly language statements of the form:

\[ \text{MOV CL,8-bit number} \]

and \( 61H \) being the 8-bit number.

During the normal 8086-family execution sequence, \text{MOV CL,61H} \text{ is moved into the instruction queue from memory prior to actual execution. Since the immediate operand (61H) is stored as part of the instruction itself (0B161H) the immediate operand is thus fetched from memory at the same time as the instruction itself, once again reducing execution time. The same is true for 16-bit immediate operands. For example, in:

\[ \text{MOV DI,2AB4H} \]

the immediate operand \( 2AB4H \) forms part of the machine code version of this assembly language instruction:

\[ \text{BF 2AB4} \]

and so the operand is fetched from memory at the same time as the instruction itself.

### 13.1.3 Direct addressing

**Direct addressing** mode is similar except that in this case the effective \textit{address} of one of the operands is taken directly from the instruction. Consider the instruction
MOV AX,TOTAL

in which TOTAL corresponds to the location with offset 210H relative to DS. The machine code version of this instruction is

A1 1002

Since 1002 is the memory representation of 0210H, MOV AX,TOTAL is an instruction which uses direct addressing mode.

Only one operand of an instruction may be directly addressed, so that if total and item_sum are identifiers of two Pascal integer variables which we assume are represented in signed 16-bit form then, for optimal execution speed not involving arithmetic in registers, the Pascal

\[ \text{total} := \text{total} + \text{item_sum} \]

could be coded by a compiler using two directly addressed instructions:

MOV AX,ITEMSUM
ADD TOTAL,AX

(where ITEMSUM and TOTAL are the locations corresponding to the Pascal identifiers item_sum and total respectively).

Also, given

\[ \text{const taxrate} = 30; \]

\[ \text{var total:integer;} \]

\[ \ldots \]

\[ \begin{align*}
\text{begin} \\
\ldots \\
\ldots \\
\text{end}
\end{align*} \]

the Pascal statement \( \text{total} := \text{taxrate} \) could be coded using immediate and direct addressing thus:

MOV AX,30D
MOV TOTAL,AX

(where we assume that TOTAL is a word in memory corresponding to the Pascal integer total).

13.1.4 Register indirect addressing

In an instruction which employs register indirect addressing mode, the effective address is found in either the BX, BP, SI or DI register. In other words, the effective address is not found directly from the instruction itself but indirectly by accessing a register, as in

SUB DX,[BX]
A Pascal compiler may use indirect addressing for manipulating pointers. For example, given:

```pascal
type customer = record
  name: array[1..30] of char;
  address: array[1..50] of char
end;

customers = file of customer;

var mail_list: customers;
```

then BX could be used to point to successive records in the `mail_list` file so that the assembly language `[BX]` corresponds to the Pascal `mail_list`.

### 13.1.5 Direct indexed addressing

We should first observe that authors often differ in the names they give to the various addressing modes. Of course, the correct names are those given to the modes by the creators of the 8086 family, Intel Corporation. But writers often believe they can make things clearer to their readers by choosing their own names. The addressing mode described in this section which we will call **direct indexed addressing** is a good example. While we are certainly not alone in using this name, many other authors call this **indexed addressing**. Our name allows us to emphasize the relation between this addressing mode and another one, called based indirect addressing, which shall be described in due course.

An instruction which uses **direct indexed addressing** mode obtains its data item from, or stores its data item in, an address which is the sum of the contents of an index register (DI or SI) and a signed 8-bit number or an unsigned 16-bit number. Thus,

```assembly
MOV AX,[SI+4]  
ADD [DI-6],CX  
MUL WORD PTR [SI-192]
```

are all examples of instructions which use direct indexed addressing.

In `MOV AX,[SI+4]`, the offset (relative to DS) of the data item to be moved into AX is found by adding 4 to the contents of SI. In `ADD [DI-6],CX` the offset relative to DS specifying the data item to which the contents of CX are to be added is found by subtracting 6 from the contents of DI.

### Alternative forms

8086-family assembly language actually requires instructions which use direct indexed addressing mode to be written in forms such as:

```assembly
MOV AX,[SI+36]  
MOV [SI-27],AX
```

but MASM also allows the alternative forms
MOV AX,36[SI]
MOV -27[SI],AX
MOV AX,[SI]+36
MOV [SI]+27,AX

Moreover, if TOTAL is defined by a pseudo-op of the form

TOTAL DW ..

and corresponds to offset 204H (say) from DS, then given the instruction

MOV AX,TOTAL[BX]

MASM will assemble this into the machine code equivalent of

MOV AX,[BX+204]

Some justification for the existence of direct indexed addressing mode

Consider implementing the Pascal assignment

account[i] := total

where we have

var
i,total:integer;
account,daily_balance:array[1..limit_constant] of integer;

Suppose further that the array account is stored in signed 16-bit form in consecutive locations corresponding to the MASM variable ACCOUNT starting with account[1]; that the value of i is in register SI; and that the MASM variable TOTAL corresponds to total. Then

MOV AX,TOTAL
MOV ACCOUNT[SI],AX

is equivalent to

account[i] := total

The facilitation of the compilation of such high-level language array accesses is precisely the reason that the 8086 family has direct indexed addressing mode, where the effective address is calculated as ‘index register contents + displacement’. An assignment of the form

account[i] := daily_balance[j]

requires two index registers (which is one reason why the 8086 family has SI and DI), and if we assume the value of j is in register DI, then

MOV AX,DAILY_BALANCE[DI]
MOV ACCOUNT[SI],AX

is the assembly language equivalent of account[i] := daily_balance[j] (assuming suitable arrangements with regard to segment registers). Similarly
for $i := 1$ to 10 do $balance[i] := balance[i] - credit[i] + debit[i];$

could be implemented

MOV CX,10D
MOV SI,1
NEXT_BALANCE: MOV AX,BALANCE[SI]
SUB AX,CREDIT[SI]
ADD AX,DEBIT[SI]
MOV BALANCE[SI],AX
INC SI
INC SI
LOOP NEXT_BALANCE

13.1.6 Base relative addressing

In base relative addressing mode, the offset of a data item is found by adding a signed 8-bit or unsigned 16-bit number to the contents of a base register (BX or BP). Thus

MOV AX,[BX-3]
SUB [BP+4],5

are examples of instructions which employ base relative addressing mode. MASM also allows these instructions to be written in the alternative forms, such as

MOV AX,[BX]-3
MOV AX,-3[BX]

Of course, addresses given in BX are taken relative to DS whereas those in BP are taken relative to SS.

In order to see why this addressing mode is provided, consider the Pascal declarations:

type marks = record
    student_number:integer;
    math:integer;
    french:integer;
    physics:integer;
    geography:integer;
    biology:integer;
end;

exam = file of marks;
var year1:exam;

and a related Pascal program fragment to find the average physics mark in year 1:

$sum := 0; count := 0;$
while not eof do
    begin
        get(year1);
        $sum := sum + year1^\text{physics}$;
count := count + 1
end;
if count <> 0 then
  writeln('average physics mark is', sum/count)
end;

To access the record component year1^physics in the machine code equivalent of this fragment, base relative addressing mode is used.

The physics component of a marks record will be stored in the word corresponding to the 6th and 7th bytes of that record (see Figure 13.2), numbering bytes in a record as 0,1,2,3,... Hence, if BX is used to point to the beginning of successive records in the year1 file so that [BX] corresponds to year1^, then the word at [BX+6] corresponds to year1^physics and

\[ \text{sum} := \text{sum + year1^physics} \]

could be implemented as:

\[
\begin{align*}
\text{MOV AX, SUM} \\
\text{ADD AX, [BX+6]} \\
\text{MOV SUM, AX}
\end{align*}
\]

For the less powerful members of the 8086 family, direct indexed addressing and base relative addressing amount to the same thing: both involve the format register + displacement. However, there is actually a big conceptual advantage to be gained by thinking of them as different. And when we come to study the 80386 chip and more powerful members of the 8086 family in Chapter 23 we shall see that, for these microprocessors, there is a genuine difference between the two modes.

The best way to think of direct indexed addressing is that it is used to

![Figure 13.2](image_url)

The file year1.
access successive elements of an array, the displacement corresponding to the start of the array and the register specifying the particular element within that array. Base relative addressing, on the other hand, allows access to an individual component of a record in a file, the base register corresponding to the start of the record and the displacement specifying the particular component in that record.

13.1.7 Base indexed addressing

An instruction which uses **base indexed addressing** mode obtains its data item from, or stores its data item at, an address which is obtained by adding together the contents of a base register and an index register and, optionally, an 8-bit (or 16-bit) signed (respectively, unsigned) number added to that. Thus

```
MOV AX,[BX+DI]
MOV [BX+SI+2],AX
INC BYTE PTR [BP+SI-45]
```

are all examples of instructions which use base indexed addressing mode.

In **MOV AX,[BX+DI]** the data item will be moved from the location having offset relative to DS obtained by adding the contents of the BX and DI registers. In **INC BYTE PTR [BP+SI-45]** the location whose contents are to be incremented will have offset relative to SS obtained by adding the contents of BP to the contents of SI and then subtracting 45.

As with the direct indexed and base relative addressing modes, MASM allows alternative forms of instruction which employ base indexed mode. Thus, all three terms may be in brackets in any order or the signed 8- or 16-bit number may be combined with either register. Thus:

```
MOV AX,[BX+SI+2]
MOV AX,[SI+BX+2]
MOV AX,[SI][BX+2]
MOV AX,[BX][SI+2]
MOV AX,[BX][SI]
```

would all be assembled by MASM into the same machine code instruction.

Based indexed addressing mode is provided to assist in the accessing of records in a file which contain array fields. Consider the Pascal declarations

```pascal
    type classrecord = record
        student_number:integer;
        age:integer;
        coursework_marks:array[1..20] of integer;
        exam_average:integer
    end;

    var class_file:file of classrecord;
```

and suppose that we want to find the average total coursework mark for all students whose details are held in class_file (see Figure 13.3.).
Assuming that, within each record, the array coursework_marks is held in bytes 4 to 43 inclusive (numbering bytes in a record 0, 1, 2, 3, 4, … as usual), then the body of the for loop in:

\[
\text{get(class\_file); no\_of\_students := 0; sum := 0; while not eof do begin no\_of\_students := no\_of\_students + 1; for } i := 1 \text{ to 20 do sum := sum + class\_file\^\_course\_work\_marks[i]; get(class\_file) end}
\]

could be implemented by using the base register BX to hold the starting address of this particular record (that is, corresponding to class\_file\^), and having the index register SI point to each particular array element to be added. Thus:

\[
\text{ADD AX,[BX+SI+4]}
\]

would correspond to

\[
\text{sum := sum + class\_file\^\_course\_work\_marks[i];}
\]

(see Figure 13.3).
13.1.8 A summary of 8086-family addressing modes

This completes our introduction to the 8086-family addressing modes. The details are summarized for easy reference in Table 13.1.

13.1.9 Addressing modes involving BP

In a Pascal compiler, indirect addressing can be useful for matching formal parameters and actual parameters and controlling the allocation of memory to procedures. Procedures in Pascal are re-entrant: that is, they may be invoked (called up) while already in execution from a previous invocation. This could occur, for example, if a procedure were recursive (and so invoked itself) or if it invoked some other procedure which in turn invoked the original procedure.

The main consequence of this is that both the parameters and any local variables must be assigned to a unique set of memory locations – one for each concurrent invocation. Otherwise the data being used by one invocation might be corrupted by a subsequent invocation. Thus, memory must be allocated for

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Operand format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>register</td>
<td>reg</td>
<td>MOV AX,BX</td>
</tr>
<tr>
<td>immediate</td>
<td>data</td>
<td>MOV AX,31</td>
</tr>
<tr>
<td>direct</td>
<td>disp</td>
<td>MOV AX,TOTAL</td>
</tr>
<tr>
<td>register indirect</td>
<td>[BX]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[BP]</td>
<td>MOV AX,[BX]</td>
</tr>
<tr>
<td></td>
<td>[SI]</td>
<td>MOV AX,[BP]</td>
</tr>
<tr>
<td></td>
<td>[DI]</td>
<td>MOV AX,[SI]</td>
</tr>
<tr>
<td>direct indexed</td>
<td>[SI]+disp</td>
<td>MOV AX,TOTAL[SI]</td>
</tr>
<tr>
<td></td>
<td>[DI]+disp</td>
<td></td>
</tr>
<tr>
<td>base relative</td>
<td>[BX]+disp</td>
<td>MOV AX,[BP]+disp</td>
</tr>
<tr>
<td></td>
<td>[BP]+disp</td>
<td>MOV AX,[BP]4</td>
</tr>
<tr>
<td>base indexed</td>
<td>[BX][SI]+disp</td>
<td>MOV AX,[BP][DI]+disp</td>
</tr>
<tr>
<td></td>
<td>[BX][DI]+disp</td>
<td>[BP][SI]+disp</td>
</tr>
<tr>
<td></td>
<td>[BP][DI]+disp</td>
<td>MOV AX,[BX+SI+3]</td>
</tr>
</tbody>
</table>

Note: reg is any 8- or 16-bit register except IP
data is an 8- or 16-bit constant value
disp is an 8- or 16-bit signed number, and is optional for base indexed addressing.
a procedure's data items (parameters and local variables) every time the procedure is called. The collection of memory locations allocated in this way is called an **activation record**. Because the memory used by a procedure is released when execution of that procedure terminates, Pascal compilers usually arrange to keep activation records on the stack: this is a natural consequence of the fact that the last procedure invoked will be the first to finish execution. (We have already encountered this idea in all but name when we discussed the passing of parameters to subroutines via the stack, in Section 8.7).

Each time a Pascal procedure is invoked, an appropriate block of memory is reserved for the activation record on the top of the stack by changing the contents of the SP register. During execution, BP is used to remember where the activation record begins. Access to items within the activation record can then be arranged by the addressing modes which use BP. For example, as we saw in Section 8.7, \([BP + \text{disp}]\) can be used to access a simple parameter within the activation record.

To be specific, consider the declarations

```
proc example(var f:integer);
var locx,locy:integer
begin
   ...
end;
var a:integer;
```

Clearly, the assembly language or machine code equivalent of `example` produced by a compiler cannot use any specific address corresponding to the variable `f` since this will not be known until a call such as

```
example(a)
```
Table 13.2 Use of memory addressing modes to implement Pascal and similar high-level languages (\(disp\) is an unsigned 16-bit number).

<table>
<thead>
<tr>
<th></th>
<th>Ordinary</th>
<th>Pointer</th>
<th>Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables of type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\texttt{integer} and \texttt{char}</td>
<td>direct</td>
<td>BX</td>
<td>BP + disp</td>
</tr>
<tr>
<td>Arrays</td>
<td>SI + disp</td>
<td>BX + SI</td>
<td>BP + SI + disp</td>
</tr>
<tr>
<td>DI + disp</td>
<td>BX + DI</td>
<td>BP + DI disp</td>
<td></td>
</tr>
<tr>
<td>Individual records</td>
<td>direct</td>
<td>BX + disp</td>
<td>BP + disp</td>
</tr>
<tr>
<td>Arrays within records</td>
<td>SI + disp</td>
<td>BX + SI + disp</td>
<td>BP + SI + disp</td>
</tr>
<tr>
<td>DI + disp</td>
<td>BX + DI disp</td>
<td>BP + SI + disp</td>
<td></td>
</tr>
</tbody>
</table>

occurs and the actual parameter \((a)\) corresponding to the formal parameter \((f)\) is known.

At the point of invocation, code will be generated by the Pascal compiler to put an activation record on the stack consisting of just three words: one for the \texttt{integer} parameter and two for the local integer variables \texttt{locx} and \texttt{locy}. If the address of the start of the activation record is put into BP then indirect addressing can be used to refer to the parameter in the form \([BP]\) (using the fact that indirect addressing with BP is relative to SS) and the local variables can be referred to using \([BP-4]\) and \([BP-6]\) (see Figure 13.4).

The uses of addressing modes in the implementation of high-level languages like Pascal are summarized in Table 13.2. It has three columns, one for ordinary variables of each type, one for pointers to variables of that type, and one for a corresponding activation record.

### EXERCISES

13.1 For each of the following say whether it is a valid 8086-family assembly language instruction and then submit the whole text to MASM to verify your answers.

\begin{verbatim}
ADD AX,[BX+CX+2]  
SUB DX,[BX+BP+8]  
SUB CH,[BX+20487] 
MOV [BX+SI+2],AL  
MOV [BH+SI+32],DL  
MOV AX,[12+SI+BP]  
ADD [SI]TOTAL,3  
SUB [DI+3+BP],2  
ADD 3[DI+BP],2  
ADC 3[DI][BP],2  
SBC TOTAL[DI][BP],2
\end{verbatim}