Chapter 4

Addressing Modes, Instruction Encoding, and Data Definition

We have seen in Chapter 1 that a **machine language program** consists of instructions and data represented as sequences of bits. A **machine language instruction** has in general an operation code (opcode) and an operand, and specifies a fundamental operation that the CPU is able to carry out. The **opcode** specifies the operation to be carried out by the CPU, whereas the **operand** specifies the data and/or the location(s) of the data on which the operation is to be performed. For some instructions, the operand also specifies the location of the result. We have also seen that an **assembly language instruction** is the symbolic representation of a machine language instruction. It has the general format:

mnemonic-opcode ***operand(s)*** **[*remark*]**

where ***mnemonic-opcode*** is the symbolic representation of an opcode.

The number of operands depends on the instruction. The Intel 8086 processor's instructions either have zero, one, or two operands. An optional ***remark*** may follow the operand(s). It begins with a semicolon and consists of a text that is used to explain or clarify the purpose of an instruction.

The mnemonic-opcode, the operand(s), and the remark are separated from each other by one or more spaces and/or tabs. The following are examples of assembly language instructions:

```
MOV   AX, BX              ;  copy contents of register BX into register AX
ADD   AX, 19h             ;  add 25 to contents of register AX
IMUL  WORD PTR [BX]
SUB   WORD PTR [DS : 200h], 15
```

You may notice in the above examples that operand(s) are also specified in symbolic forms and that two operands are separated from each other with a comma. This chapter first discusses how the operands of assembly language instructions may be specified. It then describes how they are encoded in machine language instructions, and how to allocate storage and define constant data in an assembly language program.

4.1 Addressing Modes

Addressing modes refer to the different ways in which operands of instructions of the Intel 8086 processor are specified.

Seven addressing modes are used to specify operand(s) of instructions: *register*, *immediate*, *direct*, *register indirect*, *direct indexed*, *base relative*, and *base indexed* addressing mode. The last five are used to specify the offset or the "selector:offset" address of a memory location. We discuss these addressing modes in turn and then show how they are used to implement high-level programming languages features.

Register Addressing

An instruction is said to use **register addressing** mode if it specifies an operation in which data is fetched from or moved into a register. The register is specified in the instruction by its symbolic names. For examples AX, AH, BX, CL, CS, and SS. The following table illustrates instructions with operand(s) in register addressing mode.

Instructions	*Remarks*
MOV AX, CX	two register operands: the first is register AX, and the second is CX
ADD AL, 5	the first operand is a register: register AL
IMUL BX	one register operand: register BX

Immediate Addressing

For certain instructions, the data item to be processed is specified as part of the instruction. These instructions are said to use **immediate addressing** and the data item is referred to as **immediate data**. Only one operand of a two-operand instruction may be immediate data. Immediate data may be specified in each of the following number system:

- decimal (followed by an optional "D" or "d" suffix). For examples 25, -25D, or 50d.
- binary (followed by the suffix "B" or "b"). For examples 1101B, -1101b, or 110B.
- hexadecimal (followed by the suffix "H" or "h"). For examples 4AH, -0Ch, or 0FF4h.
- octal (followed by the suffix "O", "o", "Q" or "q"). For examples 24o, 24q, or -37q.

The following table illustrates instructions with an operand in immediate addressing mode.

Instructions	*Remarks*
MOV AX, 5Fh	two operands: the second is an immediate data 5F (in hexadecimal)
ADD AL, -101101b	two operands; the second is an immediate data -101101 (in binary)
RET 4	one immediate data operand 4 (in decimal)

The five remaining addressing modes are used for instructions that have a memory location operand. The Intel 8086 processor has two types of instructions with a memory location operand: some instructions specify operations in which data is fetched from or moved into a memory location, and others specify the transfer of control to another instruction. The remaining addressing modes are used to specify the offset or the "selector:offset" address of a memory location operand.

Direct Addressing

An instruction is said to use **direct addressing** mode if the offset of a memory location is specified in the instruction as an operand. The offset is specified in the same way as an immediate data, but in square brackets. For example [DS : 20F5h], [DS :10110010b], [DS:25d], or [DS:12]. You precede an offset with a segment register in order to specify the segment in which the memory location is located. The default segment is the data segment for most instructions. The following table illustrates instructions with a memory location operand in direct addressing mode.

Instructions	***Remarks***
MOV AX, [DS : 1Ah]	two operands: the first is register AX and the second is a word memory location at offset 001Ah in the data segment.
ADD [DS : 1Ah], AL	two operands: the second is register AL, and the first is a byte memory location at offset 001Ah in the data segment.
MOV BYTE PTR [DS :1Ah], 5	two operands: the first is a byte memory location at offset 001Ah in the data segment, and the second is a byte immediate data.
IMUL WORD PTR [DS:1Ah]	one operand: a word memory location at offset 001Ah in the data segment.

Register Indirect Addressing

For some instructions with a memory location operand, the offset of the memory location operand may be specified in a register. For the Intel 8086 processor, the only registers that you can use for this purpose are registers BX, SI, and DI.

A register used to store the offset of a memory location operand is specified in an assembly language instruction in square brackets (such as [BX], [SI], or [DI]) and the instruction is said to use **register indirect addressing**. The offset of the corresponding memory location is found by first accessing the specified register. The segment containing the memory location operand is not specified in the instruction. It is implied by the instruction The following table illustrates instructions with an operand in register indirect addressing mode.

Instructions	*Remarks*
MOV AX, [BX]	two operands: the first is register AX, and the second is a word memory location in the data segment at offset in register BX.
MOV BYTE PTR [DI], 5	two operands: the first is a byte memory location in the data segment at offset in register DI, and the second is a byte immediate data.
IMUL WORD PTR [SI]	one operand: a word memory location in the data segment at offset in register SI.

Register indirect addressing is suitable for the implementation of pointer variables in high-level programming languages such as C/C++. For example, a list of values of the same type (characters or two's complement binary integers of the same size) that are stored in consecutive memory locations can be processed in a loop by using a register to hold the offset of each of these memory locations. To move from one data item in the list to the next, this register is incremented in the body of the loop by the size of a data item. For example, in the following C/C++ code segment, pointer pt could be implemented using one of the registers BX, DI, or SI..

```
int total = 0;
int * pt;
int list[] = { 5, -3, 11, -17, 3};
for (pt = list; pt  < list +5;  pt ++)
     total  = total +  *pt;
```

Direct Indexed Addressing

With **direct indexed addressing** mode, the offset of a memory location is computed by adding the contents of an index register (SI or DI) to a signed 8-bit integer or an unsigned 16-bit integer called **displacement**. The index register and the displacement may be specified in an assembly language instruction in one of the following forms:

[Reg $\pm$ Disp] or [Reg] $\pm$ Disp

where Reg is either SI or DI and the displacement is specified in the same way as an immediate operand.

Direct indexed addressing mode is appropriate for the implementation of indexed variables in high-level programming languages. For example, in the following C/C++ code segment, list[i] could be specified in an instruction by using the offset of the first element of the array as the displacement and either register SI or DI to hold the index.

```
int total = 0;
int list[] = { 5, -3, 11, -17, 3};
for (int i = 0; i < 5; i++)
     total  = total + list [i];
```

The following table illustrates instructions with an operand in direct indexed addressing mode.

Instructions	*Remarks*
MOV AX, [SI + 6]	the second operand is a word memory location in the data segment. Its offset is computed by adding 6 to the contents of register SI.
MOV BYTE PTR [DI - 3], 5	the first operand is a byte memory location in the data segment. Its offset is computed by adding -3 to the contents of register DI.
IMUL WORD PTR [SI] + 4	one operand: a word memory location in the data segment. Its offset is computed by adding 4 to the contents of register SI.

Base Relative Addressing

Base relative addressing mode is similar to direct indexed addressing mode, except that base register BX or base pointer register BP is used instead of the index register SI or DI. The operand may also be specified in one of the forms:

[Reg $\pm$ Disp] or [Reg] $\pm$ Disp

where Reg is either register BX or register BP. Register BP is used only when the memory location is in the stack.

Note that when a data item is in the data segment, its offset can be specified using either the base relative addressing or the direct indexed addressing mode. However, we find base relative addressing more appropriate for the implementation of pointer arithmetic found in high-level programming languages such as C/C++. For example, in the following C/C++ code segment, the variable *pt* could be implemented using either register BP or register BX, depending on whether the memory location is in the stack or not; and the expression pt + 2 implemented as either [BP + 2] or [BX + 2].

```
int list[] = {1, 4, 7, 10, 13},
*pt, num;
pt = list;
num  = *(pt + 2);
```

Base relative addressing is also suitable for accessing a member of an element of an array of records (or structures), For example, if we consider the C/C++ statement:

num = employee[i].hours;

employee[i] could be implemented using register BP or register BX (depending on whether the memory location is in the stack or not) to hold the offset of the first member of the i^{th} structure, and *hours* implemented by a displacement that corresponds to the number of bytes between this member and the first member of the structure.

Another situation in which base relative addressing is appropriate is when a structure is referenced by a pointer variable which is used to access a member of the structure as in the following C/C++ statement:

num = pt -> hours;

pt could be implemented using register BP or register BX (depending on whether the memory location is in the stack or not) and *hours* implemented by a displacement that corresponds to the number of bytes between this member and the first member of the structure. The following table illustrates instructions with an operand in base relative addressing mode.

Instructions	***Remarks***
MOV AX, [BP + 6]	the second operand is a word memory location in the stack segment. Its offset is computed by adding 6 to the contents of register BP.
ADD [BX - 9], AL	the first operand is a byte memory location in the data segment. Its offset is computed by adding - 9 to the contents of register BX.
MOV BYTE PTR [BX - 3], 5	the first operand is a byte memory location in the data segment. Its offset is computed by adding -3 to the contents of register BX.
IMUL WORD PTR [BP]+ 4	one operand: a word memory location in the stack segment. Its offset is computed by adding 4 to the contents of register BP.

Base Indexed Addressing

With the **base indexed addressing** mode, the offset of a memory location is computed by adding the contents of a base register (register BX or register BP) to the contents of an index register (register SI or register DI), and optionally a signed 8-bit or an unsigned 16-bit displacement. The operand may be specified with or without a displacement in one of the following forms:

i) Without displacement:

[Reg1][Reg2] or [Reg1] + [Reg2] or [Reg1+Reg2]

ii) With a displacement:

[Reg1 +Reg2 $\pm$ Disp] or [Reg1][Reg2] $\pm$ Disp

Where Reg1 is either register BX or register BP, and Reg2 is either register SI or register DI. Register BP is used instead of register BX only if the memory location is in the stack.

The base indexed addressing mode is more appropriate for the implementation of high-level programming languages indexed variables of multi-dimensional arrays. For example, if we consider the C/C++ statement:

```
Document [l][c] = 'b';
```

Document [l][c] could be implemented by using either register BX or register BP (depending on whether the memory location is in the stack or not) to implement variable *l*, and register SI or register DI to implement variable *c*; with the displacement being the offset of the first element of the array.

The following table illustrates instructions with an operand in base indexed addressing mode.

Instructions	*Remarks*
MOV AX, [BP][SI] + 6	the second operand is a word memory location in the stack segment. Its offset is computed by adding 6 to the sum of the contents of registers BP and SI.
ADD [BX + DI - 9], AL	the first operand is a byte memory location in the data segment. Its offset is computed by adding - 9 to the sum of the contents of registers BX and DI.
MOV BYTE PTR [BX][SI], 5	the first operand is a byte memory location in the data segment. Its offset is the sum of the the contents of registers BX and DI.
IMUL WORD PTR [BP + DI]	one operand: a word memory location in the stack segment. Its offset is the sum of the contents of registers BP and DI.

Exercise 4.1

What is the addressing mode of each operand of the following instructions:

Instructions	**Addressing Modes**
ADD AX, CX	
SUB BX, 1Ah	
MOV [2B5h], CX	
SUB AX, [DI]	
MOV 4[DI], BX	
IMUL BYTE PTR 4[BX + DI]	
ADD WORD PTR 5[BX], 3	
MOV AX, [BP-4]	
MOV [BP][SI], DX	

The PTR Operator

The size of a memory location operand can either be a byte, a word, . . . , etc., and must be specified in an instruction for the assembler or Debug to generate the proper machine language instruction.

In a two-operand instruction in which one of the operands is a register, this information is implied by the size of the register: in most instructions, both operands must have the same size. However, in a one-operand instruction or in a two-operand instruction with an immediate data as one of the operands, the size of the memory location or the immediate data (byte, word, . . . , etc) must explicitly be specified using the **PTR** operator.

An operand (memory location or immediate data) is considered to be a byte if it is preceded by **the BYTE PTR** operator and it is considered to be a word if it is preceded by the **WORD PTR** operator. The following table illustrates instructions in which the PTR operator is used to specify the size of memory location and immediate operands.

Instructions	***Remarks***
MOV [DS:200h], WORD PTR 5	two operands: a word in memory at offset 0200h in the data segment, and a word immediate data 0005h
ADD BYTE PTR [BX], 20	two operands: a byte in memory in the data segment at offset in register BX and a byte immediate data 14h
MOV [DS:200h], CX	two operands: a word in memory at offset 0200h in the data segment and a word in register CX. WORD PTR operator is not necessary.
MOV BL, 7	two operands: a byte in register BL and a byte immediate data. BYTE PTR operator is not necessary.
IMUL WORD PTR [SI]	one operand: a word memory location in the data segment at offset in register SI

Example 4.1 illustrates the computing of the offset and the absolute/effective address of a memory location operand specified in register indirect, direct indexed, base relative, or base indexed addressing mode. Note that register BP can only be used to specify the offset of memory locations in the stack.

Exercise 4.2

Indicate by T (true) or F (false) whether the size of the following operands must be explicitly specified using the PTR operator:

a. MOV AX, [DS:150h] T F

b. ADD [BX][SI], 5 T F

c. ADD BX, 5 T F

d. MOV [DS:150h], 2Ah T F

e. IDIV [DS:150h] T F

f. IDIV BL T F

Example 4.1 Computing the Offset and the Absolute Address

Assume the following register status:

DS = 2000 SS = 3000 BX = 012A BP = 021B DI = 0010 SI = 0020.

For each of the following memory location operands:

a. [SI + 5] **b.** [BP][SI] + 12h **c.** [BX][DI] **d.** [BP - 3]

1) compute its offset.
2) compute its absolute address.

Solutions

Operand	Offset	Absolute Address
a. [SI + 5]	0020 + 5 = 0025	2000**0** + 0025 = 20025
b. 12h[BP][SI]	021B + 0020 + 12 = 024D	3000**0** + 024D = 3024D
c. [BX][DI]	012A + 0010 = 013A	2000**0** + 013A = 2013A
d. [BP - 3]	021B - 3 = 0218	3000**0** + 0218 = 30218

Exercise 4.3

Assuming the following register status:

DS = 2000 SS = 3000 BX = 012A BP = 021B DI = 0010 SI = 0020.

1) compute the offset of each of the following memory location operands.
2) compute its absolute address.

Operand	Offset	Absolute Address
[BP + 1Ah]		
[BX][DI] + 5		
[DI + 9]		
[BP][SI]		
[SI + 7]		
[BX - 5]		

4.2 Instruction Encoding

This section discusses the different formats used to encode the machine language instructions of the Intel 8086 processor. These encoding formats are illustrated by using the syntax of assembly language instructions valid in the Debug environment

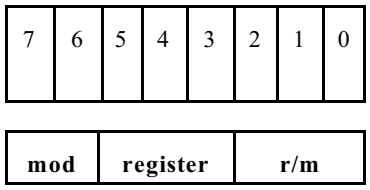
A machine language instruction consists in general of an opcode and an operand, both specified as a sequence of bits. But, the encoding of the operand depends on the addressing mode used in the instruction. The general structure of an instruction is as follows:

Opcode	'mod r/m' byte	Offset/Displacement	Immediate data

Opcode Except for some few instructions that have a two-byte opcodes, most machine language instructions for the Intel 8086 processor have a one-byte opcode. The primary role of the opcode is to specify the operation to be performed by the CPU. But, smaller encoding fields may be defined within the opcode byte to provide other information to the CPU such as the size of operands (8-bit or 16-bit), where the result should be stored, or that a 16-bit immediate data is represented in the instruction using 8 bits.

For some instructions the opcode byte also contains information about a particular register operand: this register is said to be implied by the opcode and is not specified in the operand field. The opcodes of a subset of the Intel 8086 processor's instructions are provided in Appendix 2.

'mod r/m' byte this byte consists of three fields: the *mod field* (bits 7 and 6); the *register field* (bits 5, 4 and 3); and the *r/m (register/memory) field* (bits 2, 1 and 0) as follows:



- the **mod** and the **r/m fields** are used to let the CPU know whether or not there is a memory location operand, and how to get its offset when it is the case.
- For some instructions, the **register field** is used as an opcode extension. Otherwise, it is used with the **r/m field** to hold the codes of register operands.

For each instruction in Appendix 2, we have also indicated whether or not a 'mod r/m' byte is needed; and when it is needed, there is also an indication on how its fields are used.

Offset/Displacement This field is present only if there is an operand in memory and its offset is specified in direct addressing mode or in an indirect addressing mode with a displacement.

An offset is specified here with its low and high bytes swapped. A one-byte displacement is a two's complement binary integer, whereas a word displacement is an unsigned binary integer. A word displacement is also specified with its high and low bytes swapped.

Immediate Data this field is present only if there is an operand in immediate addressing mode. Immediate data are two's complement binary integers and a word immediate data is specified with its low and high bytes swapped.

The remaining part of this section discusses the encoding of registers, then the mod and the r/m fields, and finally, the encoding of instructions.

Encoding Registers

Registers are encoded in a machine language instruction using a 3-bit code. However, the code used to encode a base or an index register depends on whether or not this register is used in register mode (that means used to hold data) or in one of the indirect addressing modes. Table 4.1 provides the codes of all registers (used in register mode), and Table 4.2 provides the codes of base and index registers when they are used in indirect addressing modes.

Table 4.1 **Encoding of Registers (in register mode)**

Codes	*16-bit register*	*segment register*	*8-bit register*
000	AX	ES	AL
001	CX	CS	CL
010	DX	SS	DL
011	BX	DS	BL
100	SP		AH
101	BP		CH
110	SI		DH
111	DI		BH

Note that the same code is assigned to two or three registers; the CPU distinguishes registers with the same code by using the context in which they are used. For example, in some contexts the code 100 will refer to register SP, but in others, it will refer to register AH. That means, register SP and AH can not be used in instructions with the same opcode.

Table 4.2 **Encoding of registers used in indirect addressing modes**

Codes	*registers*
000	[BX + SI]
001	[BX + DI]
010	[BP + SI]
011	[BP + DI]
100	[SI]
101	[DI]
110	[BP]
111	[BX]

The mod and the r/m fields

The mod and the r/m fields of the 'mod r/m' byte are used to let the CPU know if there is a memory location operand or not. When there is a memory location operand, it also specifies how to compute its offset. This information is provided in Table 4.3.

Table 4.3 using the mod and the r/m fields

mod	*r/m*	*observation*
11		no memory operand
00	110	memory operand: offset in direct addressing mode
00	code of [Reg]/[Reg1+Reg2]	memory operand: offset = [Reg] or [Reg1] + [Reg2]
01	code of [Reg]/[Reg1+Reg2]	memory operand: offset = [Reg] or [Reg1] + [Reg2] + 1 byte-displacement
10	code of [Reg]/[Reg1+Reg2]	memory operand: offset= [Reg] or [Reg1] + [Reg2] + 1 word-displacement

Exercise 4.4

Following the first two examples, use the mod and the r/m fields of the following 'mod r/m' bytes to indicate whether or not there is a memory location operand. Also specify how to compute the offset of a memory location operand (note: you do not have to compute the offset) if there is one.

'Mod r/m' Byte	Mod	Register	r/m	Remarks
D1	**11**	**010**	**001**	**No operand in memory.**
8E	**10**	**001**	**110**	**One operand in memory; offset = one word displacement + contents of register BP.**
0C				
1E				
18				
C2				
17				
91				

Encoding Instructions with No Operand

Some Intel 8086 processor's instructions such as the **CWD**, the **CBW**, or the **RETN** instruction have no operand and are encoded just by their opcode. Some examples are provided as follows:

Assembly Language	Machine Language
CWD	99h
CBW	98h
RETN	C3h

Encoding One-Operand Instructions

The operand of a one-operand instruction is either in a register, in memory, or as an immediate data. Their formats are discussed as follows:

Single Operand in a Register

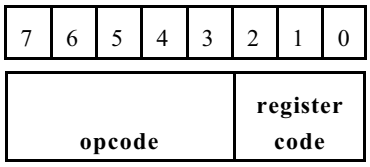
One-operand instructions that use register addressing mode have two encoding patterns: a **short form** which consists of just one byte, and a more **general form** which consists of the opcode byte followed by the 'mod r/m' byte.

The short form is used when the register operand is a 16-bit register, whereas the general form is used in any other situation. The short forms are shown in Figure 4.1, and the more general form is shown in Figure 4.2.

Figure 4.1 **Short form of an instruction with a single 16-bit register operand**

Two formats are used for the short form: one for segments registers, and another one for other 16-bit registers.

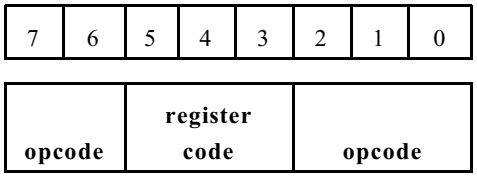
a) Short Form for 16-bit Registers that are not Segment Registers



Note that the machine language instruction that corresponds to a register may be obtained by adding to the opcode byte (in which bits 0, 1, and 2 are set to zero) the code of that register. Using the information in Appendix 2, we have the following machine language instructions:

Assembly Language	Machine Language		
PUSH AX	50h + 00	or	50h
PUSH BX	50h + 03	or	53h
INC AX	40h + 00	or	40h
INC BX	40h + 03	or	43h
INC SI	40h + 06	or	46h

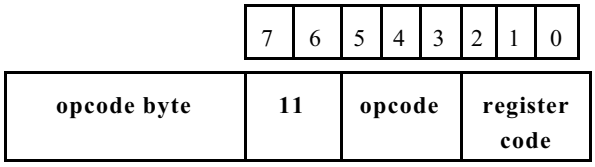
b) Short Form for Segment Registers



The 2-bit opcode in bit positions 6 and 7, and the 3-bit opcode in bit positions 0, 1, and 2 are provided in Appendix 2. Using this information, we have the following machine language instructions:

Assembly Language	Machine Language			
	Opcode	Register Code	Opcode	
PUSH ES	00	000	110	= 06h
PUSH CS	00	001	110	= 0Eh
POP DS	00	011	111	= 1Fh

Figure 4.2 **General form of an instruction with a single register operand**



Using the information in Appendix 2, we have the following machine language instructions:

Assembly Language	Opcode	mod-register-r/m			Machine Language
INC AH	FE	11	000	**100**	FE C4
			= C4h		
IMUL BX	F7	11	101	**011**	F7 EB
			= EBh		

Exercise 4.5

Provide the machine language instruction that corresponds to each of the following assembly language instructions:

a. POP CX **c.** DEC DX **e.** POP ES **g.** IMUL BL

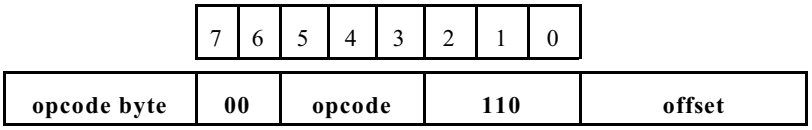
b. DEC BL **d.** IDIV CX **f.** PUSH DS

Single Operand in Memory

When the operand of a one-operand instruction is in memory, the encoding of the corresponding machine language instruction depends on the addressing mode used in the instruction. In Figure 4.3, different encoding patterns are provided with respect to the addressing mode used in the instruction.

Figure 4.3 Encoding Instructions with a Single Operand in Memory

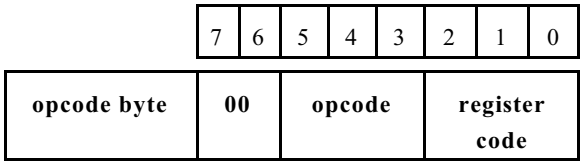
1. Direct Addressing Mode



Using the information in Appendix 2, we have the following machine language instruction:

Assembly Language	Opcode	mod-register-r/m	Machine Language
INC WORD PTR [DS:200h]	FF	00 000 110 = 06h	FF 06 00 02

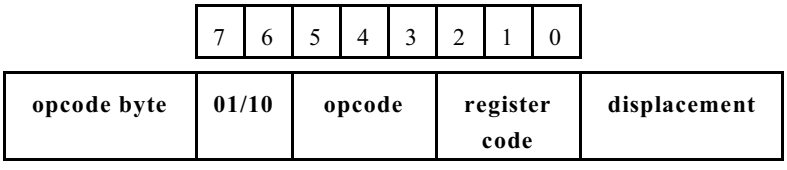
2. Indirect Addressing Mode (with no Displacement)



Using the information in Appendix 2, we have the following machine language instruction:

Assembly Language	Opcode	mod-register-r/m	Machine Language
INC BYTE PTR [BX][SI]	FE	00 000 **000** = 00h	FE 00

3. Indirect Addressing Mode with a Displacement



Using the information in Appendix 2, we have the following machine language instructions:

Assembly Language	Opcode	mod-register-r/m	Machine Language
IDIV WORD PTR [DI] +1A2Bh	F7	10 111 101 = BD	F7 BD 2B 1A
IDIV WORD PTR [SI + 2Bh]	F7	01 111 100 = 7C	F7 7C 2B

Exercise 4.6

Provide the machine language instruction that corresponds to each of the following assembly language instructions:

a. IDIV BYTE PTR [DS:15Ah]

b. IMUL WORD PTR [SI]

c. DEC WORD PTR [DS:21Eh]

d. INC BYTE PTR [BX][DI] + 1Fh

Single Immediate Data

When the only operand of an instruction is an immediate data, the machine language instruction is the opcode followed by that immediate data. For example, the code for **RETN 8** is **C2 08 00**, where **C2** is the opcode and **0008** is the immediate data. Note that a 16-bit immediate data is specified with its low and high byte swapped.

Exercise 4.7

Provide the machine language instruction that corresponds to each of the following assembly language instructions:

a. INT 3

b. INT 21h

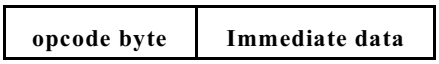
c. RET 6

Encoding Two-Operand Instructions

Ten encoding patterns are identified with respect to the addressing modes used for two-operand instructions. These encoding patterns are provided in Figure 4.4 with some examples.

Figure 4.4 **Encoding Two-Operand Instructions**

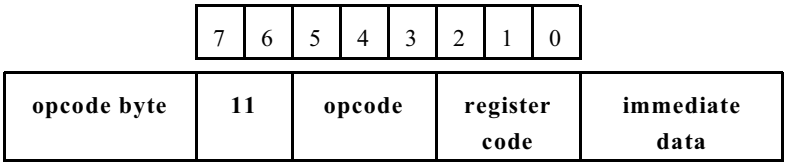
1. **Register, Immediate Operands (with no 'mod r/m' byte)**



Using the information in Appendix 2, we have the following machine language instruction:

Assembly Language	Opcode	Machine Language
ADD AX, 7	05	05 07 00

2. **Register, Immediate Operands (with 'mod r/m' byte)**

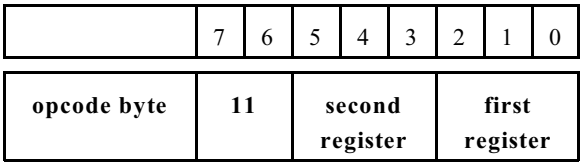


Using the information in Appendix 2, we have the following machine language instruction:

Assembly Language	Opcode	mod-register-r/m	Machine Language
CMP CX, 5	83	11 111 **001**	83 F9 05
		= F9	
	or 81		81 F9 05 00

Note that two encoding patterns are available for some instructions with an immediate data: one in which the immediate data is represented as a byte and another in which it is represented as a word. Therefore, if the immediate data can be represented as a byte, you may use either one of the encoding pattern.

3. Register, Register Operands



Using the information in Appendix 2, we have the following machine language instruction:

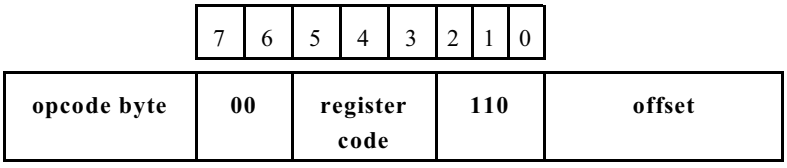
Assembly Language	Opcode	mod-register-r/m	Machine Language
SUB BX, DX	29	11 010 011 = D3h	29 D3

4. Register, Memory Operand in Direct Mode (without 'mod r/m' byte)

Using the information in Appendix 2, we have the following machine language instruction:

Assembly Language	Opcode	Machine Language
MOV [DS:200h], AL	A2	A2 00 02

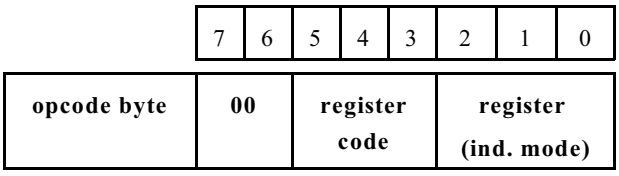
5. Register, Memory Operand in Direct Mode (with 'mod r/m' byte)



Using the information in Appendix 2, we have the following machine language instruction:

Assembly Language	Opcode	mod-register-r/m	Machine Language
Mov [DS:200h], BL	88	00 **011** 110 = 1Eh	88 1E 00 02

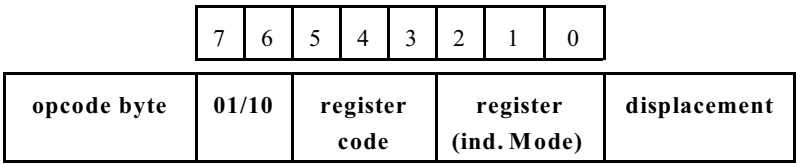
6. Register, Memory Operand in Indirect Mode (with no displacement)



Using the information in Appendix 2, we have the following machine language instruction:

Assembly Language	Opcode	mod-register-r/m	Machine Language
MOV [BX][DI], DS	8C	00 011 001 = 19h	8C 19

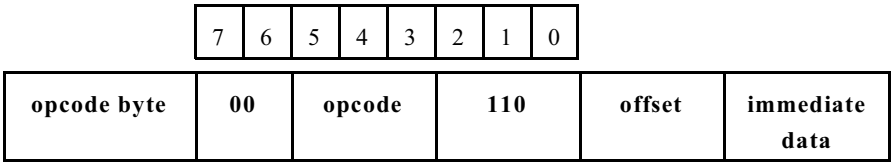
7. Register, Memory Operand in Indirect Mode (with displacement)



Using the information in Appendix 2, we have the following machine language instruction:

Assembly Language	Opcode	mod-register-r/m	Machine Language
MOV [BX][DI] + 5, DS	8C	01 011 001 = 59h	8C 59 05
ADD AX,[SI] + 1A2Bh	03	10 000 100 = 84h	03 84 2B 1A

8. Memory Operand in Direct Mode, Immediate Data

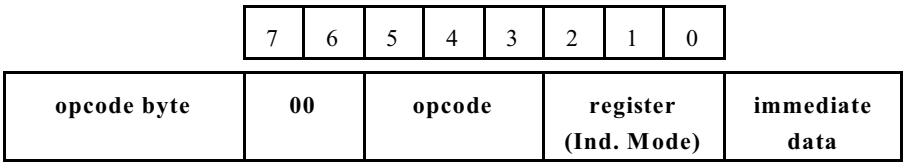


Using the information in Appendix 2, we have the following machine language instruction:

Assembly Language: SUB WORD PTR [DS:200h], 1A2Bh

Opcode	mod-register-r/m	Machine Language
81	00 101 110 = 2Eh	81 2E 00 02 2B 1A

9. Memory Operand in Indirect Mode (without Displacement), Immediate Data

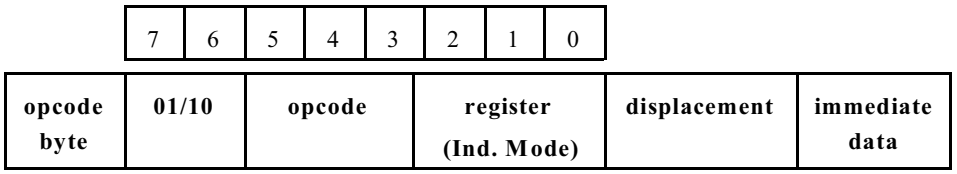


Using the information in Appendix 2, we have the following machine language instruction:

Assembly Language: SUB WORD PTR [BX], 5

Opcode	mod-register-r/m	Machine Language
83	00 101 111 = 2Fh	83 2F 05

10. Memory Operand in Indirect Mode (with a Displacement), Immediate Data



Using the information in Appendix 2, we have the following machine language instruction:

Assembly Language: SUB WORD PTR [BX] + 0E0Fh, 1A2Bh

Opcode	mod-register-r/m	Machine Language
81	10 101 111 = AFh	81 AF 0F 0E 2B 1A

Exercise 4.8

Provide the machine language instruction that corresponds to each of the following assembly language instructions:

a. CMP AX, 1Ah

b. ADD BX, 1Ah

c. MOV AX, [15Ah]

d. ADD AX, [SI]

e. MOV BYTE PTR [SI], 1Ah

f. MOV WORD PTR 2F[BX][SI], 1A1Ch

g. SUB AL, 5

h. MOV DS, AX

i. MOV BX, [15Ah]

j. ADD AX, 5[SI]

k. MOV WORD PTR [15Bh], 1A1Ch

Transfer of Control Instructions and Relative Addressing

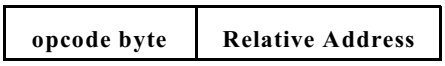
Immediately after an instruction is fetched from the main memory and before it is executed, the instruction pointer register IP is incremented by the length of that instruction so that its contents is the offset of the next instruction to be fetched from the memory. However, the Intel 8086 processor also has instructions generally referred to as *transfer of control instructions*, that alter the order in which instructions are fetched from the memory by writing a new offset into register IP.

There are two types of transfer of control instructions: those that only affect the contents of register IP, and those that also affect the contents of the code segment register CS in addition to affecting the contents of register IP.

Transfer of control instructions that only affect the contents of register IP are specified in assembly language as follows:

<mnemonic-opcode> ***<offset>***

But in machine language, this offset is replaced by an 8 or 16-bit two's complement binary integer value called *relative address*. The format of the corresponding machine language instruction follows:



The **relative address** is computed as follows:

<Offset> - (Offset of current instruction + length of current instruction)

or **<Offset> - (Offset of next instruction)**

Using the information in Appendix 2, we have the following machine language instructions:

Offset	Assembly Language	Opcode	R. Address	Machine Language
015A 015C	JL 016Eh	7C	016E - 015C = 12	7C 12
0170 0173	JMP 02FFh	E9	02FF - 0173 = 01 8C	E9 8C 01

<Offset> is referred to as a **Short-Offset** if the corresponding relative address can be represented as a byte.

Transfer of control instructions that also affect the contents of register CS in addition to affecting the contents of register IP are specified in assembly language as follows:

<mnemonic-opcode> ***<segment selector>*:*<offset>***

and the format of the corresponding machine language instruction is as follows:

opcode byte	Offset	Segment Selector

Using the information in Appendix 2, we have the following machine language instruction:

Assembly Language	Opcode	Machine Language
JMP FAR PTR 1A2Ch:15Ah	EA	EA 5A 01 2C 1A

Note that the low and the high bytes of the offset and the segment selector are swapped in the instruction.

Exercise 4.9

Provide the machine language instruction that corresponds to each of the following assembly language instructions:

	Offset	Assembly Language Instruction
a.	016B	JB 17Eh
	016D	
b.	021A	LOOP 200h
	021C	
c.	0230	JMP 3ABh
	0233	
d.		JMP 1B3Eh:4Ah

4.3 Storage Allocation

We have seen in section 4.1 that the operands of instructions are either in registers, in memory, or immediate data. In order to specify a memory location as operand of an instruction, we must first make a request to the assembler or the system program Debug to allocate this memory location. This section discusses how these requests are made.

Defining Memory Locations

In addition to the instructions that are translated into machine language, an assembly language program may also contain statements called **assembler directives** or **pseudo-operations (pseudo-ops)** that are used to provide additional information about the translation process to the system program Debug or the assembler, or to instruct Debug or the assembler to perform certain tasks during program translation. One such directive is the **define directive**: it instructs the assembler or Debug to allocate a memory location and to initialize it with one or more data constants. It takes one of the following five basic forms:

DB	defines one or more bytes of data.
DW	defines one or more words (2 bytes) of data.
DD	defines one or more doublewords (2 words or 4 bytes) of data.
DQ	defines one or more quad-words (4 words or 8 bytes) of data.
DT	defines one or more ten bytes (10 bytes) of data.

It has the following syntax:

Dx ***<list-of-values>***

where *list-of-values* is one or more values separated by commas, and Dx is either DB, DW, DD, DQ, or DT.

A value for the **DB** form is either a positive or negative integer value (specified either in decimal, hexadecimal, binary, or octal) or a sequence of zero or more characters enclosed in single or double quotes.

A value for the **DW**, **DD**, **DQ**, or **DT** form is either a positive or negative integer value (specified either in decimal, hexadecimal, binary, or octal), or a floating-point value.

Values are stored in memory in consecutive bytes starting at the offset at which the directive is specified. A character value is stored in ASCII code whereas an integer value is stored in two's complement binary integer using one byte for the **DB** form, two bytes for the **DW** form, four bytes for the **DD** form, eight bytes for the **DQ** form, and ten byte for the **DT** form.

A value for a basic form other than **DB** is stored in memory from the low byte to the high byte. Some examples are illustrated in Figure 4.5.

Figure 4.5 **Using the define directive**

For each of the following examples, assume that the directive is specified at offset 0200h.

Directives	Memory Locations					
1. DB 4	**Offset:**	0200				
	Contents:	04				
2. DB 'ABCD', 1Ah	**Offset:**	0200	0201	0202	0203	0204
	Contents:	41	42	43	44	1A
3. DW -2, 3	**Offset:**	0200	0201	0202	0203	
	Contents:	FE	FF	03	00	
4. DW 1A2Bh	**Offset:**	0200	0201			
	Contents:	2B	1A			

5. DW F5h, -2Ah	**Offset:**	0200	0201	0202	0203
	Contents:	F5	00	D6	FF
6. DW 'AB'	**Offset:**	0200	0201		
	Contents:	42	41		
7. DD 12345678h	**Offset:**	0200	0201	0202	0203
	Contents:	78	56	34	12

Notice in example 6 of Figure 4.5 that a value for the **DW** form may also be specified as two characters in single or double quotes. These characters are represented in ASCII code and stored in memory from the low byte to the high byte.

Exercise 4.10

Assuming that each of the following define directives is specified at offset 0200h, show the contents of the memory locations after the assembly of these directives.

Directives	**Memory Locations**	
1. DB 'CS-280'	**Offset:**	0200
	Contents:	
2. DB 4Fh, 'P1'	**Offset:**	0200
	Contents:	
3. DW 34Fh	**Offset:**	0200
	Contents:	

4. DW -2Fh, 5 **Offset:** 0200
   Contents:
5. DW 'C1' **Offset:** 0200
   Contents:
6. DD 2B3C4Dh **Offset:** 0200
   Contents:

If you only want to reserve a memory location for future use in a program, you may do so by specifying the question mark (**?**) as a value. The assembler will initialize the corresponding memory location with zeroes. We therefore have the following equivalent define directives:

DB 'KFC', ?, -12	is the same as	DB 'KFC', 0, -12
DW ?, 25	is the same as	DW 0, 25

However, Debug does not allow the specification of the question mark.

Two or more define directives specified one after another cause the assembler or Debug to allocate consecutive memory locations as illustrated in Figure 4.6. Also, with Dx being either DB, DW, DD, DQ, or DT,

Dx L1, L2, ... , Ln

has the same effect as:

Dx L1

Dx L2

. . .

Dx Ln

Figure 4.6 **Define directive and Consecutive Memory Locations**

Assume that the first define directive is specified at offset 0200, and that the second is specified immediately after the first, and the third is specified immediately after the second.

Directives	Memory Locations					
1. DB 4	**Offset:**	0200				
	Contents:	04				
2. DB 'ABCD', 1Ah	**Offset:**	0201	0202	0203	0204	0205
	Contents:	41	42	43	44	1A
3. DW -2, 3	**Offset:**	0206	0207	0208	0209	
	Contents:	FE	FF	03	00	

DUP Operator

The **DUP** operator is used to repeat the definition of a list of values. Its syntax is as follows:

Dx ***count*** **DUP** **(*list-of-values*)**

where Dx is either DB, DW, DD, DQ, or DT, and *count* is the number of repetition. For example, the following definitions are equivalent:

DB	4 DUP ('A')	is the same as	DB 'A', 'A', 'A', 'A'
DW	3 DUP (5, 2)	is the same as	DW 5, 2, 5, 2, 5, 2
DB	3 DUP (2,'C')	is the same as	DB 2, 'C', 2, 'C', 2, 'C'
DW	4 DUP (?)	is the same as	DW ?, ?, ?, ?

Exercise 4.11

Assuming that the first define directive is specified at offset 0200, and that each subsequent directive is specified after the previous one, show the contents of the memory locations after the processing of the following directives by the assembler or Debug.

Directives	Memory Locations
1. DB ‘CS-280'	**Offset:** 0200
	Contents:
2. DB 2 DUP(‘CD')	**Offset:**
	Contents:
3. DW 3 DUP(0)	**Offset:**
	Contents:
4. DB 4Fh, ‘P1’	**Offset:**
	Contents:
5. DW 34Fh	**Offset:**
	Contents:
6. DW -2Fh, 5	**Offset:**
	Contents:

Exercise 4.11

Assuming that the first define directive is specified at offset 0200, and that each subsequent directive is specified after the previous one, show the contents of the memory locations after the assembly of these directives.

Directives	Memory Locations
1. DB ‘CS-280'	**Offset:** 0200
	Contents:
2. DB 2 DUP(‘CD')	**Offset:**
	Contents:
3. DW 3 DUP(0)	**Offset:**
	Contents:
4. DB 4Fh, ‘P1’	**Offset:**
	Contents:
5. DW 34Fh	**Offset:**
	Contents:
6. DW -2Fh, 5	**Offset:**
	Contents:

Chapter 4: Exercises

1. What is the addressing mode of each operand of the following instructions:

   a. ADD BX, 25

   b. MOV CX, DX

   c. SUB CX, [10Ah]

   d. MOV WORD PTR [BX], 102Fh

   e. IMUL WORD PTR 5[DI]

   f. MOV AX, 10[BP]

   g. ADD [BX + SI], DX

   h. MOV 54[BX][DI], CX

2. Assuming the following register status:

   DS: 1000; SS: 2000; BX: 0A00; BP: 0B00; DI: 0050; SI: 0070

   i) Compute its offset of each of the following memory location operands.

   ii) Compute its absolute address.

   a. 0Fh[BX]

   b. 0Ah[BP]

   c. [BX][SI]

   d. 5[BP][SI]

   e. [BX][DI]- 2.

   f. [DI] - 26

3. For each of the following 'mod r/m' byte, indicate whether or not there is an operand in memory. If there is an operand in memory, also indicate how to compute the offset of that memory location (note: you do not have to compute the offset).

   a. C4 **c.** 3D; **e.** 57; **g.** 50; **i.** 94

   b. 96 **d.** 16; **f.** F8; **h.** 51; **j.** 86.

4. Using the information in Appendix 2, specify the machine language instruction that corresponds to each of the following assembly language instructions:

   a. INC CX;

   b. IMUL WORD PTR [1Ah];

   c. MOV AX, [1Ah];

   d. ETN 4;

   e. ADD BYTE PTR 8[BX][SI], 15

   f. ADD BX, [SI];

   g. SUB AL, 5;

   h. ADD BYTE PTR [BX][SI], 15

   i. IDIV WORD PTR [DI];

   j. CMP AX, CX;

   k. PUSH SS;

   l. INC BYTE PTR 5[BP][SI];

   m. MOV BX, [1Ah];

   n. IDIV BX;

   o. CMP BYTE PTR [1Ah], 15;

   p. SUB AX, 1A2Bh[BP];

   q. SUB BX, 5

   r. MOV WORD PTR [12Ah], 2C5h

   s. JMP FAR 1F25h:150h

t.

Offset	Instructions
0120	JZ 014Eh
0122	

u.

Offset	Instructions
015A	LOOP 148h
015C	

5. Assuming that each of the following define directives is specified at offset 0200, show the contents of the memory locations after they have been processed by the assembler.

a. DB ‘Dr. Mark’

b. DB ‘Math 234', 27

c. DW 136, -71

d. DW 8Ah, ‘CK’

e. DW 4 DUP (?)

f. DD 1A2B3C4Dh

g. DD 62B4h

h. DQ 1B35Eh

Chapter 4: Solutions of Exercises

1.

Instructions	Remarks
ADD BX, 25	First operand is a register; second operand is immediate.
MOV CX, DX	First and second operand are registers.
SUB CX, [10Ah]	First operand is a register; second operand is in memory location with offset in direct addressing mode.
MOV WORD PTR [BX], 102Fh	First operand is in memory location with offset in register indirect addressing mode and second operand is immediate.
IMUL WORD PTR 5[DI]	One memory location operand with offset in direct indexed addressing mode.
MOV AX, 10[BP]	First operand is a register; second operand is in memory location with offset in base relative addressing mode.
ADD [BX + SI], DX	First operand is in memory location with offset in base indexed addressing mode; and the second operand is a register.
MOV 54[BX][DI], CX	First operand is in memory location with offset in base indexed addressing mode; and the second operand is a register.

2.

Operand	Offset	Absolute Address
0Fh[BX]	000F + 0A00 = 0A0F	10000 + 0A0F = 10A0F
0AH[BP]	000A + 0B00 = 0B0A	20000 + 0B0A = 20B0A
[BX][SI]	0A00 + 0070 = 0A70	10000 + 0A70 = 10A70
5[BP][SI]	0005 + 0B00 + 0070 = 0B75	20000 + 0B75 = 20B75
-2[BX][DI]	FFFE + 0A00 + 0050 = 0A4E	10000 + 0A4E = 10A4E
-26[DI]	FFE6 + 0050 = 0036	10000 + 0036 = 10036

3.

'Mod r/m' Byte	Mod	Register	r/m	Remarks
C4	11	000	100	No operand in memory.
96	10	010	110	One operand in memory; offset = one word displacement + contents of register BP.
3D	00	111	101	One operand in memory; offset = contents of register DI.
16	00	010	110	One operand in memory with offset in direct mode.
57	01	010	111	One operand in memory; offset = one byte displacement + contents of register BX.
F8	11	111	000	No operand in memory.
50	01	010	000	One operand in memory; offset = one byte displacement + contents of register BX + contents of register SI.
51	01	010	001	One operand in memory; offset = one byte displacement + contents of register BX + contents of register DI.
94	10	010	100	One operand in memory; offset = one word displacement + contents of register SI.
86	10	000	110	One operand in memory; offset = one word displacement + contents of register BP.

4.

Assembly Language	Machine Language	Assembly Language	Machine Language
INC CX	41	SUB AX, 1A2Bh[BP]	2B862B1A
PUSH SS	16	IDIV WORD PTR [DI]	F73D
RET	C3	IMUL WORD PTR [1Ah]	F72E1A00
IDIV BX	F7FB	INC BYTE PTR 5[BP+SI]	FE4205
SUB AL, 5	2C05	ADD BYTE PTR 8[BX+SI], 15	8040080F

Assembly Language	Machine Language	Assembly Language	Machine Language
SUB BX, 5	83EB05	CMP BYTE PTR [1Ah], 15	803E1A000F
CMP AX, CX	39C8	ADD BYTE PTR [BX+SI], 15	80000F
MOV AX, [1Ah]	A11A00	MOV WORD PTR [12Ah], 2C5h	C7062A01C502
MOV BX, [1Ah]	8B1E1A00	JMP 1F25h: 150h	EA5001251F
ADD BX,[SI]	031C		
JZ 014Eh	742C	LOOP 148h	E2EC

5.

Memory Locations Offsets

	0200	**0201**	**0202**	**0203**	**0204**	**0205**	**0206**	**0207**	**0208**
DB 'Dr. Mark'	44	72	2E	20	4D	61	72	6B	
DB 'Math 234', 27	4D	61	74	68	20	32	33	34	1B
DW 136, -71	88	00	B9	FF					
DW 8Ah, 'CK'	8A	00	4B	43					
DW 4 DUP (?)	00	00	00	00	00	00	00	00	
DD 1A2B3C4Dh	4D	3C	2B	1A					
DD 62B4h	B4	62	00	00					
DQ 1B35Eh	5E	B3	01	00	00	00	00	00	