ASSEMBLY LANGUAGE MANUAL

CONTENTS

LIST OF FIGURES

LIST OF TABLES

GUIDE TO TECHNICAL DOCUMENTATION

This Manual is one of a series that documents the Convergent™ Family of Information Processing Systems. The series includes:

- Technical Summary
- Workstation Hardware Manual
- Peripherals Hardware Manual
- Central Processing Unit
- CTOS™ Operating System Manual
- Executive Manual
- Editor Manual
- BASIC Manual
- FORTRAN Manual
- COBOL Manual
- Pascal Manual
- Assembly Language Manual
- Debugger Manual
- Utilities Manual
- Data Base Management System Manual
- 3270 Emulator Manual
- System Programmer's Guide
- Operator's Guide

This section outlines the contents of these manuals.

The Technical Summary briefly describes the hardware and software of the Convergent Family of Information Processing Systems. It summarizes the other manuals in one volume. It can be helpful to read this overview before reading the other manuals.

The Workstation Hardware Manual describes the mainframe, keyboard, and video display. It specifies system architecture, printed circuit boards (motherboard, processor, I/O-memory, video control, ROM expansion, and RAM expansion), keyboard, video monitor, Multibus interface, communications interfaces, power supply, and environmental characteristics of the workstation.

The Peripherals Hardware Manual describes the disk subsystems. It specifies the disk controller motherboard, controller boards for the floppy disk and the Winchester disks, power supplies, disk drives, and environmental characteristics.

The Central Processing Unit describes the main processor, the 8086. It specifies the machine architecture, instruction set and programming at the symbolic instruction level.

The CTOS™ Operating System Manual describes the operating system. It specifies services for managing processes, messages, memory, exchanges, tasks, video, disk, keyboard, printer, timer, communications, and files. In particular, it specifies the standard file access methods.

The Executive Manual describes the command interpreter, the program that first interacts with the user when the system is turned on. It specifies commands for managing files and invoking other programs such as the Editor and the programming languages.

The Editor Manual describes the text editor.

The BASIC, FORTRAN, COBOL, Pascal, and Assembly Language Manuals describe the system's programming languages. Each manual specifies both the language itself and also operating instructions for that language. For Pascal, the manual is supplemented by a popular text, Pascal User Manual and Report.

The Debugger Manual describes the Debugger, which is designed for use at the symbolic instruction level. Together with appropriate interlistings, it can be used for debugging FORTRAN, Pascal, and assembly language programs. (BASIC and COBOL, in contrast, are more conveniently debugged using special facilities described in their respective manuals.)

The Utilities Manual describes miscellaneous programs such as the Linker, which links together separately compiled object files, and the Asynchronous Terminal Emulator.

The Data Base Management System Manual describes the data base management system. It specifies (1) the data definition language, which defines the logical structure of data bases and separately defines their physical organization, (2) the host language interfaces for accessing data bases from each of the system's programming languages, and (3) the utilities for creating, loading, unloading, and reorganizing data bases.

The 3270 Emulator Manual describes the 3270 emulator package.

The System Programmer's Guide addresses the needs of the system programmer or system manager for detailed information on operating system structure and system operation. It describes (1) diagnostics, (2) procedures for customizing the operating system, and (3) system utilities normally used only by a system programmer or manager, for example, Initialize Volume, Backup, and Restore.

The Operator's Guide addresses the needs of the average user for operating instructions. It describes the workstation switches and controls, keyboard function, and floppy disk handling.

1 INTRODUCTION

This Manual describes the Convergent assembler and assembly language. The Manual is directed towards readers who understand some assembly language reasonably well

To understand an assembler, it is usually helpful to first understand the machine architecture of the target CPU. If you are not already familiar with the machine-level architecture of the Convergent Information Processing System, you can find it useful to read the Central Processing Unit. That document also contains a brief discussion of assembly language programming at an elementary level, and it describes the instruction set in detail. So, if this Manual is too difficult, try reading the Central Processing Unit.

Since this Manual is primarily a reference work, we do not expect you to read it straight through. But if you are not entirely conversant with Convergent assembly language, you should initially read the first four sections.

Choice Among Convergent Languages

A programmer working with a Convergent Information Processing System has many different languages to choose among. The choice among languages involves several considerations.

- Does the program require the unique business features of COBOL or the scientific features of FORTRAN?

- Is an interpreted language (such as BASIC) suitable?

- Will the system programming and data structuring facilities of Convergent Pascal be particularly valuable in the program to be written?

- Should the program be divided into parts to be written in different languages and combined by the Linker?

If the program (or program part) requires direct access to processor registers and flags, then assembly language is the best choice. To the extent that memory utilization and object code efficiency are more important than development speed and programmer productivity, assembly language is a better tool than Pascal or FORTRAN.

It is rarely the case that an entire application system ought to be written in assembly language. The programmer should determine those parts in which direct access to machine features, efficiency, and memory utilization are overriding concerns, and implement those parts in assembly language, while writing the remainder of the application in an appropriate high-level language.

Features of the Assembly Language

The Convergent assembly language features a powerful instruction set, sophisticated code and data structuring mechanisms, strong typing (the ability to check that the use of data is consistent with its declaration), a conditional assembly facility, and a macro language with extensive string manipulation capabilities.

Design of the Instruction Set

A complete description of the instruction set is given in Appendix A and in the Central Processing Unit.

This assembly language differs from most other assembly languages, which usually have one instruction mnemonic for each operation code (opcode). In this assembly language, a particular instruction mnemonic can be assembled into any of several opcodes; the type of opcode depends on the type of operand.

This assembly language is a "strongly typed" language because mixed operand types are not permitted in the same operation (as, for example, moving a declared byte to a word register). You cannot inadvertently move a word to a byte destination, thereby overwriting an adjacent byte, nor can you move a byte to a word destination, thereby leaving meaningless data in an adjacent byte. However, if you need to override the typing mechanism, there is a special operation, called PTR, which allows you to do this. See Section 4.

The assembly language makes it possible to convey much information in a single, easy-to-code instruction. Consider this instruction:

```
SUB [BP][SI].field4, CH
```

The contents of the 8-bit register CH are subtracted from a memory operand; registers BP and SI are used to calculate the address of the memory operand; and the identifier field4 and the dot operator (.) are used to designate symbolically an offset within the structure pointed to by BP and SI.

The register BP points within the run-time stack and is used, as is the case in this example, when the operand is on the stack. (The segment register for the stack segment is SS, so the 16-bit contents of SS are automatically used together with BP in addressing the memory operand.)

The 16-bit contents of register SI are the offset of the data from the top of the stack. That is, the contents of BP and SI are added in the effective address calculation.

In this context, the dot operator (.) refers to a structure. (See Section 3 for a description of structure definitions.) The identifier that follows, field4, identifies a structure field. Its value gives the relative distance, in bytes, from the beginning of the structure to field4. (Offset values for each field of the structure relative to the beginning of the structure are generated by the assembler. In this way the structure can be used as a pattern of relative offset values, a "storage template.")

This instruction combines the contents of the stack segment register SS, the end of stack register BP, the index register SI, and the offset of field4, to form an absolute machine address. The contents of the 8-bit register CH are subtracted from the byte thus addressed. This instruction includes opcode, base register, index register, structure displacement and relative offset, type information, direction (register to memory), and source register. The instruction assembles into only three bytes.

Arrays

Arrays of bytes, words, doublewords, structures, and records (defined below) can be defined and initialized with, respectively, the DB, DW, DD, structure-name, and record-name directives, as shown here:

```
rgb       DB 50 DUP(66)      ;Allocate 50 bytes, named rgb,
                             ;initialize each to 66.

rgw       DW 100 DUP(0)      ;Allocate 100 words, named rgw,
                             ;initialize each to 0.

rgdd      DD 20 DUP(?)       ;Allocate 20 doublewords, named
                             ;rgdd, don't initialize them.
```

When you refer to array elements, be aware that the origin of an array is 0. This means that the first byte of the array rgb is rgb[0], not rgb[1]. Its <u>n</u>th byte is rgb[<u>n</u>-1]. Also, be aware that indexes are the number of <u>bytes</u> from the start of the array, regardless of whether the array elements are bytes, words, or doublewords.

Object Modules and Linking

An object module can contain any (or all) of the following: code, constants, variable data. The Linker (see the <u>Utilities Manual</u>) arranges the contents of a set of object modules into a memory image, typically with all code together, all constants together, and all variable data together. (This arrangement makes optimal use of the addressing structures of the 8086.) Although the Linker produces such arrangements automatically, the programmer will occasionally want to exercise explicit control. The concepts and facilities used to arrange memory are explained in Section 2.

Segments and Memory References

At assembly-time, you can define as many segments as you wish, as long as each assembly module has least one segment. (You can omit segment definition statements, in which case the default segment is assigned the name ??SEG by the assembler.) Each instruction of the program and each item of data must lie within a segment. Code and data may be mixed in the same segment, but this is generally not done because such a segment cannot be linked with object segments produced by Pascal or FORTRAN.

Here are examples of segments:

- global data segment,
- local data segment,
- stack segment, and
- main program segment (code).

A hardware segment in memory contains up to 64K bytes. It starts at an address divisible by 16, called a paragraph boundary. A paragraph number that is used to address the beginning of a hardware segment is a segment base address.

A segment defined by the programmer is a logical segment. It does not necessarily start at a paragraph boundary, so logical segments need not correspond to hardware segments.

The paragraph numbers at which segments begin are contained, at run-time, within the four 16-bit segment registers (CS, DS, ES, and SS). At any time, there are four "current" segments. CS always defines the current code segment. DS usually defines the current data segment. SS always defines the current stack segment. ES can define an auxiliary data segment.

The memory address calculations done by the processor have two components: a segment base address and an offset. The segment base address must be in one of the four segment registers (CS, DS ES, or SS).

When a program gets a data item from memory, the hardware combines the 16-bit offset and the 16-bit segment base address as follows:

    20-bit physical address = 16*(segment base address) + offset

For example, if a program is assembled at offset 2400h within the data segment, and if segment register DS is loaded with the value 3E00h, then the physical address of the data is:

    16*3E00h + 2400h = 40400h

The programmer is generally not concerned with this physical address.

Registers

The registers are:

- 16-bit segment (CS, DS, SS, ES),
- 16-bit general (AX, BX, CX, DX, SP, BP, SI, DI),
- 8-bit general (AH, AL, BH, BL, CH, CL, DH, DL),
- Base and index 16-bit (BX, BP, SI, DI), and
- 1-bit flag (AF, CF, DF, IF, OF, PF, SF, TF, ZF).

Segment registers contain segment base addresses and must be appropriately initialized at run-time. (If assembly language is used only to implement subroutines for a main program written in a high-level language, this initialization is automatic.)

Each of the 16-bit general, 8-bit general, and base and index registers can be used in arithmetic and logical operations. We frequently call AX "the accumulator," but the processor actually has eight 16-bit accumulators (AX, BX, CX, DX, SP, BP, SI, DI) and eight 8-bit accumulators (AH, AL, BH, BL, CH, CL, DH, DL). Each 8-bit accumulator is the high-order or low-order byte of AX, BX, CX, or DX

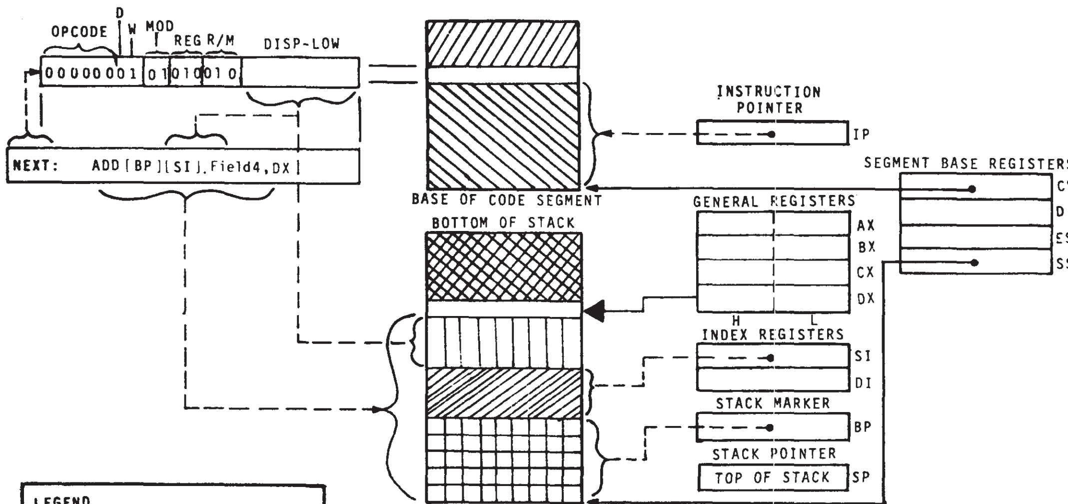
Addressing

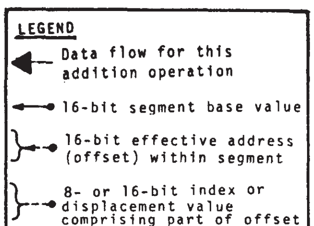
Operands can be addressed in several different ways with various combinations of base registers (BX and BP), index registers (SI and DI), displacement (adding an 8- or 16-bit value to a base or index register or to both), and direct offset (16-bit addresses used without the base or index register).

A two-operand instruction has a <u>source</u> operand, and a <u>destination</u> operand, as in:

```
MOV destination, source
```

The source operand can be an immediate value (a constant that is part of the instruction itself, such as the "7" in MOV CX, 7), a register, or a memory reference. If the <u>source</u> is an immediate value, then the <u>destination</u> operand can be either a register or a memory reference.





SAMPLE VALUE	MEANING	COMMENT
D=0	Memory destination	D=1 would mean register destination
W=1	Word operands	W=0 would be byte operands
MOD=01	Displacement 1 byte; sign-extend	*
REG=010	Use DX register	*
R/M=010	Effective address=(BP+(SI)+disp.	*

* For more encodings of MOD, REG and R/M, see the <u>Central Processing Unit</u>, page 156.

Figure 1-1. Analysis of a Sample Instruction.

Source and destination operands cannot both be memory references.

A memory reference is direct when a data item is addressed without the use of a register, as in:

```
MUL  prod, DX       ;prod is addressed toy 16-bit direct
                    ;offset.
MOV  CL, jones.bar  ;Offset of jones plus bar is 16-bit direct
                    ;offset.
```

A reference is indirect when a register is specified, as in:

```
MUL prod[BX], DX    ;Destination address is base register plus
                    ;16-bit displacement.

MOV CX, [BP][SI]    ;Source address is sum of base register
                    ;and index register.
```

See Figure 1-1 for an analysis of a sample instruction.

Procedures

The Convergent assembly language formalizes the concept of a callable procedure by providing explicit directives to identify the beginning and end of a procedure. Whereas other assembly languages start a procedure with a label and end it with a return instruction, the Convergent assembly language defines a procedure as a block of code and data delimited by PROC and ENDP statements. Thus the extent of a procedure is apparent. Here is an example:

```
WriteFile PROC
  .
  .
  .
  RET
  .
  .
  .
  RET
WriteFile ENDP
```

Procedures can be nested but must not overlap:

```
  WriteFile PROC
    .
    .
    .
    RET
    WriteLine PROC
      .
      .
      .
      RET
      .
      .
      .
    WriteLine ENDP
    .
    .
    .
    RET
  WriteFile ENDP
```

Macros

The macro capability of the assembler is used to define abbreviations for arbitrary text strings, including constants, expressions, operands, directives, sequences of instructions, comments, etc. These abbreviations can take parameters: they are string functions that are evaluated during assembly.

Fields of instruction can be parameters of macros. Macro calls can be nested. Macro definitions can be saved in a file. By including such a "macro library," the programmer can customize the assembler to include frequently used expressions, instruction sequences, and data definitions. The macro facility also provides interactive assembly by means of a macro-time console I/O facility.

Example

See Figure 1-2 for an example of a complete assembly program.

Invoking the Assembler from the Executive

Invoke the assembler with the Executive's assemble command. The following form appears:

```
  Assemble
    Source files                ____________________________________
    [Errors only?]              ____________________________________
    [GenOnly, NoGen, or Gen]    ____________________________________
    [Object file]               ____________________________________
    [List file]                 ____________________________________
    [Error file]                ____________________________________
    [List on pass 1?]           ____________________________________
```

```
Convergent Macro Assembler X1.2Factorial Subroutine                     16:00 18-Sep-80 Page    1

                     1    $TITLE(Factorial Subroutine)
                     2    FactSeg SEGMENT WORD PUBLIC
                     3            ASSUME  CS:FactSeg
                     4            PUBLIC  Factorial
                     5
                     6    ;The calling pattern is        Factorial(n, pFactorialRet): ErcType
                     7    ;       n is a word representing a positive'integer
                     8    ;       pFactorialRet is a long pointer (4 bytes) to a word where the product is to be stored
                     9    ;       ErcType is a word of error status returned in AX:
                    10    ;               0 if no error
                    11    ;               7777 if some error (e.g. overflow or invalid arg)
                    12
                    13    Factorial       PROC    FAR
     000A           14    rbn     EQU     10                             ;relative offset of n within frame
     0006           15    rbp     EQU     6                              ;relative offset of pFactorialRet within frame
0000 55             16            PUSH    BP                             ;save old frame pointer
0001 8BEC           17            MOV     BP, SP                         ;point to current stack top
0003 B80100         18            MOV     AX, 1                          ;initialize product
0006 8B4E0A         19            MOV     CX, [BP+rbn]                   ;CX gets n
0009 F7E1           20    Repeat: MUL     CX                             ;multiply by next factor
000B 700F           21            JO      Error                          ;error exit if overflow
000D E2FA           22            LOOP    Repeat                         ;decrement factor in CX and iterate
                    23    ;If control falls through the LOOP, then we're done.
000F C45E06         24            LES     BX, DWORD PTR[BP+rbp]          ;set up to store result
0012 268907         25            MOV     ES:[BX], AX                    ;store result
0015 B80000         26            MOV     AX, 0                          ;no error
0018 5D             27            POP     BP                             ;restore prior frame pointer
0019 CA0600         28            RET     6                              ;pop the 6 bytes of argument from the stack
001C B8611E         29    Error:  MOV     AX, 7777                       ;put error code into AX
001F 5D             30            POP     BP                             ;restore prior frame pointer
0020 CA0600         31            RET     6                              ;pop the 6 bytes of argument from the stack
                    32    Factorial       ENDP


                    33
                    34    FactSeg ENDS
                    35
                    36            END
                    37

There were no errors detected
```

Figure 1-2. Example of a complete Assembly program.

You need to know how to fill in a form. This is described in "Filling in a Form" in the Executive Manual.

Field Descriptions

Source files. Fill in the "Source files" field with a list of the names of the source files to be assembled. It is the only required field. If several files are specified, the result is logically like assembling the single file that is the concatenation of all the source files. (In a list of names of source files, separate each name by a space. Do not use commas.)

As an example, suppose the program is contained in Main.Asm and depends on a set of assembly-time parameters. You might maintain two source fragments to define the parameters, one for debugging, and one for production. Then "Source files" would be either:

```
ParamsDegbugging.Asm   Main.Asm
```

or:

```
ParamsProduction.Asm    Main.Asm
```

[Errors only?]. Fill in the "[Errors only?] field with "Yes" if you want a listing only of lines with errors. The listing normally contains source and object code for all source lines. Assembly produces an object file and a list file. The names of the object and list files are specified as described below. The default for "[Errors only?]" is "No", that is, a full listing.

[GenOnly, NoGen, or Gen]. Fill in the "[GenOnly, NoGen, or Gen]" field to specify how the results of macro expansion are listed. This setting can also be made in the source with the assembly control directives $GENONLY, $NOGEN, and $GEN. In GenOnly mode the results of macro expansion are listed. In NoGen mode, the listing contains the unexpanded macro invocations. In Gen mode, the listing contains invocations and full expansions, as well as intermediate stages of expansion. This last mode is most useful in debugging complex macros. Note that these controls affect only the content of the listing: the result of full expansions is always assembled to produce the object code. The default for "[GenOnly, NoGen, or Gen]" is GenOnly.

[Object file]. Fill in the "[Object file]" field to specify to which object file to write the object code that results from the assembly. The default is the last source file. That is, if you do not specify an object, a default object file is chosen as follows: treating the last source name as a character string, strip off any final suffix beginning with the character period (.), and add the characters ".Obj". The result is the name of the file. For example, if the last source file is:

[Dev]<Jones>Main

then the default object file is:

[Dev]<Jones>Main.Obj

If the last source file is:

Prog.Asm

then the default object file is:

Prog.Obj

[List File]. A listing of the assembly is written to the specified list file. The default is the last source file. That is, if no explicit listing file is specified, a file name is derived from the last source file. With the examples given above, the list files would be named, respectively:

[Dev]<Jones>Main.lst

and:

Prog.lst

[Error file]. Fill in the "[Error file]" field with the name of the file to receive the "errors only" listing if you wish to create both a full listing and a listing of just the errors. The default is to create no such listing.

[List on pass 1?]. Fill in the "[List on pass 1?]" field with "Yes" to diagnose certain errors in macros. Listings are normally generated only during the second assembly pass. However, some programming errors involving macros prevent the assembly process from ever reaching its second pass. To diagnose such errors, specify "[List on pass 1?]" as "Yes". Listings are then generated during both assembly passes. The default is "No".

2 PROGRAMS AND SEGMENTS

Segments

SEGMENT/ENDS Directives

Each of the instructions and variables of a program is within some segment. Segments can be named explicitly using the SEGMENT directive, but if no name is specified for a segment, the assembler assigns the name ??SEG. The SEGMENT directive also controls the alignment, combination, and contiguity of segments. Its format is:

```
[segname] SEGMENT [align-type] [combine-type] ['classname']
  .
  .
  .
[segname] ENDS
```

The optional fields must be in the order given. The segment is located on a memory boundary specified by [align-type], as follows:

1. PARA (the default)--the segment begins on a paragraph boundary, an address with the least significant hexadecimal digit of 0.

2. BYTE--the segment can begin anywhere.

3. WORD--the segment begins on a word boundary, i.e., an even address.

4. PAGE--the segment begins on an address divisible by 256.

Segments can be combined with other segments by the Linker as specified by [combine-type] . Segment combination permits segment elements from different assemblies to be overlaid or concatenated by the Linker. Such segment elements must have the same segname, classname, and an appropriate combine-type, as follows:

1. Not combinable (the default).

2. PUBLIC--when linked, this segment is concatenated (made adjacent) to others of the same name. The Linker controls the order of concatenation during linkage, according to your specifications.

3. AT expression--the segment is located at the 16-bit segment base address evaluated from the given expression. The expression argument is interpreted as a paragraph number. For example, if you wish the segment to begin at paragraph 3223 (absolute memory address 32230h), specify AT 3223h. You can use any valid expression that evaluates to a constant and

has no forward references. An absolute segment is permitted to establish a template for memory to be accessed at run-time; no assembly-time data or code is automatically loaded into an absolute segment.

4. STACK--the elements are overlaid such that the final bytes of each element are juxtaposed to yield a combined segment whose length is the sum of the lengths of the elements. Stack segments with the name STACK are a special case. When stack segments are combined, they are overlaid but their lengths are added together. When the Linker has combined all stack segments, it forces the total length of the aggregate stack segment to a multiple of 16 bytes. Compilers construct stack segments automatically. However, if your entire program is written in assembly language, you have to define an explicit stack segment. There are special rules regarding the use of the stack that must be observed for calls to standard object module procedures. See Section 9, "Accessing Standard Services from Assembly Code" below.

5. COMMON--the elements are overlaid such that the initial bytes of each element are juxtaposed to yield a combined segment whose length is the largest of the lengths of the elements.

The optional <u>classname</u> can be used to affect the ordering of segments in the memory image constructed by the Linker. See the <u>Utilities Manual</u> for details.

Segment Nesting

You can code a portion of one segment, start and end another, and then continue with the coding of the first. However, there is only lexical, not physical nesting, since the combination rules given above are always followed.

Lexically nested segments must end with an ENDS directive before the enclosing SEGMENT directive is closed with its ENDS directive.

The fundamental units of relocation and linkage are <u>segment elements</u>, <u>linker segments</u>, <u>class names</u>, and <u>groups</u>.

An object module is a sequence of <u>segment elements</u>. Each segment element has a segment name. An object module might consist of segment elements whose names are B, C, and D.

The Linker combines all segment elements with the same segment name from <u>all</u> object modules into a single entity called a <u>linker segment</u>. A linker segment forms a contiguous block of memory in the Fun-time memory image of the task. For example, you might use the Linker to link these two object modules:

Object Module 1
containing segment elements B, C, D

Object Module 2
containing segment elements C, D, E

Linkage produces these four linker segments:

Linker Segment B consisting of element B1
Linker Segment C consisting of elements C1, C2
Linker Segment D consisting of elements D1, D2
Linker Segment E consisting of element E2

(In each of these cases, xi denotes the segment element x in module i.)

The ordering of the various linker segments is determined by class names. (A class name is an arbitrary symbol used to designate a class.) All the linker segments with a common class name and segment name go together in memory. For example, if B1, D1, and E2 have class names Red, while C1 has class name Blue, then the ordering of linker segments in memory is:

B, D, E, C

If you look inside the linker segments, you see that the segment elements are arranged in this order:

B1, D1, D2, E2, C1, C2

(If two segment elements have different class names, then they are considered unrelated for purposes of these algorithms, even though they have the same segment name.)

As you see from this, segment names and class names together determine the ordering of segment elements in the final memory image.

The next step for the Linker is to establish how hardware segment registers address these segment elements at run-time.

A group is a named collection of linker segments that is addressed at run-time with a common hardware segment register. To make the addressing work, all the bytes within a group must be within 64K of each other.

Several linker segments can be combined into a group. For example, if B and C were combined into a group, then a single hardware segment register could be used to address segment elements B1, C1, and C2.

Segment, class, and group names can be assigned explicitly in assembler modules using appropriate assembler directives. Most compiled languages assign these names automatically. (See the individual language manuals for details.)

ASSUME Directive

The ASSUME directive declares how the instructions and data specified during assembly are to be addressed from the segment base registers during execution. The programmer must explicitly control the values in segment registers at run-time. Use of the ASSUME directive permits the assembler to verify that data and instructions will be addressable at run-time.

The ASSUME directive can be written either as:

ASSUME seg-reg:seg-name [, ...]

or:

ASSUME NOTHING

Here seg-reg is one of the segment registers.

Seg-name is one of these:

1. A segment name, as:

   ```
   ASSUME CS:codeSeg, DS:dataSeg
   ```

2. A GROUP name that has been defined earlier, as:

   ```
   ASSUME DS:DGroup, CS:CGroup
   ```

3. The expression SEG variable-name or SEG label-name, as:

   ```
   ASSUME CS:SEG Main, DS:SEG Table
   ```

4. The keyword NOTHING, as:

   ```
   ASSUME ES:NOTHING
   ```

A particular seg-reg:seg-name pair remains in force until another ASSUME assigns a different segment (or NOTHING) to the given seg-reg. To ASSUME NOTHING means to cancel any ASSUME in effect for the indicated registers. A reference to a variable whose segment is ASSUMEd automatically generates the proper object instruction; a reference to a variable whose segment is not ASSUMEd must have an explicit segment specification. (See the "Segment Override Prefix" below.)

Here is an example:

```
Tables SEGMENT
  xTab   DW 100 DUP(10)                 ;100-word array,
                                        ;initially 10’s.
  yTab   DW 500 DUP(20)                 ;500-word array
                                        ;initially 20's.
Tables ENDS

ZSeg SEGMENT
  zTab   DW 800 DUP(30)                 ;800-word array,
                                        ;initially 30’s.
ZSeg   ENDS

Sum    SEGMENT

  ASSUME CS:Sum,DS:Tables,ES:NOTHING    ;Sum addressable through
                                        ;CS and Tables through
                                        ;DS.  No assumption
                                        ;about ES.
  Start: MOV BX, xTab                   ;xTab addressable by DS:
                                        ;defined in Tables.
         ADD BX, yTab                   ;yTab addressable by DS:
                                        ;defined in Tables.
         MOV AX, SEG zTab               ;Now AX is the proper
                                        ;segment base address to
                                        ;address references to
                                        ;zTab.
         MOV ES, AX                     ;ES now holds the
                                        ;segment base address
                                        ;for ZSeg.
         MOV ES:zTab, 35                ;zTab must be addressed
                                        ;with explicit segment
                                        ;override--the
                                        ;assembler doesn’t know
                                        ;what segment register
                                        ;to use automatically.

Sum   ENDS
```

In this example, the ASSUME directive:

1. Tells the assembler to use CS to address the instructions in the segment Sum. (This fragment of program does not load CS. CS must previously have been set to point to the segment Sum. For example, CS is often initialized by a long jump or long call.)

2. Tells the assembler to look at DS for the symbolic references to xTab and yTab.

Loading Segment Registers

The CS register is loaded by a long jump (JMP), a long call (CALL), an interrupt (INT n or external interrupt), or by a hardware RESET.

The instruction INT n loads the instruction pointer (IP) with the 16-bit value stored at location 4*n of physical memory, and loads CS with the 16-bit value stored at physical memory address 4*n+2.

A hardware RESET loads CS with 0FFFFh and IP with 0.

Here is an example of defining the stack and loading the stack segment register, SS:

```
Stack       SEGMENT  STACK
    DW 1000 DUP(0)                           ;1000-words of
                                             ;stack.
StackStart LABEL WORD                        ;Stack expands
                                             ;toward low memory.
Stack ENDS

StackSetup SEGMENT
            ASSUME   CS:StackSetup
            MOV      BX, Stack
            MOV      SS, BX
            MOV      SP, OFFSET StackStart  ;start = end
                                             ;initially
StackSetup ENDS
```

This example illustrates an important point: each of the two register pairs SS/SP and CS/IP must be loaded together. The hardware has special provision to assist in this: loading a segment register by a POP or MOV instruction causes execution of the very next instruction to be protected against all interrupts. That is why the very next instruction, after the load of the stack base register, SS, must load the stack offset register, SP.

CS and its associated offset IP are loaded only by special instructions and never by normal data transfers. SS and its associated offset SP are loaded by normal data transfers but must be loaded in two successive instructions.

Segment Override Prefix

If there is no ASSUME directive for a reference to a named variable, then the appropriate segment reference can be inserted explicitly as a segment override prefix coding. This is the format:

seg-reg:

Here seg-reg is CS, DS, ES, or SS, as in:

```
DS:xyz
```

This construct does not require an ASSUME directive for the variable reference, but its scope is limited to the instruction in which it occurs.

Thus, the following two program fragments are correct and equivalent:

```
  Hohum SEGMENT
  ASSUME CS:Hohum, DS:Pond
    MOV AX, Frog
    ADD AL, Toad
    MOV Cicada, AX Hohum ENDS

  Hohum SEGMENT
    ASSUME CS:Hohum
    MOV AX, DS:Frog
    ADD AL, DS:Toad
    MOV DS:Cicada, AX
  Hohum ENDS
```

where Pond would be defined by:

```
  Pond SEGMENT
    Frog   DW     100 DUP (0)            ;100 words 0’s
    Toad   DB     500 DUP (0)            ;500 bytes 0's
    Cicada DW     800 DUP (0)            ;800 words 0's
  Pond ENDS
```

Anonymous References

Memory references that do not include a variable name are called anonymous references. These are examples:

```
  [BX]
  [BP]
```

Hardware defaults determine the segment registers for these anonymous references, unless there is an explicit segment prefix operator. These are the hardware defaults:

Addressing	Default
[BX]	DS
[BX][DI]	DS
[BX][SI]	DS
[BP]	SS
[BP][DI]	SS
[BP][SI]	SS
[DI]	DS
[SI]	DS

The exceptions to these defaults are:

1. PUSH, POP, CALL, RET, INT, and IRET always use SS and this default cannot be overridden.

2. String instructions on operands pointed to by DI always use ES and this default cannot be overridden.

Be particularly careful that an anonymous reference is to the correct segment: unless there is a segment prefix override, the hardware default is applied- For example;

```
ADD BX, [BP+5]      is the same as   ADD AX, SS:[BP+5]
MOV [BX+4], CX      is the same as   MOV DS:[BX+4], CX
SUB [BX+SI], CX     is the same as   SUB DS: [BX+SI], CX
AND [BP+DI], DX     is the same as   AND SS:[BP+DI], DX
MOV BX, [SI].one    is the same as   MOV BX, DS:[SI].one
AND [DI], CX        is the same as   AND DS:[DI], CX
```

The following examples require explicit overrides since they differ from the default usage:

```
MOV CS:[BX+2], AX
XOR SS:[BX+SI], CX
AND DS:[BP+DI], CX
MOV BX, CS:[DI].one
AND ES:[SI+4], DX
```

Memory Reference in String Instructions

The mnemonics of the string instructions are shown in Table 2-1. These include those that can be coded with operands (MOVS, etc.) and those that can be coded without operands (MOVSB, MOVSW, etc.).

Each string instruction has type-specific forms (e.g., LODSB, LODSW) and a generic form (e.g., LODS). The assembled machine instruction is always type-specific. If you code the generic form, you must provide arguments that serve only to declare the type and addressability of the arguments.

Table 2-1. String Instruction Mnemonics.

Mnemonic For Byte Operands	Mnemonic For Word Operands	Mnemonic For Symbolic Operands	Operands*
Move	MOVSB	MOVSW	MOVS
Compare	CMPSB	CMPSW	CMPS
Load AL/AX	LODSB	LODSW	LODS
Store from AL/AX	STOSB	STOSW	STOS
Compare to AL/AX	SCASB	SCASW	SCAS

*The assembler checks the addressability of symbolic operands. The opcode generated is determined by the type (BYTE or WORD) of the operands.

A string instruction must be preceded by a load of the offset of the source string into SI, and a load of the offset of the destination string into DI.

The string operation mnemonic may be preceded by a "repeat prefix" (REP, REPZ, REPE, REPNE, or REPNZ), as in REPZ SCASB. This specifies that the string operation is to be repeated the number of times contained in CX.

String operations without operands (MOVSB, MOVSW, etc.) use the hardware defaults, which are SI offset from DS, and DI offset from ES. Thus:

```
MOVSB
```

is equivalent to:

```
MOVS ES:BYTE PTR[DI],[SI]
```

If the hardware defaults are not used, both segment and type overriding are required for anonymous references, as:

```
MOVS ES:BYTE PTR[DI], SS:[SI]
```

See Section 4 below for a discussion of PTR.

String instructions can not use [BX] or [BP] addressing.

For details of string instructions and their use with a repeat prefix, see the Central Processing Unit, page 65. In particular, note that repeat and segment override should not be used together if interrupts are enabled.

GROUP Directive

The GROUP directive specifies that certain segments lie within the same 64K bytes of memory. Here is the format:

name GROUP segname [, ...]

Here name is a unique identifier used in referring to the group. segname can be the name field of a SEGMENT directive, an expression of the form SEG variable-name, or an expression of the form SEG label-name. (See "Value-Returning Operators" in Section 4 for a definition of the SEG operator.) [, ...] is an optional list of segnames. Each segname in the list is preceded by a comma.

This directive defines a group consisting of the specified segments. The group-name can be used much like a segname, except that a group-name must not appear in another GROUP statement as a segname.)

Here are three important uses of the GROUP directive:

1. Use it as an immediate value, loaded first into a general register, and then into a segment register, as in:

   ```
   MOV CX,DGroup
   MOV ES,CX
   ```

   The Linker computes the base value as the lowest segment in the group.

2. Use it an ASSUME statement, to indicate that the segment register addresses all segments of the group, as in:

   ```
   ASSUME CS:CGroup
   ```

3. Use it as an operand prefix, to specify the use of the group base value or offset (instead of the default segment base value or offset), as in

   ```
   MOV CX,OFFSET DGroup:xTab
   ```

(See "Value-Returning Operators" in Section 4 for additional information about OFFSET.)

It is not known during assembly whether all segments named in a GROUP directive will fit into 64K; the Linker checks and issues a message if they do not fit. Note that the GROUP directive is declarative only, not imperative: it asserts that segments fit in 64K, but does not alter segment ordering to make this happen. An example is:

```
DGroup GROUP dSeg, sSeg
```

An associated ASSUME directive that might be used with this group is:

```
ASSUME CS:code1, DS:DGroup, SS:DGroup
```

You can not use forward references to GROUPS.

A single segment register can be used to address all the segments in a group. This should be done carefully, however, because offsets in instructions and data are relative to the base of the group and not a particular segment.

Procedures

PROC/ENDP Directives

Procedures can be implemented using the PROC and ENDP directives. Although procedures can be executed by in-line "fall-through" of control, or jumped to, the standard and most useful method of invocation is the CALL.

Here is the format of the PROC/ENDP directives:

```
  name    PROC    [NEAR | FAR]
            .
            .
            .
          RET
            .
            .
            .
  name    ENDP
```

name is specified as type NEAR or FAR, and defaults to NEAR.

If the procedure is to be called by instructions assembled under the same ASSUME CS value, then the procedure should be NEAR. A RET (return) instruction in a NEAR procedure pops a single word of offset from the stack, returning to a location in the same segment.

If the procedure is to be called by instructions assembled under another ASSUME CS value, then the procedure should be FAR. A RET in a FAR procedure pops two words, new segment base as well as offset, and thus can return to a different segment.

Calling a Procedure

The CALL instruction assembles into one of two forms, depending on whether the destination procedure is NEAR or FAR.

When a NEAR procedure is called, the instruction pointer (IP, the address of the next sequential instruction) is pushed onto the stack, and control transfers to the first instruction in the procedure.

When a FAR procedure is called, first the content of the CS register is pushed onto the stack, then the IP is pushed onto the stack, and control transfers to the first instruction of the procedure.

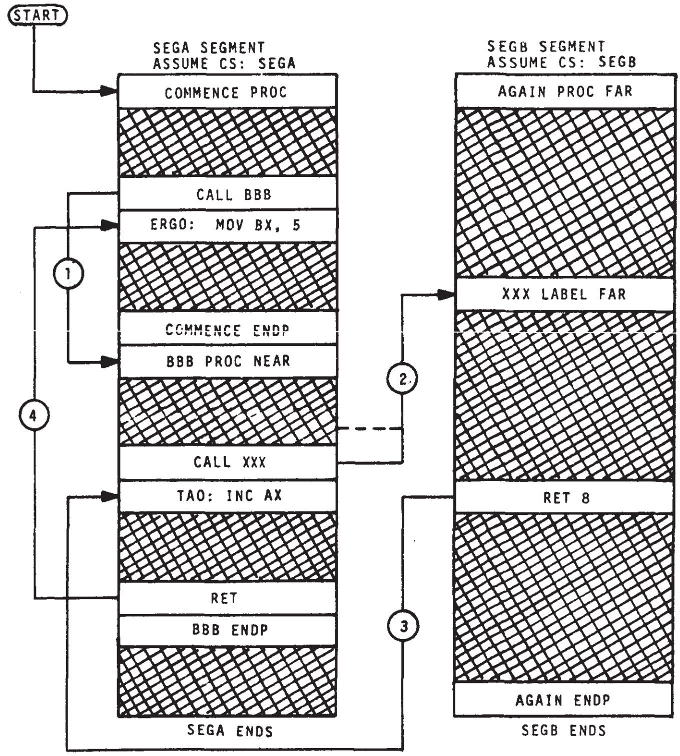
Multiple entry points to a procedure are permitted. All entry points to a procedure should be declared as NEAR or FAR, depending on whether the procedure is NEAR or FAR.

All returns from a procedure are assembled according to the procedure type (NEAR or FAR).

See Figure 2-1 for the procedure CALL/RET control flow.

Recursive Procedures and Procedure Nesting on the Stack

When procedures call other procedures, the rules are the same for declaration, calling, and returning.



KEY:

START	①	②	③	④
Comes from any of: o hardware reset o external interrupt o INT N o CALL BX o NEAR/FAR o JUMP/CALL Whatever the START, CS ←SEGA IP ←OFFSET COMMENCE	SP←SP-2 (SP)←IP IP←OFFSET BBB	SP← SP-2 (SP)← CS CS← SEGB SP← SP-2 (SP)← IP IP← OFFSET XXX	IP←(SP) SP←SP+2 CS←(SP) SP←SP+2 AND SP← SP+8 (For RET 8)	IP←(SP) SP←SP+2

Figure 2-1. CALL/RET Control Flow.

A recursive procedure is one which calls itself, or one which calls another procedure which then calls the first and so forth. Here are two points to note about recursive procedures.

1. A recursive procedure must be reentrant. This means that it must put local variables on the stack and refer to them with [BP] addressing modes

2. A recursive procedure must remove local variables from the stack before returning, by appropriate manipulation of SP.

The number of calls that can be nested (the "nesting limit") is delimited by the size of the stack segment. Two words on the stack are taken up by FAR calls, and one word by NEAR calls. Of course, parameters passed on the stack and any local variables stored on the stack take additional space.

Returning from a Procedure

The RET instruction returns from a procedure. It reloads IP from the stack if the procedure is NEAR; it reloads both IP and SP from the stack if the procedure is FAR. IRET is used to return from an interrupt handler and to restore flags.

A procedure can contain more than one RET or IRET instruction, and the instruction does not necessarily come last in the procedure.

Location Counter ($) and ORG Directive

The assembly-time counterpart of the instruction pointer is the location counter. The value contained in the location counter is symbolically represented by the dollar sign ($). The value is the offset from the current segment at which the next instruction or data item will be assembled. This value is initialized to 0 for each segment. If a segment is ended by an ENDS directive, and then reopened by a SEGMENT directive, then the location counter resumes the value it had at the ENDS.

The ORG directive is used to set the location counter to a nonnegative number. Here is the format:

```
ORG expression
```

The expression is evaluated modulo 65536 and must not contain any forward references. The expression can contain $ (the current value of the location counter), as in:

```
ORG OFFSET $+1000
```

which moves the location counter forward 1000 bytes.

An ORG directive may not have a label.

The use of the location counter and ORG are related to the use of the THIS directive, which is discussed in "Attribute Operators" in Section 4.

EVEN Directive

It is sometimes necessary to ensure that an item of code or data is aligned on a word boundary. For example, a disk sector buffer for use by the Operating System must be word aligned. The assembler implements the EVEN directive by inserting before the code or data, where necessary, a 1-byte NOP (no operation) instruction (90h). Here is an example:

```
        EVEN
Buffer  DW  256  DUP(0)
```

The EVEN directive can be used only in a segment whose alignment type, as specified in the SEGMENT directive, is WORD, PARA, or PAGE. It cannot be used in a segment whose alignment type is BYTE.

Program Linkage (NAME/END, PUBLIC, and EXTRN)

The Linker combines several different assembly modules into a single load module for execution. For more about the Linker, see the Utilities Manual.

Three program linkage directives can be used by the assembly module to identify symbolic references between modules. None of these three linkage directives can be labeled. They are:

- NAME, which assigns a name to the object module generated by the assembly. For example:

  ```
  NAME  SortRoutines
  ```

  If there is no explicit NAME directive, the module name is derived from the source file name. For example, the source file [Volname]<Dirname>Sort.Asm has the default module name Sort.

- PUBLIC, which specifies those symbols defined within the assembly module whose attributes are made available to other modules at linkage. For example:

  ```
  PUBLIC  SortExtended, Merge
  ```

  If a symbol is declared PUBLIC in a module, the module must contain a definition of the symbol.

- EXTRN, which specifies symbols that are defined as PUBLIC in other modules and referred to in the current module. Here is the format of the EXTRN directive:

```
EXTRN name.type [, ...]
```

In this format, name is the symbol defined PUBLIC elsewhere and type must be consistent with the declaration of name in its defining module. type is one of:

- BYTE, WORD, DWORD, structure name, or record name (for variables),
- NEAR or FAR (for labels or procedures), or
- ABS (for pure numbers; the implicit SIZE is WORD).

If you know the name of the segment in which an external symbol is declared as PUBLIC, place the corresponding EXTRN directive inside a set of SEGMENT/ENDS directives that use this segment name. You may then access the external symbol in the same way as if the uses were in the same module as the definition.

If you do not know the name of the segment in which an external symbol is declared as PUBLIC, place the corresponding EXTRN directive at the top of the module outside all SEGMENT/ENDS pairs. To address an external symbol declared in this way, you must do two things:

1. Use the SEG operator to load the 16-bit segment part into a segment register. (See "Value-Returning Operators" in Section 4 for a description of the SEG operator.) Here is an example:

```
MOV AX, SEG Var      ;Load segment base
MOV ES, AX           ;value into AX, and thence to ES.
```

2. Refer to the variable under control of a corresponding ASSUME (such as ASSUME ES:SEG var) or using a segment override prefix.

END Directive

The end of the source program is identified by the END directive. This terminates assembly and has the format:

```
END [expression]
```

The expression should be included only in your main program and must be NEAR or FAR and specifies the starting execution address of the program. Here is an example:

```
END Initialize
```

3 DATA DEFINITION

Introduction

The names of data items, segments, procedures, and so on, are called identifiers. An identifier is a combination of letters, digits, and the special characters question mark (?), at sign (@), and underscore (_). An identifier may not begin with a digit.

Three basic kinds of data items are accepted by the assembler.

1. Constants are names associated with pure numbers--values with no attributes. Here is an example

   ```
   Seven EQU 7   ;Seven represents the constant 7.
   ```

   While a value is defined for Seven, no location or intended use is indicated. This constant can be assembled as a byte (eight bits), a word (two bytes), or a doubleword (four bytes).

2. Variables are identifiers for data items, forming the operands of MOV, ADD, AND, MUL, and so on. Variables are defined as residing at a certain OFFSET within a specific SEGMENT. They are declared to reserve a fixed memory-cell TYPE, which is a byte, a word, a doubleword, or the number of bytes specified in a structure definition. Here is an example:

   ```
   Prune DW 8    ;Declare Prune a WORD of initial value 0008H.
   ```

3. Labels are identifiers for executable code, forming the operands of CALL, JMP, and the conditional jumps. They are defined as residing at a certain OFFSET within a specific SEGMENT. The label can be declared to have a DISTANCE attribute of NEAR if it is referred to only from within the segment in which it is defined. A label is usually introduced by writing:

   ```
   label:instruction
   ```

   which yields a NEAR label. See also PROC (under "Procedures" in Section 2) and LABEL under "Labels and the LABEL Directive" below, which can introduce NEAR or FAR labels.

Constants

There are five types of constants: binary, octal, decimal, hexadecimal, and string. Table 3-1 specifies their syntax.

Table 3-1. Constants.

Constant Type	Rules For Formation	Examples
Binary		
(Base 2)	Sequence of 0's and	10B
Octal	1’s plus letter B.	11001011B
(Base 8)		
Decima1	Sequence of digits	76540
(Base 10)	0 through 7 plus	7777Q
Hexadecimal	either letter 0 or	77777Q
(Base 16)	letter Q.	
STRING		
	Sequence of digits	9903
	0 through 9, plus	9903D
	optional letter D.	
	Sequence of digits	77h
	0 through 9 and/or	1Fh
	letters A through	0CEACh
	F plus letter h.	0DFh
	(If the first digit	
	is a letter, it must	
	be preceded by 0.)	
	Any character	'A', 'B'
	string within	'ABC
	single quotes.	'Rowrff'
	(More than two	'UP.URZ'
	characters only	
	with DB.)	

An instruction can contain 8- or 16-bit immediate values. Here is an example:

```
MOV CH, 53H         ;Byte immediate value
MOV CX, 3257H       ;Word immediate value
```

Constants can be values assigned to symbols with the EQU directive. These are examples:

```
Seven EQU 7         ;7 used wherever Seven referenced
MOV AH, Seven       ;Same as MOV AH,7.
```

See Section 4 for the complete definition of EQU. The format is:

```
symbol EQU expression
```

Here, <u>expression</u> can be any assembly language item or expression. An example is:

```
xyz EQU [BP+7]
```

Attributes of Data Items

The distinguishing characteristics of variables and labels are called attributes. These attributes influence the particular machine instructions generated by the assembler.

Attributes tell where the variable or label is defined. Because of the nature of the processor, it is necessary to know both in which SEGMENT a variable or label is defined, and the OFFSET within that segment of the variable or label.

Attributes also specify how the variable or label is used. The TYPE attribute declares the size, in bytes, of a variable. The DISTANCE attribute declares whether a label can be referred to under a different ASSUMEd CS than that of the definition.

Here is a summary of the attributes of data items.

- SEGMENT

  SEGMENT is the segment base address defining the variable or label. To ensure that variable and labels are addressable at run-time, the assembler correlates ASSUME CS, DS, ES, and SS (and segment prefix) information with variable and label references. The SEG operator (see "Value-Returning Operators" in Section 4) can be applied to a data item to compute the corresponding segment base address.

- OFFSET

  OFFSET is the 16-bit byte displacement of a variable or labels from the number of bytes from the base of the containing segment. Depending on the alignment and combine-type of the segment (see Section 2, on the SEGMENT directive), the run-time value here can be different from the assembly-time value. The OFFSET operator (see "Value-Returning Operators" in Section 4) can be used to compute this value.

- TYPE (for Data)

BYTE	1 byte
WORD	2 bytes
DWORD	4 bytes
RECORD	1 or 2 bytes (according to record definition)
STRUC	n bytes (according to structure definition)

- DISTANCE (for Code)

NEAR	Reference only in same segment as definition; definition with LABEL, PROC, or id:.
FAR	Reference in segment rather than definition; definition with LABEL or PROC.

Variable Definition (DB, DW, DD Directives)

To define variables and initialize memory or both, use the DB, DW, and DD directives. Memory is allocated and initialized by DD, DW, and DD in units of BYTES (8 bits), WORDS (2 bytes), and DWORDS (doublewords, 4 bytes), respectively. The attributes of the variable defined by DB, DW, or DD are as follows:

o The SEGMENT attribute is the segment containing the definition.

o The OFFSET attribute is the current offset within that segment.

o The TYPE is BYTE (1) for DB, WORD (2) for DW, and DWORD (4) for DD.

The general form for DB, DW and DD is either:

[variable-name] (DB | DW | DD) exp [, . . .]

or:

[variable-name] (DB | DW | DD) dup-count PUP (init [, ...]))

where variable-name is an identifier and either DB, DW, or DD must be chosen.

The DB, DW, and DD directives can be used in many ways. The possibilities are:

1 constant initialization,

2. indeterminate initialization (the reserved symbol "?"),

3. address initialization (DW and DD only),

4. string initialization,

5. enumerated initialization, and

6. DUP initialization.

Constant Initialization

One, two or four bytes are allocated. The expression is evaluated to a 17-bit constant using twos complement arithmetic. For bytes, the least significant byte of the result is used. For words, the two least significant bytes are used with the least significant byte the lower-addressed byte, and the most significant byte the higher-addressed byte. (As an example, 0AAFFh is stored with the 0FFh byte first and the 0AAh byte second. For double words, the same two bytes are used as for words, and they are followed by an additional two bytes of zeros. Here are some examples:

```
  number              DW 1F3Eh     ;3Eh at number, 1Fh at
                                   ;number + 1
                      DB 100       ;Unnamed byte
  inches_per_yard    DW 3*12       ;Assembler performs arithmetic
```

Indeterminate Initialization

To leave initialization of memory unspecified, use the reserved symbol "?".

Here are some examples:

```
  x          DW     ?              ;Define and allocate a word,
                                   ;contents indeterminate
  buffer     DB     1000 DUP(?)    ;1000 bytes.
```

(The DUP clause is explained in "Dup Initialization" below.)

Address Initialization (DW and DD Only)

[variable-name]   (DW | DD) init-addr

An address expression is computed with four bytes of precision--two bytes of segment base and two bytes of offset. All four bytes are used with DD (with the offset at the lower addresses), but only the offset is used with DW. Address expressions can be combined to form more complex expressions as follows:

- o A relocatable expression plus or minus an absolute expression is a relocatable expression with the same segment attribute.
- o A relocatable expression minus a relocatable expression is an absolute expression, but it is permitted only if both components have the same segment attribute.
- o Absolute expressions can be combined freely with each other.
- o All other combinations are forbidden.

Here are some examples of initializing using address expressions:

```
  pRequest        DD Request          ;32-bit offset and segment
                                      ;of Request
  pErc            DD Request+5        ;Offset of sixth byte in
                                      ;Request
  oRequest        DW Request          ;16-bit offset of Request
```

String Initialization

Variables can be initialized with constant strings as well as with constant numeric expressions. With DD and DW, strings of one or two characters are permitted. The arrangement in memory is tailored to the 8086 architecture this way: DW 'XY' allocates two bytes of memory containing, in ascending addresses, 'Y', 'X'. DD 'XY' allocates four bytes of memory containing in ascending addresses, 'Y', 'X', 0, 0.

With DB, strings of up to 255 characters are permitted. Characters, from left to right, are stored in ascending memory locations. For example, 'ABC' is stored as 41h, 42h, 43h.

Strings must be enclosed in single quotes ('). A single quote is included in a string as two consecutive single quotes. Here are some examples:

```
Single Quote    DB           'I''m so happy!'
Date            DB           '08/08/80'
Quote           DB           '"'
Jabberwocky     DB           '''TWAS BRILLIG AND THE
                                  SLITHY TOVES...'
Run Header      DW           'GW'
```

Enumerated Initialization

[<u>variable-name</u>]   (DB | DW | DD) <u>init</u> [, ...]

Bytes, words, or doublewords are initialized in consecutive memory locations by this directive. An unlimited number of items can be specified. Here are some examples:

```
Squares      DW     0,1,4,9,16,25,36
Digit_Codes  DB     30h,316, 32h,33h,34h,35h ,36h,37h,38h,39h
Message      DB     'HELLO, FRIEND.',0Ah
                    ;14-byte text plus new line code
```

DUP Initialization

To repeat <u>init</u> (or list of <u>init</u>) a specified number of times, use the DUP operator, in this format:

<u>dup-count</u> DUP (<u>init</u>)

The duplication count is expressed by <u>dup-count</u> (which must be a positive number). <u>init</u> can be a numeric expression, an address (if used with DW or DD), a question mark, a list of items, or a nested DUP expression.

Note that in the DB, DW, and DD directives, the name of the variable being defined is <u>not</u> followed by a colon. (This differs from many other assembly languages.) For example:

```
Name   DW  100     ;okay
Name:  DW  100     ;WRONG
```

Labels and the LABEL Directive

Labels identify locations within executable code to be used as operands of jump and call instructions. A NEAR label is declared by any of the following:

```
Start          LABEL                  ;NEAR is the default
Start          LABEL NEAR             ;NEAR can be explicit
Start:                                ;Followed by code
Start          EQU $
Start          EQU THIS NEAR
Start          PROC                   ;NEAR is the default
Start          PROC NEAR              ;NEAR can be explicit
```

A FAR label is declared by any of the following:

```
Start2  EQU THIS FAR
Start2  LABEL FAR
Start   PROC FAR
```

LABEL Directive

To create a name for data or instructions, use the LABEL directive, in the format:

<u>name</u> LABEL <u>type</u>

<u>name</u> is given segment, offset, and type attributes. The label is given a <u>segment</u> attribute specifying the current segment, an <u>offset</u> attribute specifying the offset within this segment, and a <u>type</u> as explicitly coded (NEAR, FAR, BYTE, WORD, DWORD, structure-name or record-name).

When the LABEL directive is followed by executable code, <u>type</u> is usually NEAR or FAR. The label is used for jumps or calls, but not MOVs or other instructions that manipulate data. NEAR and FAR labels cannot be indexed.

When the LABEL directive is followed by data, <u>type</u> is one of the other five classifications. An identifier declared using the LABEL directive can be indexed if assigned a data type, such as, BYTE, WORD, etc. The name is then valid in MOVs, ADDs, and so on, but not in direct jumps or calls. (See Section 4 for indirect jumps or calls.)

A LABEL directive using <u>structure-name</u> or <u>record-name</u> names data and is assigned a type attribute according to the record or structure definition.

The main uses of the LABEL directive, illustrated below, are: accessing variables by an "alternate type," defining FAR labels, and accessing code by an "alternate distance" (for example, defining a FAR label with the same segment and offset values as an existing NEAR label).

LABEL with Variables

The assembler uses the type of a variable in determining the instruction assembled for manipulating it. You can cause an instruction normally generated for a different type to be assembled by using LABEL to associate an alternative name and type with a location. For example, the same area of memory can be treated sometimes as a byte array and sometimes as a word array with the definitions:

```
rgw     LABEL     WORD
rgb     DB        200 DUP(0)
```

The data for this array can be referred to in two ways:

```
ADD AL, rgb[50]            ;Add fiftieth byte to AL
ADD AX, rgw[38]            ;Add twentieth word to AX
```

LABEL with Code

A label definition can be used to define a name of type NEAR and FAR. This is only permitted when a CS assumption is in effect; the CS assumption (<u>not</u> the segment being assembled) is used to determine the SEG and OFFSET for the defined name.

For example,

```
Place       LABEL  FAR
SamePlace   MUL CX,[BP]
```

introduces Place as a FAR label otherwise equivalent to the NEAR label SamePlace.

Label Addressability

The addressability of a label is determined by:

1. its declaration as NEAR or FAR, and

2. its use under the same or different ASSUME:CS directive as its declaration.

The four possibilities of code for each are shown in Table 3-2.

Table 3-2. Target Label Addressability.

	Near Label	Far Label
Same ASSUME CS:	NEAR Jump/Call	NEAR Jump FAR Call
Different ASSUME CS:	Not allowed	FAR Jump FAR Call

A NEAR jump or call is assembled with a 1- or 2-byte displacement using modulo 64K arithmetic. 64K bytes of the current segment can be addressed as NEAR.

A FAR jump or call is assembled with a 4-byte address. The address consists of a 16-bit offset and 16-bit segment base address. An entire megabyte of memory can be addressed as FAR.

(The semantics of PROC/ENDP directives are discussed in Section 2.)

Records

A _record_ is a format used to define bit-aligned subfields of bytes and words. The two steps in using records are:

1. define and name a record format, and

2. invoke the record name as an operator, thereby allocating and initializing memory.

Define a record by writing:

record-name RECORD _field-name_:_width_ [=_default_][, ...]

Neither record-name nor any of the field names can conflict with existing names. The sum of the _width_s of the fields can not exceed 16 bits. Each _width_ can be an expression, but must not make forward references.

The assembler divides records into two classes, those with a total width of up to 8 bits, and those with a total width of up to 16 bits. A byte is allocated for each instance of a record of the first class, and a word for each instance of a record of the second class. The data of each record instance is right-justified within the allocated memory.

The definition of a record can include a specification of how instances are to be initialized. This specification is given with the optional [=_default_] clause. For example, this definition:

```
HashEntry RECORD  state:2=3, sKey:4, rbKey:9
```

might be used in setting up a hash table. Each entry has a 2-bit state field, a 4-bit "size of key" sKey, and a 9-bit "relative byte of key in page" rbKey. The state field, being two bits wide, can hold four values. The state field is explicitly specified to default to 3. The other fields are assigned the implicit default value 0, since no explicit default is specified. A field eight bits wide can have a single character as its default value, as in bData:8='a'.

When a record is declared, the assembler associates with its field names these special values:

o the width of the field,

o the bit position of the right end of the field, and

o a mask constant for extracting the field from an instance of the record.

The width is computed with the WIDTH operator, the mask with the MASK operator, and the bit position with the field name itself. Thus, with HashEntry as above, the following holds.

```
state       =  0Dh  sKey       =    9h  rbKey       =   0h
MASK state  = E00h  MASK sKey  = 1E00h  MASK rbKey  = 1FFh
WIDTh state =   3h  WIDTh skey =    4h  WIDTh rbKey =   9h
```

As another example, let us define the format for the first two bytes of an instruction.

```
Inst2b RECORD Opcode 6, D:1, W:1, Mod:2, Reg : 3, Rm:3
```

The definition might be used in this way:

```
Inst_Table Inst2b  100 DUP(<,,,,,>)   ;Code to initialize
                                      ;Inst_Table
           MOV     AX, Inst_Table[BX] ;Load the entry at
                                      ;offset BX
           AND     AX, MASK Mod       ;Mask off all but Mod
           MOV     CL, Mod
           SHR     AX, CL             ;Now AX contains Mod
```

This example also shows how, for each record field, the bit position and MASK operator can be used to extract the field from a record.

The assembler right-justifies a record's user-defined fields when those fields do not occupy an entire word or byte. The fields are moved to the least-significant bit-positions of the byte or word defined by the record. For example, the definition:

```
Ascii_Twice  RECORD C1:7,C2:7
```

would result in the format:

```
15              14 13             7 6               0
|   (undefined)   |      (C1)      |      (C2)      |
    2 bits            7 bits           7 bits
```

Initializing Records

After records have been declared, the record name and operator can be used for allocation and initialization. There are two formats:

Format 1:

[name] record-name <[init][, ...]>

Format 2:

[name] record-name dup-count DUP (<[init] [, ...]>)

In both formats, the first byte or word (depending on the RECORD definition) of the allocated memory is optionally named. The record definition to be used is specified by record-name. Finally, the operand is a possibly empty list of initial field values. For example;

<>	Use field default values from the record definition.
<8,,10>	Set initial values of the first and third fields to 8 and 10, respectively, but use the default from the definition for the middle field.

The initial field values can be constants, constant expressions, or the indeterminate initialization "?". If the expression evaluates to a number not expressible in binary within the width of the corresponding record field, then the number is truncated on the left. For example, 11001 binary, in a 2-bit field, is truncated to 01.

With Format 2, multiple instances of the record can be allocated at once. The number of copies of the record to be allocated is given by dup-count. Note that in this format, the angle-brackets must be enclosed within parentheses as shown.

You can use a record as part or all of an expression, as in:

```
MOV AX, Inst2B<OP,D,W,MOD,REG,RM>
```

Structures

Just as records are used to format bit-aligned data at the byte or word level, structures are used to define byte-aligned fields within multibyte data structures.

Structures can be used to group together logically related data items.

For example, suppose you give the name Car to a structure. You use this structure to define individual fields of size (in bytes) 1, 2, 2, and 4 symbolically. The assembler generates the relative offsets:

```
Car     STRUC          ;No memory reserved--use this
                       ;as template for Ford below
Year    DB 0           ;Reference to .Year generates
                       ;relative offset of 0
Model   DW 0           ;Reference to .Model generates
                       ;relative offset of 1
Color   DW 0           ;Reference to .Color generates
                       ;relative offset of 3
License DB 'XXXX'      ;Reference to .License generates
                       ;relative offset of 5
Car     ENDS
```

The body of the structure definition is delimited by the STRUC and ENDS directives. The spacing of offsets within the structure is determined by the enclosed DB, DW, and DD directives.

You now allocate real memory and initialize using Car as an operator.

```
  Ford Car<63,'FL','GR','FOXY'>   ;allocate and initialize
```

Note that the programmer-assigned name Car is used here as an operator, and that the initialization of the structure is done with both integer data (63) and character data ('FL').

This use of Car as an operator is the assembly-time analog of this run-time initialization:

```
  FORD DB 8 DUP(?)                ;allocate 8 bytes
                                  ;(uninitialized)
  MOV Ford.Year,63                ;initialize Year field
  MOV Ford.Model,'FL'             ;initialize Model field
  MOV Ford.Color,'GR'             ;initialize Color field
  MOV Ford.License,'FOXY'         ;initialize License field
```

It is also possible, as described below, to specify default values during the definition of the structure, and to selectively override these defaults during memory allocation. All this can take place during assembly.

As another example, here is a structure that implements the request block for the Close File operator used with the CTOS Operating System:

```
  RqCloseFile   STRUC
    sCntInfo     DW  2
    nReqPbCb     DB  0
    nRespPbCb    DB  0
    userNum      DW  ?
    exchResp     DW  ?
    ercRet       DW  ?
    rqCode       DW  10
    fh           DW  ?
  RqCloseFile   ENDS

  rqCloseFile1  RqCloseFile<,,,1,3,,,>      ;Nondefault values
                                            ;are userNum 1,
                                            ;exchResp 3
    MOV  AX, fhNew
    MOV  rqCloseFile1.fh                    ;Fill in the fh
                                            ;field if an rq
    CMP  rqCloseFile1.ercRet, ercOk         ;Is the error return
                                            ;equal to the value
                                            ;ercOK?
```

Structures are not restricted to use with statically allocated data. For example

```
  CMP  [BP+rbRqCloseFile].rqCode,10   ;Examine rqCode in an
                                      ;anonymous instance of
                                      ;RqCloseFile that's on the
                                      ;stack
```

Here is the general format of the STRUC/ENDS statement-pair, together with the enclosed DB, DW, and DD directives:

```
  structure-name  STRUC
                    .
                    .
                    .
  [field-name] (DB | DW | DD) ( default [, ...]
                                dup-count DUP (default [, ... ] )
                    .
                    .
                    .
  structure-name  ENDS
```

In this case, DB, DW, and DD are used just as defined earlier, with the exception that there cannot be any forward references. Matching STRUC/ENDS pairs must have the matching _structure-names_. _Field-names_ are optional: if used, they must be unique identifiers.

Default Structure Fields

Default values for structure fields are as specified in the DB, DW, or DD directives. Because the STRUC/ENDS pair does not allocate memory, these default initializations have no immediate effect. The defaults are used to initialize memory later when the structure-name is used as a memory allocation operator as in the allocation of rqCloseFile1, above.

Overridable Structure Fields

When memory is allocated certain structure-field default values can be overridden by initial values specified in the allocation expression; these are called _simple fields_. Other field values that include a list or a DUP clause cannot be overridden. A DB character string is considered simple. Here are some examples of what can and cannot be overridden:

```
  Super STRUC
        DW  ?          ;Simple field: override okay
        DB  'Message'  ;Simple character string field: override
                       ;okay
        DD  5 DUP(?)   ;Multiple field: no override
        DB  ?,2,3      ;Multiple field: no override
  Super ENDS
```

Initializing Structures

After structures have been declared, they can be allocated and initialized with the structure-name as operator. The general format is similar to that for record initialization. (There are two formats.)

Format 1:

```
[name] structure-name <[init][, ...]>
```

Format 2 (with duplication):

```
[name] structure-name dup-count PUP (<[init] [, ...]>)
```

In both formats, the first byte or word (depending on the structure definition) of the allocated memory is optionally named. The structure definition to be used is specified by structure-name. Finally, the operand is a possibly empty list of initial field values. For example:

<>	Use field default values from the structure definition.
<8,,10>	Set initial values of the first and third fields to 8 and 10, respectively, but use the default from the definition for the middle field.

The initial field values can be constants, constant expressions, or the indeterminate initialization "?".

One-byte strings can override any field. Two-byte strings can override any DW or DD field. Multibyte strings can override a DB field, but only if the overriding string is no longer than the overridden string.

The number of copies of the structure to be allocated is dup-count; it must evaluate to a positive integer.

4 OPERANDS AND EXPRESSIONS

Operands

The instruction set of the 8086 makes it possible to refer to operands in a variety of ways. (The instruction set is described in the Central Processing Unit.) Either memory or a register can serve as the first operand (destination) in most two-operand instructions, while the second operand (source) can be memory a register, or a constant within the instruction. There are no memory-to-memory operations.

A 16-bit offset address can be used to directly address operands in memory. Base registers (BX or BP) or index registers (SI or DI) or both, plus an optional 8- or 16-bit displacement constant, can be used to indirectly address operands in memory.

Either memory or a register can receive the result of a two-operand operation. Any register or memory operand (but not a constant operand) can be used in single-operand operations. Either 8- or 16-bit operands can be specified for almost all operations.

Immediate Operands

An immediate value expression can be the source operand of two-operand instructions, except, for multiply, divide, and the string operations. Here are the formats:

[label:] mnemonic memory-reference, expression

and

[label:] mnemonic register expression

Here [label] is an optional identifier. mnemonic is any two-operand mnemonic (for example, MOV, ADD, and XOR). See "Memory Operands" below for the definition of memory-reference. In summary, it has a direct 16-bit offset address, and is indirect through BX or BP, SI or DI, or through BX or BP plus SI or DI, all with an optional 8- or 16-bit displacement. In the second format, register is any general-purpose (not segment) register. For a definition of expression, see the rest of this section. See Table 3-1 (Section 3) for rules on formation of constants.

The steps that the assembler follows in processing an instruction containing an immediate operand are;

- Determine if the destination is of type BYTE or WORD.

- Evaluate the expression with 17-bit arithmetic.

- If the destination operand can accommodate the result, encode the value of the expression, using twos complement arithmetic, as an 8- or 16-bit field (depending on the type, BYTE or WORD, of the destination operand) in the instruction being assembled.

In 8086 instruction formats, as in data words, the least significant byte of a word is at the lower memory address.

```
MOV  CH, 5                  ;8-bit immediate value to register
ADD  DX.3000H               ;16-bit immediate value to register
AND  Table[BX], 0FF00h      ;16-bit immediate value (where
                            ;Table is a WORD) through BX,
                            ;16-bit displacement
XOR  Table[BX+DI+100], 7    ;16-bit immediate through
                            ;BX+DI+(Table+100)
```

Register Operands

The 16-bit segment registers are CS, DS, SS, and ES. The 16-bit general registers are AX, BX, CX, DX, SP, BP, SI, and DI. The 8-bit general registers are AH, AL, BH, BL, CH, CL, DH, and DL. The 16-bit pointer and index registers are BX, BP, SI, and DI. The 1-bit flag registers are AF, CF, DF, IF, OF, PF, SF, TF, and ZF.

Segment base addresses are contained in segment registers and must be initialized by the programmer.

Arithmetic and logical operations can be performed using each of the general 8-bit, general 16-bit, and pointer and index 16-bit registers. So, even though AX is often called "the accumulator," there are actually eight separate 16-bit accumulators and eight 8-bit accumulators as listed above. Each of the 8-bit accumulators is either the high-order (H) or the low-order (L) byte of AX, BX, CX, or DX.

After each instruction, the flags are updated to reflect conditions detected in the processor or any accumulator. See Appendix A and the Central Processing Unit for the flags affected for each instruction.

These are the flag-register mnemonics:

```
AF:  Auxiliary Carry
CF:  Carry
DF:  Direction
IF:  Interrupt-enable
OF:  Overflow
PF:  Parity
SF:  Sign
TF:  Trap
ZF:  Zero
```

Explicit Register Operands

These are two-operand instructions that explicitly specify registers:

o Register to register

[label:] mnemonic reg, reg

Example.

```
ADD BX, DI    ;BX=BX+DI
```

o Immediate to register

[label:] mnemonic reg imm

Example:

```
ADD BX, 30H   ;BX=BX+30H
```

o Memory to register

[label:] mnemonic reg mem

Example:

```
ADD BX, Table[DI]     ;BX=BX+DI'th entry in Table
```

o Register to memory

[label:] mnemonic mem, reg

Example:

```
ADD Table[D1], BX      ;Increment DI'th entry in Table by BX
```

(Note that "i'th entry" means "entry at i'th byte.")

Implicit Register Operands

These instructions use registers implicitly:

Instruction	Implicit Uses
AAA, AAD, AAM, AAS	AL, AH
CBW, CWD	AL, AX or AX:DX
DAA, DAS	AL
IN, OUT	AL or AX
MUL, IMUL, DIV, IDIV	AL, AX or AX:DX
LAHF, SAHF	AH
LES	ES
LDS	DS
Shifts, Rotates	CL
String	CX, SI, DI
XLAT	AL, BX

The instructions with a single register operand have the form:

```
[label:] mnemonic reg
```

Example:

```
INC DI    ;DI=DI+1
```

Segment Registers

Segment registers are discussed in Section 2.

General Registers

When a 16-bit general register or pointer/index register is one of the operands of a two-operand instruction, the other operand must be immediate, a WORD reference to memory, or a WORD register.

When an 8-bit general register (AH, AL, BH, BL, CH, CL, DH, DL) is one of the operands of a two-operand instruction, the other operand must be an 8-bit immediate quantity, a BYTE reference to memory, or a BYTE register.

Flags

Instructions never specify the 1-bit flags as operands; flag instructions (as STC, CLC, CMC) manipulate all flags at once, and other instructions affect one or more flags implicitly (as INC, DEC, ADD, MUL, and DIV).

See Section 7 for flag operation and Appendix A for how each instruction affects the flags.

Memory Operands

Memory Operands to JMP and CALL

The JMP and CALL instructions take a simple operand. There are a number of different cases, determined by the operand. The control transfer can be' direct (with the operand specifying the target address) or indirect (with the operand specifying a word or doubleword containing the target address) . The transfer can be NEAR (in which case only IP changes) or PAR (both IP and CS change). Here are examples to illustrate the cases:

Operand to JMP/CALL	Direct/Indirect	NEAR/FAR	Target
NextIteration	Direct	NEAR[1]	NextIteration
FltMul	Direct	FAR[2]	FltMul
DX	Indirect	NEAR	CS:DX
LabelsNear[DI]	Indirect	NEAR[3]	Contained in word at LabelsNear[DI]
LabelsFar[DI]	Indirect	FAR[4]	Contained in dword at LabelsFar[DI]
DWORD PTR [BX]	Indirect	FAR FAR	Contained in dword at [BX]
WORD PTR [BX]	Indirect	NEAR	Contained in word at [BX]

[1]Assuming NextIteration is a NEAR label in the same segment or group as the jump or call.

[2]Assuming FltMul is a FAR label--a label to which control can be transferred from outside the segment containing the label.

[3]Assuming LabelsNear is an array of words.

[4]Assuming LabelsFar is an array of dwords .

CALL differs from JMP only in that a return address is pushed onto the stack. The return address is a word for a near call and a dword for a far call.

If the assembler determines that the target of a JMP or CALL is addressable by a 1-byte displacement from the instruction, it uses a special <u>short</u> jump or call instruction. Here are some examples:

```
Again: DEC  BX
       JNZ  Again     ;Short jump will be used.
       JMP  Last      ;Not short because Last is a forward
                      ;reference.
Last:  ...

       JMP  $+17      ;Short jump since displacement is in the
                      ;range -128 to 127.  BEWARE:  Variable
                      ;length instructions make it easy to get
                      ;this wrong; it's safer to use a label.
       JMP SHORT Last ;Forces assembly of a short transfer; it
                      ;will yield an error if the target is
                      ;not addressable with a 1-byte
                      ;displacement.
```

(NOTE: Do not confuse the concepts of PUBLIC and EXTRN with NEAR and FAR. PUBLICS and EXTRNs are used at assembly- and link-time only and are not run-time concepts. NEAR and FAR, in contrast, control the instructions to be executed at run-time. It is entirely possible for an EXTRN to be NEAR.)

Variables

This section covers the use of simple, indexed, and structured variables as operands. If you are unfamiliar with how to define and initialize variables, review Section 3.

Simple Variables. An unmodified identifier used the same way it is declared is a simple variable. Here is an example:

```
wData  DW  'AB'
        .
        .
        .
       MOV  BX, wData
```

Indexed Variables. A simple variable followed by a square-bracketed expression is an indexed variable. The expression in square brackets is a constant or constant expression, a base register (as BX or BP) or an index register (as SI or DI), a base or index register plus or minus a constant expression (in any order), or a base register plus an index register plus or minus a constant or constant expression (in any order) .

When you use indexed variables, be aware that the indexing is 0-origin (that is, the first byte is numbered 0), the index is always a number of bytes, and the type is the type of the simple variable to which the index is applied. For example, if the table Primes is defined by:

```
Primes  DW  250 DUP(?)
```

and register BX contains the value 12, then the instruction.

```
MOV Primes[BX], 17
```

sets the twelfth and thirteenth bytes of Primes (which are the bytes of the seventh word in Primes) to 17.

Double-Indexed Variables. Double-indexed variables use a sum of two displacements to address memory. Here is an example:

```
Primes[BX][SI+5]
```

Most forms of double indexing can be written with a more complex single index expression. For example, these two forms are completely equivalent:

```
Var[Disp1][Disp2]
```

and

```
Var[Disp1+Disp2]
```

The displacements can be constants or expressions that evaluate to constants, base or index registers (BX, BP, SI or DI) or base or index registers plus or minus a constant offset. The only restriction is that BX and BP can not both appear, and SI and DI cannot both appear in the same double-indexed variable.

These three expressions are all invalid.

```
Primes[BX+BP]
Primes[SI][2*BX]
Primes[BX][BP]
```

Indexing can be used in combination with structures. Recall the example given earlier

```
RqCloseFile   STRUC
  sCntInfo     DW  2
  nReqPbCb     DB  0
  nRespPbCb    DB  0
  userNum      DW  ?
  exchResp     DW  ?
  ercRet       DW  ?
  rqCode       DW  10
  fh           DW  7
RqCloseFile  ENDS
```

All of the following are valid:

```
MOV  RqCloseFile.sCntInfo, AX
MOV  [BX].userNum, AX
MOV  [BP][SI-4].fh
```

Attribute Operators

In addition to indexing, structure, arithmetic, and logical operators, operands can contain a class of operators called attribute operators. Attribute operators are used to override an operand's attributes, to compute the values of operand attributes, and to extract record fields.

PTR, the Type Overriding Operator

PTR is an infix operator. That is, it has two operands, and is written between them in this format:

type PTR addr-expr

type is BYTE, WORD, DWORD, NEAR, FAR, or structure-name. addr-expr is a variable, label, or number.

PTR sets or overrides the type of its operand without affecting the other attributes of the operand, such as SEGMENT and

OFFSET. Here are some examples of its use with data. Suppose rgb and rgw are declared by:

```
rgb  DB  100 DUP(?)
rgw  DW  100 DUP(?)
```

Then:

```
INC  rgb[SI]
INC  rgw[SI]
```

generate, respectively, byte-increment and word-increment instructions. Types can be overridden with:

```
INC WORD PTR rgb[SI]   ;word increment
INC BYTE PTR rgw[SI]   ;byte increment
```

Sometimes no variable is named in an instruction: the instruction uses an "anonymous" variable. In such cases the PTR operator must always be used. Thus:

```
INC  WORD PTR [BX]     ;word increment
INC  BYTE PTR [BX]     ;byte increment
INC  [BX]              ;INVALID because the operand [BX] is
                       ;"anonymous."
```

Segment Override

The segment override operator is discussed in Section 2. It is denoted by the colon, ":", and takes these three forms:

- <u>seg-reg</u>:<u>addr-expr</u>

- <u>segment-name</u> <u>addr-expr</u>

- <u>group-name</u>:<u>addr-expr</u>

The SEGMENT attribute of a label, variable, or address-expression is overridden by the segment override operator. The other attributes are unaffected. The first two forms do a direct override; the third recalculates the offset from the GROUP base.

SHORT

The single argument of the SHORT operator is an offset that can be addressed through the CS segment register. When the target code is within a 1-byte signed (twos complement) self-relative displacement, SHORT can be used in conditional jumps, jumps, and calls. This means that the target must lie within a range no more than 128 behind the beginning of the jump or call instruction, and no more than 127 bytes in front of it. (See "Memory Operands to JMP and CALL Operands" in this Section for more on SHORT.)

THIS

The single argument of the THIS operator is a type (BYTE, WORD, DWORD) or distance (NEAR, FAR) attribute. A data item with the specified type or attribute is defined at the current assembly location. Here are the formats:

```
THIS  type
THIS  distance
```

The segment and offset attributes of the defined data item are, respectively, the current segment and the current offset. The type or distance attributes are as specified. Thus the two statements:

```
byteA  LABEL  BYTE
byteA  EQU    THIS BYTE
```

have the same effect. Similarly, $ is equivalent to:

```
THIS NEAR
```

In the example:

```
E1  EQU     THIS FAR
E2: REPNZ   SCASW
```

the two addresses, E1 and E2, differ exactly in that E1 is FAR whereas E2 is NEAR.

Value-Returning Operators

Here are the value-returning operators:

- TYPE. It accepts one argument, either a variable or a label. TYPE returns, for variables, 1 for type BYTE, 2 for type WORD, 4 for type DWORD, and the number of bytes for a variable declared with a structure type. TYPE returns, for labels, either -1 or -2 (representing, respectively, NEAR or FAR) .

- LENGTH. It accepts one argument, a variable. It returns the number of units allocated for that variable. (The number returned is not necessarily bytes.) Here are examples:

  ```
  One DB 250(?)     ;LENGTH One=250
  Two DW 350(?)     ;LENGTH Two=350
  ```

- SIZE. It returns the total number of bytes allocated for a variable. SIZE is the product of LENGTH and TYPE.

- SEG. It computes the segment value of a variable or a label. Use it in ASSUME directives or to initialize segment registers, as described in Section 2.

- o OFFSET. It returns the offset of a variable or label. At time of linking, when the final alignment of the segment is frozen the value is resolved. If a segment is combined with pieces of the same segment defined in other assembly modules, or is not aligned on a paragraph boundary, the assembly-time offsets shown in the assembly listing can not be valid at run-time The offsets are properly calculated by the Linker if you use the OFFSET operator.

  The only attribute of a variable in many assembly languages is its offset. A reference to the variable's name is a reference also to its offset. Three attributes are defined by this assembly language for a variable, so to isolate the offset value, the OFFSET operator is needed. In a DW directive, however, the OFFSET operator is implicit.

  The variables in address expressions that appear in DW and DD directives have an implicit OFFSET.

  When used with the GROUP directive, the OFFSET operator does not yield the offset of a variable within the group. It returns rather the offset of the variable within its segment. Use the GROUP override operator to get the offset of the variable within the group. Here is an example:

```
DGroup  GROUP    Data,??SEG
data    SEGMENT
          .
          .
          .
xyz     DB       0
          .
          .
          .
        DW       xyz                  ;Offset within segment
        DW       DGroup:xyz           ;Offset within group
data     ENDS
        ASSUME   CS:??SEG,DS:DGroup
        MOV      CX,OFFSET xyz        ;Loads seg offset of xyz
        MOV      CX,OFFSET Dgroup:xyz ;Loads group offset of
                                      ;xyz
        LEA      CX, xyz              ;Also loads group offset
                                      ;of xyz
          .
          .
          .
```

You may not use forward references to group-names.

Record Operators

The use of operators with records is illustrated in Section 3. The definitions are repeated here for completeness. Associated with each field of a record are the following:

- Shift-count. This is the field-name of the record.

- MASK operator. This operator has one argument, which is a field-name. It returns a bit-mask that consists of 1's in the bit positions included by the field and 0's elsewhere.

- WIDTH operator. This operator returns the number of bits in a record or field.

If the definition of a record formats 8 bits, the record is of type BYTE, and if it formats 16 bits, of type WORD.

Operator Precedence in Expressions

The assembler evaluates expressions from left to right. It evaluates operators with higher precedence before other operators that come directly before or after. To override the normal order of precedence, use parentheses.

In order of decreasing precedence, here are the classes of operators:

1. Expressions within parentheses, expressions within angle brackets (records), expressions within square brackets, the structure "dot" operator, ".", and the LENGTH, SIZE, WIDTH, and MASK operators.

2. PTR, OFFSET, SEG, TYPE, THIS, and "name:" (segment override).

3. Multiplication and division: *, /, MOD, SHL, SHR.

4. Addition and subtraction: +, -.

5. Relational operators: EQ, NE, LT, LE, GT, GE.

6. Logical NOT.

7. Logical AND.

8. Logical OR and XOR.

9. SHORT.

EQU Directive

Use EQU to assign an assembly-time value to a symbol. This is the format:

<u>name</u> EQU <u>expression</u>

Here are examples to illustrate the cases:

```
y   EQU  z             ;y is made a synonym for z.
xx  EQU  [BX+DI-3]     ;xx is a synonym for an indexed reference
                       ;--note that the right side is evaluated
                       ;at use, not at definition.
x   EQU  EX:Bar[BP+2]  ;Segment overrides are also allowed.
xy  EQU  (TYPE y)*5    ;Random expressions are allowed.
RAX EQU  AX            ;Synonyms for registers are allowed.
```

PURGE Directive

Use the PURGE directive to delete the definition of a specified symbol. After a PURGE, the symbol can be redefined. The symbol's new definition is used by all occurrences of the symbol after the redefinition. You cannot purge register names, reserved words, or a symbol appearing in a PUBLIC directive.

5 FORWARD REFERENCES

The instruction set of the 8086 often provides several ways of achieving the same end. For example, if a jump is within 128 bytes of its target, the control transfer can be a SHORT jump (two bytes), a NEAR jump (three bytes), or a FAR jump (four bytes). If the assembler "knows" which case applies, it generates the optimal object code.

However, for the convenience of the programmer, the assembly language allows, in many cases, the use of a variable or label prior to its definition. When the assembler encounters such a forward reference, it must reserve space for the reference, although it does not yet know whether the label (for example) will turn out to be SHORT, NEAR, or FAR. The assembler makes a "guess," if it must, about the memory required, and proceeds on the basis of that guess.

The assembler makes two successive passes over the source program, and can always tell during the second pass whether a guess made during the first pass was correct. If a guess is too generous, the assembler can repair matters during the second pass by, for example, inserting an extra no-op instruction after an offending jump, and still produce valid output. If a guess is too conservative, however, no such remedy is available, and the assembler flags the forward reference as an error during the second pass.

The programmer can generally repair this kind of error by a small change to the source text and a reassembly. For example, the insertion of an attribute coercion such as "BYTE PTR" or "FAR PTR" is often a sufficient correction. However, the safest course is to follow programming practices that make it unnecessary for the assembler to guess. This can be done as follows:

- Put EQU directives early in programs.
- Put EXTRN directives early in programs.
- Within a multisegment source file, try to position the data segments (and hence the variable definitions) before the code segments.

6 INSTRUCTION FORMAT

The instruction format of the 8086 uses up to three fields to specify the location of an operand in a register or in memory. The assembler sets all three fields automatically when it generates code. These fields, when used, make up the second byte of an instruction, which is called the "MOD --- R/M" byte.

The two most significant bits of the "MOD --- R/M" byte are the MOD field, which specifies how to interpret the R/M field.

The next three bits are occupied by the REG field, which specifies an 8- or 16-bit register as an operand. Instead of specifying a register, the REG field can, in some instructions, refine the instruction code given in the first byte of an instruction.

The next three bits are occupied by the R/M field, which can specify either a particular register operand or the addressing MODe to select a memory operand. This occurs in combination with the MOD field.

The MOD and R/M fields determine the effective address (EA) of the memory operand and the interpretation of successive bytes of the instruction, as follows:

MOD	Interpretation
00	DISP = 0 (disp-low and disp-high are absent)
01	DISP = disp-low sign-extended to 16 bits (disp-high is absent)
10	DISP = disp-high, disp-low
11	There is no DISP (disp-low and disp-high are both absent) and R/M is interpreted as a register.

If MOD ≠ 11, then R/M is interpreted as follows:

R/M	interpretation
000	[BX]+[SI]+DISP
001	[BX]+[DI]+DISP
010	[BP]+[SI]+DISP
011	[BP]+[DI]+DISP
100	[SI]+DISP
101	[DI]+DISP
110	[BP]+DISP if MOD ≠ 0 DISP if MOD = 0
111	[BX]+DISP

If MOD = 11, then the effective address is a register designated by R/M. In word instructions, the interpretation is:

R/M	Register
000	AX
001	CX
010	DX
011	BX
100	SP
101	BP
110	SI
111	DI

In byte instructions (W = 0), the interpretation is:

R/M	Register
000	AL
001	CL
010	DL
011	BL
100	AH
101	CH
110	DH
111	BH

7 FLAGS

Flag Registers

Certain results of data manipulations are distinguished or denoted by flags. The flags that are affected by data manipulations are AF, CF, OF, PF, SF, and ZF.

The four basic mathematical operations (addition, subtraction, multiplication and division) are provided by the processor. 8- and 16-bit operations are available, as are signed and unsigned arithmetic. The representation of signed values is by standard twos complement arithmetic. The addition and subtraction operations serve as both signed and unsigned operations; the two possibilities are distinguished by the flag settings.

Arithmetic may be performed directly on unpacked decimal digits or on packed decimal representations.

Some operations indicate these results only by setting flags. For example, the processor implements "compare" as a special subtract which does not change either operand but does set flags to indicate a zero, positive, or negative result.

By using one of the conditional jump instructions, a program can test the setting of five of the flags (carry, sign, zero, overflow, and parity) . The flow of program execution can be altered based on the outcome of a previous operation. One more flag, the auxiliary carry flag, is used by the ASCII and decimal-adjust instructions.

It is important to understand which instructions set which flags. Suppose you wish to load a value into AX, and then test whether the value is 0. The MOV instruction does not set ZF, so the following does not work:

```
MOV  AX, wData
JZ   Zero
```

Instead, since ADD does set ZF, the following does work:

```
MOV  AX, wData
ADD  AX, 0
JZ   Zero
```

A flag can be set, but not tested, over the duration of several instructions. In such cases, the intervening instructions must be carefully checked to ascertain that they do not affect the flag in question. This is generally a dangerous programming practice.

(See Appendix A for the flags set by each instruction.)

Flag Usage

Most arithmetic operations set or clear six flag registers. "Set" means set to 1, and "clear" means clear to 0.

Auxiliary Carry Flag (AF)

If an operation results in a carry out of or a borrow into the low-order four bits of the result, AF is set; otherwise it is cleared. A program cannot test this flag directly: it is used solely by the decimal adjust instructions.

Carry Flag (CF)

If an operation results in a carry out of (from addition) or a borrow into (from subtraction), the high-order bit of the result, CF is set; otherwise it is cleared.

This flag usually indicates whether an addition causes a "carry" into the next higher order digit or a subtraction causes a "borrow." CF is not, however, affected by increment (INC) and decrement (DEC) instructions. CF is set by an addition that causes a carry out of the high-order bit of the destination, and cleared by an addition that does not cause a carry. CF is also affected by the logical AND, OR, and XOR instructions.

The contents of an operand are moved one or more positions to the left or right by the rotate and shift instructions. The carry flag is treated as if it were an extra bit of the operand. Only RCL and RCR preserve the original value in CF. The value does not, in these cases, remain in CF. The value is replaced with the next bit rotated out of the source. If an RCL is used, the value in CF is replaced by the high-order bit and goes into the low-order bit. If an RCR is used, the value in CF is replaced by the low-order bit and goes into the high-order bit. (This is useful in multiple-word arithmetic operations.) In other rotates and shifts, the value in CF is lost.

Overflow Flag (OF)

If a signed operation results in an overflow, OF is set; otherwise it is cleared. (That is, an operation results in a carry into the high-order bit of the result but not a carry out of the high-order bit, or vice versa.)

Parity Flag (PF)

If the modulo 2 sum of the low-order eight bits of an operation is 0 (even parity), PF is set; otherwise it is cleared (odd parity).

Following certain instructions, the number of one bits in the destination is counted and the parity flag set if the number is even and cleared if the number is odd.

Sign Flag (SF)

If the high-order bit of the result is set, SF is set; otherwise it is cleared.

Following an operation, the high-order bit of its target can be interpreted as a sign. The SF flag is set equal to this high-order bit by instructions that affect SF. Bit 7 is the high-order bit of a byte and bit 15 is the high-order bit of a word.

Zero Flag (ZF)

If the result of an operation is 0, ZF is set; otherwise it is cleared.

Following certain operations, if the destination is zero, the zero flag is set, and if the destination is not zero, the zero flag is cleared. Both ZF and CF are set by a result that has a carry and a zero. Here is an example:

```
  00110101
 +11001011
 ---------
  00000000       Carry Flag = 1
                 Zero Flag  = 1
```

8 MACRO ASSEMBLER

Introduction

The assembler supports the definition and invocation of macros: expressions, possibly taking parameters, that are evaluated during assembly to produce text. The text that results is then processed by the assembler as source code, just as if it had been literally present in the input to the assembler. For example, consider the program fragment;

```
%*DEFINE (Call2(subr,arg1,arg2))(
     PUSH  %arg1
     PUSH  %arg2
     CALL  %subr
)

%Call2 (Input,p1,p2)
```

This fragment defines a macro, Call2, of three arguments, and then invokes it. The invocation is to the expanded form:

```
PUSH  p1
PUSH  p2
CALL  Input
```

The character "%" is called the metacharacter and is used to activate all macro processing facilities: macro invocations are preceded by "%" and macro definitions by "%*". (The metacharacter can be changed; how to do this is described later in this Section.)

The simplest kind of macro definition takes the form:

%*DEFINE (MacroName ParameterList) (Body)

where MacroName is an identifier, ParameterList is a list of parameter names enclosed in parentheses, and Body is the text of the macro.

When parameter names appear in the Body, they are preceded by the "%" character. A simple macro invocation takes the form:

%MacroName (ArgList)

This expands to the corresponding macro Body with parameter names of the macro definition replaced by arguments from the macro invocation.

LOCAL Declaration

The purpose of macros is to permit the definition of a pattern--the body of the macro--that is to be recreated at each invocation of the macro. Thus two invocations of a macro normally expand to source text differing only insofar as the parameters of invocation differ. Consider however the definition:

```
  %*DEFINE  (CallNTimes(n,subr))(
              MOV   AX,%n
  Again:      DEC   AX
              JZ    Done
              PUSH  AX
              CALL  %subr
              POP   AX
              JMP   Again
  Done:)
```

An invocation such as %CallNTimes(5,FlashScreen) expands to;

```
             MOV   AX,5
  Again:     DEC   AX
             JZ    Done
             PUSH  AX
             CALL  FlashScreen
             POP   AX
             JMP   Again
  Done:
```

A second invocation of this macro produces an error because it doubly defines the labels Again and Done. The problem is that in this case we want a new, unique pair of labels created for each invocation. This can be done in a macro definition using the LOCAL declaration. The proper form is illustrated by:

```
  %*DEFINE(CallNTimes(n,subr)) LOCAL Again Done (
              MOV   AX,%n
  %Again:     DEC   AX
              JZ    %Done
              PUSH  AX
              CALL  %subr
              POP   AX
              JMP   %Again
  %Done:)
```

Conditional Assembly

In a manner carefully integrated with macro processing, the assembler also supports assembly-time expression evaluation and string manipulation facilities. These include the functions EVAL, LEN, EQS, GTS, LTS, NEX, GES, LES, and SUBSTR. Here are examples to illustrate the possibilities:

Function	Example	Evaluation of Example	Description
EVAL	%EVAL(3*(8/5))	3h	Evaluate expression
LEN	%LEN(First)	5h	Length of string
EQS	%EQS(AA,AA)	0FFFFh	String equality
GTS	%GTS(y,x)	0FFFFh	String greater
LTS	%LTS(y,x)	0h	String less
NES	%NES(AA,AB)	0FFFFh	String not equal
GES	%GES(y,y)	0FFFFh	String greater or equal
LES	%LES(z,y)	0h	String less or equal
SUBSTR	%SUBSTR(abcde,2,3)	bcd	Substring

Note that these functions evaluate to hexadecimal numbers, and that the relational functions (EQS, etc.) evaluate to 0FFFFh if the relation holds and 0h if it does not. The parameter to EVAL must evaluate to a number.

The result of a numeric computation done during macro processing can be given a symbolic name with the SET function, which is invoked in the form:

```
%SET (name, value)
```

For example:

```
%SET (xyz, 7+5)
```

sets the macro variable xyz to value 0Ch. Subsequent to the use of SET, %xyz is equivalent to 0Ch. Similarly, the invocation:

```
%SET (xyz, %xyz-1)
```

decrements the value of the macro variable xyz.

The macro facility also supports conditional and repetitive assembly with the control functions IF, REPEAT, and WHILE.

IF has two versions

%IF (<u>param1</u>) THEN (<u>param2</u>) ELSE (<u>param3</u>) FI

and

%IF (<u>param1</u>) THEN (<u>param2</u>) FI

The first parameter is treated as a truth value--odd numbers are true and even numbers false. If the first parameter is true, the IF expression is equivalent to the value of its second parameter; if the first parameter is false, the IF expression is equivalent to the value of its third parameter (or to the null string if the third parameter is omitted). For example:

```
%IF (1) THEN (aa) ELSE (bb) FI
```

is equivalent to aa, and:

```
%IF (2) THEN (aa) FI
```

is equivalent to the null string.

The IF function can be used in conjunction with macro variables to provide conditional assembly. Suppose a program contains a table that is to be searched for a value at run-time. If the table is small, a simple linear search is best; if the table is large, a binary search is preferable. Then you could code:

```
%IF (%sTable GT 10)
  THEN(
 ;binary search version here
)else(
 ;linear search here
)
```

The macro variable %sTable would have to be defined with some numeric value; otherwise the expansion of the IF would yield an error.

Sometimes it is convenient to control a conditional assembly by whether or not a symbol has been defined: in the usual case, the symbol is not defined and one alternative is selected, but if a definition for the symbol is found, a different alternative is selected. The macro processor supports this capability with the ISDEF function. ISDEF may use two forms: one tests whether a run-time symbol (for example, a label) has been defined, and the other tests whether a macro-time symbol has been defined. In both cases, the result is -1 if the symbol is defined, and 0 if the symbol is not defined. The two forms are, % ISDEF (symbol) to check a run-time symbol, and, %*ISDEF (%symbol), to check a macro-time symbol

Repetitive Assembly

REPEAT is used to assemble one of its parameters a specified number of times. The form is:

%REPEAT (<u>param1</u>) (<u>param2</u>)

For example:

```
%REPEAT (4)
(   DW  0
)
```

is equivalent to:

```
     DW  0
     DW  0
     DW  0
     DW  0
```

(Note that in this, and in most examples involving the macro facility, the parentheses are the delimiters of textual parameters, so their placement is critical.)

WHILE is used to assemble one of its parameters a variable number of times, depending on the result of an assembly-time computation to be performed before each repetition. The form is:

  %WHILE (<u>param1</u>) (<u>param2</u>)

For example, suppose %nWords has the value 3h. Then the result of:

```
  %WHILE (%nWords GT 0) (%REPEAT (%nWords)
  (   DW    %nWords
  )   %SET  (nWords, %nWords-1))
```

is:

```
     DW  3h
     DW  3h
     DW  3h
     DW  2h
     DW  2h
     DW  1h
```

When using the control functions REPEAT and WHILE it is sometimes desirable to explicitly terminate expansion. This can be done with EXIT, whose invocation stops the expansion of the enclosing REPEAT, WHILE, or macro. For example, if %n is initially 5, then the expression.

```
  %WHILE(%n GT 0)
     (%REPEAT (%n) (%IF (%n) THEN (%EXIT) FI DW %n
  )%SET (n, %n-1)
```

expands to:

```
     DW  4
     DW  4
     DW  4
     DW  4
     DW  2
     DW  2
```

Interactive Assembly (IN and OUT)

The macro capability supports interactive assembly, based on the two functions IN and OUT, which are used, respectively, to read input from the keyboard during assembly and to display information on the video display during assembly. When using IN and OUT, it is important to understand the two-pass nature of the assembler. Since the assembler makes two passes over the text, it expands all macros and macro-time functions twice. This works, but the programmer must take care:

1. that expressions involving macro-time variables generate the same code or data in both passes, and

2. that IN and OUT are not expanded twice.

The programmer may control these effects using the specially defined macro variables PASS1 and PASS2, whose values are:

	During First Pass	During Second Pass
PASS1	-1	0
PASS2	0	-1

Here is an example to illustrate these facilities. Suppose you want to prompt the user for a number at the beginning of an assembly, then use this (input) string later. Do this by inserting, near the beginning of the source, this code:

```
%IF (%PASS1 EQ -1)
   THEN (%OUT (Enter table size in bytes)
         %SET (sTable, %IN)) FI
```

The OUT and IN execute during the first pass only, and the user's input becomes the value of the macro variable sTable; this can later be referred to by %sTable.

Comments

You can write macro-time comments. The format is either:

```
%'text-not-containing-RETURN-or-apostrophe'
```

or

```
%'text-not-containing-RETURN-or-apostrophe RETURN
```

(Here RETURN designates the character generated by the Convergent RETURN key, code 0Ah.) Since the characters of the embedded text of a comment are consumed without any effect, comments may be used to insert extra returns for readability in macro definitions.

Match Operation

The special macro function MATCH is particularly useful for parsing strings during macro processing. It permits its parameters to be divided into two parts: a head and a tail. A simple form is:

```
%MATCH (var1, var2) (text)
```

For example, following the expansion of:

```
%MATCH (var1, var2) (a, b, c, d)
```

The macro variable var1 has the value "a" and var2 the value "b, c, d". This facility might be used together with LEN and WHILE. Consider the expression:

```
%WHILE (%LEN(%arg) GT 0)(%MATCH (head, arg)(%arg)
   DW %head
))
```

If %arg is initially the text 10, 20, 30, 40, then the expansion is:

```
DW 10
DW 20
DW 30
DW 40
```

Advanced Features

The form of MATCH just described, as well as the form of macro definition and call described above, are actually only special cases. In fact the separator between the parameters of MATCH or of a macro can be a user-specified separator other than comma. The remainder of this Section explains this and a number of related advanced features of the macro facility. Most programmers find the macro facilities described above quite sufficient for their needs; what follows can be deferred to a second reading.

The entities manipulated during macro processing are macro identifiers, macro delimiters, and macro parameters.

A <u>macro identifier</u> is any string of alphanumeric characters and underscores that begins with an alphabetic character.

A <u>macro delimiter</u> is a text string used as punctuation between macro parameters. There are three kinds of macro delimiters:

1. An <u>identifier delimiter</u> is the character "@" followed by an identifier.

2. An implicit blank delimiter is any text string made up of the "white space" characters space, RETURN, or TAB.

3. A literal delimiter is any other delimiter. Thus, all the preceding examples have used the comma as a literal delimiter.

A macro parameter is any text string in which parentheses are balanced. The following are valid parameters:

```
xyz
(xyz)
((xyz)()(()))
```

whereas the following are not:

```
(
(()
xy)(
```

That is, parentheses are considered balanced if the number of left and right parentheses is the same and, moreover, in reading from left to right there is no intermediate point at which more right than left parentheses have been encountered.

The most general form of macro definition is:

```
%*DEFINE (ident pattern) <locals> (body)
```

where:

1. the "*" is optional (see below for details),

2. ident is a macro identifier as defined above,

3. pattern and body are any balanced strings, and

4. <locals> is optional and, if present, consists of the reserved word LOCAL and a list of macro identifiers separated by spaces.

In all macro definitions illustrated above, the pattern has the form:

```
(id1, id2, ..., idn)
```

and all invocations are of the form:

```
%ident (param1, param2  ..., paramn)
```

Here are examples to illustrate other cases. The definition:

```
%*DEFINE (DWDW A @AND B)(DW %A
     DW %B)
```

requires an invocation such as;

```
%DWDW 1 AND 2
```

which expands to:

```
DW 1
DW 2
```

Here the delimiter preceding the formal parameter A and following the formal parameter B is an implicit space. The delimiter between the A and the B is the identifier delimiter @AND.

Bracket and Escape

The macro processor has two special functions, "bracket" and "escape," which are useful in defining invocation patterns and parameters. The bracket function has the form:

%(<u>text</u>)

where <u>text</u> is balanced. The text within the brackets is treated literally. Thus, given the definition:

```
%*DEFINE (F(A))(%(%F(2)))
```

the invocation:

```
%F(1)
```

expands to:

```
%F(2)
```

since the %F(2) is embedded within a bracket function and hence not treated as another macro call. Similarly, the definition:

```
%*DEFINE (DWDW A AND B)(DW %A
     DW %B)
```

declares three formal parameters A, AND, and B (with implicit blank delimiters), whereas the definition:

```
%*DEFINE (DWDW A %(AND) B)(DW %A
     DW %B)
```

treats the AND as a literal delimiter, so that the invocation:

```
%DWDW 1AND2
```

yields the expanded form:

```
DW 1
DW 2
```

The escape function is useful to bypass requirements for balanced text or to use special characters like "%" or "*" as regular characters.

The form is:

%<u>ntext</u>

where <u>n</u> is a digit, 0 to 9, and <u>text</u> is a string exactly <u>n</u> characters long. For example, you might define:

```
%*DEFINE (Concat(A,B))(%A%B)
```

and invoke this macro by:

```
%Concat (DW  ,%1(3+,4%1))
```

yielding the expansion:

```
DW (3+4)
```

MATCH Calling Patterns

Generalized calling patterns are applicable to MATCH just as they are to macro definition and invocation. The general form is:

%MATCH(<u>ident1</u> <u>macrodelimiter</u> <u>ident2</u>)(<u>balancedtext</u>)

For example, if "arg" is initially:

```
10 xyz 20 xyz 30
```

then:

```
%WHILE (%LEN(%arg) GT 0)(%MATCH(head @xyz arg)(%arg)
    DW  %head
)
```

expands to:

```
DW 10
DW 20
DW 30
```

Processing Macro Invocations

In processing macro invocations, the assembler expands inner invocations as they are encountered. Thus, in the invocation:

```
%F(%G(1))
```

the argument to be passed to F is the result of expanding %G(1). The expansion of inner invocations can be suppressed using the bracket and escape functions. Thus, with both of the invocations:

```
%F(%(%G(1)))
%F(%5%G(1))
```

it is the literal text %G(1), not the expansion of that text, that is the actual parameter of F.

Expanded and Unexpanded Modes

All macro processor functions can be evaluated in either of two modes, expanded and unexpanded. When the function, invocation, or definition is preceded by "%", the mode used is expanded; when preceded by "%*", the mode used is unexpanded. In either case, actual parameters are expanded and substituted for formal parameters within the body of invoked macros. In unexpanded mode, there is no further expansion. In expanded mode, macro processing specified in the body of a macro is also performed. For example, let the macros F and G be defined by:

```
%*DEFINE(F(X))(%G(%X))
%*DEFINE(G(Y))(%Y+%Y)
```

Then the invocation:

```
%*F(1)
```

expands to:

```
%G(1)
```

whereas the invocation:

```
%F(1)
```

expands to:

```
1+1
```

Nested Macro Expansion

When macro expansion is nested inner expansions are according to the mode they specify; on completion of inner expansions, processing continues in the mode of the outer expansion. An alternate way of saying this is that the parameters of user-defined macros are always processed in expanded mode. The bodies are processed in expanded mode when a "%" invocation is used, and in unexpanded mode when a "%*" invocation is used. It is also possible to invoke built-in functions in either expanded or unexpanded mode. For each built-in function, some arguments are classified as parameter-like and therefore processed in expanded mode, whereas others are classified as body-like and therefore processed in expanded mode only if the invocation is with "%".

The complete table follows:

```
DEFINE (p-arg) (b-arg)
EQS (p-arg)
EVAL (p-arg)
GES (p-arg)
GTS (p-arg)
IF (p-arg) THEN (b-arg) ELSE (b-arg)
ISDEF (b-arg)
LEN (b-arg)
LES (p-arg)
LTS (p-arg)
MATCH (p-arg) (b-arg)
METACHAR (p-arg)
NES (p-arg)
OUT (b-arg)
REPEAT (p-arg) (b-arg)
SUBSTR (b-arg, p-arg, p-arg)
WHILE (p-arg)(b-arg)
```

where *p-arg* denotes parameter-like arguments and *b-arg* denotes body-like arguments.

Assembly control directives, explained in section 10, begin with a "$" after a RETURN. If a control is encountered in expanded mode, it is obeyed; otherwise the control is simply treated as text.

A different character can be substituted for the built-in metacharacter "%" by calling the function METACHAR, in the form:

%METACHAR (*newmetacharacter*)

The metacharacter should not be a left or right parenthesis an asterisk, an alphanumeric character, or a "white space" character.

9 ACCESSING STANDARD SERVICES FROM ASSEMBLY CODE

You can access all system services from modules written in assembly language. To do so, you must follow certain standard calling conventions, register conventions, and segment/group conventions. If, in addition, you wish to use the system's virtual code management services, you must follow additional virtual code conventions.

Calling Conventions

Here we explain how CTOS™ Operating System services and standard object module procedures are invoked from programs written in assembly language. The following example of a call to the standard object module procedure ReadBsRecord is helpful in understanding this subject. The calling pattern of this procedure, described in detail in the CTOS™ Operating System Manual, is

```
ReadBsRecord (pBSWA, pBufferRet, sBufferMax
              psDataRet): ErcType
```

The Operating System and the standard object modules deal with quantities of many different sizes, ranging from single-byte quantities, such as Boolean flags, to multibyte quantities, such as request blocks and Byte Stream Working Areas. Three of these sizes are special: one byte, two bytes, and four bytes. Only quantities of these sizes are passed as parameters on the stack or returned as results in the registers. When it is necessary to pass a larger quantity as a parameter or to return a larger quantity as a result, a pointer to the larger quantity is used in place of the quantity itself. A pointer is always a 4-byte logical memory address consisting of an offset and segment base address.

For example, ReadBsRecord takes as parameters a pointer to a Byte Stream Work Area (pBSWA), a pointer to a buffer (pBufferRet), a maximum buffer size (sBufferMax), and a pointer to a word containing the size of some data (psDataRet). ReadBsRecord returns an error status of type ErcType. The pointers are all 4-byte quantities, the size is a 2-byte quantity, and the error status is a 2-byte quantity. Suppose that data is allocated by the declarations:

```
sBSWA    EQU     130
sBuffer  EQU     80

bswa     DB      sBSWA    DUP(?)
buffer   DB      sBuffer  DUP(?)
sData    DW      ?
```

Then to call ReadBsRecord, it is necessary first to push onto the stack, in order, a pointer to <u>bswa</u>, a pointer to <u>buffer</u>, the size of <u>buffer</u> (the constant <u>sBuffer</u>), and a pointer to <u>sData</u>. If DS contains the segment base address for the segment containing <u>bswa</u> <u>buffer</u> and <u>sData</u>, then this may be done by the code:

```
  PUSH  DS                ;Push the segment base address for bswa
  MOV   AX, OFFSET        ;Set BX to the offset of bswa
  PUSH  AX                ;Push the offset of bswa
  PUSH  DS                ;Ditto for the buffer
  MOV   AX, sBuffer       ;Get the buffer size into a register
  PUSH  AX                ;Push this word onto the stack
  PUSH  DS                ;Push the segment base address
  MOV   AX, OFFSET sData
  PUSH  AX                ;and then the offset of sData
  CALL  ReadBsRecord      ;Do the call
```

Note that pointers are arranged in memory with the low-order part, the offset, at the lower memory address, and the high-order part, the segment base, at the higher memory address. However, the processor architecture of the Convergent Information Processing System is such that stacks expand from high memory addresses toward low memory addresses; hence the high-order part of a pointer is pushed before the low-order part. Note also that the processor has no instruction that pushes an immediate constant: that is why the constant sBuffer must first be loaded into a register and that register pushed onto the stack. Finally, note that this sample code actually computes the various pointers at run-time. It would also be possible to have the pointers precomputed by adding to the program the declaration:

```
  pBSWA    DD      bswa
  pBuffer  DD      buffer
  psData   DD      sData
```

If this were done, then the appropriate calling sequence would be:

```
           LES     BX, pBSWA
           PUSH    ES
           PUSH    BX
           LES     BX, pBuffer
           PUSH    ES
           PUSH    BX
           MOV     AX, sBuffer
           PUSH    AX
           LES     BX, psData
           PUSH    ES
           PUSH    BX
           CALL    ReadBxRecord
```

Note that the LES instruction loads the offset part of the pointer into BX and the segment part into ES in a single instruction.

Object module and system common procedures as well as procedural references to system services must be declared EXTRN and FAR. These declarations may not be embedded in a SEGMENT/ENDS declaration. See line 6 of Figure 11-3.

The result returned by ReadBsRecord is a 2-byte quantity and according to the Convergent calling conventions, is returned in AX. If the result were a 4-byte quantity, the high-order part would be returned in ES and the low-order part in BX.

All of the 4-byte quantities dealt with in this example are pointers. There are many cases in which the Operating System and standard object module procedures deal with 4-byte quantities other than pointers, such as logical file addresses (lfa). It is important to understand that, as far as regards calling and register conventions and stack formats, such 4-byte quantities are dealt with exactly as 4-byte pointers, when they are parameters, the high-order part is pushed first and the low-order part second; when they are results, the high-order part is returned in ES and the low-order part is returned in BX.

There is one additional case, not illustrated by the example of ReadBsRecord. When a parameter is a single byte, such as a boolean flag, two bytes on the stack are actually required, although the high-order byte of these two bytes is not used. Thus the instruction:

```
  PUSH   BYTE PTR[BX]
```

adds two bytes to the stack. One of these bytes is specified by the operand of the PUSH instruction; the other is not set and <u>no</u> reference should be made to it. Similarly, when the result of a function is a single byte, that byte is returned in AL and no reference should be made to the contents of AH.

Register Usage Conventions

When writing in assembly language a call to a standard object module procedure or to the Operating System, be aware of the Convergent standard register conventions. The contents of CS, DS, SS, SP, and BP are preserved across calls: they are the same on the return as they were just prior to the pushing of the first argument. It is assumed that SS and SP point, respectively, to the base of the stack and the top of the stack, and this stack will, in general, be used by the called service. (Do <u>not</u> put temporary variables in the stack area below SS.SP; see "Interrupts and the Stack" below for details.) These conventions place no particular requirement on the contents of BP unless virtual code segment management services are being used. (See

"Virtual Code Segment Management and Assembly Code" below for details of BP usage with virtual code.) The other registers and the flags are not automatically preserved across calls to the Operating System or the standard object module procedures. Any other registers which must be saved in a particular application must be saved explicitly by the caller. Although there is not an absolute requirement that these register usage conventions be followed in parts of an application that do not call standard Convergent services, failing to follow them is not recommended in the Convergent programming environment.

Segment and Group Conventions

Main Program

A main program module written in assembly language must declare its stack segment and starting address in a special way. This is illustrated in the sample module of Figure 11-2. In particular:

- o The stack segment must have the combine type Stack. (See line 22.)
- o The starting address must be specified in the END statement. (See line 27.)

When the program is run, the Operating System performs the following steps:

- o It loads the program.
- o It initializes SS to the segment base address of the program's stack.
- o It initializes SP to the top of the stack.
- o It transfers control to the starting address with interrupts enabled.

SS and DS When Calling Object Module Procedures

If the program calls Convergent object module procedures, there are additional requirements. The program format used in Figure 11-2 does not suffice. A correct program is given Figure 11-3, illustrating the following points:

- o The stack segment must have segment name Stack, combine type Stack, and classname 'Stack'. See line 44.
- o Although not required, it is standard practice that user code be contiguous in memory with Convergent code and that code be at the front of the memory image. This is achieved if all

code segments have classname 'Code' and this class is mentioned before any other in the module. See lines 11-12.

- It is desirable to avoid forward references to constants. It is also standard, though not required, to make user constants contiguous with Convergent constants in the memory image and to locate constants directly after code. You can achieve both goals by giving all constant segments the classname 'Const' and by mentioning this classname before any other save 'Code'. See lines 17-22

- It is desirable to avoid forward references to data. It is also standard, though not required, to make user data contiguous with Convergent data in the memory image, and to locate data directly after constants. You can achieve both goals by giving all data segments the classname 'Data' and by mentioning this classname before any others save 'Code' and 'Const'. See lines 27-36. Note that EXTRN declarations for data declared in object module procedures must be embedded in the data SEGMENT/ENDS declarations.

- At any time that a call is made to an object module procedure, DS and SS must contain the segment base address of a special group named DGroup. This group contains the Data Const, and Stack segments, and is declared as illustrated in line 53. In addition, at the time of a call to an object module procedure, SP must address the top of a stack area to be used by the called procedure. A correct initialization of SS, SP and DS is illustrated in lines 62-68. These values need not be maintained constantly, but, if they are changed, they should be restored (using the appropriate top of stack value in SP if it has changed) for any call to an object module procedure. Note that the Operating System's interrupt handlers save the user registers by pushing them onto the stack defined by SS:SP. Therefore, some valid stack must be defined at all times that interrupts are enabled.

Interrupts and the Stack

If interrupts are enabled, interrupt routines use the stack as defined by SS and SP. Therefore you should <u>never</u>, even temporarily, put data in the stack segment at a memory address less than SS:SP.

Use of Macros

The instructions to set up parameters on the stack before a call and to examine the result on return have a number of cases, as discussed above. The instructions that must be executed differ slightly according to whether a parameter is in a register, a static variable, an immediate constant, a word, or a doubleword. If you are programming a particular assembly module in which not all of this variability occurs, it may be simplest to program the required calling sequences just once, to include them in your program as macro definitions, and to invoke them using the assembler's macro expansion capability.

For example, the procedural interface to the Write operation is given in the CTOS™ Operating System Manual as;

```
Write (fh, pBuffer, sBuffer, lfa, psDataRet): ErcType
```

where fh and sBuffer are 2-byte quantities and pBuffer, lfa, and psDataRet are 4-byte quantities. The corresponding external declaration and macro definition would be;

```
EXTRN    Write:  FAR
%*DEFINE(Write(fh pBuffer sBuffer lfa psDataRet))
                (PUSH  %fh
                 PUSH  WORD PTR %pBuffer[2]
                 PUSH  WORD PTR %pBuffer[0]
                 PUSH  %sBuffer
                 PUSH  WORD PTR %lfa[2]
                 PUSH  WORD PTR %lfa[0]
                 PUSH  WORD PTR %psDataRet[2]
                 PUSH  WORD PTR %psDataRet[0]
                 CALL  Write
)
```

Note that the 4-byte quantities are treated slightly differently from the 2-byte quantities, requiring first a PUSH of the high-order word, then a PUSH of the low-order word.

Here is an example of the use of this macro with "static" actual parameters:

```
fh1        DW    ?
           EVEN
buffer     DB    512 DUP(?)
sBuf       DW    SIZE buffer
pBuf       DD    buffer
lfa1       DD    ?
sDataRet   DW    ?
psDataRet  DD    sDataRet
           .
           .
           .
    ;code to initialize fh1, buffer, and lfa1
           .
           .
           .
       %Write(fh pBuffer sBuffer lfa psDataRet)
```

You might, instead, want to invoke this macro with actual parameters on the stack. Suppose that the quantities rbfh1, rbsBuf, rbpBuf, rblfa1, and rbpsData are on the stack and that the top of stack pointer is in register BX. Here is a sample invocation:

```
  rbfh1   EQU   -6
  rbsBuf  EQU   -8
  rbpBuf  EQU   -10
  rblfa1  EQU   -14
  rbpsDat EQU   -18
          %Write([BP+rbfh1] [BP+rbpBuf]
                 [BP+rbsBuf] [BP+rblfa1]
                 [BP+rbpsData]
```

Virtual Code Segment Management and Assembly Code

The virtual code segment management services of the Convergent Information Processing System permit the programmer to configure a program (written in any of the Convergent compiled languages, in assembly language, or in a mixture of these) into overlays. Although data cannot be overlaid with these services, code can be overlaid. Moreover, the run-time operations whereby code overlays are read into memory and discarded from memory are entirely automatic. The programmer need only specify, when linking the program, which modules are to be overlaid, and need make no change to the program apart from inserting at its start a single procedure call to initialize virtual code segment management services. (See the CTOS™ Operating System Manual for details.)

The correct automatic operation of the virtual code facility requires certain assumptions about stack formats and register usage in the run-time environment to be satisfied. These assumptions are automatically satisfied by the compiled languages of the Convergent System; however, the assembly language programmer must follow some simple rules if virtual code segment management is to be used. If a program contains no calls to overlaid modules from assembly language code or from procedures called from assembly language code, then the presence of assembly language code in the program has no affect on the operation of virtual code segment management services. In this case, there are no additional rules that the assembly language programmer must follow.

An overlay fault is defined as a call to or return to an overlaid module that is not in memory. An overlay fault automatically invokes virtual code segment management services to read the required overlay into memory and possibly to discard one or more other overlays from memory. The virtual code segment management services do this, in part, by examining the run-time stack. Therefore, if there are control paths in a program such that the stack may contain entries created by assembly language code when an overlay fault occurs, the assembly language programmer is subject to additional rules. These are the rules:

1. The register usage conventions discussed earlier must be followed. The intervention of virtual code segment management services preserves the registers SS, SP, DS, and BP, and, if an overlay fault occurs during the return from a function, preserves registers AX, BX, and ES where results may be returned. Other registers are not, in general, preserved, and therefore cannot be used to contain parameters or return results.

2. The stack segment must be named STACK and must be part of DGroup. (If a program is a mixture of assembly language code and compiled code, and all code shares the same stack, this happens automatically; if a main program is written in assembly language, it must be done explicitly. See the example of an assembly language main program for details.)

3. All procedures must be declared using the PROC and ENDP directives. Procedure bodies may not overlap. That is, the pattern:

```
Outer  PROC    FAR
       ;Code of Outer
Inner  PROC    FAR
       ;Code of Inner
Inner  ENDP
       ;More code of Outer
Outer  ENDP
```

   is not permitted and must be replaced by the pattern

```
Outer  PROC    FAR
       ;Code of Outer
       ;More code of Outer
Outer  ENDP
Inner  PROC    FAR
       ;Code of Inner
Inner  ENDP
```

   Note that this is only a restriction on syntactic nesting; there is no restriction on nested calls, and <u>Outer</u> can, in any case, contain calls to <u>Inner</u>.

4. If all of these conventions are followed, then when control enters an assembly language procedure, the most recent entry on the stack is the return address. In addition to preserving the value of BP, as discussed above, the procedure must push this value of BP onto the stack before it makes any nested call. No values may be pushed onto the stack between the return address and the pushed BP. This convention enables the virtual code segment management services to scan the stack during an overlay fault; its violation is not detected as an error but causes the overlaid program to fail

in unpredictable ways. Naturally, the pushed BP must be popped during the procedure's exit sequence.

5. All code must be in a class named CODE.

6. The SEG operator may not be used on an operand in class CODE nor in any segment that is part of an overlay. In particular, an instruction such as:

```
MOV    AX, SEG Procedure
```

is not permitted.

7. If a procedural value (that is, a value that points to a procedure) is to be constructed, this must be done in a class other than CODE by either:

```
pProc    DD  Procedure
```

or:

```
pProc    DW  Procedure
         DW  SEG Procedure
```

Such procedural values do not point directly at the procedure (since the procedure may be in an overlay), but at a special resident transfer vector created by the Linker. Such a procedural value may be invoked by the code:

```
CALL    DWORD  PTR pProc
```

8. If a procedure is known to be resident, and it is desired to address, not its entry in the resident transfer vector, but the procedure code directly, this may be done using, in place of SEG and OFFSET, the operators RSEG and ROFFSET. If RSEG or ROFFSET is applied to a value in an overlay, an error is detected during linking.

System Programming Notes

The rest of this Section describes some of the algorithms and data structures that make up the virtual code segment management facility. An understanding of these details is not needed by the user of the virtual code segment management facility--they are included for the information of the system programmer desiring a model of the internal workings of the virtual code segment management facility.

When you invoke the Linker, if you specify the use of overlays, then the Linker creates in the run file a special segment in the resident part of the program called the statics segment. This segment contains a transfer vector (an array of 5-byte entries called stubs with one stub for each public procedure in the program). A stub consists of one byte containing an operation code, either JUMP or CALL, and four bytes containing a long address. The Linker notes each call to a public procedure in an overlaid program and transforms it to an intersegment indirect call through the address part of the corresponding stub.

The contents of the address part of a stub for a procedure which is in memory (i.e., either resident or overlaid but currently swapped in) is the actual starting address of the procedure; thus the call of such a procedure is slower than it would be in a nonoverlaid program by only one memory reference.

The contents of the address part of a stub for a procedure not in memory is the address of a procedure in the virtual code segment management facility. Thus a call of such a procedure actually transfers to the virtual code segment management facility. Such a call of the virtual code segment management facility is a "call fault." When a call fault occurs, the virtual code segment management facility reads the needed overlay into the swap buffer. Before control is transferred to the called procedure, two other steps are taken.

1. The address in all stubs for procedures in the overlay is changed to the swapped-in address of the procedure.

2. If some overlays had to be deleted from the swap buffer to make room for the new overlay, the stubs for their procedures reset to the address of the procedure in the virtual code segment management facility that deals with call faults. (It is possible for an overlay to be deleted from memory even though control is nested within it--i.e., even though a return into it is pushed onto the stack. This situation is handled properly: all such stacked return addresses are modified to be the address of a procedure in the virtual code segment management facility that subsequently swaps the overlay back into memory when a "return fault" occurs.)

The user will observe that, in the preceding discussion, no use is made of the first byte of a stub the operation code. This byte is, in fact, only used for calls of procedural values. The virtual code segment management facility arranges that the operation code is a jump instruction for an overlay in memory; thus an invocation of a procedural argument for such a procedure results in a call to a jump instruction which then transfers control to the procedure. The virtual code segment management facility arranges that the operation code for an overlay not in memory is a call; since the address part of such a stub is the address of the virtual code segment management facility, the invocation of such a procedure results instead in the activation of the virtual code segment management facility.

10 ASSEMBLY CONTROL DIRECTIVES

The Convergent assembly language contains facilities to control the format of the assembly listing and to sequence the reading of "included" source files. These facilities are invoked by assembly control directives. Assembly control directives must occur on one or more separate lines within the source (i.e., not intermixed on the same line as other source code) . An assembly control line must begin with the character "$". Such a line may contain one or more controls, separated by spaces. Here is an example:

```
$TITLE(Parse Table Generator) PAGEWIDTH(132) EJECT
```

The meanings of the individual controls are described below.

EJECT

The control line containing EJECT begins a new page.

GEN

All macro calls and macro expansions, including intermediate levels of expansion, appear in the listing.

NOGEN

Only macro calls, not expansions, are listed. However, if an expansion contains an error, it is listed.

GENONLY

Only the final results of macro expansion, and not intermediate expansions or calls, are listed. This is the default mode.

INCLUDE (file)

Subsequent source lines are read from the specified file until the end of the file is reached. At the end of the included file, source input resumes in the original file just after the INCLUDE control line.

LIST

Subsequent source lines appear in the listing.

NOLIST

Subsequent source lines do not appear in the listing.

PAGELENGTH (n)

Pages of the listing are formatted n lines long.

PAGEWIDTH (n)

Lines of the listing are formatted a maximum of n characters wide.

PAGING

The listing is separated into numbered pages. This is the default.

NOPAGING

The listing is continuous, with no page breaks inserted.

SAVE

The setting of the LIST/NOLIST flag and the GEN/NOGEN/GENONLY flag is stacked, up to a maximum nesting of 8.

RESTORE

The last SAVEd flags are restored.

TITLE (text)

The text is printed as a heading on subsequent listing pages. The default title is the null string. The text must have balanced parentheses. (See Section 8 for details.)

Using a Printer with Assembly Listings

The listing produced by the assembler is paginated with titles and page numbers. Since the entire page image is formatted in such a listing, it should be printed by APPENDing or COPYing to [Lpt] rather than with the Executive's PRINT command. (The PRINT command can be used to print such a listing, but only by overriding many of its default values; these values were chosen to make the printing of text files created with the Editor most convenient.)

11 SAMPLE ASSEMBLER MODULES

This section contains three complete sample assembler modules. The first, Figure 11-1, is a source module of the assembler itself. It is the module that translates the assembler's internal error numbers into textual error messages.

The second module Figure 11-2, is a skeleton of a "standalone" assembler main program, and illustrates how the run-time stack is allocated in an assembler module. This example follows a bare minimum of the standard system conventions and does not link properly to standard object module procedures.

The third module, Figure 11-3, is an assembler main program compatible with Convergent conventions and linkable with standard object module procedures, as described above in Section 9, "Accessing Standard Services from Assembly Code."

```
;Error message module for the assembler.  Suitable for loading into an overlay in orcer to save space in the resident.

PUBLIC PAscizFromErc
;pAsciz = PAscizFromErc(erc, ofUpArrow)
;
;Given an error code in DS:[BP+8] (1st arg).
;
;Returns ES:BX = pointer to 0 terminated ascii string.
;       Stores flag indicating whether uparrow is to accompany error message in location pointed at by DS:[BP+6] (2nd arg.)

;Define the segments we are going to use. Do this here to get them in the desired physical order
;The storage layout consists of the procedure code followed by a packed group of ascii strings, followed by two parallel arrays

AsmErr SEGMENT WORD PUBLIC 'CODE'       ;Segment for the code of PAscizFromErc
AsmErr ENDS

AsmEr1 SEGMENT WORD PUBLIC 'ERRORS'     ;Segment for the ascii text of messages
AsmEr1 ENDS

AsmEr2 SEGMENT WORD PUBLIC 'ERRORS'     ;Segment for offsets to text, indexed by erc
rgRaRgch LABEL WORD
AsmEr2 ENDS

AsmEr3 SEGMENT WORD PUBLIC 'ERRORS'     ;Segment for array of fUparrow flags, indexed by erc
rgfUpArrow LABEL BYTE
AsmEr3 ENDS

;Address everything in this module thru CS: (which points to the base of ErrGroup)
ErrGroup GROUP AsmErr, AsmEr1, AsmEr2, AsmEr3

AsmErr SEGMENT
ASSUME CS:ErrGroup                      ;Tell the assembler what to expect in CS

PAscizFromErc PROC FAR                  ;Procedure entry point
        PUSH    BP
        MOV     BP,SP                   ;Save callers BP, set up local frame pointer

        MOV     BX,[BP+8]               ;BX = erc
        CMP     BX,ercMax               ;Compare against maximum error #
        JB      Ok
        MOV     BX,ercMax-1             ;Too big: use "Internal error" message
Ok:
        MOV     AL,rgfUpArrow[BX]       ;Fetch uparrow flag for this erc
        MOV     DI,[BP+6]               ;Fetch Callers DS relative pointer to where he wanted it stored
        MOV     [DI],AL                 ;Store it

        SHL     BX,1                    ;BX = erc*2 so as to index array of words
        MOV     BX,rgRaRgch[BX]         ;Fetch CS relative offset to error message text
        MOV     AX,CS
        MOV     ES,AX                   ;Return segment of text in ES

        POP     BP                      ;Restore callers BP
        RET     4H                      ;Dump args from stack and return
PAscizFromErc ENDP
```

Figure 11-1. Error Message Module Program. (Page 1 of 3.)

```
AsmErr ENDS

AsmEr1 SEGMENT

;This macro generates the text and the 2 parallel arrays

%*Define(Err(fUpArrow, erc, rgch))
(%IF (%erc GT ercMax) THEN (ercMax       EQU     %erc) FI
orgch   EQU $                             %'Remember where we started the string'
        DB      '%rgch',0                 %'The zero terminated ascii string'
AsmEr2  SEGMENT
        ORG     %erc*2
        DW      ErrGroup:orgch            %'The ErrGroup relative offset (i.e. CS rel.) of ascii string'
AsmEr2  ENDS
AsmEr3  SEGMENT
        ORG     %erc
        DB      %fUpArrow                 %'The uparrow flag'
AsmEr3  ENDS
)

;Do the work

ercMax  EQU     0                         ;Initialize max. defined error code

%Err(1,00,Invalid numeric constant)
%Err(1,01,Syntax error)
%Err(0,02,Expression too complex)
%Err(0,03,Internal error #1)
%Err(0,04,Invalid arithmetic operation for relocatable or external expression)
%Err(1,05,Invalid use of register in expression)
%Err(0,06,Invalid use of PTR, must operate upon address expression)
%Err(1,07,Undefined symbol)
%Err(0,08,Forward reference to EQU''ed register not permitted)
%Err(0,09,SIZE and LENGTH must operate upon data symbol)
%Err(1,10,Invalid argument to ASSUME, must not be forward reference)
%Err(0,11,PROC/ENDP nesting too deep)
%Err(0,12,Mismatched PROC/ENDP)
%Err(0,13,Invalid origin for absolute segment)
%Err(0,14,Invalid redefinition of symbol)
%Err(0,15,Mismatched SEGMENT/ENDS)
%Err(0,16,Expression must be absolute)
%Err(0,17,Value too large for field)
%Err(1,18,Strings > 2 characters allowed only in DB)
%Err(0,19,Invalid SEGMENT/GROUP prefix)
%Err(0,20,Label phase error, Pass 2 value differs from Pass 1 value)
%Err(0,21,No ASSUME CS: in effect, NEAR label cannot be defined)
%Err(0,22,Invalid GROUP member, must be a SEGMENT name)
%Err(0,23,Limit of 255 EXTRN symbols per object module exceeded)
%Err(0,24,Duplicate declaration for symbol)
%Err(1,25,Not an address expression)
%Err(0,26,Argument to END must be a NEAR/FAR label defined in this module)
%Err(0,27,Invalid argument to ORG, not absolute or offset)
%Err(0,28,Too many GROUPs)
```

Figure 11-1. Error Message Module Program. (Page 2 of 3.)

```
%Err(0,29,Too many SEGMENTs)
%Err(0,30,Too many GROUP members)
%Err(0,31,SEGMENT nesting too deep)
%Err(0,32,Invalid destination operand)
%Err(0,34,Operand must be a BYTE, WORD or DWORD)
%Err(0,35,Operands not reachable thru segment registers)
%Err(0,36,Too little space reserved due to forward reference)
%Err(0,37,Invalid combination of index and base registers)
%Err(0,38,Invalid types of operands for this instruction)
%Err(0,39,May not move immediate value to segment register)
%Err(0,40,Invalid shift count)
%Err(0,41,RET outside of PROC/ENDP)
%Err(0,42,Operand must be NEAR or FAR)
%Err(0,43,NEAR jump to different ASSUME CS:)
%Err(0,44,Conditional jump to FAR label)
%Err(0,45,SHORT jump to further away than 128 bytes)
%Err(0,46,Segment size exceeds 64K bytes)
%Err(0,47,No END statment or open SEGMENT/ENDS PROC/ENDP)
%Err(1,48,Missing right ''%1)'')
%Err(1,49,Invalid character following the Metacharacter)
%Err(0,50,Invalid control)
%Err(0,51,Undefined macro or control)
%Err(1,52,Invalid call pattern)
%Err(1,53,Invalid pattern argument to MATCH)
%Err(1,54,Invalid LOCAL symbol definition)
%Err(0,55,Macro or INCLUDE nesting level too deep)
%Err(0,56,Invalid PAGEWIDTH or PAGELENGTH)
%Err(0,57,SAVE/RESTORE nesting level too deep)
%Err(0,58,RESTORE without matching SAVE)
%Err(0,59,Attempt to redefine builtin function)
%Err(0,60,Macro attempts to redefine itself)
%Err(0,61,Instruction always uses ES:, may not be overridden)
%Err(0,62,May not index NEAR or FAR expression)
%Err(0,63,Attempt to divide or MOD by 0)
%Err(0,64,Two memory operands are illegal)
%Err(1,65,DUP factor must be positive integer)
%Err(0,66,Internal Error #2)

AsmEr1 ENDS
END
```

Figure 11-1. Error Message Module Program. (Page 3 of 3.)

```
Convergent Macro Assembler X1.2                                          15:45 18-Sep-80 Page    1

                        1    ;Skeleton main program
                        2
                        3    Main     SEGMENT WORD
                        4    ASSUME   CS:Main
                        5
                        6    Begin:
                        7
                        8    ;Put program here, in place of this code which beeps the beeper
0000 B040               9    Loopx:   MOV      AL,40H
0002 E644              10             OUT      44H,AL
0004 B9FFFF            11             MOV      CX,OFFFFH
0007 E2FE              12             LOOP     $                      ;beeper ON for about a second
0009 33C0              13             XOR      AX,AX
000B E644              14             OUT      44H,AL
000D B9FFFF            15             MOV      CX,OFFFFH
0010 E2FE              16             LOOP     $                      ;beeper OFF for about a second
0012 EBEC              17             JMP      Loopx
                       18    ;End of beeper code
                       19
                       20    Main     ENDS
                       21
                       22    Stack    SEGMENT STACK                   ;Stack must have STACK combine type
0000 (   96            23             DW       60H     DUP (?)        ;Need about 60H word stack min. to run
     0000)

                       24                                             ;under CTOS and use debugger
                       25    Stack    ENDS
                       26
                       27    END      Begin
                       28

There were no errors detected
```

Figure 11-2. Standalone Main Program.

```
Convergent Macro Assembler X1.2                                         15:45 18-Sep-80 Page   1

                                 ;Sample main program which links with Convergent Object module procedures
                                 ;This program forever outputs the string "Now is the time ..." followed by an
                                 ;iteration count to the video.

                                 ;Declare the OS and Object module procedures external, accessable by FAR CALL's
                                 EXTRN   WriteBsRecord:FAR, WriteByte:FAR, ErrorExit:FAR

                                 ;First declare code segment so that it is loaded first, class = Code so that it
                                 ;will be physically near Convergent code.  Note that it need not be PUBLIC

                                 Main    SEGMENT WORD              'Code'
                                 Main    ENDS

                                 ;Next declare segment containing all constant cata which will be combined with
                                 ;Convergent segment of same name and class

                                 Const   SEGMENT WORD    PUBLIC  'Const'

0000 4E6F772069732074            rgchMsg DB      'Now is the time for all good men to come to the aid of their party'
     68652074696D6520
     666F7220616C6C20
     676F6F64206D656E
     20746F20636F6D65
     20746F2074686520
     616964206F662074
     6865697220706172
     7479
0042 4200                        cbMsg   DW      SIZE rgchMsg            ;Count of bytes in msg

                                 Const   ENDS

                                 ;Next declare segment containing all variable data which will be combined with
                                 ;Convergent segment of same name and class

                                 Data    SEGMENT WORD    PUBLIC  'Data'
                                 EXTRN   bsVid:BYTE                      ;We write to video using SAM's preopened
                                                                         ;bytestream which is located in the Data segment
                                                                         ;It is important that this declaration be embedded
                                                                         ;in Data SEGMENT/ENDS directives as here

0000 0000                        cloop   DW      0                       ;Count of loops
0002 0000                        cbWrittenRet    DW      ?               ;Word for WriteBsRecord to return bytes written

                                 Data    ENDS

                                 ;Stack segment. Should have name and class of Stack so as to be combined with
                                 ;Convergent Stack segments (which contain space estimates for stack used by Convergent
                                 ;software). Space allocated here need only be sufficient for procedures in this
                                 ;module plus a fixed overhead of about 60H words(i.e. 192 decimal bytes) which allows
                                 ;for interrupts and OS calls

                                 Stack   SEGMENT STACK             'Stack' ;Note especially the combine type = STACK,
                                                                           ;not PUBLIC.
```

Figure 11-3.   Convergent-Compatible Main Program.   (Page 1 of 3.)

```
Convergent Macro Assembler X1.2                                                    15:45 18-Sep-80 Page   2

0000 (    96             47          DW       60H      DUP (?)
     0000)
     00C0                48    wLimStack      EQU      THIS     WORD   ;Initial top of stack label. Because of the
                         49                                            ;way STACK segments are combined, this will
                         50                                            ;label the end of the combined Stack segment
                         51    STACK    ENDS
                         52
                         53    Dgroup   GROUP    Const, Data, Stack    ;All addressing of variables/constants thru
                         54                                            ;a GROUP named Dgroup which is known to all
                         55                                            ;Object module procedures and must  be loaded into SS and DS

                         56
                         57    ;Here is the program code
                         58
                         59    Main     SEGMENT
                         60    ASSUME   CS:Main                        ;All code in Main will be relative to start of Main
                         61
0000 B8----              62    Begin:   MOV      AX,Dgroup             ;Load Dgroup into SS and DS
0003 8ED0                63             MOV      SS,AX
                         64    ASSUME   SS:Dgroup                      ;Tell the assembler about new seg register contents
0005 BC0000          R   65             MOV      SP,OFFSET Dgroup:wLimStack;Init stack pointer, must immediately follow SS load
                         66                                            ;Note that stack must be relative to Dgroup since
                         67                                            ;that is what is in SS
0008 8ED8                68             MOV      DS,AX
                         69    ASSUME   DS:Dgroup                      ;Tell the assembler about new seg register contents
                         70
000A C70600000000    R   71             MOV      cloop,0               ;initialize loop counter
                         72
                         73    Loopx:
                         74    ;CALL WriteBsRecord(pbsVid, prgchMsg, cbMsg, pcbWrittenRet)
                         75
0010 1E                  76             PUSH     DS                    ;(1) pbsVid
0011 8D060000        E   77             LEA      AX,bsVid
0015 50                  78             PUSH     AX
0016 1E                  79             PUSH     DS                    ;(2) prgchMsg
0017 8D060000        R   80             LEA      AX,rgchMsg
001B 50                  81             PUSH     AX
001C FF364200        R   82             PUSH     cbMsg                 ;(3) cbMsg
0020 1E                  83             PUSH     DS                    ;(4) pcbWrittenRet
0021 8D060200        R   84             LEA      AX,cbWrittenRet
0025 50                  85             PUSH     AX
0026 9A0000----      E   86             CALL     WriteBsRecord
002B 23C0                87             AND      AX,AX
002D 754C                88               JNE    Error                 ;Test erc, jump if an error occurred
                         89
002F A10000          R   90             MOV      AX,cloop
0032 E81800              91             CALL     PrintHex              ;print loop counter
0035 FF060000        R   92             INC      cloop                 ;and bump it
                         93
                         94    ;CALL WriteByte(pbsVid, 0AH)
0039 1E                  95             PUSH     DS                    ;(1) pbsVid
003A 8D060000        E   96             LEA      AX,bsVid
003E 50                  97             PUSH     AX
003F B00A                98             MOV      AL,0AH
0041 50                  99             PUSH     AX                    ;(2) 0AH
```

Figure 11-3. Convergent-Compatible Main Program. (Page 2 of 3.)

```
Convergent Macro Assembler X1.2                                                   15:45 18-Sep-80 Page    3

0042 9A0000----        E  100          CALL    WriteByte
0047 23C0                 101          AND     AX,AX
0049 7530                 102            JNE   Error                   ;Test erc, jump if an error occurred
                          103
004B EBC3                 104          JMP     Loopx                   ;Loop forever
                          105
                          106  ;Local procedure to convert number in AX to hex and output to video
                          107
                          108  PrintHex        PROC    NEAR
004D B90400               109          MOV     CX,4                    ;Init digit count
0050 51                   110  Print1: PUSH    CX                      ;preserve digit count
0051 B104                 111          MOV     CL,4
0053 D3C0                 112          ROL     AX,CL                   ;position next digit
0055 50                   113          PUSH    AX                      ;save rotated word since WriteByte clobbers
                          114                                          ;all the registers
0056 8BD8                 115          MOV     BX,AX
0058 80E30F               116          AND     BL,0FH                  ;mask it off
005B 80C330               117          ADD     BL,'0'                  ;convert to ascii
005E 80FB39               118          CMP     BL,'9'
0061 7603                 119            JBE   Print2                  ;jump if 0-9
0063 80C307               120          ADD     BL,'A'-'0'-10.          ;else in range A-F
                          121  Print2:
                          122  ;CALL    WriteByte(pbsVid, BL)
0066 1E                   123          PUSH    DS                      ;(1) pbsVid
0067 8D060000          E  124          LEA     AX,bsVid
006B 50                   125          PUSH    AX
006C 53                   126          PUSH    BX                      ;(2) BL
006D 9A0000----        E  127          CALL WriteByte
0072 23C0                 128          AND     AX,AX
0074 7505                 129            JNE   Error                   ;jump if non zero erc
                          130
0076 58                   131          POP     AX                      ;restore word to output
0077 59                   132          POP     CX                      ;restore digit count
0078 E2D6                 133          LOOP    Print1                  ;Loop 4 times
007A C3                   134          RET
                          135  PrintHex        ENDP
                          136
                          137  ;Here on fatal error, AX = erc
                          138
                          139  ;CALL ErrorExit(erc)
007B 50                   140  Error:  PUSH    AX                      ;(1) erc
007C 9A0000----        E  141          CALL ErrorExit
                          142
                          143  Main    ENDS
                          144
                          145  END     Begin                           ;Specify start address of Begin
                          146

There were no errors detected
```

Figure 11-3. Convergent-Compatible Main Program. (Page 3 of 3.)

Appendix A: INSTRUCTION SET

Table A-3 lists the instruction set in numeric order of instruction code. Table A-4 lists the instruction set in alphabetical order of instruction mnemonic. This instruction set is described in detail in the Central Processing Unit.

Legend

Each table contains seven columns.

The column labeled "Op Cd" is the operand code. "Memory Organization" is explained in Section 6. The "Instruction" column is the instruction mnemonic. The "Operand," if there is one, is the operand acted upon by the instruction.

The "Summary" column contains a brief summary of each instruction. Parentheses surrounding an item means "the contents of." For example, "(EA)" means "the contents of memory location EA," and "(SS)" means "the contents of register SS." The infix operators (+, -, OR, XOR, etc.) denote the standard arithmetic or logical operation. CMP denotes a subtraction wherein the result is discarded and only the values of the flags are changed. "TEST" denotes a logical "AND" wherein the result is discarded and only the values of the flags are changed.

The "clocks" column is the clock time for each instruction. (See Table A-1 below.) Where two clock times are given in the conditional instructions, the first is the time if the jump (or loop) is performed, and the second if it is not. In all instructions with memory (EA) as one of the operands, a second clock time is given in parentheses. This is because in all these instructions memory may be replaced by a register. In such cases, the faster clock time applies. Where repetitions are possible, a second clock time is also given in parentheses, in the form "x+y/rep", where "x" is the base clock time, "y" is the clock time to be added for each repetition, and "rep" is the number of repetitions.

The "flags" column enumerates the flag conditions, according to this code:

```
      S = set (to 1)
      C = cleared (to 0)
      X = altered to reflect operation result
      U = undefined (code should not rely on these values)
      R = replaced from memory (e.g., POPF)
  blank = unaffected
```

These are the flags:

0 = Overflow flag
D = Direction flag
1 = Interrupt-enable flag
T = Trap flag
S = Sign flag
Z = Zero flag
A = Auxiliary Carry flag
P = Parity flag
C = Carry flag

These symbols are used in the tables:

Symbol	Interpretation
bAddr	16-bit offset within a segment of a word (addressed without use of base or indexing)
bData	byte immediate constant
bEA	effective address of a byte
bREG	8-bit register (AH, AL, BH, CH, CL, DH, or DL)
CF	value (0 or 1) of the carry flag
Ext(b)	word obtained by sign extending byte b
FLAGS	values of the various flags
off	16-bit offset within a segment
Sign(w)	word of all 0's if w is positive, all 1's if w is negative
sba	segment base address
SR	segment register (CS, DS, ES, or SS)
wAddr	16-bit offset within a segment of a word (addressed without use of base or indexing)
wData	word immediate constant
wEA	effective address of a word
wREG	16-bit register (AX, BX, CX, DX, SP, BP, SI, or DI)

Effective Address (EA) calculation time is according to Table A-1 below:

Table A-1. Effective Address Calculation Time.		
EA Components		Clocks*
Displacement only		6
Base or Index only	(BX, BP, SI, DI)	5
Displacement + Base or Index	(BX, BP, SI, DI)	9
Base + Index	[BP+DI],[BX+SI]	7
	[BP+SI],[BX+DI]	8
Displacement + Base + Index	[BP+DI]+DISP [BX+SI]+DISP	11
	[BP+SI]+DISP [BX+DI]+DISP	12

*Add two clocks for segment override. Add four clocks for each 16-bit word transfer with an odd address.

Alternate Mnemonics

These instructions have synonymous alternate mnemonics:

Table A-2. Alternate Mnemonics.

Instruction	Synonym	Description
JA	JNBE	Jump if not below or equal
JAE	JNB	Jump if not below
JAE	JNC	Jump if not carry
JB	JNAE	Jump if not above or equal
JB	JC	Jump if carry
JBE	JNA	Jump if not above
JG	JNLE	Jump if not less or equal
JGE	JNL	Jump if not less
JL	JNGE	Jump if not greater or equal
JLE	JNG	Jump if not greater
JNZ	JNE	Jump if not equal
JPE	JP	Jump if parity
JPO	JNP	Jump if no parity
JZ	JE	Jump if equal
LOOPNZ	LOOPNE	Loop (CX) times while not equal
LOOPZ	LOOPE	Loop (CX) times while equal
REPZ	REP	Repeat string operation
REPZ	REPE	Repeat string operation while equal
REPNZ	REPNE	Repeat while $(CX) \neq 0$ and $(ZF) = 1$
SHL	SAL	Byte shift EA left 1 bit

Table A-3. Instruction Set in Numeric Order of Instruction Code. (Page 1 of 7.)

Op Cd	Memory Organization	Instruc- tion	Operand	Summary	Clocks	Flags ODITSZAPC
00	MOD REGR/M	ADD	bEA,REG	(bEA)=(bEA)+(bREG)	16+EA(3)	X XXXXX
01	MOD REGR/M	ADD	wEA,REG	(wEA)=(wEA)+(wREG)	16+EA(3)	X XXXXX
02	MOD REGR/M	ADD	REG,bEA	(bREG)=(bREG)+(bEA)	9+EA(3)	X XXXXX
03	MOD REGR/M	ADD	REG,wEA	(wREG)=(wREG)+(wEA)	9+EA(3)	X XXXXX
04		ADD	AL,bData	(AL)=(AL)+bData	4	X XXXXX
05		ADD	AX,wData	(AX)=(AX)+wData	4	X XXXXX
06		PUSH	ES	Push (ES) onto stack	10	
07		POP	ES	Pop stack to ES	8	
08	MOD REGR/M	OR	bEA,REG	(bEA)=(bEA) OR (bREG)	16+EA(3)	C XXUXC
09	MOD REGR/M	OR	wEA,REG	(wEA)=(wEA) OR (wREG)	16+EA(3)	C XXUXC
0A	MOD REGR/M	OR	REG,bEA	(bREG)=(bREG) OR (bEA)	9+EA(3)	C XXUXC
0B	MOD REGR/M	OR	REG,wEA	(wREG)=(wREG) OR (wEA)	9+EA(3)	C XXUXC
0C		OR	AL,bData	(AL)=(AL) OR bData	4	C XXUXC
0D		OR	AX,wData	(AX)=(AX) OR wData	4	C XXUXC
0E		PUSH	CS	Push (CS) onto stack	11	
0F		(not used)				
10	MOD REGR/M	ADC	EA,REG	(bEA)=(bEA)+(bREG)+CF	16+EA(3)	X XXXXX
11	MOD REGR/M	ADC	EA,REG	(wEA)=(wEA)+(wREG)+CF	16+EA(3)	X XXXXX
12	MOD REGR/M	ADC	REG,EA	(bREG)=(bREG)+(bEA)+CF	9+EA(3)	X XXXXX
13	MOD REGR/M	ADC	REG,EA	(wREG)=(wREG)+(wEA)+CF	9+EA(3)	X XXXXX
14		ADC	AL,bData	(AL)=(AL)+bData+CF	4	X XXXXX
15		ADC	AX,wData	(AX)=(AX)+wData+CF	4	X XXXXX
16		PUSH	SS	Push (SS) onto stack	11	X XXXXX
17		POP	SS	Pop stack to SS	8	
18	MOD REGR/M	SBB	bEA,REG	(bEA)=(bEA)-(bREG)-CF	16+EA(3)	X XXXXX
19	MOD REGR/M	SBB	wEA,REG	(wEA)=(wEA)-(wREG)-CF	16+EA(3)	X XXXXX
1A	MOD REGR/M	SBB	REG,bEA	(bREG)=(bREG)-(bEA)-CF	9+EA(3)	X XXXXX
1B	MOD REGR/M	SBB	REG,wEA	(wREG)=(wREG)-(wEA)-CF	9+EA(3)	X XXXXX
1C		SBB	AL,bData	(AL)=(AL)-bData-CF	4	X XXXXX
1D		SBB	AL,wData	(AX)=(AX)-wData-CF	4	X XXXXX
1E		PUSH	DS	Push (DS) onto stack	10	
1F		POP	DS	Pop stack to DS	8	
20	MOD REGR/M	AND	bEA,REG	(bEA)=(bEA) AND (bREG)	16+EA(3)	C XXUXC
21	MOD REGR/M	AND	wEA,REG	(wEA)=(wEA) AND (wREG)	16+EA(3)	C XXUXC
22	MOD REGR/M	AND	REG,bEA	(bREG)=(bREG) AND (bEA)	9+EA(3)	C XXUXC
23	MOD REGR/M	AND	REG,wEA	(wREG)=(wREG) AND (wEA)	9+EA(3)	C XXUXC
24		AND	AL,bData	(AL)=(AL) AND bData	4	C XXUXC
25		AND	AX,wData	(AX)=(AX) AND wData	4	C XXUXC
26		ES:		ES segment override	2	
27		DAA		Decimal adjust for ADD	4	X XXXXX
28	MOD REGR/M	SUB	bEA,REG	(bEA)=(bEA)-(bREG)	16+EA(3)	X XXXXX
29	MOD REGR/M	SUB	wEA,REG	(wEA)=(wEA)-(wREG)	16+EA(3)	X XXXXX
2A	MOD REGR/M	SUB	REG,bEA	(bREG)=(bREG)-(bEA)	9+EA(3)	X XXXXX
2B	MOD REGR/M	SUB	REG,wEA	(wREG)=(wREG)-(wEA)	9+EA(3)	X XXXXX
2C		SUB	AL,bData	(AL)=(AL)-bData	4	X XXXXX
2D		SUB	AX,wData	(AX)=(AX)-wData	4	X XXXXX
2E		CS:		CS segment override	2	
2F		DAS		Decimal adjust for subtract	4	U XXXXX
30	MOD REGR/M	XOR	bEA,REG	(bEA)=(bEA) XOR (bREG)	16+EA(3)	C XXUXC
31	MOD REGR/M	XOR	wEA,REG	(wEA)=(wEA) XOR (wREG)	16+EA(3)	C XXUXC
32	MOD REGR/M	XOR	REG,bEA	(bREG)=(bREG) XOR (bEA)	9+EA(3)	C XXUXC
33	MOD REGR/M	XOR	REG,wEA	(wREG)=(wREG) XOR (wEA)	9+EA(3)	C XXUXC
34		XOR	AL,bData	(AL)=(AL) XOR bData	4	C XXUXC
35		XOR	AX,wData	(AX)=(AX) XOR wData	4	C XXUXC
36		SS:		SS segment override	2	
37		AAA		ASCII adjust for add	4	U UUXUX
38	MOD REGR/M	CMP	bEA,bREG	FLAGS=(bEA) CMP (bREG)	9+EA	X XXXXX
39	MOD REGR/M	CMP	wEA,wREG	FLAGS=(wEA) CMP (wREG)	9+EA	X XXXXX
3A	MOD REGR/M	CMP	bREG,bEA	FLAGS=(bREG) CMP (bEA)	9+EA	X XXXXX

Table A-3. Instruction Set in Numeric Order of Instruction Code. (Page 2 of 7.)						
Op Cd	Memory Organization	Instruction	Operand	Summary	Clocks	Flags ODITSZAPC
3B	MOD REGR/M	CMP	wREG,wEA	FLAGS=(wREG) CMP (wEA)	9+EA	X XXXXX
3C		CMP	AL,bData	FLAGS=(AL) CMP (bData)	4	X XXXXX
3D		CMP	AX,wData	FLAGS=(AX) CMP (wData)	4	X XXXXX
3E		DS:		DS segment override	2	
3F		AAS		ASCII adjust for subtract	4	U UUXUX
40		INC	AX	(AX)=(AX)+1	2	X XXXX
41		INC	CX	(CX)=(CX)+1	2	X XXXX
42		INC	DX	(DX)=(DX)+1	2	X XXXX
43		INC	BX	(BX)=(BX)+1	2	X XXXX
44		INC	SP	(SP)=(SP)+1	2	X XXXX
45		INC	BP	(BP)=(BP)+1	2	X XXXX
46		INC	SI	(SI)=(SI)+1	2	X XXXX
47		INC	DI	(DI)=(DI)+1	2	X XXXX
48		DEC	AX	(AX)=(AX)-1	2	X XXXX
49		DEC	CX	(CX)=(CX)-1	2	X XXXX
4A		DEC	DX	(DX)=(DX)-1	2	X XXXX
4B		DEC	BX	(BX)=(BX)-1	2	X XXXX
4C		DEC	SP	(SP)=(SP)-1	2	X XXXX
4D		DEC	BP	(BP)=(BP)-1	2	X XXXX
4E		DEC	SI	(SI)=(SI)-1	2	X XXXX
4F		DEC	DI	(DI)=(DI)-1	2	X XXXX
50		PUSH	AX	Push (AX) onto stack	11	
51		PUSH	CX	Push (CX) onto stack	11	
52		PUSH	DX	Push (DX) onto stack	11	
53		PUSH	BX	Push (BX) onto stack	11	
54		PUSH	SP	Push (SP) onto stack	11	
55		PUSH	BP	Push (BP) onto stack	11	
56		PUSH	SI	Push (SI) onto stack	11	
57		PUSH	DI	Push (DI) onto stack	11	
58		POP	AX	Pop stack to AX	8	
59		POP	CX	Pop stack to CX	8	
5A		POP	DX	Pop stack to DX	8	
5B		POP	BX	Pop stack to BX	8	
5C		POP	SP	Pop stack to SP	8	
5D		POP	BP	Pop stack to BP	8	
5E		POP	SI	Pop stack to SI	8	
5F		POP	DI	Pop stack to DI	8	
60		(not used)				
61		(not used)				
62		(not used)				
63		(not used)				
64		(not used)				
65		(not used)				
66		(not used)				
67		(not used)				
68		(not used)				
69		(not used)				
6A		(not used)				
6B		(not used)				
6C		(not used)				
6D		(not used)				
6E		(not used)				
6F		(not used)				
70		JO	bDISP	Jump if overflow	16 or 4	
71		JNO	bDISP	Jump if no overflow	16 or 4	
72		JB	bDISP	Jump if below	16 or 4	
73		JAE	bDISP	Jump if above or equal	16 or 4	
74		JZ	bDISP	Jump if zero	16 or 4	
75		JNZ	bDISP	Jump if not zero	16 or 4	

Table A-3. Instruction Set in Numeric Order of Instruction Code. (Page 3 of 7.)

Op Cd	Memory Organization	Instruction	Operand	Summary	Clocks	Flags ODITSZAPC
76		JBE	bDISP	Jump if below or equal	16 or 4	
77		JA	bDISP	Jump if above	16 or 4	
78		JS	bDISP	Jump if sign	16 or 4	
79		JNS	bDISP	Jump if no sign	16 or 4	
7A		JPE	bDISP	Jump if parity even	16 or 4	
7B		JPO	bDISP	Jump if parity odd	16 or 4	
7C		JL	bDISP	Jump if less	16 or 4	
7D		JGE	bDISP	Jump if greater or equal	16 or 4	
7E		JLE	bDISP	Jump if less or equal	16 or 4	
7F		JG	bDISP	Jump if greater	16 or 4	
80	MOD 000 R/M	ADD	bEA,bData	(bEA)=(bEA)+bData	17+EA	X XXXXX
80	MOD 001 R/M	OR	bEA,bData	(bEA)=(bEA) OR bData	17+EA	C XXUXC
80	MOD 010 R/M	ADC	bEA,bData	(bEA)=(bEA)+bData+CF	17+EA	X XXXXX
80	MOD 011 R/M	SBB	bEA,bData	(bEA)=(bEA)-bData-CF	17+EA	X XXXXX
80	MOD 100 R/M	AND	bEA,bData	(bEA)=(bEA) AND bData	17+EA	C XXUXC
80	MOD 101 R/M	SUB	bEA,bData	(bEA)=(bEA)-bData	17+EA	X XXXXX
80	MOD 110 R/M	XOR	bEA,bData	(bEA)=(bEA) XOR bData	17+EA	C XXUXC
80	MOD 111 R/M	CMP	bEA,bData	FLAGS=(bEA) CMP bData	10+EA	X XXXXX
81	MOD 000 R/M	ADD	wEA,wData	(WEA)=(wEA)+wData	17+EA	X XXXXX
81	MOD 001 R/M	OR	wEA,wData	(wEA)=(wEA) OR wData	17+EA	C XXUXC
81	MOD 010 R/M	ADC	wEA,wData	(wEA)-(wEA)+wData+CF	17+EA	X XXXXX
81	MOD 011 R/M	SBB	wEA,wData	(wEA)=(wEA)-wData-CF	17+EA	X XXXXX
81	MOD 100 R/M	AND	wEA,wData	(wEA)=(wEA) AND wData	17+EA	C XXUXC
81	MOD 101 R/M	SUB	wEA,wData	(wEA)=(wEA)-wData	17+EA	X XXXXX
81	MOD 110 R/M	XOR	wEA,wData	(wEA)=(wEA) XOR wData	17+EA	C XXUXC
81	MOD 111 R/M	CMP	wEA,wData	FLAGS=(wEA) XOR wData	10+EA	X XXXXX
82	MOD 000 R/M	ADD	bEA,bData	(bEA)=(bEA)+bData	17+EA	X XXXXX
82	MOD 001 R/M		(not used)			
82	MOD 010 R/M	ADC	bEA,bData	(bEA)=(bEA)+bData+CF	17+EA	X XXXXX
82	MOD 011 R/M	SBB	bEA,bData	(bEA)=(bEA)-bData-CF	17+EA	X XXXXX
82	MOD 100 R/M		(not used)			
82	MOD 101 R/M	SUB	bEA,bData	(bEA)=(bEA)-bData	17+EA	X XXXXX
82	MOD 110 R/M		(not used)			
82	MOD 111 R/M	CMP	bEA,bData	FLAGS=(bEA) CMP bData	10+EA	X XXXXX
83	MOD 000 R/M	ADD	wEA,bData	FLAGS=(wEA)+Ext(bData)	17+EA	X XXXXX
83	MOD 001 R/M		(not used)			
83	MOD 010 R/M	ADC	wEA,bData	(wEA)=(wEA)+Ext(bData)+CF	17+EA	X XXXXX
83	MOD 011 R/M	SBB	wEA,bData	(wEA)=(wEA)-Ext(bData)-CF	17+EA	X XXXXX
83	MOD 100 R/M		(not used)			
83	MOD 101 R/M	SUB	wEA,bData	(wEA)=(wEA)-Ext(bData)	17+EA	X XXXXX
83	MOD 110 R/M		(not used)			
33	MOD 111 R/M	CMP	wEA,bData	FLAGS=(wEA) CMP Ext(bData)	10+EA	X XXXXX
84	MOD REGR/M	TEST	bEA,bREG	FLAGS=(bEA) TEST (bREG)	9+EA(3)	C XXUXC
85	MOD REGR/M	TEST	wEA,wREG	FLAGS=(wEA) TEST (wREG)	9+EA(3)	C XXUXC
86	MOD REGR/M	XCHG	bREG,bEA	Exchange bREG, bEA	17+EA(4)	
87	MOD REGR/M	XCHG	wREG,wEA	Exchange wREG, wEA	17+EA(4)	
88	MOD REGR/M	MOV	bEA,bREG	(bEA)=(bREG)	9+EA(2)	
89	MOD REGR/M	MOV	wEA,wREG	(wEA)=(wREG)	9+EA(2)	
8A	MOD REGR/M	MOV	bREG,bEA	(bREG)=(bEA)	8+EA(2)	
8B	MOD REGR/M	MOV	wREG,wEA	(wREG)=(wEA)	8+EA(2)	
8C	MOD 0SR R/M	MOV	wEA,SR	(wEA)=(SR)	9+EA(2)	
8C	MOD 1-- R/M	(not used)				
8D	MOD REGR/M	LEA	REG, EA	(REG)=effective address	2+EA(2)	
8E	MOD 0SR R/M	MOV	SR,wEA	(SR)=(wEA)	8+EA(2)	
8E	MOD -- R/M	(not used)				
8F	MOD 000 R/M	POP	EA	Pop stack to EA	17+EA	
8F	MOD 001 R/M	(not used)				
8F	MOD 010 R/M	(not used)				
8F	MOD 011 R/M	(not used)				

Op Cd	Memory Organization	Instruc-tion	Operand	Summary	Clocks	Flags ODITSZAPC
8F	MOD 100 R/M	(not used)				
8F	MOD 101 R/M	(not used)				
8F	MOD 110 R/M	(not used)				
8F	MOD 111 R/M	(not used)				
90		XCHG	AX,AX	NOP	3	
91		XCHG	AX,CX	Exchange (AX), (CX)	3	
92		XCHG	AX,DX	Exchange (AX), (DX)	3	
93		XCHG	AX,BX	Exchange (AX), (BX)	3	
94		XCHG	AX,SP	Exchange (AX), (SP)	3	
95		XCHG	AX,BP	Exchange (AX), (BP)	3	
96		XCHG	AX,SI	Exchange (AX), (SI)	3	
97		XCHG	AX,DI	Exchange (AX), (DI)	3	
98		CBW		(AX)=Ext(AL)	2	
99		CWD		(DX)=Sign(AX)	5	
9A		CALL	off:sba	Direct FAR call	28	
9B		WAITX		Wait for TEST signal	3+WAITX	
9C		PUSHF		Push FLAGS onto stack	10	
9D		POPF		Pop stack to FLAGS	8	RRRRRRRRR
9E		SAHF		(FLAGS)=(AH)	4	RRRRRRRRR
9F		LAHF		(AH)=(FLAGS)	4	
A0		MOV	AL,bAddr	(AL)=(bAddr)	10	
A1		MOV	AX,wAddr	(AX)=(wAddr)	10	
A2		MOV	bAddr,AL	(bAddr)=(AL)	10	
A3		MOV	wAddr,AX	(wAddr)=(AX)	10	
A4		MOVSB		Move byte string	18 (9+17/rep)	
A5		MOVSW		Move word string	18 (9+17/rep)	
A6		CMPSB		Compare byte string	22 (9+22/rep)	X XXXXX
A7		CMPSW		Compare word string	22 (9+22/rep)	X XXXXX
A8		TEST	AL,bData	FLAGS=(AL) TEST (bData)	4	X XXUXC
A9		TEST	AX,bData	FLAGS=(AX) TEST (wData)	4	X XXUXC
AA		STOSB		Store byte string	11 (9+10/rep)	
AB		STOSW		Store word string	11 (9+10/rep)	
AC		LODSB		Load byte string	12 (9+13/rep)	
AD		LODSW		Load word string	12 (9+13/rep)	
AE		SCASB		Scan byte string	15 (9+15/rep)	X XXXXX
AF		SCASW		Scan word string	15 (9+15/rep)	X XXXXX
B0		MOV	AL,bData	(AL)=bData	4	
B1		MOV	CL,bData	(CL)=bData	4	
B2		MOV	DL,bData	(DL)=bData	4	
B3		MOV	BL,bData	(BL)=bData	4	
B4		MOV	AH,bData	(AH)=bData	4	
B5		MOV	CH,bData	(CH)=-bData	4	
B6		MOV	DH,bData	(DH)=bData	4	
B7		MOV	BH,bData	(BH)=bData	4	
B8		MOV	AX,wData	(AX)=wData	4	
B9		MOV	CX,wData	(CX)=wData	4	
BA		MOV	DX,wData	(DX)=wData	4	
BB		MOV	BX,wData	(BX)=wData	4	
BC		MOV	SP,wData	(SP)=wData	4	

Table A-3. Instruction Set in Numeric Order of Instruction Code. (Page 4 of 7.)

Table A-3. Instruction Set in Numeric Order of Instruction Code. (Page 5 of 7.)

Op Cd	Memory Organization	Instruction	Operand	Summary	Clocks	Flags ODITSZAPC
BD		MOV	BP,wData	(BP)=wData	4	
BE		MOV	SI,wData	(SI)=wData	4	
BF		MOV	DI,wData	(DI)=wData	4	
C0		(not used)				
C1		(not used)				
C2		RET	wData	NEAR return; (SP)=(SP)+ wData	12	
C3		RET		NEAR return	8	
C4	MOD REGR/M	LES	REG,EA	ES:REG=(wEA+2):(wEA)	16+EA	
C5	MOD REGR/M	LDS	REG,EA	DS:REG=(wEA+2):(wEA)	16+EA	
C6	MOD 000 R/M	MOV	bEA,bData	(bEA)=(bData)	10+EA	
C6	MOD 001 R/M	(not used)				
C6	MOD 010 R/M	(not used)				
C6	MOD 011 R/M	(not used)				
C6	MOD 100 R/M	(not used)				
C6	MOD 101 R/M	(not used)				
C6	MOD 110 R/M	(not used)				
C6	MOD 111 R/M	(not used)				
C7	MOD 000 R/M	MOV	EA,wData	(wEA)=wData	10+EA	
C7	MOD 001 R/M	(not used)				
C7	MOD 010 R/M	(not used)				
C7	MOD 011 R/M	(not used)				
C7	MOD 100 R/M	(not used)				
C7	MOD 101 R/M	(not used)				
C7	MOD 110 R/M	(not used)				
C7	MOD 111 R/M	(not used)				
C8		(not used)				
C9		(not used)				
CA		RET	wData	FAR return, ADD data to REG SP	17	
CB		RET		FAR return	18	
CC		INT	3	Type 3 interrupt	52	CC
CD		INT	bData	Typed interrupt	51	CC
CE		INTO		Interrupt if overflow	53 or 4	CC
(Simple execution of the instruction takes 4 clocks, and actual interrupt, 53.)						
CF		IRET		Return from interrupt	24	RRRRRRRRR
D0	MOD 000 R/M	ROL	bEA,1	Rotate bEA left 1 bit	15+EA	X X
D0	MOD 001 R/M	ROR	bEA,1	Rotate bEA right 1 bit	15+EA	X X
D0	MOD 010 R/M	RCL	bEA,1	Rotate bEA left through carry 1 bit	15+EA	X X
D0	MOD 011 R/M	RCR	bEA,1	Rotate bEA right through carry 1 bit	15+EA	X X
D0	MOD 100 R/M	SHL	bEA,1	Shift bEA left 1 bit	15+EA	X X
D0	MOD 101 R/M	SHR	bEA,1	Shift bEA right 1 bit	15+EA	X X
D0	MOD 110 R/M	(not used)				
D0	MOD 111 R/M	SAR	bEA,1	Shift signed bEA right 1 bit	15+EA	X XXUXX
D1	MOD 000 R/M	ROL	wEA,1	Rotate wEA left 1 bit	15+EA	X X
D1	MOD 001 R/M	ROR	wEA,1	Rotate wEA right 1 bit	15+EA	X X
D1	MOD 010 R/M	RCL	wEA,1	Rotate wEA left through carry 1 bit	15+EA	X X
D1	MOD 011 R/M	RCR	wEA,1	Rotate wEA right through carry 1 bit	15+EA	X X
D1	MOD 100 R/M	SHL	wEA,1	Shift wEA left 1 bit	15+EA	X X
D1	MOD 101 R/M	SHR	wEA,1	Shift wEA right 1 bit	15+EA	X X
D1	MOD 110 R/M	(not used)				
D1	MOD 111 R/M	SAR	wEA,1	Shift signed wEA right 1 bit	15+EA	X XXUXX

Table A-3. Instruction Set in Numeric Order of Instruction Code. (Page 6 of 7.)

Op Cd	Memory Organization	Instruction	Operand	Summary	Clocks	Flags ODITSZAPC
D2	MOD 000 R/M	ROL	bEA,CL	Rotate bEA left (CL) bits	20+EA +4/bit	X X
D2	MOD 001 R/M	ROR	bEA,CL	Rotate bEA right (CL) bits	20+EA +4/bit	X X
D2	MOD 010 R/M	RCL	bEA,CL	Rotate bEA left through carry (CL) bits	20+EA +4/bit	X X
D2	MOD 011 R/M	RCR	bEA,CL	Rotate bEA right through carry (CL) bits	20+EA +4/bit	X X
D2	MOD 100 R/M	SHL	bEA,CL	Shift bEA left (CL) bits	20+EA +4/bit	X X
D2	MOD 101 R/M	SHR	bEA,CL	Shift bEA right (CL) bits	20+EA +4/bit	X X
D2	MOD 110 R/M	(not used)				
D2	MOD 111 R/M	SAR	bEA,CL	Shift signed bEA right (CL) bits	20+EA +4/bit	X XXUXX
D3	MOD 000 R/M	ROL	wEA,CL	Rotate wEA left (CL) bits	20+EA +4/bit	X X
D3	MOD 001 R/M	ROR	wEA,CL	Rotate wEA right (CL) bits	20+EA +4/bit	X X
D3	MOD 010 R/M	RCL	wEA,CL	Rotate wEA left through carry (CL) bits	20+EA +4/bit	X X
D3	MOD 011 R/M	RCR	wEA,CL	Rotate wEA right through carry (CL) bits	20+EA +4/bit	X X
D3	MOD 100 R/M	SHL	wEA,CL	Shift wEA left (CL) bits	20+EA +4/bit	X X
D3	MOD 101 R/M	SHR	wEA,CL	Shift wEA right (CL) bits	20+EA +4/bit	X X
D3	MOD 110 R/M	(not used)				
D3	MOD 111 R/M	SAR	wEA,CL	Shift signed wEA right (CL) bits	20+EA +4/bit	X XXUXX
D4	00001010		AAM	ASCII adjust for multiply	83	U XXUXU
D5	00001010		AAD	ASCII adjust for divide	60	U XXUXU
D6		(not used)				
D7		XLAT	TABLE	Translate using (BX)	11	
D8	MOD -- R/M	ESC	EA	Escape to external device	8+EA	
E0		LOOPNZ	bDISP	Loop (CX) times while not zero	19 or 5	
E1		LOOPZ	bDISP	Loop (CX) times while zero	18 or 6	
E2		LOOP	bDISP	Loop (CX) times	17 or 5	
E3		JCXZ	bDISP	Jump if (CX)=0	18 or 6	
E4		IN	AL,bPort	Input from bPort to AL	10	
E5		IN	AX,wPort	Input from wPort to AX	10	
E6		OUT	bPort,AL	Output (AL) to bPort	10	
E7		OUT	wPort,AX	Output (AX) to wPort	10	
E8		CALL	wDISP	Direct near call	11	
E9		JMP	wDISP	Direct near jump	7	
EA		JMP	wDISP, wSEG	Direct far jump	7	
EB		JMP	bDISP	Direct near jump	7	
EC		IN	AL,DX	Byte input from port (DX) to REG AL	8	
ED		IN	AX,DX	Word input from port (DX) to REG AX	8	
EE		OUT	DX,AL	Byte output (AL) to port (DX)	8	
EF		OUT	DX,AX	Word output (AX) to port (DX)	8	
F0		LOCK		Bus lock prefix	2	
F1		(not used)				

Table A-3. Instruction Set in Numeric Order of Instruction Code. (Page 7 of 7.)

Op Cd	Memory Organization	Instruction	Operand	Summary	Clocks	Flags ODITSZAPC
F2		REPNZ		Repeat while (CX)≠0 AND (ZF)=0	2	
F3		REPZ		Repeat while (CX)≠0 AND (ZF)=1	2	
F4		HLT		Halt	2	
F5		CMC		Complement carry flag	2	`        X`
F6	MOD 000 R/M	TEST	bEA,bData	FLAGS=(bEA) TEST bData	10+EA	`C   XXUXC`
F6	MOD 001 R/M	(not used)				
F6	MOD 010 R/M	NOT	bEA	Byte invert bEA	16+EA	
F6	MOD 011 R/M	NEG	bEA	Byte negate bEA	16+EA	`X   XXXXS`
(Note: Carry Flag is C if destination is 0.)						
F6	MOD 100 R/M	MUL	bEA	Unsigned multiply by (bEA)	71	`X   UUUUX`
F6	MOD 101 R/M	IMUL	bEA	Signed multiply by (bEA)	90	`X   UUUUX`
F6	MOD 110 R/M	DIV	bEA	Unsigned divide by (bEA)	90	`U   UUUUU`
F6	MOD 111 R/M	IDIV	bEA	Signed divide by (bEA)	112	`U   UUUUU`
F7	MOD 000 R/M	TEST	wEA,wData	FLAGS=(wEA) TEST wData	10+EA	`C   XXUXC`
F7	MOD 001 R/M	(not used)				
F7	MOD 010 R/M	NOT	wEA	Invert wEA	16+EA	
F7	MOD 011 R/M	NEG	wEA	Negate wEA	16+EA	`X   XXXXS`
(Note: Carry Flag is C if destination is 0.)						
F7	MOD 100 R/M	MUL	wEA	Unsigned multiply by (wEA)	124	`X   UUUUX`
F7	MOD 101 R/M	IMUL	wEA	Signed multiply by (wEA)	144	`X   UUUUX`
F7	MOD 110 R/M	DIV	wEA	Unsigned divide by (wEA)	155	`U   UUUUU`
F7	MOD 111 R/M	IDIV	wEA	Signed divide by (wEA)	177	`U   UUUUU`
F8		CLC		Clear carry flag	2	`        C`
F9		STC		Set carry flag	2	`        S`
FA		CLI		Clear interrupt flag	2	`  C`
FB		STI		Set interrupt flag	2	`  S`
FC		CLD		Clear direction flag	2	` C`
FD		STD		Set direction flag	2	` C`
FE	MOD 000 R/M	INC	bEA	(bEA)=(bEA)+1	15+EA	`X   XXXX`
FE	MOD 001 R/M	DEC	bEA	(bEA)=(bEA)-1	15+EA	`X   XXXX`
FE	MOD 010 R/M	(not used)				
FE	MOD 011 R/M	(not used)				
FE	MOD 100 R/M	(not used)				
FE	MOD 101 R/M	(not used)				
FE	MOD 110 R/M	(not used)				
FE	MOD 111 R/M	(not used)				
FF	MOD 000 R/M	INC	wEA	(wEA)=(wEA)+1	15+EA	`X   XXXX`
FF	MOD 001 R/M	DEC	wEA	(wEA)=(wEA)-1	15+EA	`X   XXXX`
FF	MOD 010 R/M	CALL		Indirect NEAR call	13+EA	
FF	MOD 011 R/M	CALL		Indirect FAR call	29+EA	
FF	MOD 100 R/M	JMP		Indirect NEAR jump	7+EA	
FF	MOD 101 R/M	JMP		Indirect FAR jump	16+EA	
FF	MOD 110 R/M	PUSH		Push (EA) onto stack	16+EA	
FF	MOD 111 R/M	(not used)				

Table A-3. Instruction Set in Alphabetic Order of Instruction Mnemonic. (1 of 6.)

Instruction	Operand	Summary	Op Cd	Memory Organization	Clocks	Flags ODITSZAPC
AAA		ASCII adjust for add	37		4	U UUXUX
AAD		ASCII adjust for divide	D5	00001010	60	U XXUXU
AAM		ASCII adjust for multiply	D4	00001010	83	U XXUXU
AAS		ASCII adjust for subtract	3F		4	U UUXUX
ADC	AL,bData	(AL)=(AL)+bData+CF	14		4	X XXXXX
ADC	AX,wData	(AX)=(AX)+wData+CF	15		4	X XXXXX
ADC	bEA,bData	(bEA)=(bEA)+bData+CF	80	MOD 010 R/M	17+EA	X XXXXX
ADC	wEA,wData	(wEA)=(wEA)+wData+CF	81	MOD 010 R/M	17+EA	X XXXXX
ADC	bEA,bData	(bEA)=(bEA)+bData+CF	82	MOD 010 R/M	17+EA	X XXXXX
ADC	wEA,bData	(wEA)=(wEA)+Ext(bData)+CF	83	MOD 010 R/M	17+EA	X XXXXX
ADC	bEA,REG	(bEA)=(bEA)+(bREG)+CF	10	MOD REGR/M	16+EA(3	X XXXXX
ADC	wEA, REG-	(wEA)=(wEA)+(wREG)+CF	11	MOD REGR/M	16+EA(3	X XXXXX
ADC	REG,bEA	(bREG)=(bREG)+(bEA)+CF	12	MOD REGR/M	9+EA(3)	X XXXXX
ADC	REG,wEA	(wREG)=(wREG)+(wEA)+CF	13	MOD REGR/M	9+EA(3)	X XXXXX
ADD	AL,bData	(AL)=(AL)+bData	04		4	X XXXXX
ADD	AX,wData	(AX)=(AX)+wData	05		4	X XXXXX
ADD	bEA,REG	(bEA)=(bEA)+(bREG)	00	MOD REGR/M	16+EA(3	X XXXXX
ADD	wEA,REG	(wEA)=(wEA)+(wREG)	01	MOD REGR/M	16+EA(3)	X XXXXX
ADD	REG,bEA	(bREG)=(bREG)+(bEA)	02	MOD REGR/M	9+EA(3)	X XXXXX
ADD	REG,wEA	(wREG)=(wREG)+(wEA)	03	MOD REGR/M	9+EA(3)	X XXXXX
ADD	bEA,bData	(bEA)=(bEA)+bData	80	MOD 000 R/M	17+EA	X XXXXX
ADD	wEA,wData	(wEA)=(wEA)+wData	81	MOD 000 R/M	17+EA	X XXXXX
ADD	bEA,bData	(bEA)=(bEA)+bData	82	MOD 000 R/M	17+EA	X XXXXX
ADD	wEA,bData	FLAGS=(wEA)+Ext(bData)	83	MOD 000 R/M	17+EA	X XXXXX
AND	AL,bData	(AL)=(AL) AND bData	24		4	C XXUXC
AND	AX,wData	(AX)=(AX) AND wData	25		4	C XXUXC
AND	bEA,REG	(bEA)=(bEA) AND (bREG)	20	MOD REGR/M	16+EA(3	C XXUXC
AND	wEA,REG	(wEA)=(wEA) AND (wREG)	21	MOD REGR/M	16+EA(3	C XXUXC
AND	REG, bEA	(bREG)=(bREG) AND (bEA)	22	MOD REGR/M	9+EA(3)	C XXUXC
AND	REG,wEA	(wREG)=(wREG) AND (wEA)	23	MOD REGR/M	9+EA(3)	C XXUXC
AND	bEA,bData	(bEA)=(bEA) AND bData	80	MOD 100 R/M	17+EA	C XXUXC
AND	wEA,wData	(wEA)=(wEA) AND wData	81	MOD 100 R/M	17+EA	C XXUXC
CALL.	off:sba	Direct FAR call	9A		28	
CALL	wDISP	Direct NEAR call	E8		11	
CALL	EA	Indirect NEAR call	FF	MOD 010 R/M	13+EA	
CALL	EA	Indirect FAR call	FF	MOD 011 R/M	29+EA	
CBW		(AX)=Ext(AL)	98		2	
CLC		Clear carry flag	F8		2	C
CLD		Clear direction flag	FC		2	C
CLI		Clear interrupt flag	FA		2	C
CMC		Complement carry flag	F5		2	X
CMP	AL,bData	FLAGS=(AL) CMP (bData)	3C		4	X XXXXX
CMP	AX, wData	FLAGS=(AX) CMP (wData)	3D		4	X XXXXX
CMP	bEA,bREG	FLAGS=(bEA) CMP (bREG)	38	MOD REGR/M	9+EA	X XXXXX
CMP	wEA,wREG	FLAGS=(wEA) CMP (wREG)	39	MOD REGR/M	9+EA	X XXXXX
CMP	bREG,bEA	FLAGS=(bREG) CMP (bEA)	3A	MOD REGR/M	9+EA	X XXXXX
CMP	wREG,wEA	FLAGS=(wREG) CMP (wEA)	3B	MOD REGR/M	9+EA	X XXXXX
CMP	bEA,bData	FLAGS=(bEA) CMP bData	80	MOD 111 R/M	10+EA	X XXXXX
CMP	bEA,bData	FLAGS=(bEA) CMP bData	82	MOD 111 R/M	10+EA	X XXXXX
CMP	wEA,wData	FLAGS=(wEA) CMP wData	81	MOD 111 R/M	10+EA	X XXXXX
CMP	wEA,bData	FLAGS=(wEA) CMP Ext(bData)	83	MOD 111 R/M	10+EA	X XXXXX
CMPSB		Compare byte string	A6		22 (9+22/rep)	X XXXXX
CMPSW		Compare word string	A7		22 (9+22/rep)	X XXXXX
CS:		CS segment override	2E		2	
CWD		(DX)=Sign(AX)	99		5	
DAA		Decimal adjust for ADD	27		4	X XXXXX

Table A-3. Instruction Set in Alphabetic Order of Instruction Mnemonic. (2 of 6.)

Instruction	Operand	Summary	Op Cd	Memory Organization	Clocks	Flags ODITSZAPC
DAS		Decimal adjust for subtract	2F		4	U XXXXX
DEC	AX	(AX)=(AX)-1	48		2	X XXXX
DEC	BP	(BP)=(BP)-1	4D		2	X XXXX
DEC	BX	(BX)=(BX)-1	4B		2	X XXXX
DEC	CX	(CX)=(CX)-1	49		2	X XXXX
DEC	DI	(DI)=(DI)-1	4F		2	X XXXX
DEC	DX	(DX)=(DX)-1	4A		2	X XXXX
DEC	bEA	(bEA)=(bEA)-1	FE	MOD 001 R/M	15+EA	X XXXX
DEC	wEA	(wEA)=(wEA)-1	FF	MOD 001 R/M	15+EA	X XXXX
DEC	SP	(SP)=(SP)-1	4C		2	X XXXX
DEC	SI	(SI)=(SI)-1	4E		2	X XXXX
DIV	bEA	Unsigned divide by (bEA)	F6	MOD 110 R/M	90	U UUUUU
DIV	wEA	Unsigned divide by (wEA)	F7	MOD 110 R/M	155	U UUUUU
DS:		DS segment override	3E		2	
ES:		ES segment override	26		2	
ESC	EA	Escape to external device	D8	MOD --- R/M	8+EA	
HLT		Halt	F4		2	
IDIV	bEA	Signed divide by (bEA)	F6	MOD 111 R/M	112	U UUUUU
IDIV	wEA	Signed divide by (wEA)	F7	MOD 111 R/M	177	U UUUUU
IMUL	bEA	Signed multiply by (bEA)	F6	MOD 101 R/M	30	X UUUUX
IMULT	WEA	Signed multiply by (wEA)	F7	MOD 101 R/M	144	X UUUUX
IN	AL,DX	Byte input from port (DX) to REG AL	EC		8	
IN	AL,bPort	Input from bPort to AL	E4		10	
IN	AX,DX	Word input from port (DX) to REG AX	ED		8	
IN	AX,wPort	Input from wPort to AX	E5		10	
INC	AX	(AX)=(AX)+1	40		2	X XXXX
INC	BP	(BP)=(BP)+1	45		2	X XXXX
INC	BX	(BX)=(BX)+1	43		2	X XXXX
INC	CX	(CX)=(CX)+1	41		2	X XXXX
INC	DI	(DI)=(DI)+1	47		2	X XXXX
INC	DX	(DX)=(DX)+1	42		2	X XXXX
INC	bEA	(bEA)=(bEA)+1	FE	MOD 000 R/M	15+EA	X XXXX
INC	wEA	(wEA)=(wEA)+1	FF	MOD 000 R/M	15+EA	X XXXX
INC	SP	(SP)=(SP)+1	44		2	X XXXX
INC	SI	(SI)=(SI)+1	46		2	X XXXX
INT	bData	Typed interrupt	CD		51	CC
INT	3	Type 3 interrupt	CC		52	CC
INTO		Interrupt if overflow	CE		53 or 4	CC
Simple execution of the instruction takes 4 clocks and actual interrupt, 53.)						
IRET		Return from interrupt	CF		24	RRRRRRRRR
JA	bDISP	Jump if above	77		16 or 4	
JAE	bDISP	Jump if above or equal	73		16 or 4	
JB	bDISP	Jump if below	72		16 or 4	
JBE	bDISP	Jump if below or equal	76		16 or 4	
JC	(Same as JB, JNAE.)					
JCXZ	bDISP	Jump if (CX)=0	E3		18 or 6	
JE	(Same as JZ.)					
JG	bDISP	Jump if greater	7F		16 or 4	
JGE	bDISP	Jump if greater or equal	7D		16 or 4	
JL	bDISP	Jump if less	7C		16 or 4	
JLE	bDISP	Jump if less or equal	7E		16 or 4	
JMP	bDISP	Direct NEAR jump	EB		7	
JMP	wDISP	Direct NEAR jump	E9		7	
JMP	wDISP, wSEG	EA Direct FAR jump			7	
JMP	EA	Indirect FAR jump	FF	MOD 101 R/M	16+EA	
JMP	EA	Indirect NEAR jump	FF	MOD 100 R/M	7+EA	

Table A-3. Instruction Set in Alphabetic Order of Instruction Mnemonic. (3 of 6.)

Instruction	Operand	Summary	Op Cd	Memory Organization	Clocks	Flags ODITSZAPC
JNA	(Same as JBE.)					
JNB	(Same as JAE.)					
JNBE	(Same as JA.)					
JNG	(Same as JLE.)					
JNGE	(Same as JL.)					
JNL	(Same as JGE.)					
JNLE	(Same as JG.)					
JNO	bDISP	Jump if no overflow	71		16 or 4	
JNP	(Same as JPO.)					
JNS	bDISP	Jump if no sign	79		16 or 4	
JNZ	bDISP	Jump if not zero	75		16 or 4	
JO	bDISP	Jump if overflow	70		16 or 4	
JPE	bDISP	Jump if parity even	7A		16 or 4	
JPO	bDISP	Jump if parity odd	7B		16 or 4	
JS	bDISP	Jump if sign	78		16 or 4	
JZ	bDISP	Jump if zero	74		16 or 4	
LAHF		(AH)=(FLAGS)	9F		4	
LDS	REG,EA	DS:REG=(wEA+2):(wEA)	C5	MOD REGR/M	16+EA	
LEA	REG,EA	(REG)=effective address	8D	MOD REGR/M	2+EA(2)	
LES	REG,EA	ES:REG=(wEA+2):(wEA)	C4	MOD REGR/M	16+EA	
LODSB		Load byte string	AC		12 (9+13/rep)	
LODSW		Load word string	AD		12 (9+13/rep)	
LOCK		Bus lock prefix	F0		2	
LOOP	bDISP	Loop (CX) times	E2		17 or 5	
LOOPE	(Same as LOOPZ.)					
LOOPNE	(Same as LOOPNZ.)					
LOOPNZ	bDISP	Loop (CX) times while not zero	E0		19 or 5	
LOOPZ	bDISP	Loop (CX) times while zero	E1		18 or 6	
MOV	bAddr,AL	(bAddr)=(AL)	A2		10	
MOV	wAddr,AX	(wAddr) = (AX)	A3		10	
MOV	AH,bData	(AH)=bData	B4		4	
MOV	AL,bAddr	(AL)=(bAddr)	A0		10	
MOV	AL,bData	(AL)=bData	B0		4	
MOV	AX,wAddr	(AX)=(wAddr)	A1		10	
MOV	AX,wData	(AX)=wData	B8		4	
MOV	BH,bData	(BH)=bData	B7		4	
MOV	BL,bData	(BL)=bData	B3		4	
MOV	BP,wData	(BP)=wData	BD		4	
MOV	BX,wData	(BX)=wData	BB		4	
MOV	CH,bData	(CH)=bData	B5		4	
MOV	CL,bData	(CL)=bData	B1		4	
MOV	CX,wData	(CX)=wData	B9		4	
MOV	DH,bData	(DH)=bData	B6		4	
MOV	DI,wData	(DI)=wData	BF		4	
MOV	DL,bData	(DL)=bData	B2		4	
MOV	DX,wData	(DX)=wData	BA		4	
MOV	bEA,bData	(bEA)=(bData)	C6	MOD 000 R/M	10+EA	
MOV	wEA,wData	(wEA)=(wData)	C7	MOD 000 R/M	10+EA	
MOV	bEA,bREG	(bEA)=(bREG)	88	MOD REGR/M	9+EA(2)	
MOV	wEA,wREG	(wEA)=(wREG)	89	MOD REGR/M	9+EA(2)	
MOV	wEA,SR	(wEA)=(SR)	8C	MOD 0SR R/M	9+EA(2)	
MOV	bREG,bEA	(bREG)=(bEA)	8A	MOD REGR/M	8+EA(2)	
MOV	wREG,wEA	(wREG)=(wEA)	8B	MOD REGR/M	8+EA(2)	
MOV	SI,wData	(SI)=wData	BE		4	
MOV	SP,wData	(SP)=wData	BC		4	
MOV	SR,wEA	(SR)=(wEA)	8E	MOD 0SR R/M	8+EA(2)	

Table A-3. Instruction Set in Alphabetic Order of Instruction Mnemonic.

Instruction	Operand	Summary	Op Cd	Memory Organization	Clocks	Flags ODITSZAPC
MOVS	(Use MOVSB, MOVSW.)					
MOVSB		Move byte string	A4		18 (9+17/rep)	
MOVSW		Move word string	A5		18 (9+17/rep)	
MUL	bEA	Unsigned multiply by (bEA)	F6	MOD 100 R/M	71	X UUUUX
MUL	wEA	Unsigned multiply by (wEA)	F7	MOD 100 R/M	124	X UUUUX
NEG	bEA	Byte negate bEA	F6	MOD 011 R/M	16+EA	X XXXXS
(Note: Carry Flag is C if destination is 0.)						
NEG	wEA	Negate wEA	F7	MOD 011 R/M	16+EA	X XXXXS
(Note: Carry Flag is C if destination is 0.)						
NOP	(Same as XCHG AX,AX)					
NOT	bEA	Byte invert bEA	F6	MOD 010 R/M	16+EA	
NOT	wEA	Invert wEA	F7	MOD 010 R/M	16+EA	
OR	AL,bData	(AL)=(AL) OR bData	0C		4	C XXUXC
OR	AX,wData	(AX)=(AX) OR wData	0D		4	C XXUXC
OR	bEA,bData	(bEA)=(bEA) OR bData	80	MOD 001 R/M	17+EA	C XXUXC
OR	wEA,wData	(wEA)=(wEA) OR wData	81	MOD 001 R/M	17+EA	C XXUXC
OR	bEA,REG	(bEA)=(bEA) OR (bREG)	08	MOD REGR/M	16+EA(3)	C XXUXC
OR	wEA,REG	(wEA)=(wEA) OR (wREG)	09	MOD REGR/M	16+EA(3)	C XXUXC
OR	REG,bEA	(bREG)=(bREG) OR (bEA)	0A	MOD REGR/M	9+EA(3)	C XXUXC
OR	REG,wEA	(wREG)=(wREG) OR (wEA)	0B	MOD REGR/M	9+EA(3)	C XXUXC
OUT	DX, AL	Byte output (AL) to port (DX)	EE		8	
OUT	DX, AX	Word output (AX) to port (DX)	EF		8	
OUT	bPort,AL	Output (AL) to bPort	E6		10	
OUT	wPort,AX	Output (AX) to wPort	E7		10	
POP	AX	Pop stack to AX	58		8	
POP	BX	Pop stack to BX	5B		8	
POP	BP	Pop stack to BP	5D		8	
POP	CX	Pop stack to CX	59		8	
POP	DI	Pop stack to DI	5F		8	
POP	DS	Pop stack to DS	1F		8	
POP	DX	Pop stack to DX	5A		8	
POP	EA	Pop stack to EA	8F	MOD 000 R/M	17+EA	
POP	ES	Pop stack to ES	07		8	
POP	SI	Pop stack to SI	5E		8	
POP	SP	Pop stack to SP	5C		8	
POP	SS	Pop stack to SS	17		8	
POPF		Pop stack to FLAGS	9D		8	RRRRRRRRR
PUSH	AX	Push (AX) onto stack	50		11	
PUSH	BP	Push (BP) onto stack	55		11	
PUSH	BX	Push (BX) onto stack	53		11	
PUSH	CS	Push (CS) onto stack	0E		11	
PUSH	CX	Push (CX) onto stack	51		11	
PUSH	DI	Push (DI) onto stack	57		11	
PUSH	DS	Push (DS) onto stack	1E		10	
PUSH	DX	Push (DX) onto stack	52		11	
PUSH	EA	Push (EA) onto stack	FF	MOD 110 R/M	16+EA	
PUSH	ES	Push (ES) onto stack	06		10	
PUSH	SI	Push (SI) onto stack	56		11	
PUSH	SP	Push (SP) onto stack	54		11	
PUSH	SS	Push (SS) onto stack	16		11	X XXXXX
PUSHF		Push FLAGS onto stack	9C		10	
RCL	bEA,1	Rotate bEA left thru carry 1 bit	D0	MOD 010 R/M	15+EA	X X
RCL	wEA,1	Rotate wEA left thru carry 1 bit	DI	MOD 010 R/M	15+EA	X X

Table A-3. Instruction Set in Alphabetic Order of Instruction Mnemonic. (5 of 6.)

Instruction	Operand	Summary	Op Cd	Memory Organization	Clocks	Flags ODITSZAPC
MOVS						
MOVSB		Move byte string	A4		18	
MOVSW		Move word string	A5		18	
RCR	bEA,CL	Rotate bEA right thru carry (CL) bits	D2	MOD 011 R/M	20+EA +4/bit	X X
RCR	wEA,CL	Rotate wEA right thru carry (CL) bits	D3	MOD 011 R/M	20+EA +4/bit	X X
RCR	bEA,1	Rotate bEA right thru carry 1 bit	D0	MOD 011 R/M	15+EA	X X
RCR	wEA,1	Rotate wEA right thru carry 1 bit	D1	MOD 011 R/M	15+EA	X X
REP	(Same as REPZ.)					
REPE	(Same as REPZ.)					
REPNE	(Same as REPNZ.)					
REPNZ		Repeat while (CX)≠0 AND (ZF)=0	F2		2	
REPZ		Repeat while (CX)≠0 AND (ZF)=1	F3		2	
RET	wData	FAR return, ADD data to REG SP	CA		17	
RET		FAR return	CB		18	
RET		NEAR return	C3		8	
RET	wData	NEAR return; (SP)=(SP)+ (wData)	C2		12	
ROL	bEA,CL	Rotate bEA left (CL) bits	D2	MOD 000 R/M	20+EA +4/bit	X X
ROL	wEA,CL	Rotate wEA left (CL) bits	D3	MOD 000 R/M	20+EA +4/bit	X X
ROL	bEA,1	Rotate bEA left 1 bit	D0	MOD 000 R/M	15+EA	X X
ROL	wEA,1	Rotate wEA left 1 bit	D1	MOD 000 R/M	15+EA	X X
ROR	bEA,CL	Rotate bEA right (CL) bits	D2	MOD 001 R/M	20+EA +4/bit	X X
ROR	wEA,CL	Rotate wEA right (CL) bits	D3	MOD 001 R/M	20+EA +4/bit	X X
ROR	bEA,1	Rotate bEA right 1 bit	D0	MOD 001 R/M	15+EA	X X
ROR	wEA,1	Rotate wEA right 1 bit	D1	MOD 001 R/M	15+EA	X X
SAHF		(FLAGS)=(AH)	9E		4	RRRRRRRRR
SAL	(Same as SHL.)					
SAR	bEA,CL	Shift signed bEA right (CL) bits	D2	MOD 111 R/M	20+EA +4/bit	X XXUXX
SAR	wEA,CL	Shift signed wEA right (CL) bits	D3	MOD 111 R/M	20+EA +4/bit	X XXUXX
SAR	bEA,1	Shift signed bEA right 1 bit	D0	MOD 111 R/M	15+EA	X XXUXX
SAR	wEA,1	Shift signed wEA right 1 bit	D1	MOD 111 R/M	15+EA	X XXUXX
SBB	AL,bData	(AL)=(AL)-bData-CF	1C		4	X XXXXX
SBB	AL,wData	(AX)=(AX)-wData-CF	1D		4	X XXXXX
SBB	bEA,bData	(bEA)=(bEA)-bData-CF	80	MOD 011 R/M	17+EA	X XXXXX
SBB	bEA,bData	(bEA)=(bEA)-bData-CF	82	MOD 011 R/M	17+EA	X XXXXX
SBB	wEA,wData	(wEA)=(wEA)-wData-CF	81	MOD 011 R/M	17+EA	X XXXXX
SBB	wEA,bData	(wEA)=(wEA)-Ext(bData)-CF	83	MOD 011 R/M	17+EA	X XXXXX
SBB	bEA,REG	(bEA)=(bEA)-(bREG)-CF	18	MOD REG R/M	16+EA(3)	X XXXXX
SBB	wEA,REG	(wEA)=(wEA)-(wREG)-CF	19	MOD REG R/M	16+EA(3)	X XXXXX
SBB	REG,bEA	(bREG)=(bREG)-(bEA)-CF	1A	MOD REG R/M	9+EA(3)	X XXXXX
SBB	REG,wEA	(wREG)=(wREG)-(wEA)-CF	1B	MOD REG R/M	9+EA(3)	X XXXXX
SCASB		Scan byte string	AE		15 (9+15/rep)	X XXXXX
SCASW		Scan word string	AF		15 (9+15/rep)	X XXXXX
SHL	bEA,CL	Shift bEA left (CL) bits	D2	MOD 100 R/M	20+EA +4/bit	X X

Table A-3. Instruction Set in Alphabetic Order of Instruction Mnemonic. (6 of 6.)						
Instruction	Operand	Summary	Op Cd	Memory Organization	Clocks	Flags ODITSZAPC
SHL	wEA,CL	Shift wEA left (CL) bits	D3	MOD 100 R/M	20+EA +4/bit	X X
SHL	bEA,1	Shift bEA left 1 bit	D0	MOD 100 R/M	15+EA	X X
SHL	wEA,1	Shift wEA left 1 bit	D1	MOD 100 R/M	15+EA	X X
SHR	bEA,CL	Shift bEA right (CL) bits	D2	MOD 101 R/M	20+EA +4/bit	X X
SHR	wEA,CL	Shift wEA right (CL) bits	D3	MOD 101 R/M	20+EA +4/bit	X X
SHR	bEA,1	Shift bEA right 1 bit	D0	MOD 101 R/M	15+EA	X X
SHR	wEA,1	Shift wEA right 1 bit	D1	MOD 101 R/M	15+EA	X X
SS:		SS segment override	36		2	
STC		Set carry flag	F9		2	S
STD		Set direction flag	FD		2	C
STI		Set interrupt flag	FB		2	S
STOSB		Store byte string	AA		11 (9+10/rep)	
STOSW		Store word string	AB		11 (9+10/rep)	
SUB	AL,bData	(AL)=(AL)-bData	2C		4	X XXXXX
SUB	AX,wData	(AX)=(AX)-wData	2D		4	X XXXXX
SUB	bEA,bData	(bEA)=(bEA)-bData	80	MOD 101 R/M	17+EA	X XXXXX
SUB	bEA,bData	(bEA)=(bEA)-bData	82	MOD 101 R/M	17+EA	X XXXXX
SUB	wEA,wData	(wEA)=(wEA)-wData	81	MOD 101 R/M	17+EA	X XXXXX
SUB	wEA,bData	(wEA)=(wEA)-Ext(bData)	83	MOD 101 R/M	17+EA	X XXXXX
SUB	bEA,REG	(bEA)=(bEA)-(bREG)	28	MOD REGR/M	16+EA(3)	X XXXXX
SUB	wEA,REG	(wEA)=(wEA)-(wREG)	29	MOD REGR/M	16+EA(3)	X XXXXX
SUB	REG,bEA	(bREG)=(bREG)-(bEA)	2A	MOD REGR/M	9+EA(3)	X XXXXX
SUB	REG,wEA	(wREG)=(wREG)-(wEA)	2B	MOD REGR/M	9+EA(3)	X XXXXX
TEST	AL,bData	FLAGS=(AL) TEST (bData)	A8		4	X XXUXC
TEST	AX,bData	FLAGS=(AX) TEST (wData)	A9		4	X XXUXC
TEST	bEA,bData	FLAGS=(bEA) TEST bData	F6	MOD 000 R/M	10+EA	C XXUXC
TEST	wEA,wData	FLAGS=(wEA) TEST wData	F7	MOD 000 R/M	10+EA	C XXUXC
TEST	bEA,bREG	FLAGS=(bEA) TEST (bREG)	84	MOD REGR/M	9+EA(3)	C XXUXC
TEST	wEA,wREG	FLAGS=(wEA) TEST (wREG)	85	MOD REGR/M	9+EA(3)	C XXUXC
WAITX		Wait for TEST signal	9B		3+WAITX	
XCHG	AX,AX	NOP	90		3	
XCHG	AX,BP	Exchange (AX), (BP)	95		3	
XCHG	AX,BX	Exchange (AX), (BX)	93		3	
XCHG	AX,CX	Exchange (AX), (CX)	91		3	
XCHG	AX,DI	Exchange (AX), (DI)	97		3	
XCHG	AX,DX	Exchange (AX), (DX)	92		3	
XCHG	AX,SI	Exchange (AX), (SI)	96		3	
XCHG	AX,SP	Exchange (AX), (SP)	94		3	
XCHG	bREG,bEA	Exchange bREG, bEA	86	MOD REGR/M	17+EA(4)	
XCHG	wREG,wEA	Exchange wREG, wEA	87	MOD REGR/M	17+EA(4)	
XLAT	TABLE	Translate using (BX)	D7		11	
XOR	AL,bData	(AL)=(AL) XOR bData	34		4	C XXUXC
XOR	AX,wData	(AX)=(AX) XOR wData	35		4	C XXUXC
XOR	bEA,bData	(bEA)=(bEA) XOR bData	80	MOD 101 R/M	17+EA	C XXUXC
XOR	wEA,wData	(wEA)=(wEA) XOR, wData	81	MOD 101 R/M	17+EA	C XXUXC
XOR	bEA,REG	(bEA)=(bEA) XOR (bREG)	30	MOD REGR/M	16+EA(3)	C XXUXC
XOR	wEA,REG	(wEA)=(wEA) XOR (wREG)	31	MOD REGR/M	16+EA(3)	C XXUXC
XOR	REG,bEA	(bREG)=(bREG) XOR (bEA)	32	MOD REGR/M	9+EA(3)	C XXUXC
XOR	REG,wEA	(wREG)=(wREG) XOR (wEA)	33	MOD REGR/M	9+EA(3)	C XXUXC

Appendix B: RESERVED WORDS

A
AAA
AAD
AAM
AAS
ABS
ADC
ADD
AH
AL
AND
ASSUME
AT
AX
BH
BL
BP
BX
BYTE
CALL
CBW
CH
CL
CLC
CLD
CLI
CMC
CMP
CMPS
CMPSB
CMPSW
COMMON
CS
CWD
CX
DAA
DAS
DB
DD
DEC
DH
DI
DIV
DL
DS
DUP
DW
DWORD
DX
EJECT
END
ENDP
ENDS
EQ
EQU
ES
ESC
EVEN
EXTRN
FAC
FALC
FAR
GE
GEN
GENONLY
GROUP
GT
HIGH
HLT
IDIV
IMUL
IN
INC
INCLUDE
INT
INTO
IRET
JA
JAE
JB
JBCZ
JBE
JC
JE
JGE
JL
JLE
JMP
JNA
JNAE
JNB
JNBE
JNC
JNE
JNG
JNGE
JNLE
JNO
JNP
JNS
JNZ
JO
JP
JPE
JPO
JS
JZ
LABEL
LAHF
LDS
LE
LEA
LENGTH
LES
LIST
LOCK
LODS
LODSB
LODSW
LOOP
LOOPE
LOOPNZ
LOOPZ
LOW
LT
MASK
MEMORY
MOD
MOV
MOVS
MOVSB
MOVSW
MUL
NAME
NE
NEAR
NEG
NIL
NOGEN
NOLIST
NOPAGING
NOT
NOTHING
NOXREF
OFFSET
OR
ORG
OUT
PAGE
PAGELENGTH
PAGEWIDTH
PAGING
PARA
POP
POPF
PROC
PTR
PUBLIC
PURGE
PUSH
PUSHF
RCL
RCR
RECORD
REPE
REPNE
REPNZ
REPZ
RESTORE
RET
ROR
SAL
SAR
SAVE
SBB
SCAS
SCASB
SCASW
SEG
SEGMENT
SHL
SHORT
SHR
SI
SIZE
SP
SS
STACK
STC
STD
STI
STOS
STOSB
STOSW
SUB
TEST
THIS
TITLE
TYPE
WAIT
WIDTH
WORD
XCHG
XLAT
XLATB
XOR
?
??SEG