APPENDIX C: ASSEMBLER DIRECTIVES AND NAMING RULES

This appendix consists of two sections. The first section describes some of the most widely used directives in 80x86 Assembly language programming. In the second section Assembly language rules and restrictions for names and labels are discussed and a list of reserved words is provided.

SECTION C.1: 80x86 ASSEMBLER DIRECTIVES

Directives, or as they are sometimes called, pseudo-ops or pseudo-instructions, are used by the assembler to help it translate Assembly language programs into machine language. Unlike the microprocessor's instructions, directives do not generate any opcode; therefore, no memory locations are occupied by directives in the final ready-to-run (exe) version of the assembly program. To summarize, directives give directions to the assembler program to tell it how to generate the machine code; instructions are assembled into machine code to give directions to the CPU at execution time. The following are descriptions of the some of the most widely used directives for the 80x86 assembler. They are given in alphabetical order for ease of reference.

ASSUME

The ASSUME directive is used by the assembler to associate a given segment's name with a segment register. This is needed for instructions that must compute an address by combining an offset with a segment register. One ASSUME directive can be used to associate all the segment registers. For example:

```
ASSUME      CS:name1,DS:name2,SS:name3,ES:name4
```

where name1, name2, and so on, are the names of the segments. The same result can be achieved by having one ASSUME for each register:

```
ASSUME      CS:name1
ASSUME      DS:name2
ASSUME      SS:name3
ASSUME      ES:nothing
ASSUME      nothing
```

The key word "nothing" can be used to cancel a previous ASSUME directive.

DB (Define Byte)

The DB directive is used to allocate memory in byte-sized increments. Look at the following examples:

```
DATA1    DB    23
DATA2    DB    45,97H,10000011B
DATA3    DB    'The planet Earth'
```

In DATA1 a single byte is defined with initial value 23. DATA2 consists of several values in decimal (45), hex (97H), and binary (10000011B). Finally, in DATA3, the DB directive is used to define ASCII characters. The DB directive is normally used to define ASCII data. In all the examples above, the address location for each value is assigned by the assembler. We can assigned a specific offset address by the use of the ORG directive.

DD (Define Doubleword)

To allocate memory in 4-byte (32-bit) increments, the DD directive is used. Since word-sized operands are 16 bits wide (2 bytes) in 80x86 assemblers, a doubleword is 4 bytes.

```
VALUE1   DD    4563F57H
RESULT   DD    ?           ;RESERVE 4-BYTE LOCATION
DAT4     DD    25000000
```

It must be noted that the values defined using the DD directive are placed in memory by the assembler in low byte to low address and high byte to high address order. This convention is referred to as little endian. For example, assuming that offset address 0020 is assigned to VALUE1 in the example above, each byte will reside in memory as follows:

```
DS:20=(57)
DS:21=(3F)
DS:22=(56)
DS:23=(04)
```

DQ (Define Quadword)

To allocate memory in 8-byte increments, the DQ directive is used. In the 80x86 a word is defined as 2 bytes; therefore, a quadword is 8 bytes.

```
DAT_64B  DQ    5677DD4EE4FF45AH
DAT8     DQ    10000000000000
```

DT (Define Tenbytes)

To allocate packed BCD data, 10 bytes at a time, the DT directive is used. This is widely used for memory allocation associated with BCD numbers.

```
DATA     DT    399977653419974
```

Notice there is no H for the hexadecimal identifier following the number. This is a characteristic particular to the DT directive. In the case of other directives (DB, DW, DD, DQ), if there is no H at the end of the number, it is assumed to be in decimal and will be converted to hex by the assembler. Remember that the little endian convention is used to place the bytes in memory, with the least significant byte going to the low address and the most significant byte to the high address. DT can also be used to allocated decimal data if "d" is placed after the number:

```
DATA     DT    65535d ;stores hex FFFF in a 10-byte location
```

DUP (Duplicate)

The DUP directive can be used to duplicate a set of data a certain number of times instead of having to write it over and over.

```
DATA1      DB      20 DUP (99)          ;DUPLICATE 99 20 TIMES
DATA2      DW      6 DUP (5555H)        ;DUPLICATE 5555H 6 TIMES
DATA3      DB      10 DUP (?)           ;RESERVE 10 BYTES
DATA4      DB      5 DUP (5 DUP (0))    ;25 BYTES  INITIALIZED TO ZERO
DATA5      DB      10 DUP (00,FFH)      ;20 BYTES ALTERNATE 00, FF
```

DW (Define Word)

To allocate memory in 2-byte (16-bit) increments, the DW directive is used. In the 80x86 family, a word is defined as 16 bits.

```
DATAW_1  DW      5000
DATAW_2  DW      7F6BH
```

Again, in terms of placing the bytes in memory the little endian convention is used with the least significant byte going to the low address and the most significant byte going to the high address.

END

Every program must have an entry point. To identify that entry point the assembler relies on the END directive. The label for the entry and end point must match.

```
HERE:     MOV       AX,DATASEG   ;ENTRY POINT OF THE PROGRAM
          ...
          ...
          END       HERE         ;EXIT POINT OF THE PROGRAM
```

If there are several modules, only one of them can have the entry point, and the name of that entry point must be the same as the name put for the END directive as shown below:

```
;from the main program:
              EXTRN  PROG1:NEAR
              ...
MAIN_PRO:     MOV    AX,DATASG           ;THE ENTRY POINT
              MOV    DS,AX
              ...
              CALL   PROG1
              ...
              END    MAIN_PRO            ;THE EXIT POINT

;from the module PROG1:
              PUBLIC PROG1
PROG1         PROC
              ...
              RET                        ;RETURN TO THE MAIN MODULE
PROG1         ENDP
              END                        ;NO LABEL IS GIVEN
```

Notice the following points about the above code:

1. The entry point must be identified by a name. In the example above the entry point is identified by the name MAIN_PRO.
2. The exit point must be identified by the same name given to the entry point, MAIN_PRO.
3. Since a given program can have only one entry point and exit point, all modules called (either from main or from the submodules) must have directive END with nothing after it.

ENDP (see the PROC directive)

ENDS (see the SEGMENT and STRUCT directives)

EQU (Equate)

To assign a fixed value to a name, one uses the EQU directive. The assembler will replace each occurrence of the name with the value assigned to it.

```
FIX_VALU  EQU    1200
PORT_A    EQU    60H
COUNT     EQU    100
MASK_1    EQU    00001111B
```

Unlike data directives such as DB, DW, and so on, EQU does not assign any memory storage; therefore, it can be defined at any time at any place, and can even be used within the code segment.

EVEN

The EVEN directive forces memory allocation to start at an even address. This is useful due to the fact that in 8086, 286, and 386SXmicroprocessors, accessing a 2-byte operand located at an odd address takes extra time. The use of the EVEN directive directs the assembler to assign an even address to the variable.

```
          ORG    0020H
DATA_1    DB     34H
          EVEN
DATA_2    DW     7F5BH
```

The following shows the contents of memory locations:

```
DS:0020 = (34)
DS:0021 = (? )
DS:0022 = (5B)
DS:0023 = (7F)
```

Notice that the EVEN directive caused memory location DS:0021 to be bypassed, and the value for DATA_2 is placed in memory starting with an even address.

EXTRN (External)

The EXTRN directive is used to indicate that certain variables and names used in a module are defined by another module. In the absence of the EXTRN directive, the assembler would search for the definition and give an error when it couldn't find it. The format of this directive is

```
          EXTRN  name1:typea [,name2:typeb]
```

where type will be NEAR or FAR if name refers to a procedure, or will be BYTE, WORD, DWORD, QWORD, TBYTE if name refers to a data variable.

```
;from the main program:
          EXTRN  PROG1:NEAR
          PUBLIC DATA1
          ...
MAIN_PRO  MOV    AX,DATASG          ;THE ENTRY POINT
          MOV    DS,AX
          ...
          CALL   PROG1
          ...
          END    MAIN_PRO           ;THE EXIT POINT

;PROG1 is located in a different file:
          EXTRN DATA1:WORD
          PUBLIC PROG1
PROG1     PROC
          ...
          MOV    BX,DATA1
          ...
          RET                       ;RETURN TO THE MAIN MODULE
PROG1     ENDP
          END
```

Notice that the EXTRN directive is used in the main procedure to identify PROG1 as a NEAR procedure. This is needed because PROG1 is not defined in that module. Correspondingly, PROG1 is defined as PUBLIC in the module where it is defined. EXTRN is used in the PROG1 module to declare that operand DATA1, of size WORD, has been defined in another module. Correspondingly, DATA1 is declared as PUBLIC in the calling module.

GROUP

The GROUP directive causes the named segments to be linked into the same 64K byte segment. All segments listed in the GROUP directive must fit into 64K bytes. This can be used to combine segements of the same type, or different classes of segments. An example follows:

```
SMALL_SYS    GROUP          DTSEG,STSEG,CDSEG
```

The ASSUME directive must be changed to make the segment registers point to the group:

```
ASSUME    CS:SMALL_SYS,DS:SMALL_SYS,SS:SMALL_SYS
```

The group will be listed in the list file, as shown below:

Segments and Groups:

Name	Length	Align	Combine	Class
SMALL_SYS	GROUP			
STSEG	0040	PARA	NONE	
DTSEG	0024	PARA	NONE	
CDSEG	005A	PARA	NONE	

INCLUDE

When there is a group of macros written and saved in a separate file, the INCLUDE directive can be used to bring them into another file. In the program listing (.lst file), these macros will be identified by the symbol "C" (or "+" in some versions of MASM) before each instruction to indicate that they are copied to the present file by the INCLUDE directive.

LABEL

The LABEL directive allows a given variable or name to be referred to by multiple names. This is often used for multiple definition of the same variable or name. The format of the LABEL directive is

```
        name    LABEL  type
```

where type may be BYTE, WORD, DWORD, QWORD. For example, a variable name DATA1 is defined as a word and also needs to be accessed as 2 bytes, as shown in the following:

```
DATA_B    LABEL    BYTE
DATA1     DW       25F6H

          MOV      AX,DATA1            ;AX=25F6H
          MOV      BL,DATA_B           ;BL=F6H
          MOV      BH,DATA_B +1        ;BH=25H
```

The following shows the LABEL directive being used to allow accessing a 32-bit data item in 16-bit portions.

```
DATA_16   LABEL    WORD
DATDD_4   DD       4387983FH
          ...
          MOV      AX,DATA_16          ;AX=983FH
          MOV      DX,DATA_16 + 2      ;DX=4387H
```

The following shows its use in a JMP instruction to go to a different code segment.

```
          ....
          JMP      PROG_A
          ....
PROG_A    LABEL    FAR
INITI:    MOV      AL,12H
          OUT      PORT,AL
```

In the program above the address assigned to the names "PROG_A" and "INITI" are exactly the same. The same function can be achieved by the following:

```
          JMP  FAR PTR INITI
```

LENGTH

The LENGTH operator returns the number of items defined by a DUP operand. See the SIZE directive for an example.

OFFSET

To access the offset address assigned to a variable or a name, one uses the OFFSET directive. For example, the OFFSET directive was used in the following example to get the offset address assigned by the assembler to the variable DATA1:

```
        ORG     5600H
DATA1   DW      2345H
        ...
        MOV     SI,OFFSET DATA1     ;SI=OFFSET OF DATA1 = 5600H
```

Notice that this has the same result as "LEA SI,DATA1".

ORG (Origin)

The ORG directive is used to assign an offset address for a variable or name. For example, to force variable DATA1 to be located starting from offset address 0020, one would write

```
        ORG     0020H
DATA1   DW41F2H
```

This ensures the offset addresses of 0020 and 0021 with contents 0020H = (F2) and 0021H = (41).

PAGE

The PAGE directive is used to make the ".lst" file print in a specific format. The format of the PAGE directive is

```
PAGE [lines],[columns]
```

The default listing (meaning that no PAGE directive is coded) will have 66 lines per page with a maximum of 80 characters per line. This can be changed to 60 and 132 with the directive "PAGE 60,132". The range for number of lines is 10 to 255 and for columns is 60 to 132. A PAGE directive with no numbers will generate a page break.

PROC and ENDP (Procedure and End Procedure)

Often, a group of Assembly language instructions will be combined into a procedure so that it can be called by another module. The PROC and ENDP directives are used to indicate the beginning and end of the procedure. For a given procedure the name assigned to PROC and ENDP must be exactly the same.

```
name1   PROC    [attribute]
        ...
name1   ENDP
```

There are two choices for the attribute of the PROC: NEAR or FAR. If no attribute is given, the default is NEAR. When a NEAR procedure is called, only IP is saved since CS of the called procedure is the same as the calling program. If a FAR procedure is called, both IP and CS are saved since the code segment of the called procedure is different from the calling program.

PTR (Pointer)

The PTR directive is used to specify the size of the operand. Among the options for size are BYTE, WORD, DWORD, and QWORD. This directive is used in many different ways, the most common of which are explained below.

1. PTR can be used to allow an override of a previously defined data directive.

```
DATA1   DB      23H,7FH,99H,0B2H
DATA2   DW      67F1H
DATA3   DD      22229999H
        ...
        MOV     AX, WORD PTR DATA1          ;AX=7F23
        MOV     BX, WORD PTR DATA1 + 2      ;BX,B299H
```

Although DATA1 was initially defined as DB, it can be accessed using the WORD PTR directive.

```
        MOV     AL, BYTE PTR DATA2   ;AL=F1H
```

In the above code, notice that DATA2 was defined as WORD but it was accessed as BYTE with the help of BYTE PTR. If this had been coded as "MOV AL,DATA2", it would generate an error since the sizes of the operands do not match.

```
        MOV     AX, WORD PTR DATA3              ;AX=9999H
        MOV     DX, WORD PTR DATA3 + 2      ;DX=2222H
```

DATA3 was defined as a 4-byte operand but registers are only 2 bytes wide. The WORD PTR directive solved that problem.

2. The PTR directive can be used to specify the size of a directive in order to help the assembler translate the instruction.

```
        INC     [DI]            ;will cause an error
```

This instruction was meant to increment the contents of the memory location(s) pointed at by [DI]. How does the assembler know whether it is a byte operand, word operand, or doubleword operand? Since it does not know, it will generate an error. To correct that, use the PTR directive to specify the size of the operand as shown next.

```
INC     BYTE PTR [SI]       ;increment a byte pointed by SI
or
INC     WORD PTR [SI]               ;increment a word pointed by SI
or
INC     DWORD PTR [SI]      ;increment a doubleword pointed by SI
```

3. The PTR directive can be used to specify the distance of a jump. The options for the distance are FAR and NEAR.

```
JMP     FAR PTR INTI   ;ensures that it will be a 5-byte instruction
...
```

```
INITI:  MOV   AX,1200
```

See the LABEL directive to find out how it can be used to achieve the same result.

PUBLIC

To inform the assembler that a name or symbol will be referenced by other modules, it is marked by the PUBLIC directive. If a module is referencing a variable outside itself, that variable must be declared as EXTRN. Correspondingly, in the module where the variable is defined, that variable must be declared as PUBLIC in order to allow it to be referenced by other modules. See the EXTRN directive for examples of the use of both EXTRN and PUBLIC.

SEG (Segment Address)

The SEG operator is used to access the address of the segment where the name has been defined.

```
DATA1    DW      2341H
         ...
         MOV     AX,SEG DATA1 ;AX=SEGMENT ADDRESS OF DATA1
```

This is in contrast to the OFFSET directive, which accesses the offset address instead of the segment.

SEGMENT and ENDS

In full segment definition these two directives are used to indicate the beginning and the end of the segment. They must have the same name for a given segment definition. See the following example:

```
DATSEG   SEGMENT
DATA1    DB      2FH
DATA2    DW      1200
DATA3    DD      99999999H
DATSEG   ENDS
```

There are several options associated with the SEGMENT directive, as follows:

```
name1  SEGMENT  [align] [combine] [class]

name1  ENDS
```

ALIGNMENT: When several assembled modules are linked together, this indicates where the segment is to begin. There are many options, including PARA (paragraph = 16 bytes), WORD, and BYTE. If PARA is chosen, the segment starts at a hex address divisible by 10H. PARA is the default alignment. In this alignment, if a segment for a module finished at 00024H, the next segment will start at address 00030H, leaving from 00025 to 0002F unused. If WORD is chosen, the segment is forced to start at a word boundary. In BYTE alignment, the segment starts at the next byte and no memory is wasted. There is also the PAGE option, which aligns segments along the 100H (256) byte boundary. While all these options are supported by many assemblers, such as MASM and TASM, there is another option supported only by assemblers that allow system development. This option is AT. The AT option allows the program to assign a physical address. For example, to burn a program into ROM starting at physical address F0000, code

```
ROM_CODE    SEGMENT     AT F000H
```

Due to the fact that option AT allows the programmer to specify a physical address that conflicts with DOS's memory management responsibility, many assemblers such as MASM will not allow option AT.

COMBINE TYPE: This option is used to merge together all the similar segments to create one large segment. Among the options widely used are PUBLIC and STACK. PUBLIC is widely used in code segment definitions when linking more than one module. This will consolidate all the code segments of the various modules into one large code segment. If there is only one data segment and that belongs to the main module, there is no need to define it as PUBLIC since no other module has any data segment to combine with. However, if other modules have their own data segments, it is recommended that they be made PUBLIC to create a single data segment when they are linked. In the absence of that, the linker would assume that each segment is private and they would not be combined with other similar segments (codes with codes and data with data). Since there is only one stack segment, which belongs to the main module, there is no need to define it as PUBLIC. The STACK option is used only with the stack segment definition and indicates to the linker that it should combine the user's defined stack with the system stack to create a single stack for the entire program. This is the stack that is used at run time (when the CPU is actually executing the program).

CLASS NAME: Indicates to the linker that all segments of the same class should be placed next to each other by the LINKER. Four class names commonly used are 'CODE', 'DATA', 'STACK', and 'EXTRA'. When this attribute is used in the segment definition, it must be enclosed in single apostrophes in order to be recognized by the linker.

SHORT

In a direct jump such as "JMP POINT_A", the assembler has to choose either the 2-byte or 3-byte format. In the 2-byte format, one byte is the opcode and the second byte is the signed number displacement value added to the IP of the instruction immediately following the JMP. This displacement can be anywhere between −128 and +127. A negative number indicates a backward JMP and a positive number a forward JMP. In the 3-byte format the first byte is the opcode and the next two bytes are for the signed number displacement value, which can range from −32,768 to 32,767. When assembling a program, the assembler makes two passes through the program. Certain tasks are done in the first pass and others are left to the second pass to complete. In the first pass the assembler chooses the 3-byte code for the JMP. After the first pass is complete, it will know the target address and fill it in during the second pass. If the target address indicates a short jump (less than 128) bytes away, it fills the last byte with NOP. To inform the assembler that the target address is no more than 128 bytes away, the SHORT directive can be used. Using the SHORT directive makes sure that the JMP is a 2-byte instruction and not 3-byte with 1 byte as NOP code. The 2-byte JMP requires 1 byte less memory and is executed faster.

SIZE

The size operator returns the total number of bytes occupied by a name. The three directives LENGTH, SIZE, and TYPE are somewhat related. Below is a description of each one using the following set of data defined in a data segment:

```
DATA1  DQ    ?
DATA2  DW    ?
DATA3  DB    20 DUP (?)
DATA4  DW    100 DUP (?)
DATA5  DD    10 DUP (?)
```

TYPE allows one to know the storage allocation directive for a given variable by providing the number of bytes according to the following table:

bytes	
1	DB
2	DW
4	DD
8	DQ
10	DT

For example:

```
MOV     BX,TYPE DATA2      ;BX=2
MOV     DX,TYPE DATA1      ;DX=8
MOV     AX,TYPE DATA3      ;AX=1
MOV     CX,TYPE DATA5      ;CX=4
```

When a DUP is used to define the number of entries for a given variable, the LENGTH directive can be used to get that number.

```
MOV     CX,LENGTH DATA4    ;CX=64H        (100 DECIMAL)
MOV     AX,LENGTH DATA3    ;AX=14H        (20 DECIMAL)
MOV     DX,LENGTH DATA5    ;DX=0A(10 DECIMAL)
```

If the defined variable does not have any DUP in it, the LENGTH is assumed to be 1.

```
MOV     BX,LENGTH DATA1    ;BX=1
```

SIZE is used to determine the total number of bytes allocated for a variable that has been defined with the DUP directive. In reality the SIZE directive basically provides the product of the TYPE times LENGTH.

```
MOV     DX, SIZE DATA4     ;DX=C8H=200 (100 x 2=200)
MOV     CX, SIZE DATA5     ;CX=28H=40 (4 x 10=40)
```

STRUC (Structure)

The STRUC directive indicates the beginning of a structure definition. It ends with an ENDS directive, whose label matches the STRUC label. Although the same mnemonic ENDS is used for end of segment and end of structure, the assembler knows which is meant by the context. A structure is a collection of data types that can be accessed either collectively by the structure name or individually by the labels of the data types within the structure. A structure type must first be defined and then variables in the data segment may be allocated as that structure type. Looking at the following example, the data directives between STRUC and ENDS declare what structure ASC_AREA looks like. No memory is allocated for such a structure definition. Immediately below the structure definition is the label ASC_INPUT, which is declared to be of type ASC_AREA. Memory is allocated for the variable ASC_INPUT. Notice in the code segment that ASC_INPUT can be accessed either in its entirety or by its component parts. It is accessed as a whole unit in "MOV DX,OFFSET ASC_INPUT". Its component parts are accessed by the variable name followed by a period, then the component's name. For example, "MOV BL,ASC_INPUT.ACT_LEN" accesses the actual length field of ASC_INPUT.

```
;from the data segment:
ASC_AREA    STRUC                       ;defines struc for string input
MAX_LEN     DB      6                   ; maximum length of input string
ACT_LEN     DB      ?                   ; actual length of input string
ASC_NUM     DB      6 DUP (?)           ; input string
ASC_AREA    ENDS                        ;end struc definition
ASC_INPUT   ASC_AREA      <>            ;allocates memory for struc

;from the code segment:
            ...
GET_ASC:    MOV     AH,0AH
            MOV     DX,OFFSET ASC_INPUT
            INT     21H
            ...
            MOV     SI,OFFSET ASC_INPUT.ASC_NUM  ;SI points to ASCII num
            MOV     BL,ASC_INPUT.ACT_LEN         ;BL holds string length
            ...
```

TITLE

The TITLE directive instructs the assembler to print the title of the program on top of each page of the ".lst" file. What comes after the TITLE pseudo-instruction is up to the programmer, but it is common practice to put the name of the program as stored on the disk right after the TITLE pseudo-instruction and then a brief description of the function of the program. Whatever is placed after the TITLE pseudo-instruction cannot be more than 60 ASCII characters (letters, numbers, spaces, punctuation).

TYPE

The TYPE operator returns the number of bytes reserved for the named data object. See the SIZE directive for examples of its use.

SECTION C.2: RULES FOR LABELS AND RESERVED NAMES

Labels in 80x86 Assembly language for MASM 5.1 and higher must follow these rules:

1. Names can be composed of:
   alphabetic characters: A - Z and a - z
   digits: 0 - 9
   special characters: "?" "." "@" "_" "$"

2. Names must begin with an alphabetic or special character. Names cannot begin with a digit.

3. Names can be up to 31 characters long.

4. The special character "." can only be used as the first character.

5. Uppercase and lowercase are treated the same. "NAME1" is treated the same as "Name1" and "name1".

Assembly language programs have five types of labels or names:

1. Code labels, which give symbolic names to instructions so that other instructions (such as jumps) may refer to them

2. Procedure labels, which assign a name to a procedure

3. Segment labels, which assign a name to a segment

4. Data labels, which give names to data items

5. Labels created with the LABEL directive

Code labels

These labels will be followed by a colon and have the type NEAR. This enables other instructions within the code segment to refer to the instruction. The labels can be on the same line as the instruction:

```
          ...
ADD_LP:   ADD  AL,[BX]          ;label is on same line as the instruction
          ...
          ...
          LOOP ADD_LP
```

or on a line by themselves:

```
          ...
ADD_LP:                         ;label is on a line by itself
          ADD  AL,[BX]          ;ADD_LP refers to this instruction
          ...
          ...
          LOOP ADD_LP
```

Procedure labels

These labels assign a symbolic name to a procedure. The label can be NEAR or FAR. When using full segment definition, the default type is NEAR. When using simplified segment definition, the type will be NEAR for compact or small models but will be FAR for medium, large, and huge models. For more information on procedures, see PROC in Section C.1.

Segment labels

These labels give symbolic names to segments. The name must be the same in the SEGMENT and ENDS directives. See SEGMENT in Section C.1 for more information. Example:

```
DAT_SG   SEGMENT
SUM      DW       ?
DAT_SG   ENDS
```

Data labels

These labels give symbolic names to data items. This allows them to be accessed by instructions. Directives DB, DW, DD, DQ, and DT are used to allocate data. Examples:

```
DATA1    DB       43H
DATA2    DB       F2H
SUM      DW       ?
```

Labels defined with the LABEL directive

The LABEL directive can be used to redefine a label. See LABEL in Section C.1 for more information.

Reserved Names

The following is a list of reserved words in 80x86 Assembly language programming. These words cannot be used as user-defined labels or variable names.

Register Names:

AH	AL	AX	BH	BL	BP	BX	CH	CL	CS	CX	DH
DI	DL	DS	DX	ES	SI	SP	SS				

Instructions:

AAA	AAD	AAM	AAS	ADC	ADD
AND	CALL	CBW	CLC	CLD	CLI
CMC	CMP	CMPS	CWD	DAA	DAS
DEC	DIV	ESC	HLT	IDIV	IMUL
IN	INC	INT	INTO	IRET	JA
JAE	JB	JBE	JCXZ	JE	JG
JGE	JL	JLE	JMP	JNA	JNAE
JNB	JNBE	JNE	JNG	JNGE	JNL
JNLE	JNO	JNP	JNS	JNZ	JO
JP	JPE	JPO	JS	JZ	LAHF
LDS	LEA	LES	LOCK	LODS	LOOP
LOOPE	LOOPNE	LOOPNZ	LOOPZ	MOV	MOVS
MUL	NEG	NIL	NOP	NOT	OR
OUT	POP	POPF	PUSH	PUSHF	RCL
RCR	REP	REPE	REPNE	REPNZ	REPZ
RET	ROL	ROR	SAHF	SAL	SAR
SBB	SCAS	SHL	SHR	STC	STD
STI	STOS	SUB	TEST	WAIT	XCHG
XLAT	XOR				

Assembler operators and directives

$ * + -	. / = ?	[ ]			
ALIGN	ASSUME	BYTE	COMM	COMMENT	DB
DD	DF	DOSSEG	DQ	DS	DT
DW	DWORD	DUP	ELSE	END	ENDIF
ENDM	ENDS	EQ	EQU	EVEN	EXITM
EXTRN	FAR	FWORD	GE	GROUP	GT
HIGH	IF	IFB	IFDEF	IFDIF	IFE
IFIDN	IFNB	IFNDEF	IF1	IF2	INCLUDE
INCLUDELIB	IRP	IRPC	LABEL	LE	LENGTH
LINE	LOCAL	LOW	LT	MACRO	MASK
MOD	NAME	NE	NEAR	NOTHING	OFFSET
ORG	PAGE	PROC	PTR	PUBLIC	PURGE
QWORD	RECORD	REPT	REPTRD	SEG	SEGMENT
SHORT	SIZE	STACK	STRUC	SUBTTL	TBYTE
THIS	TITLE	TYPE	WIDTH	WORD	
.186	.286	.286P	.287	.386	.386P
.387	.8086	.8087	.ALPHA	.CODE	.CONST
.CREF	.DATA	.DATA?	.ERR	.ERR1	.ERR2
.ERRB	.ERRDEF	.ERRDIF	.ERRE	.ERRIDN	.ERRNB
.ERRNDEF	.ERRNZ	.FARDATA	.FARDATA?	.LALL	.LFCOND
.LIST	.MODEL	%OUT	.RADIX	.SALL	.SEQ
.SFCOND	.STACK	.TFCOND	.TYPE	.XALL	.XCREF
.XLIST					