ASM286 Assembly Language Reference

April 18, 2020

Contents

Chapter 1

Introduction

About This Manual

ASM286 supports the 80286 and 8086 microprocessors, and 80287 and 8087 floating-point coprocessors. Throughout this manual, the word "processor" refers to any of the above microprocessors and the words "floating-point coprocessor" refer to any of the above coprocessors.

This manual is a reference for the ASM286 assembly language. It assumes that you are familiar with assembly language programming and 8086/80286 processor architecture. If you aren't, see the Intel 80286 Programmer's Reference Manual.

About This Chapter

This chapter introduces the assembly language. It has three major sections:

- Lexical Elements

  This section describes the assembler character set, tokens, separators, identifiers, comments, and the difference between source file lines and logical statement lines.

- Statements

  This section introduces the assembler directives, processor instruction set, and floating-point instruction set.

- Program Structure

This section provides a template for assembler programs together with a simple example program (see Appendix B for another example program). It summarizes the essential parts of every ASM286 program.

Lexical Elements

This section describes the lexical elements of the assembly language, except for its keywords and reserved words. See also: Keywords and reserved words, Appendix C

Character Set

The assembler character set is a subset of the ASCII character set. Each character in a source file should be one of the following:

Alphanumerics:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
0123456789
```

SpecialCharacters:

```
+-*/()[]<>;'."_:?@$&
```

Logical Delimiters: space, tab, and newline

If a program contains any character that is not in the preceding set, the assembler treats the character as a logical space.

Uppercase and lowercase letters are not distinguished from each other except in character strings. For example, xyz and XYZ are interchangeable, but 'xyz' and 'XYZ' are not equivalent character strings.

The special characters and combinations of special characters have particular meanings in a program, as described throughout this manual.

See also: ASCII character set, Appendix D

Tokens and Separators

A token is the smallest meaningful unit of a source program, much as words are the smallest meaningful units of a sentence. A token is one of the following:

- An end of statement

- A delimiter
- An identifier
- A constant
- An assembler keyword or reserved word

A separator that is a logical space or a delimiter must be specified between two adjacent tokens that are identifiers, constants, keywords, and/or reserved words. The most commonly used separator is the space character.

The end of statement token must be specified between two adjacent statements. The most commonly used statement terminator is the newline character.

See also: Constants, Chapter 4 keywords and reserved words, Appendix C

Logical Spaces

Any unbroken sequence of spaces can be used wherever a single space character is valid. Horizontal tabs are also used as token separators. The assembler interprets horizontal tabs as a single logical space. However, tabs are reproduced as multiple space characters in the print (listing) file to maintain the appearance of the source file.

See also: Print file, ASM286 Macro Assembler Operating Instructions

Logical spaces may not be specified within tokens such as identifiers, constants, keywords, or reserved words. The assembler treats any invalid character(s) in the context of a source file as a separator.

Delimiters

Like logical spaces, delimiters mark the end of a token, but each delimiter has a different special meaning. Some examples are commas and colons.

When a delimiter is present, a logical space between two tokens need not be specified. However, extra space or tab characters often make programs easier to read.

Delimiters are described in context throughout this manual.

Identifiers

An identifier is a name for a programmer-defined entity such as a segment, variable, label, or constant. Valid identifiers conform to the following rules:

- The initial character must be a letter (A...Z or a...z) or one of the following special characters:
  - ? A question mark (ASCII value: 3FH)
  - @ An at sign (ASCII value: 40H)
  - _ An underscore (ASCII value: 5FH)
- The remaining characters may be letters, digits (0..9), and the preceding special characters. Separators may not be specified within identifiers.
- An identifier may be up to 255 characters in length; it is considered unique only up to 31 characters.
- Every identifier within a program module represents one and only one entity. A named entity is accessible from anywhere in the module when it is referenced by name. The assembler does not have identifier scope rules that allow you to specify the same name for two distinct entities in different contexts.

Continued Statements and Comments

An assembler statement usually occupies a single source file line. A source file line is a sequence of characters ended by a newline character.

However, the end of line in a source file is not necessarily the logical end of a statement. Assembler statements do terminate with a newline character, but logical statements can extend over several lines by using the continuation character (&).

The end of line in a source file always terminates a comment. The semicolon (;) is the initial character of a comment.

Valid comments and statements conform to the following rules:

- A comment begins with a semicolon (;) and ends when the line that contains it is terminated. The assembler ignores comments.
- A statement or comment may be continued on subsequent continuation lines. The first character following the line terminator that is not a logical space must be an ampersand (&).
- Statements and comments may extend over many source file lines if they conform to the following:
  - Symbols (such as identifiers, keywords, and reserved words) cannot be broken across continuation lines.

- Character strings must be closed with an apostrophe on one line and reopened with an apostrophe on a subsequent continuation line, with an intervening comma (,) after the ampersand. Space and tab characters within a character string are significant; they are not treated as logical spaces.
- If a comment is continued, the first character following the ampersand that is not a logical space must be a semicolon (;).

Examples

The following examples illustrate the difference between the end of a source file line and the logical end of an assembler statement. The notation <newline> represents the newline character. Both examples are equivalent.

1. This example has a single statement on a single source file line. The end of the source file line and the logical end of the statement are the same.

```
;         1         2         3         4<newline>
; 234567890123456789012345678901234567890<newline>
<newline>                 ; interpreted as logical space
MOV AX, FOO<newline>
```

2. This example has many ends of lines in the source file, but it has only one logical end of statement.

```
;         1         2         3         4<newline>
; 234567890123456789012345678901234567890<newline>
<newline>                 ; interpreted as logical space
MOV                       ; this ASM286<newline>
& AX,                     ; statement extends<newline>
&                         ; <newline>
&                         ; <newline>
&                         ; over<newline>
&                         ; several lines<newline>
& FOO                     ; statement ends here<newline>
<newline>
```

Chapter 2

Segmentation

This chapter contains three main sections:

- Overview of Segmentation

  This section briefly describes processor segmentation, together with the assembler directives that define and set up access to logical program segments.

- Defining Logical Segments

  This section explains the SEGMENT..ENDS and STACKSEG directives. These directives define code, data, and stack segments in assembler programs.

- Assuming Segment Access

  This section explains the ASSUME directive. This directive specifies which segments in an assembler program are accessed by the processor segment registers at any given point in the program's code.

Overview of Segmentation

The processor addresses 16 megabytes of physical memory. Processor memory is segmented. For programmers, processor memory appears to consist of up to four accessible segments at a time:

- One code segment containing the executable instructions
- One stack segment containing the run-time stack
- Up to two data segments, each containing part of the data

Assembler program segments are called logical segments, because they represent logical addresses that must be mapped to processor physical addresses before program execution.

The maximum size of a program segment is 64K bytes.

See also: Processor memory organization, Appendix A, and operand addressing and the USE attribute, Chapters 5 and 6.

At run time, the physical base address of a program segment will be accessed by an immediate value loaded into a segment register. This value is called a selector. A selector points (indirectly in processor protected mode and directly in processor real address mode) to the physical location of a segment. The processor segment registers are CS, DS, and SS, which access code, data, and stack segments, respectively, and ES, which access an additional data segment.

Logical segments are created in an assembler module with the SEGMENT (code and data) and STACKSEG (stackorstack-and-data) directives. These directives specify a segment name; this name defines a logical address for the segment. A segment name can appear in several contexts throughout a program:

- In data initializations, because it stands for the value of the selector
- In segment register initializations
- In an ASSUME statement, which tells the assembler which segment registers contain which selectors

See also: ASSUME statement, in this chapter selectors, Chapter 4, data and segment register initializations, Chapter 1.

After program code is assembled, the system utilities map assembler program segments to processor physical addresses. A named segment becomes a sequence of contiguous physical addresses. A logical segment becomes physically accessible when the segment name is loaded into a processor segment register during program execution.

Defining Code, Data, and Stack Segments

The SEGMENT..ENDS directive defines an assembler program's code and data segments. The STACKSEG directive defines the stack (or mixed stack and data) segment. Both directives specify a name for each logical segment defined in a program.

Because program segments are named, assembler logical segments need not be contiguous lines of source code. Within a source module, a named segment can be closed with ENDS

and reopened with another SEGMENT.. that specifies the same name. Logical segments can also be coded in more than one source module.

See also: Logical segments in source modules, Chapter 3

SEGMENT..ENDS Directive

Syntax

```
name SEGMENT[access][combine]
::
[instructions, directives, and/or data initializations]
::
name ENDS
```

Where:

name is an identifier for the segment; name must be unique within the module. name represents the logical address of the beginning of the program segment. The segment's contents (specified between SEGMENT..ENDS) represent logical addresses that are offsets from the segment name.

access is an optional RO (read only), EO (execute only), ER (execute and read), RW (read and write).

combine is unspecified (default), PUBLIC, or COMMON. If neither PUBLIC nor COMMON is specified for name, the segment is non-combinable: the entire segment is in this module and it will not be combined with segments of the same name from any other module. However, separate pieces of a non-combinable segment within a module will be combined.

If a SEGMENT PUBLIC or SEGMENT COMMON directive has been specified for the segment name, the combine specification for segments with the same name in other modules must be the same.

Discussion

The SEGMENT..ENDS directive defines all or part of a logical program segment whose name is name. The contents of the segment consist of the assembled instructions, directives, label declarations, and/or data initializations that occur between SEGMENT and ENDS.

These contents will be mapped to a contiguous sequence of processor physical addresses by the system utilities. When a segment name is used as an instruction operand, it is an immediate value.

Within a single source module, each occurrence of SEGMENT..ENDS that has the same name is considered part of a single program segment. All ASM386 source code must be specified within a SEGMENT..ENDS. Every named variable and label in an assembler program must also be defined within a SEGMENT..ENDS.

Access, use, and combine are optional; they may be specified in any order. Specifying EO, ER, RO, or RW Access

access is an assembler (and processor) protection feature; it specifies the kind(s) of access permitted to the segment.

The assembler issues a warning for the initial definition of a segment if the access specification is omitted. The assembler also assigns an access value according to the contents of the segment. For a segment that contains data only, the value is RW; for a segment that contains code only, it is EO. For mixed code and data, the value is ER.

After a named segment has been defined with a SEGMENT statement, access can be omitted when the segment is reopened. However, its value may not be changed when the segment is reopened.

Specifying PUBLIC or COMMON

combine specifies how the segment will be combined with segments of the same name from other modules to form a single physical segment in memory. The actual combination of modules occurs at bind time.

If a SEGMENT directive specifying PUBLIC or COMMON already exists for a named segment, combines pecifications in other modules must match it. The named segment's combine attribute should be specified (at least) for the initial segment definition in subsequent modules. The following explains how a logical segment in more than one module is combined:

- All segments of the same name that are defined as PUBLIC will be concatenated to form one physical segment. Control the order of combination with the binder. The length of the combined PUBLIC segment will equal approximately the sum of the lengths of the SEGMENT..ENDS pieces. For a segment declared PUBLIC, there is no guarantee that the beginning of a particular segment part within the module will have an offset of zero within the final combined segment.

- All segments of the same name that are defined as COMMON will be overlapped to form one physical segment. Each module's version of the segment begins at offset zero within the segment, so each version has the same physical address. The length of the combined COMMON segment will be equal to the longest individual length within any of its defining modules. A COMMON segment may not specify EO or ER access.

  If neither PUBLIC nor COMMON is specified, the segment is non-combinable. The entire logical segment must be contained in a single source module. It cannot be combined with segments from other program modules.

Multiple Definitions for a Segment

Assembler segments can be opened and closed (with the SEGMENT..ENDS directive) within a source module as many times as you wish. All separately defined parts of the segment are concatenated by the assembler and treated as if they were defined within a single SEGMENT..ENDS.

Assembler procedure, codemacro, and structure definitions may not overlap segment boundaries.

When a segment is reopened, it is unnecessary to respecify its access, use, and combine attributes, if any. Do not change the combine or use attribute when a segment is reopened.

If a segment's access is respecified, both access specifications must form a compatible set. The following are compatible sets:

- RO and RW are a compatible set with a resulting access attribute of RW.
- Any combination of RO, EO, and ER form a compatible set with a resulting access attribute of ER.

There are no other compatible sets for access specifications.

Chapter 3

Defining And Initializing Data

This chapter has four major sections:

- An overview of assembler labels, variables, and data This section explains:
  - Assembler label and variable types
  - The relationship between assembler variable types and the values associated with variables: the processor or floating-point coprocessor data types
  - How to specify data values in assembler programs
- Assembler variables This section explains:
  - Storage allocations for variables
  - V ariable attributes
  - Defining and initializing simple-type variables with the DBIT, DB, DW, DD, DP, DQ, and DT directives
  - Defining compound types with the RECORD and STRUC directives; defining and initializing variables of these types (records and structures)
  - Defining and initializing variables with DUP clause(s)
- Assembler labels This section explains:
  - Label attributes
  - The location counter and the ORG and EVEN directives
  - The LABEL directive

  - Defining implicit NEAR labels

  - The PROC directive

- Using symbolic data, including named variables and labels, with the EQU and PURGE directives

Specifying Assembler Data Values

Assembler data can be expressed in binary, hexadecimal, octal, decimal, or ASCII form. Decimal values that represent integers or reals can be specified with a minus sign; a plus sign is redundant but accepted. Real numbers can also be expressed in floating-point decimal or in hexadecimal notations. Table 4-2 summarizes the valid ways of specifying data values in assembler programs.

Table 4-2. Assembler Data Value Specification Rules

Value in	Examples		Rules of Formulation
Binary	1100011B	110B	A sequence of 0's and 1's followed by the letter B.
Octal	7777O	4567Q	A sequence of digits in the range 0..7 followed by the letter O or the letter Q.
Decimal	3309	3309D	A sequence of digits in the range 0..9 followed by an optional letter D.
Hexadecimal	55H	4BEACH	A sequence of digits in the range 0..9 and/or letters A..F followed by the letter H. A digit must begin the sequence.
ASCII	'AB'	'UPDATE.EXT'	Any ASCII string enclosed in single quotes.
Decimal	-1.	1E-32 3.14159	A rational number that may be preceded by a sign and followed by an optional exponent. A decimal point is required if no exponent is present but is optional otherwise. The exponent begins with the letter E followed by an optional sign and a sequence of digits in the range 0..9.
Hexadecimal	40490FR	0C0000R	A sequence of digits in the range 0..9 and/or letters A..F followed by the letter R. The sequence must begin with a digit, and the total number of digits and letters must be (8/16/20) or (9/17/21 with the first digit 0).

A real hexadecimal specification must be the exact sequence of hex digits to fill the internal floating-point coprocessor representation of the floating-point number. For this reason, such values must have exactly 8, 16, or 20 hexadecimal digits, corresponding to the single, double, and extended precision reals that the floating- point coprocessor and the floating-point instructions handle. Such values can have 9, 17, or 21 hexadecimal digits only if the initial digit must be a zero because the value begins with a letter.

Data values can be specified in an assembler program in a variety of formats, as shown

in Table 4-2. The way the processor or floating-point coprocessor represents such data internally is called its storage format.

See also: Processor storage formats, Appendix A floating-point coprocessor storage formats, Chapter 7

Initializing Variables

Assembler variables can be initialized by:

- Variable or segment names that represent logical addresses
- Constants (see Table 4-2)
- Constant expressions

A series of operands and operators is called an expression. An expression that yields a constant value is called a constant expression.

See also: Assembler expressions, Chapter 5

The assembler evaluates constant expressions in programs.

How the Assembler Evaluates Constant Expressions

The assembler can perform arithmetic operations on 8-, and 16-bit numbers. The assembler interprets these numbers as integer or ordinal data types.

An integer value specified with a sign is a constant expression. The assembler evaluates integer or ordinal operands and expressions using 32-bit two's complement integer arithmetic.

By using this arithmetic, the assembler can evaluate expressions whose operands' sizes might extend beyond the storage type of the result. As long as the expression's value fits in the storage type of the destination, the assembler does not generate an error when intermediate results are too large. The assembler does generate an error if the final result is too large to fit in the destination.

Variables

A variable defines a logical address for the storage of value(s). An assembler variable is not required to have a name as long as its associated value(s) are accessible. But, every variable has a type; records and structures have a compound type.

Assembler variables must be defined with a storage allocation statement. A storage allocation specifies a type (storage size in bytes) and defines a logical address for a variable that gives access to the variable's value(s). A storage allocation statement may also specify initial value(s) for a variable.

Use the DB, DW, DD, DP, DQ, or DT directive to allocate storage for simple-type variables of the following sizes:

DB	8-bits (byte)
DW	16-bits (word)
DD	32-bits (dword)
DP	48-bits (pword)
DQ	64-bits (qword)
DT	80-bits (10 bytes)

Use a DUP clause within any assembler data allocation statement to allocate and optionally initialize a sequence of storage units of a single variable type. DUP defines an array-like variable whose element values are accessed by an offset from the variable name or from the initially specified storage unit.

Syntax

```
[name] dtyp init [,...]
```

Where:

name — is the name of the variable. Within the module, it must be a unique identifier.

dtyp — is DB, DW, DD, DP, DQ, or DT.

init — is the initial value to be stored in the allocated space. init can be a numeric constant (expressed in binary, hexadecimal, decimal, or octal), an ASCII string, or the question mark character (?), which specifies storage with undefined value(s). dtyp restricts the values that may be specified for init.

Defining and Initializing Variables of a Simple Type

All assembler variable definitions use the DB, DW, DD, DQ, DP, or DT directives. The template components of compound variable types are simple types defined with these directives.

DB Directive

Syntax

```
[name] DB init [,...]
```

Where:

name	is the name of the variable. Within the module, it must be a unique identifier.
init	is a question mark (?), a constant expression, or a string of up to 255 ASCII characters enclosed in single quotes (').

Discussion

DB reserves storage for and optionally initializes a variable of type BYTE. ? reserves storage with an undefined value.

Numeric initial values can be specified in binary, octal, decimal, or hexadecimal (see Table 4-2). The specified constant or constant expression must evaluate to a number in the range 0..255 (processor ordinal) or -128..127 (processor integer).

The components of character string values must be ASCII characters and the whole string must be enclosed in single quotes. To include a single quote character within such a string, specify two single quotes (").

Each ASCII character requires a byte of storage. In BYTE strings, successive characters occupy successive bytes. The name of the variable represents the logical address of the first character in such a string.

Examples

1. This example initializes the variable ABYTE to the constant value 100 (decimal). It reserves storage for another byte with an undefined value.

   ```
   ABYTE DB 100
   DB ?
   ```

2. This example initializes three successive bytes to the values 4, 10, and 200, respectively.

```
BYTES3 DB 4,10,200
```

3. This example initializes seven bytes containing the ASCII values of the characters A, B, C, ' , D, E, and F, respectively.

```
STRGWQUOT DB 'ABC''DEF'
```

DW Directive

Syntax

```
[name] DW init [,...]
```

Where:

name	is the name of the variable. Within the module, it must be a unique identifier.
init	is a question mark (?), a constant expression, or a string of up to 2 characters enclosed in single quotes (').

Discussion

DW defines storage for and optionally initializes a 16-bit variable of type WORD. ? reserves storage with an undefined value.

Numeric initial values can be specified in binary, octal, decimal, or hexadecimal (see Table 4-2). The specified constant or constant expression must evaluate to a number in the range 0..65535 (processor ordinal) or -32768..32767 (processor integer).

A variable or label name yields an initial value that is the offset of the variable or label. It is an error to initialize a WORD variable with the name of a variable or label that has been defined in a USE32 segment; its offset is too large (32-bits). A segment name yields an initial value that is the segment selector.

A 1- or 2-character string yields an initial value that is interpreted and stored as a number. The assembler stores a 2-byte value even if the specified string has only one character:

- It stores the specified initial value in the least significant byte.
- It zeros the remaining byte.

Examples

1. This example tells the assembler to reserve storage for two uninitialized words.

   ```
   UNINIT DW ?,?
   ```

2. This example initializes WORD variables with numeric values.

   ```
   CONST DW 5000 ; decimal constant
   HEXEXP DW OFFFH -10 ; expression
   ```

3. This example initializes VAR1OFF to the offset of VAR1 (both variables are within a USE16 segment) and CODESEL to the selector of a segment named CODE.

   ```
   VAR1OFF DW VAR1
   CODESEL DW CODE
   ```

4. This example initializes NUMB to the ASCII value (interpreted as a number) of the letters AB.

   ```
   NUMB DW 'AB' ; equivalent to NUMB DW 4142H
   ```

DD Directive

```
[name] DW init [,...]
```

Where:

name	is the name of the variable. Within the module, it must be a unique identifier.
init	is a question mark (?), a constant expression, the name of a variable or label, or a string of up to 4 characters enclosed in single quotes (').

Discussion

DD defines storage for and optionally initializes a 32-bit variable of type DWORD. ? reserves storage with an undefined value.

Integer initial values can be specified in binary, octal, decimal, or hexadecimal (see Table 4-2). The specified constant or constant expression must evaluate to a number in

the range: $-2^{31}..2^{31-1}$ (processor integer or floating-point coprocessor short integer) Or, $0..2^{32-1}$ (processor ordinal)

Real initial values can be specified in floating-point decimal or in hexadecimal (see Table 4-2). A decimal constant must evaluate to a real in the ranges: -3.4E38..-1.2E-38, 0.0, 1.2E-38..3.4E38 (floating-point coprocessor single precision real)

A constant expressed as a hexadecimal real must be the exact sequence of hex digits to fill the internal floating-point coprocessor representation of a single precision real (8 hexadecimal digits or 9 hexadecimal digits, including an initial 0).

A USE16 variable or label name yields an initial value that fills the dword. Its high-order word contains the segment selector and its low-order word contains the offset of the USE16 variable or label.

A USE32 variable or label name yields an initial value that is the offset (from the segment base) of the variable or label.

A string (up to four characters) yields an initial value that is interpreted and stored as a number. The assembler stores a 4-byte value even if the specified string has fewer than four characters:

- It stores the specified initial values in the least significant bytes.
- It zeros the remaining bytes.

Examples

1. This example defines two variables, a floating-point coprocessor short integer and a single precision real.

   ```
   INTVAR DD 1234567
   REALVAR DD 1.6E25
   ```

2. In this example, LAB1 was defined in a USE16 segment and LAB2 was defined in a USE32 segment.

   ```
   LAB1_ADD DD LAB1 ; LAB1_ADD contains offset and ; segment selector of LAB1
   LAB2_ADD DD LAB2 ; LAB2_ADD contains offset of LAB2
   ```

3. This example initializes three unnamed dwords. The first contains an undefined value. The second contains the ASCII numeric value of the letter A. The third contains the integer 450 (decimal).

   ```
   DD ?, 'A', 450
   ```

Chapter 4

Processor Instructions

4.1 Processor Instructions

AAA

ASCII Adjust after Addition

Opcode	Instruction	Clocks	Description
37	AAA	4	ASCII adjust AL after addition

Operation

```
IF ((AL AND 0FH) > 9) OR (AF = 1) THEN
  AL := AL + 6;
  AH := AH +1;
  AF := 1;
  CF := 1;
ELSE
  CF := 0;
  AF := 0;
ENDIF
AL := AL AND 0FH;
```

Discussion

Code AAA only following an ADD instruction that leaves a byte result in the AL register. The lower nibbles of the ADD operands should be in the range 0 through 9 (BCD digits) so that AAA adjusts AL to contain the correct decimal digit result. If ADD produced a decimal carry, AAA increments the AH register and sets the carry (CF) and auxiliary carry (AF) flags to 1. If ADD produced no decimal carry, AAA clears the carry and auxiliary flags (0) and leaves AH unchanged. In either case, AL is left with its upper nibble set to 0. To convert AL to an ASCII result, follow the AAA instruction with OR AL, 30H.

Flags Affected

AF and CF as described in the Discussion section; OF, SF, ZF, and PF are undefined.

Exceptions by Mode

Protected

None

Real Address

None

Virtual 8086

None

AAD

ASCII Adjust AX before Division

Opcode	Instruction	Clocks	Description
D5 0A	AAD	19	ASCII adjust AX before division

Operation

```
AL := AH * 0AH + AL;
AH := 0;
```

Discussion

AAD prepares 2 unpacked BCD digits (the least significant digit in AL, the most significant digit in AH) for a division operation that will yield an unpacked result. This is done by setting AL to AL + (10 * AH), and then setting AH to 0. AX is then equal to the binary equivalent of the original unpacked 2-digit number.

Flags Affected

SF, ZF, and PF as described in Appendix A; OF, AF, and CF are undefined

Exceptions by Mode

Protected

None

Real Address

None

Virtual 8086

None

AAM

ASCII adjust AX after multiply

Opcode	Instruction	Clocks	Description
D4 0A	AAM	17	ASCII adjust AX after multiply

Operation

```
AH := AL / 0AH;
AL := AL MOD 0AH;
```

Discussion

Code AAM only following a MUL instruction on two unpacked BCD digits that leaves the result in the AX register. AL contains the MUL result, because it is always less than 100. AAM unpacks this result by dividing AL by 10, leaving the quotient (most significant digit) in AH and the remainder (least significant digit) in AL.

Flags Affected

F, ZF, and PF as described in Appendix A; OF, AF, and CF are undefined

Exceptions by Mode

Protected

None

Real Address

None

Virtual 8086

None

AAS

ASCII Adjust AL after Subtraction

Opcode	Instruction	Clocks	Description
3F	AAS	4	ASCII adjust AL after subtraction

Operation

```
IF (AL AND 0FH) > 9 OR AF = 1 THEN
  AL := AL - 6;
  AH := AH -1;
  AF := 1;
  CF := 1;
ELSE
  CF := 0;
```

```
  AF := 0;
ENDIF
AL := AL AND 0FH;
```

Discussion

Code AAS only following a SUB instruction that leaves the byte result in the AL register. The lower nibbles of the SUB operands should be in the range 0 through 9 (BCD digits) so that AAS adjusts AL to contain the correct decimal digit result. If SUB produced a decimal carry, AAS decrements the AH register and sets the carry (CF) and auxiliary carry (AF) flags to 1. If SUB produced no decimal carry, AAS clears the carry and auxiliary carry flags (0) and leaves AH unchanged. In either case, AL is left with its upper nibble set to 0. To convert AL to an ASCII result, follow the AAS with OR AL, 30H.

Flags Affected

AF and CF as described in the Discussion section; OF, SF, ZF, and PF are undefined

Exceptions by Mode

Protected

None

Real Address

None

Virtual 8086

None

Appendix A

Keywords And Reserved Words

This appendix lists assembler keywords and reserved words. Keywords consist of processor and numerics coprocessor mnemonics. Reserved words consist of all predefined keywords except the mnemonics.

Note that AND, NOT, OR, XOR, SHR, and SHL function as both processor instructions and assembler operators.

Table A.1: Assembler Keywords

AAA	FFREE	FSTP	JNLE	POPF
AAD	FIADD	FSTSW	JNO	PUSH
AAM	FICOM	FSUB	JNP	PUSHA
AAS	FICOMP	FSUBP	JNS	PUSHF
ADC	FIDIV	FSUBR	JNZ	RCL
ADD	FIDIVR	FSUBRP	JO	RCR
AND	FILD	FTST	JP	REP
ARPL	FIMUL	FWAIT	JPE	REPE
BOUND	FINCSTP	FXAM	JPO	REPNE
BSWAP	FINIT	FXCH	JS	REPNZ
CALL	FIST	FXTRACT	JZ	RET
CBW	FISTP	FYL2X	LAHF	ROL
CLC	FISUB	FYL2XP1	LAR	ROR
CLD	FISUBR	HLT	LDS	SAHF
CLI	FLD	IDIV	LEA	SAL
CLTS	FLD1	IMUL	LEAVE	SAR
CMC	FLDCW	IN	LES	SBB
CMP	FLDENV	INC	LGDT	SCAS
CMPS	FLDL2E	INS	LIDT	SCASB
CMPSB	FLDL2T	INSB	LLDT	SCASW
CMPSW	FLDLG2	INSW	LMSW	SETBE
CMPXCHG	FLDLN2	INT	LOCK	SGDT
CWD	FLDPI	INTO	LODS	SHL
DAA	FLDZ	INVD	LODSB	SHR
DAS	FMUL	INVLPG	LODSW	SIDT
DEC	FMULP	IRET	LOOP	SLDT
DIV	FNCLEX	JA	LOOPE	SMSW
ENTER	FNDISI	JAE	LOOPNE	STC
ESC	FNENI	JB	LOOPNZ	STD
F2XM1	FNINIT	JBE	LOOPZ	STI
FABS	FNOP	JC	LSL	STOS
FADD	FNSAVE	JCXZ	MOV	STOSB
FADDP	FNSTCW	JE	MOVS	STOSW
FBLD	FNSTENV	JG	MOVSB	STR
FBSTP	FNSTSW	JGE	MOVSW	SUB
FCHS	FPATAN	JL	MUL	TEST
FCLEX	FPREM	JLE	NEG	VERR
FCOM	FPTAN	JMP	NIL	VERW
FCOMP	FRNDINT	JNA	NOP	WAIT
FCOMPP	FRSTOR	JNAE	NOT	WBINVD
FDECSTP	FSAVE	JNB	OR	XADD
FDISI	FSCALE	JNBE	OUT	XCHG
FDIV	FSETPM	JNC	OUTS	XLAT
FDIVP	FSQRT	JNE	OUTSB	XLATB
FDIVR	FST	JNG	OUTSW	XOR
FDIVRP	FSTCW	JNGE	POP	
FENI	FSTENV	JNL	POPA	

Table A.2: Assembler Reserved Words

ABS	DH	EXTRN	OR	SHR
AH	DI	FAR	ORG	SI
AL	DL	GE	PREFIX66	SIZE
ALIGN	DQ	GT	PREFIX67	SP
AND	DS	HIGH	PREFX	SS
ASSUME	DT	LABEL	PROC	ST
AX	DUP	LE	PROCLEN	STACKSEG
BH	DW	LENGTH	PTR	STACKSTART
BL	DWORD	LOW	PUBLIC	STRUC
BP	DX	LT	QWORD	TBYTE
BX	END	MASK	RECORD	THIS
BYTE	ENDM	MOD	RELB	TR3
CH	ENDP	MODRM	RELD	TR4
CL	ENDS	NAME	RELW	TR5
CODEMACRO	EO	NE	RO	TYPE
COMMON	EQ	NEAR	RW	WARNING
CS	EQU	NOSEGFIX	SEG	WC
CX	ER	NOT	SEGMENT	WIDTH
DB	ES	NOTHING	SHL	WORD
DD	EVEN	OFFSET	SHORT	XOR