PLUME

LIBRARY OF FORTH ROUTINES AND UTILITIES

TEXT-PROCESSING ROUTINES

A MINI-DATABASE MANAGER

GRAPHICS ROUTINES

ASSEMBLY LANGUAGE ROUTINES

APPLICATION FRONT ENDS

BY JAMES D. TERRY

FOREWORD BY MIKE EDELHART

A SHADOW LAWN PRESS BOOK

1

Introduction

Programming languages are like human languages: their main purpose is communication. Programming languages are how we communicate with computers, how we instruct them to carry out the actions we want them to perform. Just as the language we speak and write affects how well we can communicate with our fellow human beings, so too does our choice of a programming language control how effectively we can communicate with our computers. Forth is one of the many programming languages you can use to communicate with your IBM-PC. In this book we try to increase your Forth vocabulary, and make it easier for you to communicate with your IBM-PC. *The Library of Forth Routines* is not an introduction. It assumes that you have at least a working knowledge of Forth. It will, however, take you from your working knowledge to working Forth programs. The ready-to-use "toolkits" provided here should enable you to increase your Forth programming speed and efficiency. The Forth words presented in this book can be used without restriction in any private or commercial program.

FORTH DIALECTS

Each version of Forth available can be thought of as a dialect of the language. The Forth words contained in this book have all been implemented and tested in actual programs, hence they are written in a specific version of Forth. That version is, naturally enough, one published by the author's company, and is known as Atila. You can order Atila and the source for all the words in this book using the coupon you will find in the back of the book. If you are using Atila, you will run into no dialect or version problems.

The words in this book have been written to be universal to almost all Forth dialects. You should also be able to use these programs with other versions of Forth. No unusual or esoteric words specific to Atila have been used whenever possible. Appendix B includes sources for any Atila words that

might possibly not be in your Forth. Additionally, all words have been defined in uppercase, and with the first three letters and length unique, to avoid problems with Forths that have these restrictions. Every effort has been made to present you with Forth code that you can use, whatever version of Forth you have.

THE IBM-PC

While this book is directed toward the IBM-PC and compatibles, most of the words presented could be used in any Forth system on any computer. Only Chapters 4, 5, 6, and 9 are truly dependent on the IBM-PC. Chapter 5 could be applied to any 8088/8086 system, and Chapter 6 to any computer that uses the 8087. This leaves 9 of the book's 13 chapters that can be implemented on any Forth system not run on a PC.

The version of Forth used in this book, and most other Forths available for the IBM-PC, is a 16-bit or small memory model Forth: This means that it uses address data that are 16 bits wide. While the IBM-PC can have up to 640K of memory, 16 bits only allows 64K to be addressed. To utilize the extra memory, most small memory model Forths come with a set of words to access the extra memory using 32-bit pointers, which take up two stack entries. There is, as yet, no standardization in the Forth community for these words. Presented below are the Atila words for accessing the extra memory on the IBM-PC. If you are not using Atila, you will need to refer to the documentation provided with your version of Forth to find equivalent words. (*Note:* The stack notation used in this book is described in Appendix A).

>X (– A)	Leave the segment address Atila is executing in, used primarily to convert Atila addresses to 32-bit format.
X! (N A1 A2 –)	Store N in the cell at segment A2, offset A1.
X@ (A1 A2 – N)	Leave N, the cell at segment A2, offset A1.
X <CMOVE (A1 A2 A3 A4 N –)	Move N bytes from segment A2, offset A1, to segment A4, offset A3. Move backwards in memory.
XC! (N A1 A2 –)	Store N in the byte at segment A2, offset A1.
XC@ (A1 A2 – N)	Leave N, the byte at segment A2, offset A1.

XCMOVE (A1 A2 A3 A4 N –)	Move N bytes from segment A2, offset A1, to segment A4, offset A3. Move forward in memory.
XFILL (A1 A2 N1 N2 –)	Fill N1 bytes of memory with byte N2. Start at segment A2, offset A1.

These words are used only in Chapters 4 and 9.

CONCLUSION

The toolkits provided in this book are building blocks you can use in your Forth programs. Forth is an extensible language, and the words in this book can also be extended. Throughout the text suggestions for extensions have been provided. It is the author's hope that, as you use the words in this book, you find many interesting and challenging ways to extend the building blocks that have been provided.

2

CASE Statements

Words Defined in This Chapter:

CASE:	Vectored case defining word.
CASE	Branching case word.
=OF	Equal condition in a case statement.
>OF	Greater than condition in a case statement.
<OF	Less than condition in a case statement.
RNG–OF	Range of number condition in a case statement.
END–OF	End of condition in a case statement.
ENDCASE	End of branching case.

CASE STATEMENTS

In this chapter we present a set of words that implement case statements in Forth. Standard Forth contains almost all other standard branching and iteration constructs. Case statements are not included, but this chapter corrects that omission. Two types of case statements make up this chapter. The first is a simple vectored case word. This word will be called CASE:, or "case-colon". It enables us to define a single word that expects an integer on the stack. It will execute the word corresponding to that number in its list, then exit itself. Here is an example:

```
ATILA OK    : ONE ." Word One " ; [return]
ATILA OK    : TWO ." Word Two " ; [return]
ATILA OK    : THREE ." Word Three " ; [return]
```

```
ATILA OK    CASE: EXAMPLE ONE TWO THREE ; [return]
```

We now have created a word, EXAMPLE, that is a vectored case statement. If we pass it a "one," it will execute the first word in its list.

```
ATILA OK 1 EXAMPLE [return]
Word One ATILA OK
```

Passing it a two would execute the word TWO, passing a three would cause THREE to be executed. Passing a number not in the range of one to three would result in unpredictable behavior. There is no limit to how many words may be in any single vectored case word. No other code can be in such a word, however. The list of words in a vectored case word is just used as a list in this case.

The next kind of case statement we wish to present is more general; it allows different codes to be executed, depending on the value of an integer at the start of the construct. A typical example might look like this:

(Number of eyes monster has is on the stack.)

```
CASE
  0 =OF ." A blind monster smells you." END-OF
  1 =OF ." A Cyclops stares at you." END-OF
  2 =OF ." It looks normal enough." END-OF
  NOT-OF ." A multi-eyed creature is eying you." END-OF
ENDCASE
```

As you can see, this construct provides a simple way to express what would be awkward with nested IF statements, although anything we do with a case statement we could also do with an IF. First let's present the words, then we'll get to a complete description.

The additional words <OF, >OF and RNG-OF enable us to check for less than, greater than, and range of numbers, respectively, in a case statement. This short example should demonstrate:

```
#IN CASE
  1 =OF ." You typed a one." END-OF
  5 <OF ." Less than 5! " END-OF
  6 12 RNG-OF ." Between 6 and 12 " END-OF
  NOT-OF ." Greater then 12! " END-OF
ENDCASE
```

CASE:

Define a vectored case word.

Stack on Entry: (Compile Time) Empty.
(Run Time) Number of word in list to execute.

Stack on Exit: (Compile Time) Empty.
(Run Time) Undetermined, depends on word executed.

Example of Use: See previous text.

Algorithm: Put the language in compile mode. This will cause all words to have their addresses enclosed in the dictionary. At run time, multiply number passed by two (two bytes for each address). Add this to the start address and fetch the proper address to execute.

Suggested Extensions: Define an ENDCASE word for this construct that will store the number of words in the list. Use this number to make sure undefined words don't get executed.

Definition:

```
:CASE: <BUILDS [COMPILE] ] DOES> 2* + @ EXECUTE ;
```

CASE

Define a general case word.

Stack on Entry: (Compile Time) Empty.
(Run Time) Number to be used in comparisons.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See previous text.

Algorithm: Place the number to be compared against on the return stack. Then store the depth of the data stack in #OFS. This will be used by ENDCASE to determine how many ENDIFs to compile.

Suggested Extensions: Define new case statements that use strings or float-

ing points after we have introduced them in the following chapters.

Definition:

```
0 VARIABLE #OFS
: CASE COMPILE >R SP@ #OFS ! ; IMMEDIATE
```

=OF

Start a branch in a case statement, when an equal condition is met.

Stack on Entry: (Compile Time) Empty.
(Run Time) Number to be compared against case value.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See previous text.

Algorithm: The word CRS will get a copy of the number being held on the return stack. Compare it, using the equal word, to the number on the stack. Compile an IF statement that will handle the branching based on the comparison.

Suggested Extensions: None.

Definition:

```
: CRS R> DUP >R ;
: /=OF CRS = ;
: =OF COMPILE /=OF [COMPILE] IF ; IMMEDIATE
```

<OF

Start a branch in a case statement, when a "less than" condition is met.

Stack on Entry: (Compile Time) Empty.
(Run Time) Number to be compared against case value.

Stack on Exit: (Compile Time) Empty.

(Run Time) Empty.

Definition:

```
: /<OF CRS > ;
: <OF COMPILE /<OF [COMPILE] IF ; IMMEDIATE
```

>OF

Start a branch in a case statement, when a "greater than" condition is met.

Stack on Entry: (Compile Time) Empty.
(Run Time) Number to be compared against case value.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Definition:

```
: />OF CRS < ;
: >OF COMPILE />OF [COMPILE] IF ; IMMEDIATE
```

RNG-OF

Start a branch in a case statement, when a range condition is met.

Stack on Entry: (Compile Time) Empty.
(Run Time) Lower limit of range.
Upper limit of range.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Definition:

```
: /RNG-OF CRS SWAP OVER >= LROT <= AND ;
: RNG-OF COMPILE /RNG-OF [COMPILE] IF ; IMMEDIATE
```

NOT-OF

Start an unconditional branch in a case statement.

Stack on Entry: (Compile Time) Empty.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Definition:

```
: NOT-OF -1 LITERAL [COMPILE] IF ; IMMEDIATE
```

ENDCASE

End a case statement.

Stack on Entry: (Compile Time) One entry for each OF .. END-OF pair.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See previous text.

Algorithm: At run time clear the return stack. At compile time, compile an ENDIF for each IF compiled by an OF word. Use the variable #OFS to check against the data stack, which should hold an address for each OF statement.

Suggested Extensions: None.

Definition:

```
: ENDCASE COMPILE R> COMPILE DROP BEGIN
SP@ #OFS @ <> WHILE
[COMPILE] ENDIF
REPEAT ; IMMEDIATE
```

END-OF

End an OF branch of a CASE statement.

Stack on Entry: (Compile Time) Empty.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See previous text.

Algorithm: At compile time, compile an ELSE that will be executed if the IF compiled by the OF statement fails.

Suggested Extensions: None.

Definition:

```
: END-OF [COMPILE] ELSE ; IMMEDIATE.
```

3

A Programmer's Calculator

In this chapter, we write a simple programmer's calculator. It is provided as an example of a complete Forth program. A programmer's calculator is a useful "stand-alone" tool. It will enable us to input numbers in decimal, binary, hex, and octal bases, to perform the normal arithmetic calculations on them (addition, subtraction, division, and multiplication), and to do some logical operations (like AND, OR and NOR). Our calculator will also enable us to convert between numbers and ASCII.

STEP 1: THE DESIGN

Before we start to write any program, we should design it completely. The design includes both how the program will look to the user and how the program itself will work internally. In this case, let's start with how the user will see our calculator. Our programmer's calculator will use postfix math, just like Forth itself. The user will be able to enter numbers or operations. The available operations will include: the math operators addition, subtraction, multiplication, and division; the logical operators AND, OR, and Exclusive-OR; the ability to display the current number in bases 2, 8, 10, and 16; and the ability to display the ASCII equivalent of a number or vice versa. Additionally, we'll give our calculator a memory function that enables it to hold and recall a single number. Of course, the user can at any time clear the display or the memory. Our display on the screen will look like this the following illustration (see Figure 3-1).

Internally our program will use the Forth data stack as the stack for our calculator. Obviously, we will need words to perform each of the operations allowed in the above description. What else will we need? A routine that takes the input from the user and determines if it is a number or if an operation will be needed. So will a word that displays the top number on the stack on the first

line of our calculator. And, we'll want a word that will write out our menu on the display.

CURRENT NUMBER: _____ BASE: _____ STACK DEPTH: _____

OPERATIONS: (Precede with a /)

+ – ADD – – SUBTRACT	* – MULTIPLY / – DIVIDE	& – AND | – OR	X – Exclusive–OR N – NOT
B – Binary Display D – Decimal Display	O – Octal Display H – HEX Display	U – Unsigned S – Signed	A – ASCII Display
! – Store Memory	@ – Fetch Memory	M – Clear Memory	
C – Clear Display	E – END Program		

ENTER NUMBER OR OPERATION => _____

Figure 3-1

STEP 2: START CODING

Let's start by trying to save the user of our calculator some problems. Many of the operations listed above will require a specific number of arguments. Before our calculator tries to add two numbers we should make sure that it has two numbers on the stack to add. Some operations, like memory store, for example, will only require a single number on the stack. Here is a word that will check to make sure there are enough numbers on the stack. It will take as an argument the number of stack entries needed and will return a flag. The flag will be true if there are enough numbers on the stack, false if there are too few. This word, STACK_CHECK, will have to make sure not to count its arguments when it looks at the stack. Here goes:

```
: STACK_CHECK DEPTH 1-<= ;
```

Whenever there are too few arguments we'll want to tell the user. Instead of repeating the message after each call to STACK_CHECK, it would be easier to make it part of the word itself. It did look too easy, didn't it? Here is a new version:

```
( Make sure there are enough entries )
( N – F )
( N – Number of entries requires )
```

```
( F – True or False )
: STACK_CHECK DEPTH 1– <= DUP NOT IF
     2 VTAB 0 HTAB
     ." Not enough data for operation."
  ENDIF ;
```

The VTAB and HTAB words position the cursor on line two of the display. These are not standard Forth words; no standard words exist for this purpose. Consult your manual to see how to do this in your version of Forth. With error checking in hand we can proceed to write some of the operators. Let's start with the math operators.

```
( Add two numbers )
: ADD 2 STACK_CHECK IF + ENDIF ;
```

ADD makes sure we have enough data using our error-checking word; if there is, it proceeds. The same will hold true for all the following.

```
( Subtract two numbers )
: SUB 2 STACK_CHECK IF – ENDIF ;

( Multiply two numbers )
: MULT 2 STACK_CHECK IF * ENDIF ;

( Divide two numbers )
: DIVIDE 2 STACK_CHECK IF / ENDIF ;

( AND two numbers )
: CAND 2 STACK_CHECK IF AND ENDIF ;
```

Notice how we had to call our AND something else to avoid redefining the Forth word AND.

```
( OR two numbers )
: COR 2 STACK_CHECK IF OR ENDIF ;

( Exclusive–OR two numbers )
: CXOR 2 STACK_CHECK IF XOR ENDIF ;
```

Now, to break the monotony, we'll write an operation on a single number. The logical inverse of a number can be found by Exclusive-ORing it with all ones.

```
( NOT, or logical inverse a number. )
: CNOT 1 STACK_CHECK IF –1 XOR ENDIF ;
```

Next let's deal with the memory function of our calculator. First, we need a variable that holds the actual memory value.

```
0 VARIABLE MEMORY
```

Our calculator can perform three operations: store to memory, fetch from memory, and clear memory. Here are the words that will accomplish that.

```
( Store top number in memory. )
: !MEM 1 STACK_CHECK IF DUP MEMORY ! ENDIF ;

( Fetch number from memory. )
: @MEM MEMORY @ ;

( Clear memory. )
: 0MEM MEMORY 0SET ;
```

Now we can start to deal with the display operations. These operations don't really affect the numbers on the stack, but rather how the top number is displayed. Forth has a built in variable, BASE, that determines what base that the numbers printed by DOT (.) will be displayed in. This makes our job a lot easier. A few variables will hold whether or not we want the output displayed signed or unsigned, numeric or ASCII. Here is the code:

```
( Signed or unsigned variable )
( Hold true if signed output )
0 VARIABLE SIGNED?

( Numeric or ASCII variable )
(Hold true if numeric output )
0 VARIABLE NUMERIC?

( Display Numeric ) : YES_NUMERIC -1 NUMERIC? ! ;

( Make display binary )
: 2BASE 2 BASE ! YES_NUMERIC ;

( Make display octal )
: 8BASE 8 BASE ! YES_NUMERIC ;

( Make display decimal )
( An already existing Forth word. )
: 10BASE DECIMAL YES_NUMERIC ;

( Make display hex )
( An already existing Forth word. )
: 16BASE HEX YES_NUMERIC ;
```

```
( Display Signed )
: YES__SIGNED -1 SIGNED? ! ;

( Display Unsigned)
: NOT__SIGNED SIGNED? 0SET ;

( Display ASCII )
: NOT__NUMERIC NUMERIC? 0SET ;
```

The last two operations clear the display (which for our purpose means empty the stack) and end the program.

```
( Drop numbers until the stack is empty )
: CLEAR__STACK BEGIN DEPTH 0 <> WHILE DROP REPEAT ;

( Clear the screen, leave the program. )
: END__CALCULATOR HOME ." Calculator complete. " CR ABORT ;
```

HOME is another nonstandard Forth word that clears the display screen. See the documentation that came with your Forth to determine the appropriate word in your version of Forth.

We need a number of words to draw the top line of the display. The first prints the top number on the stack. It must see if the stack is empty, in which case it will print seven blanks, or if numeric or ASCII output is desired. If the current output is numeric it must check SIGNED? to see how to print the number. It ends up looking like this:

```
: NUMBER__DISPLAY
  0 VTAB 15 HTAB DEPTH 0= IF
    7 0 DO SPACE LOOP
  ELSE
    NUMERIC? IF
      SIGNED? IF
        DUP .
      ELSE
        DUP U.
      ENDIF
    ELSE
      EMIT
    ENDIF
  ENDIF ;
```

The second display word must print out what the current base is.

```
: BASE__DISPLAY
  0 VTAB 25 HTAB BASE @ DUP 2 = IF
    ." Binary "
```

```
  ELSE
    DUP 8 = IF
      ." Octal "
    ELSE
      10 = IF
        ." Decimal "
      ELSE
        ." Hex "
  ENDIF ENDIF ENDIF ;
```

The final display word will print out how many numbers are on the stack.

```
: DEPTH_DISPLAY 0 VTAB 35 HTAB DEPTH . ;
```

We join all three into a single word for convenience:

```
: DISPLAY NUMBER_DISPLAY BASE_DISPLAY DEPTH_DISPLAY ;
```

When we start the program we want to draw the screen display. Here is that start-up word:

```
: MENU HOME
." CURRENT NUMBER:   BASE:   STACK DEPTH:" CR CR
." OPERATIONS: (Precede with a /)" CR CR
```

." + – ADD	* – MULTIPLY	& – AND	X – Exclusive–OR" CR
." – – SUBTRACT	/– DIVIDE	| – OR	N – NOT" CR CR
." B – Binary Display	O – Octal Display	U – Unsigned	A – ASCII Display" CR
." D – Decimal Display	H – Hex Display	S – Signed" CR CR	
." ! – Store Memory	@ – Fetch Memory	M – Clear Memory" CR CR	
." C – Clear Display	E – END Program" CR CR		

```
." ENTER NUMBER OR OPERATION => " CR ;
```

STEP 3: PUTTING THE PIECES TOGETHER

Now we have almost all the words that our calculator program will use. We make a list in memory of all the possible operations, one character representing each possible operation. A word will search this list when an operation is entered, and then execute the word that implements that operation. First, we must make the list. The list is a one-dimensional array. The first entry is how many elements are in the list.

```
( Convert ASCII to integer and enclose it in dictionary )
: AI BL WORD 1+ C@ C, ;

( Array of commands; there are 20)
20 CVARIABLE COMS AI + AI * AI & AI X AI - AI / AI|   AI N
AI B AI O AI U AI A AI D AI H AI S AI ! AI @ AI M AI C AI E 0 C,
```

We'll use the vectored case word to execute the commands, (hold your breath):

```
CASE: DO__COMS ADD MULT CAND CXOR SUB DIVIDE COR CNOT
2BASE 8BASE NOT__SIGNED NOT__NUMERIC 10BASE 16BASE
YES__SIGNED !MEM @MEM CLEAR__MEM CLEAR__STACK END__
PROGRAM ;
```

Here is the word that will search the list for the command passed on the stack:

```
: SEARCH__COMS 0 COMS C@ 1 DO
    OVER COMS I + C@ = IF
      DROP I LEAVE
    LOOP
    SWAP DROP ;
```

SEARCH__COMS will leave a zero on the stack if the command passed is not found in the list; it will leave the number of the command if it is found. We'll use SEARCH__COMS in this word to actually implement command execution.

```
: EXECUTE__A__COMMAND SEARCH__COMS ?DUP IF
    DO__COMS
  ELSE
    1 VTAB 0 HTAB ." Invalid command"
  ENDIF ;
```

Before we can go any further, we need a word that will handle input. This word must input a number or an operation. If the string input starts with a

slash, this word will leave a true flag and the character following the slash. This should be an operation. If no slash is found, it will convert the input to a number. If the NUMERIC? variable is false, it will take that number as the ASCII value of the first character input. If NUMERIC? is true, it will convert the string input to a number, in the current base. In either case, it will push that number on the stack. Here is the code:

```
77 CCONSTANT SLASH

: INPUT 12 VTAB 18 HTAB 20 0 DO SPACE LOOP 12 VTAB 18 HTAB
   QUERY >IN 0SET BL WORD DUP 1+ C@ SLASH = IF
     2+ C@ EXECUTE_A_COMMAND
   ELSE
     NUMERIC? IF
       >R 0, R> >BINARY DROP
     ELSE
       1+ C@
     ENDIF
   ENDIF ;
```

Step 4: THE FINAL PROGRAM

All the words we need for our final word are now ready. Our final word will be an endless loop. The END_PROGRAM word will be the only exit, by means of an ABORT. Without further hesitation:

```
: CALCULATOR MENU BEGIN
     INPUT
     DISPLAY
   0 UNTIL ;
```

This is an example of a complete, though simple, Forth program. To run the program we just type CALCULATOR at our display.

Full-Screen Editor

Forth can be programmed directly by simply entering word definitions at the OK prompt. After a few typos that require you to retype entire lines, you might wish there was some other way. This chapter will present another way: a full-screen editor for Forth blocks. It will enable us to easily enter and modify source text. The word EDIT will invoke the editor. EDIT will expect a screen or block number to edit on the stack.

The editor will place the text onto the screen and we will be able to use the arrow keys to move the cursor around and enter text. Screens will be 24 lines of 40 characters. If an 80-column screen is available, the right-hand side will hold some instructions for the user. Control keys will offer more commands. Here is a summary:

Key	Effect
Up Arrow	*Move the cursor up a single line.*
Down Arrow	*Move the cursor down a single line.*
Left Arrow	*Move the cursor left a single character.*
Right Arrow	*Move the cursor right a single character.*
Del	*Delete the character at the current cursor position.*
Ins	*Place the editor in insert mode.*
CTRL–O	*Insert a line on the screen.*
CTRL–F	*Delete a line.*
CTRL–K	*Erase the screen.*
CTRL–X	*Exit the editor, saving the current screen.*
CTRL–Q	*Exit the editor without saving the current screen.*

Home	*Place the cursor in the upper left-hand corner of the screen.*
End	*Place the cursor in the middle of the screen.*
Enter	*If in insert mode, take editor out of insert mode. Otherwise, move cursor to start of next line.*

The editor will work by directly manipulating screen memory. The variable S__START holds the starting segment address. Because screen memory is outside of Forth's 64K segment, the extra memory words discussed in Chapter 1 are used.

Suggested Extensions: The editor presented in this chapter, while sufficient for productive use, does have a number of possible extensions. One of the most useful would be the ability to move blocks of text, especially between screens. Another might be the ability to move to the next or previous screen while remaining in the editor.

Extending the editor to edit more than a screen at a time would be a useful, but quite major revision.

S__START (– N)

A constant holding the starting segment of screen memory.

Stack on Entry: Empty.

Stack on Exit: (N) – The starting segment of screen memory.

Example of Use: See words defined below.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
( For a Monochrome Display )
HEX B000 CONSTANT S__START DECIMAL

( For a Color Display )
HEX B800 CONSTANT S__START DECIMAL
```

XDL (– N)

A constant holding the number of bytes used to store each line on the display.

Stack on Entry: Empty.

Stack on Exit: (N) – The number of bytes used for each line.

Example of Use: See words defined below.

Algorithm: The IBM display uses to two bytes for each character. See your *Technical Reference Manual* for details.

Suggest Extensions: None.

Definition:

```
( For Forty-Column Display )
40 CCONSTANT XDL

( For a Color Display)
80 CCONSTANT XDL
```

TABLE–SEARCH (A N1 – (N2) F)

Search a table of integers for a specific entry.

Stack on Entry: (A) – The address of the table to search. The first entry of the table must be its length.
(N1) – The integer to search for.

Stack on Exit: (N2) – The position of the integer, if it was found.
(F) – A Boolean flag, true if the integer was found in the table, false otherwise.

Example of Use:

If A_TABLE was a table of integers and 99 was not in that table, then

```
A_TABLE 99 TABLE-SEARCH
```

would leave a false flag on the stack.

Algorithm: Place a false flag on the stack. Obtain the length of the table and start a loop through the table's entries. If a matching entry is found, change the false flag on the stack to true and leave the loop index on the stack, then exit the loop. If the loop falls through, the false flag will be left on the stack.

Suggested Extensions: None.

Definition:

```
: TABLE-SEARCH 0 LROT DUP C@ 1 DO
     DUP I + C@ 3 PICK = IF
       >R >R NOT J SWAP R> R> LEAVE
     ENDIF
  LOOP DROP DROP ;
```

XCUR (– A)

A variable used to hold the x position of the cursor.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of XCUR.

Example of Use: See words defined below.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE XCUR
```

YCUR (– A)

A variable used to hold the y position of the cursor.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of YCUR.

Example of Use: See words defined below.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE YCUR
```

PUTON (N –)

Move text from disk to screen memory.

Stack on Entry: (N) – The block number to move.

Stack on Exit: Empty.

Example of Use:

```
19 PUTON
```

This would move the contents of block 19 into screen memory.

Algorithm: Each Forth screen (at least in this editor) is made up of 24 lines of 40 characters. This word uses two loops. The outer loop of 24 is executed once for each line, the inner of 40 moves each individual character. S_START and XDL control where in memory the characters are moved.

Suggested Extensions: None.

Definition:

```
: PUTON BLOCK 24 0 DO
     40 0 DO
       DUP C@ J XDL * I 2* + S_START XC! 1+
     LOOP
  LOOP DROP ;
```

PUTBACK (N –)

Move text from screen memory to disk.

Stack on Entry: (N) The disk block to move text to.

Stack on Exit: Empty.

Example of Use:

```
19 PUTBACK
```

This would move the text in screen memory to block 19.

Algorithm: First clear the disk block. Two loops are used, as in PUTON. The outer is executed once for each line, the inner once for each character in a line. Characters are moved one at a time from screen memory to the disk block.

Suggested Extensions: None.

Definition:

```
: PUTBACK DUP BLOCK 960 + 64 ERASE
   BLOCK 24 0 DO
      40 0 DO
         J XDL * I 2* + S__START XC@ OVER C! 1+
      LOOP
   LOOP DROP ;
```

EKEY (- C)

Wait for a keypress and return its value.

Stack on Entry: Empty.

Stack on Exit: (C) A character representing the keypress.

Example of Use:

```
: WAITA BEGIN EKEY 65 = UNTIL ;
```

This word will wait until an uppercase A is pressed.

Algorithm: The Atila word KEY normally returns a cell value with the lower byte being zero to indicate an extended key code. This word converts that into

a single byte. This was done so SEARCH-TABLE could be used with byte-length integers.

Suggested Extensions: None.

Definition:

```
: EKEY KEY DUP 128 > IF
    256 / 128 +
  ENDIF ;
```

CHECK (-)

Make sure that XCUR and YCUR hold valid values.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: This word does four distinct checks: XCUR is first checked to see if it has gone past the left edge of the screen. If it has, it is set to 39, the right edge of the screen, and YCUR is decremented. XCUR is then checked to see if it has moved off the right edge of the screen. YCUR is checked to see if it has moved off the top or bottom of the screen.

Suggested Extensions: None.

Definition:

```
: CHECK XCUR C@ 255 = IF
    39 XCUR C! -1 YCUR C+!
  ENDIF
  XCUR C@ 40 = IF
    XCUR C0SET 1 YCUR C+!
  ENDIF
  YCUR C@ 255 = IF
    23 YCUR C!
  ENDIF
  YCUR C@ 24 = IF
    YCUR C0SET
  ENDIF ;
```

Q-ESC (– F)

Handle a control Q.

Stack on Entry: Empty.

Stack on Exit: (F) - A false flag, which causes the editor to terminate.

Example of Use: See words defined below.

Algorithm: Control Q is used to leave the editor without saving the screen being edited. It leaves a false flag on the stack, causing SCREENEDIT (defined below) to fall through and exit.

Suggested Extensions: None.

Definition:

```
: Q-ESC 0 ;
```

X-ESC (- F)

Handle a control X.

Stack on Entry: Empty.

Stack on Exit: (F) - A false flag, which causes the editor to terminate.

Example of Use: See words defined below.

Algorithm: Control X is used to exit the editor normally. An UPDATE is done to mark the screen being worked on for writing to disk. It leaves a false flag on the stack, causing SCREENEDIT (defined below) to fall through and exit.

Suggested Extensions: None.

Definition:

```
: X-ESC UPDATE 0 ;
```

UP (– F)

Handle the up arrow key.

Stack on Entry: Empty.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Algorithm: Decrement the value held in YCUR. Use CHECK to ensure that YCUR still holds a valid value. Leave a true flag on the stack so SCREEN-EDIT (defined below) will not fall through.

Suggested Extensions: None.

Definition:

```
: UP -1 YCUR C+! CHECK -1 ;
```

DOWN (- F)

Handle the down arrow key.

Stack on Entry: Empty.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Example of Use: See words defined below.

Algorithm: Increment the value held in YCUR. Use CHECK to ensure that YCUR still holds a valid value. Leave a true flag on the stack so SCREEN-EDIT (defined below) will not fall through.

Suggested Extensions: None.

Definition:

```
: DOWN 1 YCUR C+! CHECK -1 ;
```

LEFT (– F)

Handle the left arrow key.

Stack on Entry: Empty.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Example of Use: See words defined below.

Algorithm: Decrement the value held in XCUR. Use CHECK to ensure that XCUR still holds a valid value. Leave a true flag on the stack so SCREEN-EDIT (defined below) will not fall through.

Suggested Extensions: None.

Definition:

```
: LEFT -1 XCUR C+! CHECK -1 ;
```

RIGHT (– F)

Handle the right arrow key.

Stack on Entry: Empty.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Example of Use: See words defined below.

Algorithm: Increment the value held in XCUR. Use CHECK to ensure that XCUR still holds a valid value. Leave a true flag on the stack so SCREEN-EDIT (defined below) will not fall through.

Suggested Extensions: None.

Definition:

```
: RIGHT 1 XCUR C+! CHECK -1 ;
```

EATLEFT (– F)

Handle the backspace key.

Stack on Entry: Empty.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Example of Use: See words defined below.

Algorithm: Emit a block to erase the character at the current character position. Decrement the value held in XCUR. Use CHECK to ensure that XCUR still holds a valid value. Leave a true flag on the stack so SCREENEDIT (defined below) will not fall through.

Suggested Extensions: None.

Definition:

```
: EATLEFT 32 EMIT -1 XCUR C+! CHECK -1 ;
```

ONIM (–)

Update the screen to show that insert mode is on.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: If the screen is 80 characters wide, then print "ON " at x position 65, y position 16.

Suggested Extensions: None.

Definition:

```
: ONIM XDL 80 > IF
     16 VTAB 65 HTAB ." ON "
  ENDIF ;
```

OFFIM (–)

Update the screen to show that insert mode is off.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: If the screen is 80 characters wide, then print "OFF" at x position 65, y position 16.

Suggested Extensions: None.

Definition:

```
: OFFIM XDL 80 > IF
     16 VTAB 65 HTAB ." OFF"
  ENDIF ;
```

F–ESC (– F)

Handle the control F key. Control F is used to delete a line.

Stack on Entry: Empty.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Example of Use: See words defined below.

Algorithm: If the cursor is not on the bottom line, move each line below the current line up one full line. Erase the bottom line. Leave a true flag on the stack so SCREENEDIT (defined below) will not fall through.

Suggested Extensions: None.

Definition:

```
: F–ESC YCUR C@ 23 <> IF
     24 YCUR C@ 1+ DO
       I XDL * S_START
```

```
    I 1- XDL * S__START
    80 XCMOVE
  LOOP
  3680 80 0 DO
    I OVER + 32 SWAP S__START XC!
  2 +LOOP DROP
ENDIF -1 ;
```

O-ESC (- F)

Handle the control O key. Control O is used to insert a line.

Stack on Entry: Empty.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Example of Use: See words defined below.

Algorithm: If the cursor is not on the bottom line, move each line below the current line down one full line. Erase the current line. Leave a true flag on the stack so SCREENEDIT (defined below) will not fall through.

Suggested Extensions: None.

Definition:

```
: O-ESC YCUR C@ 23 <> IF
    YCUR C@ 1- 22 DO
      I XDL * S__START
      I 1+ XDL * S__START
      80 XCMOVE
    -1 +LOOP
    YCUR C@ XDL * 80 0 DO
      I OVER + 32 SWAP S__START XC!
    2 +LOOP DROP
  ENDIF -1 ;
```

INSERT (- A)

A Boolean variable used to hold whether or not the editor is in insert mode.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of insert.

Example of Use: See words defined below.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE INSERT
```

RETURN (– F)

Handle the return key.

Stack on Entry: Empty.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Example of Use: See words defined below.

Algorithm: When return is pressed, and the editor is not in insert mode, it causes the cursor to be moved to the start of the next line. XCUR is set to zero and YCUR is incremented. CHECK is used to make sure that XCUR and YCUR still hold valid values. If the editor was in insert mode, take it out of insert mode. Leave a true flag on the stack so SCREENEDIT (defined below) will not fall through.

Suggest Extensions: None.

Definition:

```
: RETURN INSERT C@ IF
    INSERT C0SET OFFIM
  ELSE
    XCUR C0SET 1 YCUR C+! CHECK
  ENDIF -1 ;
```

I–ESC (– F)

Handle the insert key.

Stack on Entry: Empty.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Example of Use: See words defined below.

Algorithm: Invert the value of insert, and print the current insert mode on the screen using either OFFIM or ONIM. Leave a true flag on the stack so SCREENEDIT (defined below) will not fall through.

Suggested Extensions: None.

Definition:

```
: I–ESC INSERT DUP C@ IF
     C0SET OFFIM
  ELSE
     C1SET ONIM
  ENDIF –1 ;
```

D–ESC (– F)

Handle the delete key.

Stack on Entry: Empty.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Example of Use: See words defined below.

Algorithm: Move all the characters to the right of the cursor on the current line left on positon. Blank the rightmost character on the line. Leave a true flag on the stack so SCREENEDIT (defined below) will not fall through.

Suggested Extensions: None.

Definition:

```
: D-ESC YCUR C@ XDL * XCUR C@ 2* + DUP
   2+ S__START ROT S__START 80 XCUR C@
   2* - XCMOVE 32 YCUR C@ XDL * 78 +
   S__START XC! -1 ;
```

PLUG (C – F)

Place a character on the screen.

Stack on Entry: (C) – The character to place on the screen.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Example of Use: See words defined below.

Algorithm: If insert mode is on, move all the characters to the right of the cursor on the current line one position to the right. Emit the character passed to PLUG onto the screen and use RIGHT to move the cursor to the next location. RIGHT will also leave a true flag on the stack so SCREENEDIT (defined below) will not fall through.

Suggested Extensions: None.

Definition:

```
: PLUG INSERT C@ IF
     YCUR C@ XDL * XCUR C@ 2* + DUP 2+
     SWAP S__START ROT S__START 78 XCUR C@
     2* - X<CMOVE
  ENDIF EMIT RIGHT ;
```

HOMEKEY (– F)

Handle the home key.

Stack on Entry: Empty.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Example of Use: See words defined below.

Algorithm: Move the cursor to the upper left-hand corner of the screen by setting XCUR and YCUR to zero. Leave a true flag on the stack so SCREENEDIT (defined below) will not fall through.

Suggested Extensions: None.

Definition:

```
: HOMEKEY YCUR C0SET XCUR C0SET -1 ;
```

ENDKEY (– F)

Handle the end key.

Stack on Entry: Empty.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Example of Use: See words defined below.

Algorithm: Move the cursor to the middle of the screen by setting YCUR to 12 and XCUR to zero. Leave a true flag on the stack so SCREENEDIT (defined below) will not fall through.

Suggested Extensions: None.

Definition:

```
: ENDKEY 12 YCUR C! XCUR C0SET -1 ;
```

HSBE (– A)

A variable used to hold the block number being edited.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of HSBE.

Example of Use: See words defined below.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 VARIABLE HSBE
```

SMESS (–)

Print out editor information.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Clear the screen. If the screen is 80 characters wide, print out some helpful information.

Suggested Extensions: Mode information could be printed if desired, help screens could be added to appear in this portion of the screen.

Definition:

```
: SMESS HOME XDL 80 > IF
      DUP HSBE !
      8 VTAB 50 HTAB ." Screen -> " .
      10 VTAB 50 HTAB ." CTRL-X - Save"
      11 VTAB 50 HTAB ." CTRL-Q - Quit"
      12 VTAB 50 HTAB ." CTRL-O - Insert Line"
      13 VTAB 50 HTAB ." CTRL-F - Delete Line"
      14 VTAB 50 HTAB ." CTRL-K - Clear Screen"
      16 VTAB 50 HTAB ." Insert Mode => OFF"
      ELSE
      DROP
  ENDIF INSERT COSET ;
```

K-ESC (– F)

Handle the control K key. Control K will clear the screen.

Stack on Entry: Empty.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Example of Use: See words defined below.

Algorithm: Use HOME to clear the screen. Set XCUR and YCUR to zero and use SMESS to print out some screen information. Leave a true flag on the stack so SCREENEDIT (defined below) will not fall through.

Suggested Extensions: None.

Definition:

```
: K-ESC HOME XCUR C0SET YCUR C0SET
  HSBE @ SMESS -1 ;
```

DO-ESC (N – F)

Execute an editor command.

Stack on Entry: (N) – The number of the command to execute.

Stack on Exit: (F) – A true flag, used to signal that the editor whether or not it should terminate.

Example of Use: See words defined below.

Algorithm: DO-ESC is a vectored case word. It will use the number on the stack to find the proper word to execute. All editor words leave a flag on the stack that will be used by SCREENEDIT (defined below) to determine whether or not to continue.

Suggested Extensions: Any commands you wish to add to the editor would have to be added here and in the ESC-TABLE defined below.

Definition:

```
CASE: DO-ESC Q-ESC X-ESC K-ESC RETURN
      UP DOWN LEFT RIGHT EATLEFT
      D-ESC F-ESC I-ESC O-ESC HOMEKEY
      ENDKEY ;
```

ESC-TABLE (- A)

The list of key codes for the editor commands.

Stack on Entry: Empty.

Stack on Exit: (A) - The address of the table.

Example of Use: See words defined below.

Algorithm: None.

Suggested Extensions: Any commands you wish to add to the editor would have to be added here and to DO-ESC defined above.

Definition:

```
16 CVARIABLE ESC-TABLE 17 C, 24 C, 11 C,
   13 C, 200 C, 208 C, 203 C, 205 C, 8 C,
   211 C, 6 C, 210 C, 15 C, 199 C, 207 C,
   0 ,
```

?ESC (C - F)

If a character is an editor command, execute it.

Stack on Entry: (C) - The character to CHECK.

Stack on Exit: (F) - Flag, true if an editor command was not found.

Example of Use: See words defined below.

Algorithm: Search the ESC-TABLE, if an entry is found, use DO-ESC to execute it.

Suggested Extensions: None.

Definition:

```
: ?ESC ESC-TABLE TABLE-SEARCH DUP IF
     DROP DO-ESC -1
   ENDIF NOT ;
```

SCREENEDIT (–)

Edit the text on the screen.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Place the cursor at the appropriate place using VTAB and HTAB. Get a key. If it is an editor command, ?ESC will handle it. If it is not, pass it to PLUG. Continue until a false flag is left on the stack by an editor command.

Suggested Extensions: None.

Definition:

```
: SCREENEDIT BEGIN
     XCUR C@ HTAB YCUR C@
     VTAB EKEY DUP ?ESC IF
       PLUG
     ELSE
       SWAP DROP
     ENDIF NOT
   UNTIL ;
```

EDIT (N –)

Edit a block.

Stack on Entry: (N) – The block to edit.

Stack on Exit: Empty.

Example of Use:

```
19 EDIT
```

This would invoke the editor for screen 19.

Algorithm: Print the starting messages. Use PUTON to place the text onto the screen. Use SCREENEDIT to edit the text. PUTBACK will place the text back to the disk buffer. Clear the screen.

Suggested Extensions: None.

Definition:

```
: EDIT DUP SMESS DUP PUTON XCUR C0SET
   YCUR C0SET SCREENEDIT PUTBACK HOME ;
```

5

8088 Macro Assembler

Words Defined in This Chapter:

ASSEMBLER	Define a vocabulary for the assembler words.
1byte	Define single-byte opcodes.
AAA	Assemble the 8088 adjust result of ASCII addition instruction.
AAS	Assemble the 8088 adjust result of ASCII subtraction instruction.
CBW	Assemble the 8088 convert byte to word instruction.
CLC	Assemble the 8088 clear carry flag instruction.
CLD	Assemble the 8088 clear direction flag instruction.
CLI	Assemble the 8088 clear interrupt flag instruction.
CMC	Assemble the 8088 complement the carry flag instruction.
CWD	Assemble the 8088 convert word to double word instruction.
DAA	Assemble the 8088 decimal adjust accumulator after addition instruction.
DAS	Assemble the 8088 decimal adjust accumulator after subtraction instruction.
HLT	Assemble the 8088 halt the processor instruction.
INTO	Assemble the 8088 interrupt if overflow instruction.

IRET	Assemble the 8088 return from interrupt instruction.
LAHF	Assemble the 8088 load 8080 flags into AH register instruction.
LOCK	Assemble the 8088 bus lock prefix instruction.
NOP	Assemble the 8088 no operation instruction.
POPF	Assemble the 8088 pop into the flag register instruction.
PUSHF	Assemble the 8088 push from the flag register instruction.
SAHF	Assemble the 8088 store the AH register into the 8080 flags instruction.
STC	Assemble the 8088 set the carry flag instruction.
STD	Assemble the 8088 set the direction flag instruction.
STI	Assemble the 8088 set the interrupt flag instruction.
WAIT	Assemble the 8088 wait for signal on test instruction.
XLAT	Assemble the 8088 table lookup instruction.
REPE	Assemble the 8088 repeat string instruction until zero flag is not set prefix instruction.
REPZ	Assemble the 8088 repeat string instruction until zero flag is not set prefix instruction.
REP	Assemble the 8088 repeat string instruction until zero flag is not set prefix instruction.
REPNE	Assemble the 8088 repeat string instruction until zero flag is set prefix instruction.
REPNZ	Assemble the 8088 repeat string instruction until zero flag is set prefix instruction.
AAD	Assemble the 8088 adjust AX register for division instruction.
AAM	Assemble the 8088 adjust AX register for multiplication instruction.
b/w	A variable that holds the size data the instruction being assembled will operate on, byte or word.
byte	Cause the next instruction to use a byte-length operand.
cell	Cause the next instruction to use a word-length operand.
tipe	A two-cell variable that holds the addressing mode for each possible operand of an instruction.

value	A two-cell variable holding address values.
#	Mark an operand as having a immediate addressing mode.
]	Mark an operand as having a indirect addressing mode.
reset	Reset "tipe" for the start of a new instruction.
00mod	Define a word that assembles as a 00 mod in the addressing mode byte.
[BX+SI]	Set "tipe" to the addressing mode indirect, employing the BX and SI registers.
(BX+DI)	Set "tipe" to the addressing mode indirect, employing the BX and DI registers.
[BP+SI]	Set "tipe" to the addressing mode indirect, employing the BP and SI registers.
(BP+DI)	Set "tipe" to the addressing mode indirect, employing the BP and DI registers.
[SI]	Set "tipe" to the addressing mode indirect, employing the SI register.
(DI)	Set "tipe" to the addressing mode indirect, employing the DI register.
[BX]	Set "tipe" to the addressing mode indirect, employing the BX register.
ES	Set "tipe" to the addressing mode implied, employing the ES register.
CS	Set "tipe" to the addressing mode implied, employing the CS register.
SS	Set "tipe" to the addressing mode implied, employing the SS register.
DS	Set "tipe" to the addressing mode implied, employing the DS register.
01mod	Define a word that assembles as a 01 mod in the addressing mode byte.
[BX+SI+#]	Set "tipe" to the addressing mode indirect, employing the BX and SI registers and an offset.
(BX+DI+#)	Set "tipe" to the addressing mode indirect, employing the BX and DI registers and an offset.
[BP+SI+#]	Set "tipe" to the addressing mode indirect, employing the BP and SI registers and an offset.
(BP+DI+#)	Set "tipe" to the addressing mode indirect, employing the BP and DI registers and an offset.
[SI+#]	Set "tipe" to the addressing mode indirect, employing the SI register and an offset.

(DI+#)	Set "tipe" to the addressing mode indirect, employing the DI register and an offset.
[BP+#]	Set "tipe" to the addressing mode indirect, employing the BP register and an offset.
[BX+#]	Set "tipe" to the addressing mode indirect, employing the BX register and an offset.
11mod	Define a word that assembles as a 11 mod in the addressing mode byte.
AL	Set "tipe" to the addressing mode implied, employing the AL register.
AX	Set "tipe" to the addressing mode implied, employing the AX register.
CL	Set "tipe" to the addressing mode implied, employing the CL register.
CX	Set "tipe" to the addressing mode implied, employing the CX register.
DL	Set "tipe" to the addressing mode implied, employing the DL register.
DX	Set "tipe" to the addressing mode implied, employing the DX register.
BL	Set "tipe" to the addressing mode implied, employing the BL register.
BX	Set "tipe" to the addressing mode implied, employing the BX register.
AH	Set "tipe" to the addressing mode implied, employing the AH register.
SP	Set "tipe" to the addressing mode implied, employing the SP register.
CH	Set "tipe" to the addressing mode implied, employing the CH register.
BP	Set "tipe" to the addressing mode implied, employing the BP register.
DH	Set "tipe" to the addressing mode implied, employing the DH register.
SI	Set "tipe" to the addressing mode implied, employing the SI register.
BH	Set "tipe" to the addressing mode implied, employing the BH register.
DI	Set "tipe" to the addressing mode implied, employing the DI register.
direction	Set the direction bit in the opcode being formed.

size	Set the size bit in the opcode being formed.
r/m	Set the r/m field in the addressing mode byte being formed.
md	Set the mod field in the addressing mode byte being formed.
,value	Store address of offset for the current instruction being assembled.
1reg@	Get the register value stored in the first position of "tipe".
reg@	Get the register value stored in the current position of "tipe".
1reg	Set the register field in the opcode being formed.
reg	Set the register field in the addressing mode byte being formed.
a-mode	Form and store an addressing mode byte. Store the instructions address values.
16-bit-reg?	Is a 16-bit register being used?
seg-reg?	Is a segmentation register being used?
DEC	Assemble the 8088 decrement instruction.
INC	Assemble the 8088 increment instruction.
POP	Assemble the 8088 move data from the stack instruction.
PUSH	Assemble the 8088 move data too the stack instruction.
ac,data?	Check if the operands for the current instruction are a move of data into an accumulator.
immediate?	Check if there is an immediate operand for this instruction.
eb/w	Store a displacement or address in memory.
,data	Store the displacement or address for an instruction.
s-bit	Set the sign bit in an opcode.
e-mode	From a complete addressing mode byte.
swap-dir	Mark an instruction's operands as moving data in the opposite direction.
dir?	Check the direction an instruction's operands are moving data. Swap the directions if appropriate.
1kind	Define a word to assemble the 8088 arithmetic and logical instructions which use two operands.
ADD	Assemble the 8088 add instruction.
ADC	Assemble the 8088 add with carry instruction.

And	Assemble the 8088 logical AND instruction.
CMP	Assemble the 8088 compare instruction.
Or	Assemble the 8088 logical OR instruction.
SBB	Assemble the 8088 subtract with borrow instruction.
SUB	Assemble the 8088 subtract instruction.
XOR	Assemble the 8088 Exclusive–OR instruction.
TEST	Assemble the 8088 test of data instruction.
c–bit	Set the c bit for shift and rotate instructions.
srkind	Define a word to assemble the 8088 shift and rotate instructions.
RCL	Assemble the 8088 rotate through carry-left instruction.
RCR	Assemble the 8088 rotate through carry-right instruction.
ROL	Assemble the 8088 rotate-left instruction.
ROR	Assemble the 8088 rotate-right instruction.
SAR	Assemble the 8088 shift arithmetic right instruction.
SHL	Assemble the 8088 shift-left instruction.
SHR	Assemble the 8088 shift-right instruction.
LDS	Assemble the 8088 load register and DS instruction.
LEA	Assemble the 8088 load effective address instruction.
LES	Assemble the 8088 load register and ES instruction.
mkind	Define a word to assemble the 8088 mathematical instructions.
DIV	Assemble the 8088 divide instruction.
IDIV	Assemble the 8088 unsigned divide instruction.
IMUL	Assemble the 8088 unsigned multiply instruction.
MUL	Assemble the 8088 multiply instruction.
NEG	Assemble the 8088 negate instruction.
NOT	Assemble the 8088 ones complement instruction.
2kind	Define a word to assemble the 8088 string instructions.
CMPS	Assemble the 8088 string compare instruction.
LODS	Assemble the 8088 load accumulator from memory instruction.
MOVS	Assemble the 8088 string move instruction.

SCAS	Assemble the 8088 compare against memory instruction.
STOS	Assemble the 8088 store accumulator to memory instruction.
IN	Assemble the 8088 transfer data from a port to accumulator instruction.
OUT	Assemble the 8088 transfer data from an accumulator to a port instruction.
INT	Assemble the 8088 software interrupt instruction.
SEG	Assemble the 8088 segment override instruction.
XCHG	Assemble the 8088 exchange data instruction.
LONG-CALL	Assemble the 8088 long (intersegment) call instruction.
CALL	Assemble the 8088 short (intrasegment) call instruction.
RET	Assemble the 8088 short (intrasegment) return instruction.
LONG–RET	Assemble the 8088 long (intersegment) return instruction.
LONG–BRANCH	Assemble the 8088 long (intersegment) jump instruction.
BRANCH	Assemble the 8088 short (intrasegment) jump instruction.
MOV	Assemble the 8088 data move instruction.
bopc	Define words to assemble 8088 branching opcodes.
JA	Assemble the 8088 jump on above instruction.
JNBE	Assemble the 8088 jump on not below or equal instruction.
JAE	Assemble the 8088 jump on above or equal instruction.
JNB	Assemble the 8088 jump on not below instruction.
JB	Assemble the 8088 jump on below instruction.
JNAE	Assemble the 8088 jump on not above or equal instruction.
JBE	Assemble the 8088 jump on below or equal instruction.
JNA	Assemble the 8088 jump on not above instruction.

JCXZ	Assemble the 8088 jump if CX equals zero instruction.
JE	Assemble the 8088 jump on equals instruction.
JZ	Assemble the 8088 jump on zero instruction.
JG	Assemble the 8088 jump on greater than instruction.
JNLE	Assemble the 8088 jump on not less or equal instruction.
JGE	Assemble the 8088 jump on greater or equal instruction.
JNL	Assemble the 8088 jump on not less instruction.
JL	Assemble the 8088 jump on less instruction.
JNGE	Assemble the 8088 jump on not greater or equal instruction.
JLE	Assemble the 8088 jump on less than or equal instruction.
JNG	Assemble the 8088 jump on not greater than instruction.
JNE	Assemble the 8088 jump on not equal to instruction.
JNZ	Assemble the 8088 jump on not zero instruction.
JNO	Assemble the 8088 jump on not overflow instruction.
JNP	Assemble the 8088 jump on no parity instruction.
JPO	Assemble the 8088 jump on odd parity instruction.
JNS	Assemble the 8088 jump on not sign instruction.
JO	Assemble the 8088 jump on overflow instruction.
JP	Assemble the 8088 jump on parity even instruction.
JPE	Assemble the 8088 jump on parity even instruction.
JS	Assemble the 8088 jump on sign instruction.
LOOP	Assemble the 8088 loop if CX is not equal to zero instruction.
LOOPE	Assemble the 8088 loop if CX is equal to zero and the zero flag is set instruction.
LOOPZ	Assemble the 8088 loop if CX is equal to zero and the zero flag is set instruction.
LOOPNE	Assemble the 8088 loop if CX is not equal to zero and the zero flag is not set.

LOOPNZ	Assemble the 8088 loop if CX is not equal to zero and the zero flag is not set.
J	Assemble the 8088 jump always instruction.
FLABEL	Complete a forward branch instruction.
ZIF	Start an /IF construct, check if the zero flag is set.
NZIF	Start an /IF construct, check if the zero flag is not set.
SIF	Start an /IF construct, check if the sign flag is set.
NSIF	Start an /IF construct, check if the sign flag is not set.
CIF	Start an /IF construct, check if the carry flag is set.
NCIF	Start an /IF construct, check if the carry flag is set.
/BEGIN	Mark the beginning of a structured loop construct.
/REPEAT	Mark the end of a /BEGIN .../IF .../REPEAT construct.
NEXT	Mark the end of a word defined in assembler.
END-SUB	Mark the end of a subroutine defined with the word SUBROUTINE.
MEND	Mark the end of a macro definition.
/ENDIF	Mark the end of a conditional branching construct.
/ELSE	Allow an alternative branch in a conditional branching construct.
CODE	Start the definition of a word in assembler.
SUBROUTINE	Start the definition of an assembly language subroutine.
MACRO	Start the definition of a macro.

This chapter presents a complete 8088 structured macro assembler for Forth. An assembler is an integral part of any language system, a part neglected by most other languages. In Forth, the assembler is used to define individual words in machine language. Because words are used, it is easy to intermix high-level Forth and assembler in the same program.

Why would we want any part of our programs to be in assembly (or machine) language? As its name implies, machine language is *the* language of our computer. In the case of the IBM-PC, the 8088 CPU is what all programs execute on. Anything the computer can do is controlled by this very central processing unit. There may be times when even Forth is unable to access a specific machine feature. In these cases, we must have an assembler to accomplish our job at all.

Some cases need an assembler for efficiency. Whenever the 8088 CPU executes a Forth program there are the intermediate steps of stepping through all the words, until the lowest level machine language words are found. Most of the time, the IBM seems to have enough time to process our programs with no delay, but on occasion we need it to be faster. Rewriting a high-level Forth word in machine language using an assembler will speed up the execution of that word considerably. With a Forth assembler you can fine tune the performance of your programs.

In order to program in assembly language we must know a lot about the 8088 CPU. We must know how many registers it has. We must be able to see how it accesses memory. We must understand its instruction set. We need to know the function of each instruction the 8088 is capable of executing. But to actually write an assembler we need to know more. We need to know how the 8088 itself looks into memory and decodes its instructions. Learning assembly language can seem difficult; it is an involved subject. There are many

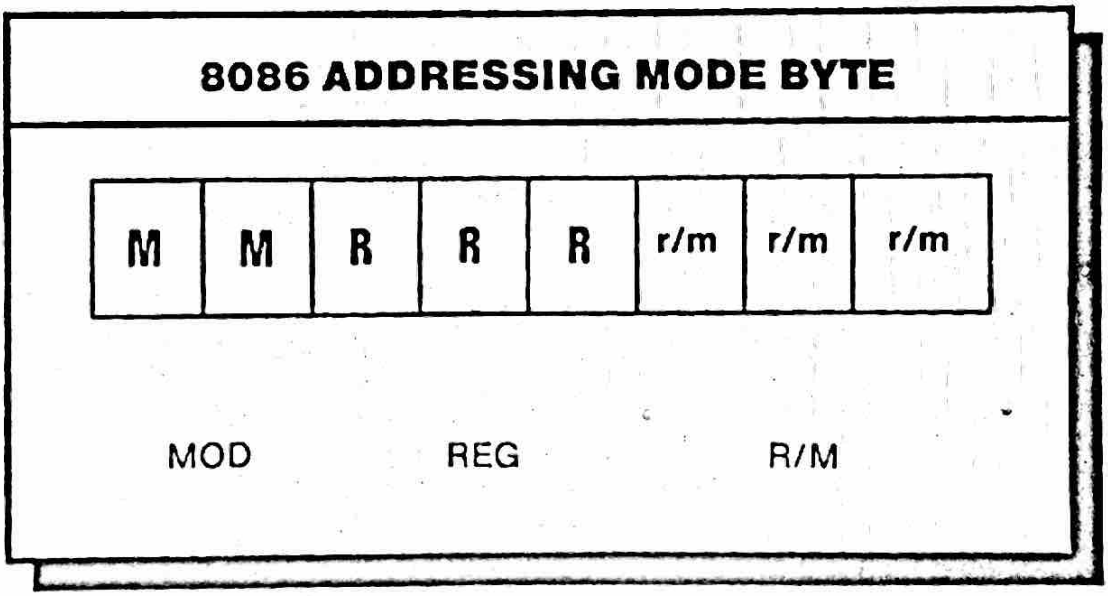


Figure 5-1

8086 EFFECTIVE ADDRESS DETERMINATION

R/M	MOD 00	MOD 01	MOD 10	MOD 11	MOD 11
000	[BX+SI]	[BX+SI+#]	[BX+SI+#]	AL	AX
001	(BX+DI)	(BX+DI+#)	(BX+DI+#)	CL	CX
010	[BP+SI]	[BX+SI+#]	[BX+SI+#]	DL	DX
011	(BP+DI)	(BX+DI+#)	(BX+DI+#)	BL	BX
100	[SI]	[SI+#]	[SI+#]	AH	SP
101	(DI)	(DI+#)	(DI+#)	CH	BP
110	Memory	[BP+#]	[BP+#]	DH	SI
111	[BX]	[BX+#]	[BX+#]	BH	DI
				w=0	w=1

Figure 5-2

books available on this subject, but since this book does not try to teach assembly-language programming, from this point on in this chapter we will assume a basic familiarity with the 8088 and assembly language.

Our Forth assembler will be used to define words. As in Forth itself, we'll have numerous structured constructs so we can write structured code. Because our scope is only a single word, we won't need a symbol table or many of the other features of a stand-alone assembler. We will have macros, though, mostly because they are so easy to implement in Forth.

The assembler will assemble each instruction into memory at the current dictionary pointer. A thorough study of the instruction set of the 8088 must be done before an assembler is written. A number of instructions are one- or two-byte constants, which are simple to write. They will use comma or c-comma to store their values in the dictionary. More complex instructions must deal with

the addressing modes of the 8088. The effective address calculation of an 8088 instruction employs an addressing mode byte that is of the form in Figure 5-1.

The REG field determines which register is used, or if no register is required by the instruction, it can be an extension of the opcode itself. The MOD and R/M bit fields of an addressing byte can be cross referenced to determine how the effective address of an instruction is determined.

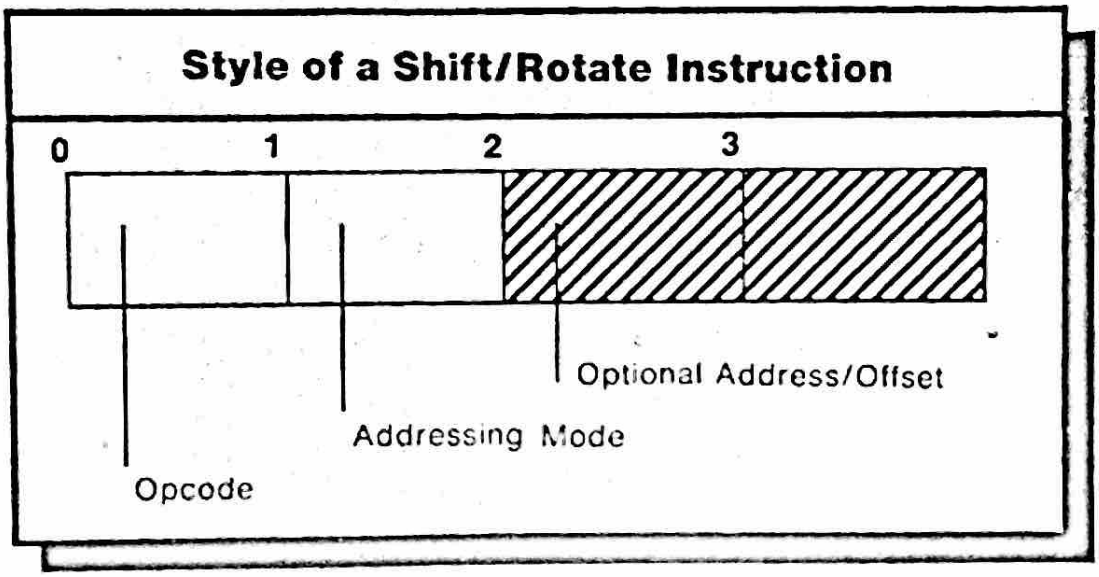
The W or size bit is held in the opcode itself.

We will use a two-celled variable called "tipe" to hold the type of addressing being used. As each operand is specified, "tipe" will be filled in appropriately.

A careful examination of the 8088 instruction set will find instructions that can be grouped together. Examples are the arithmetic and logical instructions, the shift and rotate instructions, and the branching opcodes. Each of these groups of opcodes has a very similar construction at the bit level. This information will allow us to write defining words that can cover the groups of instructions.

Let's take a look at the shift and rotate instructions, one of the simplest groups. Each shift and rotate instruction's base opcode has a unique seven bits, followed by an addressing mode byte, and by address or offset information, if it is required. (See Figure 5-3.) In each opcode there is a c bit, which the 8088 uses to determine how many bits are to be shifted. Since all the shift and rotate words only differ by the first seven bits, we can write one word that will assemble them all.

Other information we can derive from looking at the 8088 instruction set at the bit level will be used in our assembler. Every time a register is specified, for example, three bits are used. Throughout the instruction set they retain the same meaning. This enables us to use data from "tipe" for inclusion in the opcodes and addressing mode bytes.



Coding Instructions

When you write an instruction for a normal assembler, you specify the addressing mode of the instruction symbolically. MOV AX,BX and MOV AX, [BX] are different instructions because of the square brackets that specify indirect, instead of direct, addressing. Some instructions have no addressing mode, like PUSHF or NOP. Our 8088 assembler will use a similar method to specify addressing modes. Instructions that require no addressing mode in a normal assembler, like LAHF or CBW, will require none in our assembler. Every instruction that uses an address will, however, require an addressing mode indicator. Even those that, in a normal 8088 assembler, assume a default mode require an addressing mode indicator in our 8088 assembler. Here is a summary of the addressing modes used:

Forth Indicator	Normal	Forth
Register (Implied)	MOV AL,DL	AL DL MOV
# – Immediate	ADD AX,10	AX 10 # ADD
% – Direct	MOV AX,OFFSET JUNK	AX JUNK % MOV
[BX+SI] – Indirect	MOV AX, [BX+SI]	AX [BX+SI] MOV
(BX+DI) – Indirect	MOV DH, [BX+DI]	DH (BX+DI) MOV
[BP+SI] – Indirect	SUB AX, [BP+SI]	AX [BP+SI] SUB
(BP+DI) – Indirect	MOV AX,(BP+DI)	AX (BP+DI) MOV
[SI] – Indirect	MOV BX, [SI]	BX [SI] MOV
(DI) – Indirect	MOV AX, [DI]	AX (DI) MOV
[BX] – Indirect	MOV AL, [BX]	AL [BX] MOV
[BX+SI+#] – Ind. w/ offset	ADD AH, [BX+SI+3]	AH 3 [BX+SI+#] ADD
(BX+DI+#) – Ind. w/ offset	MOV AX, [BX+SI+9]	AX 9 (BX+DI+#) MOV
[BP+SI+#] – Ind. w/ offset	ADD DL, [BP+SI+1]	DL 1 [BP+SI+#] ADD
(BP+DI+#) – Ind. w/ offset	ADD DL,(BP+DI+22)	DL 22 (BP+DI+#) ADD
[SI+#] – Ind. w/ offset	MOV BX, [SI+325]	BX 325 [SI+#] MOV
(DI+#) – Ind. w/ offset	ADD AX,(DI+2)	AX 2 (DI+#) ADD
[BP+#] – Ind. w/ offset	ADD AX, [BP+12]	AX 12 [BP+#] ADD
[BX+#] – Ind. w/ offset	MOV CL, [BX+6]	CL 6 [BX+#] MOV

Note that all indirect instructions that use DI use parentheses instead of square brackets. This is because of the 3 character and length method of naming words found in many Forths.

Using the Assembler to Define Words

CODE and NEXT are the assembler defining words. For example, to code a word to ADD 10 to the top number on the stack (let's call it ADD10):

```
CODE ADD10 AX POP AX 10 # ADD AX PUSH NEXT
```

CODE will create a word with the name that follows in the input stream (ADD10 in this case) and set the context vocabulary to ASSEMBLER. NEXT will reset the context vocabulary to Forth. Two other defining words are available: SUBROUTINE and END–SUB. SUBROUTINE can be used to create a word that can be called with an assembler CALL or the high–level word CALL. Words created with CODE and NEXT cannot be called in this fashion—they are to be used in high-level defining words, that is, within words created by colon. For example, here is a subroutine that will transfer the AX register to the BX, CX, and DX registers.

```
SUBROUTINE TAX BX AX MOV CX AX MOV DX AX MOVE END–SUB
```

Remember that in subroutines the return address is the top value on the stack. The word SUBROUTINE creates a subroutine with the name next in the input stream and sets the context vocabulary to ASSEMBLER. END–SUB resets the context vocabulary to Forth.

Structured Assembler Words

Our Forth 8088 assembler includes a number of structured constructs to make coding in assembler easier. For looping, the /BEGIN ... cond structure is provided with the following conditional branching keywords: JA, JNBE, JAE, JNB, JB, JNAE, JBE, JNA, JCXZ, JE, JZ, JG, JNLE, JGE, JNL, JL, JNGE, JLE, JNG, JNE, JNZ, JNO, JNP, JPO, JNS, JO, JP, JPE, JS, LOOP, LOOPE, LOOPZ, LOOPNE, LOOPNZ, and J. The code segment:

```
AH 0 MOV AL 8 # MOV /BEGIN [BX] AH MOV BX INC AL DEC JNE
```

would zero out eight bytes pointed to by the BX register. /BEGIN ... cond loops can be nested to any desired level, but all jumps must be contained with the −126 to +129 byte limit of the 8088 conditional branching opcodes.

Conditional branching constructs are also provided in the form of IF ... / ELSE ... /ENDIF. The forms of IF provided include: ZIF, NZIF, SIF, and NSIF. They can be used in the following fashion:

```
CODE TEST NZIF AX PUSH /ENDIF NEXT
```

TEST will push the AX register on the stack if the zero flag is not set. /ELSE is

also available. For example:

```
CODE TEST NZIF AX -1 # MOV /ELSE AX 0 # MOV /ENDIF NEXT
```

This version of TEST will push a true flag if the zero flag is not set and a false flag if it is. The /ELSE branch is also limited to jumps of −126 to +129 bytes.

The /BEGIN ... WHILE ... /REPEAT construct is also provided. The WHILE is replaced by an IF−type word. This code segment will move a string that is pointed to by the BX register to the memory that is pointed to by the SI register, assuming that the string is terminated by a zero byte.

```
/BEGIN AX [BX] MOV NZIF [SI] AX MOV SI INC BX INC /REPEAT
```

MACROS

Our 8088 assembler contains the word MACRO, which will begin the definition of a macro. Macros are essentially a sort of text substitution used during the definition of assembler words. In Forth, macros are more powerful, since they are words themselves and can do processing that is useful. Most often, however, macros are used when the same portion of code must be utilized at many places during the definition of words in assembler. A macro can be defined once for a sequence of instructions, and then the single macro can be used in the places where the entire sequence was desired. For instance, let's say that while writing some words in assembler you often need to take the top number off the stack, add a base address to it, and place it back on the stack. We could define a macro like this:

```
MACRO ADD−BASE AX POP AX BASE # ADD AX PUSH MEND
```

The word MEND ends macro definitions. ADD−BASE could be used anyplace in an assembler word in place of the code in the macro. For example:

```
CODE EXAMP ADD−BASE NEXT
```

would be exactly equivalent to

```
CODE EXAMP AX POP AX BASE # ADD AX PUSH NEXT
```

Macros are more powerful than just shorthand. For example, let's say we often take the top number off the stack and store it in memory. This macro would handle it:

```
MACRO −>MEM AX POP % AX MOV MEND
```

This macro will expect a number on the stack when it is used and will generate code to store the top number on the stack at that address. If we wished to store the top number on the stack at memory location 800 (in the data segment)

```
CODE STOREAT800 800 ->MEM NEXT
```

would be exactly equivalent to

```
CODE STOREAT800 AX POP 800 % AX MOV NEXT
```

Not only do macros result in less text, but their proper use can result in much more readable assembler code.

NONSTRUCTURED CONSTRUCTS

The word FLABEL can be used to create nonstructured forward jumps by putting a zero on the stack before a branching conditional and FLABEL at the point of the desired forward branch. For example:

```
0 JZ AX POP AX POP FLABEL
```

This code segment would jump over the pulling of the top two words if the zero flag was set.

ASSEMBLER NOTES

Important: In Atila the following registers must be preserved by all assembler words: BP, DI, SI, CS, DS, ES, SS. The CS, DS, and ES registers all have the same value. Each version of Forth will have different requirements for which registers must be preserved and which may be destroyed. Please check the manual for the version of Forth you are using for this information.

In Atila the processor stack is used as the data stack. However, in some Forths the normal processor stack may not be the data stack. In this case, instructions like AX POP will not be able to be used for accessing the data stack. Again, check the documentation for your particular version of Forth.

When assembling a word, Forth is in interpretive mode and the context vocabulary is ASSEMBLER. Stack manipulation words, variables, and constants can all be used for address calculations. For instance, if you desired to access the third byte of the PAD in assembler, you could use the phrase PAD 3 + to obtain the desired address. Being in interpretive mode also means that if an error occurs, the word being assembled will not automatically be forgotten as it is when an error occurs in compile mode. You will find it necessary to

use the word FORGET to remove the definition from the dictionary.

In Atila, all numbers are stored on the stack as words. Double words have the most significant word most accessible.

Example:

This word will zero a byte in memory. It uses the top two words on the stack as a segment and offset, allowing access to all possible 8088 memory.

```
CODE 0SET-X
   DX ES MOV ( Save the ES register)
   ES POP ( Get the segment value)
   BX POP ( Get the address in the segment)
   ES SEG BX 0 # byte MOV ( Use extra segment, and zero the byte)
   ES DX MOV ( Restore the ES register)
NEXT
```

Suggested Extensions: The most useful extension of this assembler would be to include more thorough error checking for illegal or invalid instructions.

ASSEMBLER

Define a vocabulary for the assembler words.

```
( Define the vocabulary )
VOCABULARY ASSEMBLER

( Cause definitions to be entered into it)
ASSEMBLER DEFINITIONS HEX
```

1byte (N –) (–)

Define single-byte opcodes.

Stack on Entry: (Compile Time) N – Opcode value for instruction.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of use:

90 1byte NOP

Create the no operation instruction; when used, a 90 will be enclosed in the dictionary.

Algorithm: Store the value in the definition of the word. At run time, fetch that value and store it in the dictionary.

Suggested Extensions: None.

Definition:

```
: 1byte <BUILDS C, DOES> C@ C ,;

( Define all the single byte opcodes.)
37 1byte AAA 3F 1byte AAS 98 1byte CBW
F8 1byte CLC FC 1byte CLD FA 1byte CLI
F5 1byte CMC 99 1byte CWD 27 1byte DAA
2F 1byte DAS F4 1byte HLT CE 1byte INTO
CF 1byte IRET 9F 1byte LAHF
F0 1byte LOCK 90 1byte NOP
9D 1byte POPF 9C 1byte PUSHF
9E 1byte SAHF F9 1byte STC FD 1byte STD
FB 1byte STI 9B 1byte WAIT
D7 1byte XLAT F3 1byte REPE
F3 1byte REPZ F3 1byte REP
F2 1byte REPNE F2 1byte REPNZ
```

```
CODE EXAMPLE AX POP AAD CL 3 # MOV CL DIV AAM AX PUSH
NEXT
```

EXAMPLE will divide a two-digit BCD number on the stack by three and return the result in BCD on the top of the stack.

Algorithm: This is a two-byte constant opcode; use comma to enclose its opcode in the dictionary.

Suggested Extensions: None.

Definition:

```
AAD AD5 , ;
```

AAD (–)

Assemble the adjust AX register for division (AAD) instruction.

Stack unaffected.

Example of Use:

```
CODE EXAM:E AX POP AAD CL 3 # MOV CL DIV AAM AX PUSH NEXT
```

EXAMPLE will divide a two-digit BCD number on the stack by three and return the result in BCD on the top of the stack.

Algorithm: This is a two-byte constant opcode; use comma to enclose its opcode in the dictionary.

Suggested Extensions: None.

Definition:

```
AAD AD5 , ;
```

AAM (–)

Assemble the adjust result of BCD multiplication (AAM) instruction.

Stack unaffected.

Definition:

```
: AAM AD4 , ;
```

byte (–)

Cause the next instruction to use a byte-length operand.

Stack unaffected.

Example of Use:

... byte [BX] 0400 MOV ...

This would assemble into a move of the byte from address 400 to the address held in the BX register. Without "byte" the MOV instruction would not be able to determine if a byte or word length move should be encoded.

Algorithm: Set a variable (b/w) that will be checked when the opcode is stored in memory. This allows byte to work with all the instructions.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE b/w
: byte b/w C0SET ;
```

cell (–)

Cause the next instruction to use a word-length operand.

Stack unaffected.

Definition:

```
: cell b/w C1SET ;
```

tipe (– A)

A two-cell array holding the addressing mode.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of "tipe".

Example of Use: See following definitions.

Algorithm: The variable will hold the addressing mode that words we will define shortly need to store. This information will be accessed by the actual opcode words to determine the final value to store in memory. This is a two-cell array, because there can be values for both the source and destination operand of an instruction.

Suggested Extensions: None.

Definition:

```
0 VARIABLE tipe 0 ,
```

value (– A)

A two-cell array holding address values.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of "value".

Example of Use: See following definitions.

Algorithm: The variable will hold the address values that words we will define shortly need to store. This information will be accessed by the actual opcode words to determine the final value to store in memory. This is a two-cell array, because there can be values for both the source and destination operand of an instruction.

Suggested Extensions: None.

Definition:

```
0 VARIABLE value 0 ,
```

f/s+ (N1 – N2)

Point storage of operands to the correct field; source or destination.

Stack on Entry: (N1) – The address of "tipe" or "value".

Stack on Exit: (N2) – The proper value for storing data in "tipe" or "value".

Example of Use: See following words.

Algorithm: When an instruction is being assembled, there is a source and a destination operand. This word uses the variable f/s to keep track of which is currently being addressed. It will increment the value on the stack by two if

the second operand is being addressed. This enables the storing of the proper values in the variables "tipe" and "value," which are described below.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE f/s
: f/s+ f/s C@ IF 2+ ENDIF ;
```

next+ (–)

Increment f/s to allow addressing of the next operand in an instruction.

Stack unaffected.

Definition:

```
: next+ f/s C@ NOT f/s C! ;
```

(N –)

Cause an immediate operand to be assembled.

Stack on Entry: (N) – The immediate value to be used.

Stack on Exit: Empty.

Example of Use:

```
... AL 255 # MOV ...
```

This code would assemble an instruction to move the number 255 into the AL register.

Algorithm: Store the number in "value"; store a 45 as the type of addressing in "tipe".

Suggested Extensions: None.

Definition:

```
: # value f/s+ ! 45 tipe f/s+ ! next+ ;
```

](N -)

Cause an indirect memory addressing operand to be assembled.

Stack on Entry: (N) – The address value to be used.

Stack on Exit: Empty

Example of Use:

```
... cell 800 ] 0 # MOV ...
```

This code would assemble an instruction that would move a zero word into memory address 800 (as referenced by the data segment).

Algorithm: Store the address in "value"; store a 6 as the type of addressing in "tipe".

Suggested Extensions: None.

Definition:

```
: ]6 tipe f/s+ ! value f/s+ ! next+ ;
```

reset (–)

Reset "tipe" for the start of a new instruction.

Stack unaffected.

Example of Use: See words defined below.

Algorithm: A 99 is stored in both places in "tipe" to mark it as not in use. Zero was not used because it is a legal value in the r/m field of the addressing mode byte.

Suggested Extensions: None.

Definition:

```
: reset 99 tipe ! 99 tipe 2+ ! f/s COSET ;
```

00mod (N –) (–)

Define a word that assembles as a 00 mod in the addressing mode byte.

Stack on Entry: (Compile Time) (N) The value to store in "tipe".
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See words defined below.

Algorithm: The values to be stored in "tipe" are equivalent to the values that will have to be stored in the R/M field of the addressing mode byte. At compile time, store the value to be fetched at run time. At run time, store that value in "tipe".

Suggested Extensions: None.

Definition:

```
: 00mod <BUILDS C, DOES> C@ tipe f/s+ !
   next+ ;

( Define all 00 mod addressing modes)
0 00mod [BX+SI]          1 00mod (BX+DI)
2 00mod [BP+SI]          3 00mod (BP+DI)
4 00mod [SI]             5 00mod (DI)
                         7 00mod [BX]
40 00mod ES              41 00mod CS
42 00mod SS              43 00mod DS
```

01mod (N –) (–)

Define a word that assembles as a 01 mod in the addressing mode byte.

Stack on Entry: (Compile Time) (N) The value to store in "tipe".
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See words defined below.

Algorithm: The values to be stored in "tipe" are equivalent to the values that will have to be stored in the r/m field of the addressing mode byte. At compile time, store the value to be fetched at run time. At run time, fetch that value. Since all 00 mod operands require an offset, check to see if that offset is word or byte in length. If it is a word, add eight to the value to be stored in "tipe". Store the value in "tipe," then store the offset in "value".

Suggested Extensions: None.

Definition:

```
: 01mod <BUILDS C, DOES> C@ OVER 256 U>
   IF 8 + ENDIF tipe f/s+ !
   value f/s+ ! next+ ;

   8 01mod [BX+SI+#]          9 01mod (BX+DI+#)
   10 01mod [BP+SI+#]         11 01mod (BP+DI+#)
   12 01mod [SI+#]            13 01mod (DI+#)
   14 01mod [BP+#]            15 01mod [BX+#]
```

11mod (N1 N2 -) (-)

Define a word that assembles as a 11 mod in the addressing mode byte.

Stack on Entry: (Compile Time) (N1) The byte or word value.
The value to store in "tipe".
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See words defined below.

Algorithm: Again the values to be stored in "tipe" are equivalent to the values that will have to be stored in the R/M field of the addressing mode byte. At compile time, we store both the byte or word value and the "tipe" value. At run time, these values are fetched and stored in "tipe" and "b/w".

Suggested Extensions: None.

Definition:

```
: 11mod <BUILDS C, C, DOES> DUP 1+ C@
   b/w C! C@ tipe f/s+ ! next+ ;
```

```
( Define the 11 mod instructions.)

0 24 11mod AL    1 24 11mod AX
0 25 11mod CL    1 25 11mod CX
0 26 11mod DL    1 26 11mod DX
0 27 11mod BL    1 27 11mod BX
0 28 11mod AH    1 28 11mod SP
0 29 11mod CH    1 29 11mod BP
0 30 11mod DH    1 30 11mod SI
0 31 11mod BH    1 31 11mod DI
```

direction (N1 – N2)

Set the direction bit in the opcode being formed.

Stack on Entry: (N1) The opcode.

Stack on Exit: (N2) The opcode with the direction bit set properly.

Example of Use: See words defined below.

Algorithm: Use the variable "dir" to hold the direction. Fetch the value from the dir variable and use "asl" to move it into the proper place in the instruction. (Bit one in this case).

Suggested Extensions: None.

Definition:

```
: asl 0 DO 2* LOOP ;
0 CVARIABLE dir
: direction dir C@ 1 asl OR ;
```

Size (N1 – N2)

Set the size bit in the opcode being formed.

Stack on Entry: (N1) The opcode.

Stack on Exit: (N2) The opcode with the size bit set properly.

Example of Use: See words defined below.

Algorithm: Use the variable "b/w", which holds the size of the current operand, byte or word. The size bit is bit zero in the operand.

Suggested Extensions: None.

Definition:

```
: size b/w C@ OR ;
```

r/m (N1 – N2)

Set the r/m field in the addressing mode byte being formed.

Stack on Entry: (N1) The addressing mode byte.

Stack on Exit: (N2) The addressing mode byte with the r/m field set properly.

Example of Use: See words defined below.

Algorithm: The variable "tipe" holds both the r/m and mod field information. Extract the r/m data from "tipe" and "OR" it into the addressing mode byte. The r/m field is the first three bits of the addressing mode byte, so no shifting is required.

Suggested Extensions: None.

Definition:

```
: r/m@ tipe @ 8 MOD ;
: r/m r/m@ OR ;
```

md (N1 – N2)

Set the mod field in the addressing mode byte being formed.

Stack on Entry: (N1) The addressing mode byte.

Stack on Exit: (N2) The addressing mode byte with the mod field set properly.

Example of Use: See words defined below.

Algorithm: The variable "tipe" holds both the r/m and mod field information. Extract the mod data from "tipe" and "OR" it into the addressing mode byte. Since the mod field is the two high bits of the addressing mode byte, it will be neccessary to shift the data 6 bits before "ORing" it.

Suggested Extensions: None.

Definition:

```
: md@ tipe @ 8 / ;
: md md@ 6 asl OR ;
```

,value (–)

Set the mod field in the addressing mode byte being formed.

Stack unaffected.

Example of Use: See words defined below.

Algorithm: Check the mod field to see if we are accessing a byte or word register. Also check for a memory access. Store the proper address from the variable "value" as a byte or word in the dictionary using the comma words.

Suggested Extensions: None.

Definition:

```
: ,value md@ 1 = IF
    value C@ C,
  ELSE
    md@ 2 = tipe @ 6 = OR IF
      value @ ,
    ENDIF
  ENDIF ;
```

reg@ (– N)

Get the register value stored in the first position of "tipe".

Stack on Entry: Empty.

Stack on Exit: (N) The register value stored in "tipe".

Example of Use: See words defined below.

Algorithm: Fetch the proper value and extract the first three bits.

Suggested Extensions: None.

Definition:

```
: 1reg@ tipe @ 8 MOD ;
```

reg@ (– N)

Get the register value stored in the current position of "tipe".

Stack on Entry: Empty.

Stack on Exit: (N) The register value stored in "tipe".

Example of Use: See words defined below.

Algorithm: Fetch the proper value and extract the first three bits.

Suggested Extensions: None.

Definition:

```
: reg@ tipe 2+ @ 8 MOD ;
```

1reg (N1 – N2)

Set the register field in the opcode being formed.

Stack on Entry: (N1) The opcode.

Stack on Exit: (N2) The opcode with the mod field set properly.

Example of Use: See words defined below.

Algorithm: Fetch the proper value and extract the first three bits. Shift the value three places to the left and "OR" it into the opcode.

Suggested Extensions: None.

Definition:

```
: 1reg 1reg@ 3 asl OR ;
```

reg (N1 – N2)

Set the register field in the addressing mode byte being formed.

Stack on Entry: (N1) The addressing mode byte.

Stack on Exit: (N2) The addressing mode byte with the mod field set properly.

Example of Use: See words defined below.

Algorithm: Fetch the proper value and extract the first three bits. Shift the value three places to the left and "OR" it into the addressing mode byte.

Suggested Extensions: None.

Definition:

```
: reg reg@ 3 asl OR ;
```

a–mode (N –)

Form and store an addressing mode byte. Store the instruction's address values.

Stack on Entry: (N) – The addressing mode byte base.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Set the mod and r/m fields, then store the byte. Use ",value" to store the address information.

Suggested Extensions: None.

Definition:

```
: a-mode md r/m C, ,value ;
```

16-bit-reg? (- B)

Is a 16 bit register being used?

Stack on Entry: Empty.

Stack on Exit: (B) Boolean value determining whether or not a 16-bit register is being used in this instruction.

Example of Use: See words defined below.

Algorithm: Check the value held in "tipe".

Suggested Extensions: None.

Definition:

```
: 16-bit-reg? tipe @ DUP 23 > SWAP 32 <
  AND b/w C@ AND ;
```

seg-reg? (- B)

Is a segment register being used?

Stack on Entry: Empty.

Stack on Exit: (B) Boolean value determining whether or not a segment register is being used in this instruction.

Example of Use: See words defined below.

Algorithm: Check the value held in "tipe".

Suggested Extensions: None.

Definition:

```
: seg-reg? tipe @ DUP 39 > SWAP 44 <
   AND ;
```

DEC

Assemble a decrement instruction.

Stack unaffected.

Example of Use:

```
... AX DEC ...
```

This code would assemble and decrement the AX register instruction.

Algorithm: Form the instruction by setting the base value and then using the words we have defined that fill in each field of an opcode. Reset "tipe" for the next instruction.

Suggested Extensions: None.

Definition:

```
: DEC 16-bit-reg? IF
      1reg@ 72 OR C,
   ELSE
      254 size C,
      8 a-mode
   ENDIF reset ;
```

INC

Assemble an increment instruction.

Stack unaffected.

Example of Use:

```
.. BX INC ...
```

This code would assemble and increment the BX register instruction.

Algorithm: Form the instruction by setting the base value and then using the words we have defined that fill in each field of an opcode. Reset "tipe" for the next instruction.

Suggested Extensions: None.

Definition:

```
: INC 16-bit-reg? IF
     1reg@ 64 OR C,
  ELSE
     254 size C,
     0 a-mode
  ENDIF reset ;
```

POP

Assemble a pop-the-stack instruction.

Stack unaffected.

Example of Use:

```
... DX POP ...
```

This code would assemble a pop of the DX register instruction.

Algorithm: Form the instruction by setting the base value and then using the words we have defined that fill in each field of an opcode. Reset "tipe" for the next instruction. The base value will depend on the type of register or memory being popped.

Suggested Extensions: None.

Definition:

```
: POP seg-reg? IF
```

```
    7 tipe @ 40 - 3 asl OR C,
  ELSE
    16-bit-reg? IF
      1reg@ 88 OR C,
    ELSE
      143 C, 0 a-mode
    ENDIF
  ENDIF reset ;
```

PUSH

Assemble a stack push instruction.

Stack unaffected.

Example of Use:

```
... AX PUSH ...
```

This code would assemble a push of the AX register instruction.

Algorithm: Form the instruction by setting the base value and then using the words we have defined that fill in each field of an opcode. Reset "tipe" for the next instruction. The base value will depend on the type of register or memory being popped.

Suggested Extensions: None.

Definition:

```
: PUSH seg-reg? IF
     6 tipe @ 40 - 3 asl OR C,
   ELSE
     16-bit-reg? IF
       1reg@ 80 OR C,
     ELSE
       255 C, 48 a-mode
     ENDIF
   ENDIF reset ;
```

ac,data? (- B)

Check if the operands for this instruction are a move of data into an accumulator.

Stack on Entry: Empty.

Stack on Exit: (B) – Flag, true if this is a move of data into an accumulator.

Example of Use: See words defined below.

Algorithm: Check the values stored in "tipe" by the operands.

Suggested Extensions: None.

Definition:

```
: ac,data? tipe @ 24 = tipe 2+ @ 45 =
   AND ;
```

immediate? (– B)

Check if there is an immediate operand for this instruction.

Stack on Entry: Empty.

Stack on Exit: (B) – Flag, true if this instruction has an immediate operand.

Example of Use: See words defined below.

Algorithm: Check the values stored in "tipe" by the second operand.

Suggested Extensions: None.

Definition:

```
: immediate? tipe 2+ @ 45 = ;
```

eb/w (N –)

Store a displacement or address into memory.

Stack on Entry: (N) The displacement or address to store.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Check the variable b/w to see if a byte or word should be stored in memory.

Suggested Extensions: None.

Definition:

```
: eb/w b/w C@ IF
    .
  ELSE
    C,
  ENDIF ;
```

,data (–)

Store the displacement or address for an instruction.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Use the second entry in the variable "value".

Suggested Extensions: None.

Definition:

```
: ,data value 2+ @ eb/w ;
```

s–bit (N1 – N2)

Set the s or sign Extension bit in an opcode.

Stack on Entry: (N1) The opcode.

Stack on Exit: (N2) The opcode with the s bit set.

Example of Use: See words defined below.

Algorithm: Determine if the immediate data can be represented in a single, signed byte. If it can, set the s bit, which is bit 1 in the opcode.

Suggested Extensions: None.

Definition:

```
: s-bit value 2+ @ DUP 128 < SWAP -128 > AND IF
    2 OR
  ENDIF ;
```

e-mode (-)

Form a complete addressing mode byte.

Stack unaffected.

Example of Use: See words defined below.

Algorithm: This word is used for opcodes that utilize a complete addressing mode byte. Place a zero on the stack, then use the md, reg, and r/m words to form the addressing mode byte. Store the addressing mode byte in memory and then store immediate or address data with ,value.

Suggested Extensions: None.

Definition:

```
: e-mode 0 md reg r/m C, ,value ;
```

swap-dir (-)

Mark an instruction's operands as moving data in the opposite direction.

Stack unaffected.

Example of Use: See words defined below.

Algorithm: Swap the data in the two positions of "tipe" and "value". Set the direction flag.

Suggested Extensions: None.

Definition:

```
: swap-dir
        tipe DUP @ SWAP 2+ @ tipe !
        tipe 2+ !
        value DUP @ SWAP 2+ @ value !
        value 2+ !
        dir C1SET ;
```

dir? (–)

Check the direction an instruction's operands are moving data. Swap the direction, if appropriate.

Stack unaffected.

Example of Use: See words defined below.

Algorithm: Moves to a register or from a segmentation register are valid; all others are suspect and should be switched. Check "tipe" to determine the course of action to be taken.

Suggested Extensions: Extend this word to check for illegal combinations of opcodes.

Definition:

```
: dir? tipe @ 23 > tipe 2+ @ DUP 39 >
   SWAP 44 < AND NOT AND IF
      swap-dir dir C1SET
   ELSE
      dir C0SET
   ENDIF ;
```

1kind (N1 N2 –) (–)

Assemble the arithmetic and logical instructions that use two operands.

Stack on Entry: (Compile Time) (N1) – The base opcode for an operation that will involve immediate data and the accumulator.

(Compile Time) (N2) – The base opcode for other kinds of operations.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See words defined below.

Algorithm: At compile time, store the base operands in the dictionary. At run time, determine the nature of the instruction and form the opcodes and addressing mode bytes accordingly. These type of instructions have a full range of addressing options and have the leftover 8080 opcode equivalents.

Suggested Extensions: Extend this word to check for illegal combinations of opcodes.

Definition:

```
: 1kind <BUILDS C, C, DOES>
   ac,data? IF
      C@ size C, ,data
   ELSE
      immediate? IF
         128 s-bit size C,
         1+ C@ a-mode ,data
      ELSE
         dir?
         1+ C@ direction size C, e-mode
      ENDIF
   ENDIF reset ;

   ( Define the logic and arithmetic opcodes.)

   0 4 1kind ADD                16 20 1kind ADC
   32 36 1kind AND              56 60 1kind CMP
   8 12 1kind OR                24 28 1kind SBB
   40 44 1kind SUB              48 52 1kind XOR
```

TEST (–)

Assemble the 8088 test instruction.

Stack unaffected.

Example of Use:

```
... byte 1 [BX+#] 4 TEST ...
```

This sequence would assemble a test of the byte pointed to by the BX register plus one.

Algorithm: Determine what operands are being used for this instruction and form the opcodes and addressing mode bytes accordingly. This instruction has a full range of addressing options and has a leftover 8080 opcode equivalent.

Suggested Extensions: Extend this word to check for illegal combinations of opcodes.

Definition:

```
: TEST ac,data? IF
     168 size C, ,data
   ELSE
     immediate? IF
       246 size C,
       0 a-mode ,data
     ELSE
       dir? 132 size C, e-mode
     ENDIF
   ENDIF reset ;
```

c-bit (N1 – N2)

Set the c bit for shift and rotate instructions.

Stack on Entry: (N1) The opcode.

Stack on Exit: (N2) The opcode with the c bit set appropriately.

Example of Use: See words defined below.

Algorithm: See if the CL register was specified as the second operand. If it was not, assume a single-bit shift or rotate.

Suggested Extensions: Extend this word to check for values in "tipe" for other than the CL register.

Definition:

```
: c-bit tipe 2+ @ 25 = IF 2 OR ENDIF ;
```

srkind (N –) (–)

Assemble the rotate and shift instructions.

Stack on Entry: (Compile Time) (N) – The base opcode for the instruction being defined.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See words defined below.

Algorithm: At compile time, store the base opcode in the dictionary. At run time, form the instruction by checking to see if the CL register specifies the number of bits to rotate.

Suggested Extensions: Extend this word to check for illegal combinations of opcodes.

Definition:

```
: srkind <BUILDS C, DOES>
   208 c-bit size C, C@ a-mode reset ;

   ( Define the shift and rotate instructions.)
   16 srkind RCL          24 srkind RCR
   0 srkind ROL           8 srkind ROR
   56 srkind SAR          32 srkind SHL
   32 srkind SHL          40 srkind SHR
```

LDS (–)

Assemble the 8088 load of a register and DS from memory instruction.

Stack unaffected.

Example of Use:

... AX [BX] LDS ...

This sequence would assemble a load of the AX and DS registers from the four bytes pointed to by the BX register.

Algorithm: Enclose the opcode constant in the dictionary, then form the addressing mode byte using e–mode.

Suggested Extensions: None.

Definition:

```
: LDS 197 C, dir? e–mode reset ;
```

LEA (–)

Assemble the 8088 load of a register with the effective address of an instruction.

Stack unaffected.

Example of Use:

... AX 6 [BX+SI+#] LEA ...

This sequence would assemble a load of the AX register with the sum of BX, SI, and six.

Algorithm: Enclose the opcode constant in the dictionary, then form the addressing mode byte using e–mode.

Suggested Extensions: None.

Definition:

```
: LEA 141 C, dir? e–mode reset ;
```

LES (–)

Assemble the 8088 load of a register and ES from memory instruction.

Stack unaffected.

Example of Use:

```
... DX 125 [BX+#] LES ...
```

This sequence would assemble a load of the DX and ES registers from the four bytes pointed to by the BX register plus 125.

Algorithm: Enclose the opcode constant in the dictionary, then form the addressing mode byte using e–mode.

Suggested Extensions: None.

Definition:

```
: LES 196 C, dir? e–mode reset ;
```

mkind (N –) (–)

Assemble the mathematical instructions.

Stack on Entry: (Compile Time) (N) – The base opcode for the instruction being defined.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See words defined below.

Algorithm: At compile time, store the base opcode in the dictionary. At run time, form the instruction by checking "tipe" to determine the addressing mode being used by the instruction.

Suggested Extensions: Extend this word to check for illegal combinations of opcodes.

Definition:

```
: mkind <BUILDS C, DOES>
   246 size C, C@ a–mode reset ;

   ( Define the mathematical opcodes)
```

```
48 mkind DIV          56 mkind IDIV
40 mkind IMUL         32 mkind MUL
24 mkind NEG          16 mkind Not
```

2kind (N –) (–)

Assemble the mathematical instructions.

Stack on Entry: (Compile Time) (N) – The base opcode for the instruction being defined.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See words defined below.

Algorithm: At compile time, store the base opcode in the dictionary. At run time, set the size bit in the opcode and store it.

Suggested Extensions: Extend this word to check for illegal combinations of opcodes.

Definition:

```
: 2kind <BUILDS C, DOES>
   C@ size C, reset ;

   ( Define the string opcodes.)

   166 2kind CMPS          172 2kind LODS
   164 2kind MOVS          174 2kind SCAS
   170 2kind STOS
```

IN (–)

Assemble the 8088 transfer of data from a port to accumulator instruction.

Stack unaffected.

Example of Use:

```
... AL 8 # IN ...
```

This sequence would assemble an instruction to transfer data from I/O port eight to the AL register.

Algorithm: Check the operands by looking at "tipe". Only accumulator and DX addressing are valid. Store the opcode in the dictionary.

Suggested Extensions: Extend this word to check for illegal combinations of operands.

Definition:

```
: IN tipe @ 26 = IF
     236 size C,
  ELSE
     228 size C, value 2+ @ C,
  ENDIF reset ;
```

OUT (–)

Assemble the 8088 transfer of data from the accumulator to a port instruction.

Stack unaffected.

Example of Use:

```
... DX AX OUT ...
```

This sequence would assemble an instruction to transfer data from the AX register to the I/O port specified by the value in the DX register.

Algorithm: Check the operands by looking at "tipe". Only accumulator and DX addressing are valid. Store the opcode in the dictionary.

Suggested Extensions: Extend this word to check for illegal combinations of operands.

Definition:

```
: OUT tipe @ 26 = IF
     238 size C,
  ELSE
     230 size C, value @ C,
  ENDIF reset ;
```

INT (–)

Assemble the 8088 software interrupt instruction.

Stack unaffected.

Example of Use:

```
... 10 # INT ...
```

This sequence would assemble a number ten interrupt.

Algorithm: Check the operands by looking at "tipe". Only immediate mode or implicit addressing is valid. Store the opcode in the dictionary.

Suggested Extensions: Extend this word to check for illegal combinations of operands.

Definition:

```
: INT tipe @ 45 = IF
     205 C, value @ C,
  ELSE
     204 C,
  ENDIF reset ;
```

SEG (–)

Assemble the 8088 segment override instruction.

Stack unaffected.

Example of Use:

```
... ES SEG AX [BX] MOV ...
```

This sequence would assemble an extra segment override instruction that would cause the effective address of the following MOV instruction to reference data in the extra segment instead of the data segment.

Algorithm: Check the operands by looking at "tipe". The segmentation registers (ES, CS, SS, and DS) are the only valid operands. Store the opcode in the dictionary.

Suggested Extensions: Extend this word to check for illegal operands.

Definition:

```
: SEG 38 tipe @ 40 - 3 asl OR C, reset ;
```

XCHG (-)

Assemble the 8088 exchange data instruction.

Stack unaffected.

Example of Use:

```
... AX BX XCHG ...
```

This sequence would assemble an exchange of the AX and BX registers instruction.

Algorithm: Determine what operands are being used for this instruction and form the opcodes and addressing mode bytes accordingly.

Suggested Extensions: Extend this word to check for illegal operands.

Definition:

```
: XCHG tipe @ DUP 23 > SWAP 32 < AND
  b/w C@ AND IF
     144 tipe @ 24 - OR C,
  ELSE
     swap-dir 134 size C, e-mode
  ENDIF reset ;
```

LONG-CALL (-)

Assemble the 8088 long (intersegment) subroutine call instruction.

Stack unaffected.

Example of Use:

```
... 50 0 LONG-CALL ...
```

This sequence would assemble a call to offset 50 in segment zero.

Algorithm: Determine what operands are being used for this instruction and form the opcodes and addressing mode bytes accordingly. If "tipe" is not set, assume the long address is on the stack.

Suggested Extensions: Extend this word to check for illegal operands.

Definition:

```
: LONG-CALL tipe @ 45 = IF
     255 C, 24 a-mode
  ELSE
     154 C, SWAP , ,
  ENDIF reset ;
```

CALL (-)

Assemble the 8088 short (intrasegment) subroutine call instruction.

Stack unaffected.

Example of Use:

```
... [BX] CALL ...
```

This sequence would assemble a call to the address in the BX register.

Algorithm: Determine what operands are being used for this instruction and form the opcodes and addressing mode bytes accordingly. If an immediate address is being called, determine the offset from the current address for inclusion in the assembled code.

Suggested Extensions: Extend this word to check for illegal operands.

Definition:

```
: CALL tipe @ 99 <> IF
     255 C, 16 a-mode
  ELSE
     HERE 232 C, - 3 - ,
  ENDIF reset ;
```

RET (–)

Assemble the 8088 short (intrasegment) subroutine return instruction.

Stack unaffected.

Example of Use:

```
... RET ...
```

This sequence would assemble a return instruction.

Algorithm: Check "tipe" for the addressing mode being used. Only immediate and implicit modes are valid. The immediate mode causes the stack pointer to be adjusted upon return from the subroutine.

Suggested Extensions: Extend this word to check for illegal operands.

Definition:

```
: RET tipe @ 45 = IF
     194 C, value @ ,
  ELSE
     195 C,
  ENDIF reset ;
```

LONG–RET (–)

Assemble the 8088 long (intrasegment) subroutine return instruction.

Stack unaffected.

Example of Use:

```
... 6 LONG-RET ...
```

This sequence would assemble a long return instruction that would also add 6 to the stack pointer.

Algorithm: Check "tipe" for the addressing mode being used. Only immediate and implicit modes are valid. The immediate mode causes the stack pointer to be adjusted upon return from the subroutine.

Suggested Extensions: Extend this word to check for illegal operands.

Definition:

```
: LONG-RET tipe @ 45 = IF
    202 C, value @ ,
  ELSE
    203 C ,
  ENDIF reset ;
```

LONG-BRANCH (-)

Assemble the 8088 long (intersegment) jump instruction.

Stack unaffected.

Example of Use:

```
... 500 0 LONG-BRANCH ...
```

This sequence would assemble a call to offset 500 in segment zero.

Algorithm: Determine what operands are being used for this instruction and form the opcodes and addressing mode bytes accordingly. If "tipe" is not set, assume the long address is on the stack.

Suggested Extensions: Extend this word to check for illegal operands.

Definition:

```
: LONG-BRANCH tipe @ 99 <> IF
    255 C, 40 a-mode
  ELSE
    234 C, SWAP ,,
  ENDIF ;
```

BRANCH (-)

Assemble the 8088 short (intrasegment) jump instruction.

Stack unaffected.

Example of Use:

```
... [BX] BRANCH ...
```

This sequence would assemble a jump to the address in the BX register.

Algorithm: Determine what operands are being used for this instruction and form the opcodes and addressing mode bytes accordingly. If an immediate address is being called, determine the offset from the current address for inclusion in the assembled code. Note that immediate addressing can have a byte or word offset.

Suggested Extensions: Extend this word to check for illegal operands.

Definition:

```
: BRANCH tipe @ 99 <> IF

    255 C, 32 a-mode
  ELSE
    HERE - DUP DUP 127 < SWAP -127 >
    AND IF
      235 C, 3 - C,
    ELSE
      233 C, 3 - ,
    ENDIF
  ENDIF reset ;
```

MOV (-)

Assemble the 8088 data move instruction.

Stack unaffected.

Example of Use:

```
... DX 14 [BX+SI+#] MOV ...
```

This sequence would assemble a move of the word found at the address specified by the sum of the BX register, SI register, and 14 to the DX register. The address is in the data statement.

Algorithm: Determine what operands are being used for this instruction and form the opcodes and addressing mode bytes accordingly. Note that MOV encompasses all the different addressing modes.

Suggested Extensions: Extend this word to check for illegal operands.

Definition:

```
: MOV dir? tipe @ 24 = tipe 2+ @ 6 = AND IF
    160 direction size C, value 2+ @ ,
  ELSE
    tipe 2+ @ DUP 40 >= SWAP 44 <= AND IF
      140 direction C, e-mode
    ELSE
      tipe @ 45 = tipe 2+ @ DUP 23 > SWAP 32 < AND AND IF
        176 tipe 2+ @ 24 - OR b/w C@ 3 asl
        OR C, value @ eb/w
      ELSE
        tipe 2+ @ 45 = IF
          198 size C, 0 a-mode ,data
        ELSE
          136 direction size C, e-mode
        ENDIF
      ENDIF
    ENDIF
  ENDIF reset ;
```

bopc (N –) (N –)

Assemble the 8088 branching opcodes.

Stack on Entry: (Compile Time) (N) – The base opcode for the instruction.
(Run Time) (N) The address to jump to or zero if a forward jump is being assembled.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See words defined below.

Algorithm: At compile time, store the base operands in the dictionary. At run time, check the stack for the address to jump to. If the address is zero, leave the address of the current instruction on the stack and set the length of the jump

to zero. If the address on the stack is not zero, determine the offset to that address and enclose it in the dictionary.

Suggested Extensions: Extend this word to check for branches out of range.

Definition:

```
HEX
: bopc <BUILDS C, DOES> C@ C, DUP 0= IF
      C, HERE
   ELSE
      HERE - 1 - C,
   ENDIF reset ;

   ( Assemble the branching opcodes )

   77 bopc JA        77 bopc JNBE      73 bopc JAE
   73 bopc JNB       72 bopc JB        72 bopc JNAE
   76 bopc JBE       76 bopc JNA       E3 bopc JCXZ
   74 bopc JE        74 bopc JZ        7F bopc JG
   7F bopc JNLE      7D bopc JGE       7D bopc JNL
   7C bopc JL        7C bopc JNGE      7E bopc JLE
   7E bopc JNG       75 bopc JNE       75 bopc JNZ
   71 bopc JNO       7B bopc JNP       7B bopc JPO
   79 bopc JNS       70 bopc JO        7A bopc JP
   7A bopc JPE       78 bopc JS        E2 bopc LOOP

E1 bopc LOOPE E1 bopc LOOPZ
E0 bopc LOOPNE E0 bopc LOOPNZ
EB bopc J
DECIMAL
```

FLABEL (A –)

Complete a forward branch instruction.

Stack on Entry: (A) The address the branch instruction is at.

Stack on Exit: Empty.

Example of Use:

```
... 0 JZ AH INC FLABEL ...
```

This sequence would assemble a jump around the increment AH instruction if the zero flag was set.

Algorithm: Determine the offset from the current position to the address of the branch on the stack. Fill in the length of the jump in the branch instruction.

Suggested Extensions: Extend this word to check for jumps out of range.

Definition:

```
: FLABEL 1- DUP HERE - 1+ ABS SWAP C! ;
```

ZIF (-)

Start an /IF construct, checking whether the zero flag is set.

Stack unaffected.

Example of Use:

```
... ZIF SKIPIT CALL /ENDIF ...
```

This sequence would assemble the instructions to call the subroutine SKIPIT if the zero flag is set.

Algorithm: Assemble a forward jump so that if the zero flag is not set (JNZ), /ELSE or /ENDIF will fill in.

Suggested Extensions: None.

Definition:

```
: ZIF 0 JNZ ;
```

NZIF (-)

Start an /IF construct, checking whether the zero flag is not set.

Stack unaffected.

Example of Use:

```
... NZIF SKIPIT CALL /ENDIF ...
```

This sequence would assemble the instructions to call the subroutine SKIPIT if the zero flag is not set.

Algorithm: Assemble a forward jump so that if the zero flag is set (JZ), /ELSE or /ENDIF will fill in.

Suggested Extensions: None.

Definition:

```
: NZIF 0 JZ ;
```

SIF (–)

Start an /IF construct, checking whether the sign flag is set.

Stack unaffected.

Example of Use:

```
... SIF SKIPIT CALL /ENDIF ...
```

This sequence would assemble the instructions to call the subroutine SKIPIT if the sign flag is set.

Algorithm: Assemble a forward jump so that if the sign flag is not set (JNS), / ELSE or /ENDIF will fill in.

Suggested Extensions: None.

Definition:

```
: SIF 0 JNS ;
```

NSIF (–)

Start an /IF construct, checking whether the sign flag is not set.

Stack unaffected.

Example of Use:

```
... NSIF SKIPIT CALL /ENDIF ...
```

This sequence would assemble the instructions to call the subroutine SKIPIT if the sign flag is not set.

Algorithm: Assemble a forward jump so that if the sign flag is set (JS), /ELSE or /ENDIF will fill in.

Suggested Extensions: None.

Definition:

```
: NSIF 0 JS ;
```

CIF (–)

Start an /IF construct, checking whether the carry flag is set.

Stack unaffected.

Example of Use:

```
... CIF SKIPIT CALL /ENDIF ...
```

This sequence would assemble the instructions to call the subroutine SKIPIT if the carry flag is set.

Algorithm: Assemble a forward jump so that if the carry flag is not set (JNB), /ELSE or /ENDIF will fill in.

Suggested Extensions: None.

Definition:

```
: CIF 0 JNB ;
```

NCIF (–)

Start an /IF construct, checking whether the carry flag is not set.

Stack unaffected.

Example of Use:

```
... NCIF SKIPIT CALL /ENDIF ...
```

This sequence would assemble the instructions to call the subroutine SKIPIT if the carry flag is not set.

Algorithm: Assemble a forward jump so that if the carry flag is set (JB), /ELSE or /ENDIF will fill in.

Suggested Extensions: None.

Definition:

```
: NCIF 0 JB ;
```

/BEGIN (–)

Mark the begining of a structured loop construct.

Stack unaffected.

Example of Use:

```
... /BEGIN READY? CALL JNZ ...
```

This sequence would assemble a loop of calling the subroutine READY? until the zero flag was set.

Algorithm: Leave the current dictionary pointer on the stack. This will be the address that the branching words will want to assemble a jump to.

Suggested Extensions: None.

Definition:

```
: /BEGIN HERE ;
```

/REPEAT (A1 A2 –)

Mark the end of a /BEGIN .. /IF .. /REPEAT construct.

Stack on Entry: (A1) – The address the /IF instruction's branch is at.
(A2) – The address the /BEGIN left on the stack.

Stack on Exit: Empty.

Example of Use:

```
... /BEGIN READY? CALL NZIF AL 0 # MOV 8 # AL OUT /REPEAT ...
```

This sequence would assemble the instructions to call the subroutine READY? and keep sending a zero out to port eight until READY? returns with the zero flag set.

Algorithm: First assemble the absolute jump back to the /BEGIN, then fill in the length byte of the branch instruction that can exit the loop.

Suggested Extensions: None.

Definition:

```
: /REPEAT SWAP BRANCH FLABEL ;
```

NEXT (–)

Mark the end of a word defined in assembler.

Stack unaffected.

Example of Use:

```
... CODE OUT AX POP DX POP DX AX OUT NEXT ...
```

This is a complete word that would cause a specific value to be sent out a specific port. NEXT ends the definition and can be thought of as the equivalent of ";" (semi).

Algorithm: Assemble a jump to the address in the BP register, which in Atila holds the address of the inner interpreter NEXT routine. Set the dictionary to be searched back to FORTH.

Suggested Extensions: None.

Definition:

```
: NEXT BP BRANCH FORTH ;
```

END–SUB (–)

Mark the end of a subroutine defined with the word SUBROUTINE.

Stack unaffected.

Example of Use:

```
... SUBROUTINE INC–BIG–NUM AX INC ZIF BX INC /ENDIF
END–SUB ...
```

This is a complete subroutine that increments a 32-bit number held in the AX and BX registers. END–SUB assembles a return instruction and ends the definition and can be thought of as the equivalent of NEXT or ";" (semi).

Algorithm: Assemble a short return instruction, then set the dictionary back to the normal FORTH for future definitions.

Suggested Extensions: None.

Definition:

```
: END–SUB RET FORTH DEFINITIONS ;
```

MEND (–)

Mark the end of a macro definition.

Stack unaffected.

Example of Use:

```
... MACRO 32INC DUP INC ZIF 2+ INC /ENDIF MEND ...
```

This is a complete macro that increments a 32-bit number at a specific memory address. MEND marks the end of the macro definition.

Algorithm: Compile the word semi and make FORTH both the search and define dictionaries.

Suggested Extensions: None.

Definition:

```
: MEND COMPILE ; FORTH DEFINITIONS ;
  IMMEDIATE
```

/ENDIF (A –)

Mark the end of a conditional branching construct.

Stack on Entry: (A) The address the branch instruction is at.

Stack on Exit: Empty.

Example of Use:

```
... NZIF AH INC /ENDIF ...
```

This sequence would assemble a jump around the increment AH instruction if the zero flag was set.

Algorithm: This word is the same as FLABEL and is used to improve readability of code.

Suggested Extensions: None.

Definition:

```
: /ENDIF FLABEL ;
```

/ELSE (A –)

Allow an alternative branch in a conditional branching construct.

Stack on Entry: (A) The address of the branch assembled by the /IF instruction.

Stack on Exit: Empty.

Example of Use:

```
... NZIF AH INC /ELSE AH 0 # MOV /ENDIF ...
```

This sequence would assemble an increment of the AH register if the zero flag is not set and a load of the AH register with zero if it is set.

Algorithm: Assemble an absolute forward jump for /ENDIF to fill in. Then fill in the length of the branch assembled by the /IF word.

Suggested Extensions: None.

Definition:

```
: /ELSE 0 J SWAP FLABEL ;
```

CODE (–)

Start the definition of a word coded in assembler.

Stack unaffected.

Example of Use:

```
... CODE OUT AX POP DX POP DX AX OUT NEXT ...
```

This is a complete word that would cause a specific value to be sent out a specific port.

Algorithm: Use CREATE to enter the name of the word in the dictionary. Set the vocabulary to be searched to ASSEMBLER.

Suggested Extensions: None.

Definition:

```
ATILA DEFINITIONS
: CODE CREATE ASSEMBLER ;
```

SUBROUTINE (–)

Start the definition of an assembly language subroutine.

Stack unaffected.

Example of Use:

```
... SUBROUTINE INC-BIG-NUM AX INC ZIF BX INC /ENDIF
END-SUB ...
```

This is a complete subroutine that increments a 32-bit number held in the AX and BX registers.

Algorithm: Make ASSEMBLER both the search and define dictionaries. Use <BUILDS DOES> to define a word that leaves its address on the stack when it is executed.

Suggested Extensions: None.

Definition:

```
: SUBROUTINE ASSEMBLER DEFINITIONS
   <BUILDS DOES> ;
```

MACRO (–)

Start the defintion of a macro.

Stack unaffected.

Example of use:

```
... MACRO 32INC DUP INC ZIF 2+ INC /ENDIF MEND ...
```

This is a complete macro that increments a 32-bit number at a specific memory address. MEND marks the end of the macro definition.

Algorithm: Use ":" (colon) to define a word. Make ASSEMBLER both the search and define dictionaries.

Suggested Extensions: None.

Definition:

```
: MACRO ASSEMBLER DEFINITIONS : ;
```

8087 Numerical Coprocessor

Words Defined in This Chapter:

mftype	Define a word to store a value in b/w.
SHORT__REAL	Define the memory format of the next instruction to be 8087 short real.
SHORT__INTEGER	Define the memory format of the next instruction to be 8087 short integer.
LONG__REAL	Define the memory format of the next instruction to be 8087 long real.
WORD__INTEGER	Define the memory format of the next instruction to be 8087 word integer.
TEMP__REAL	Define the memory format of the next instruction to be 8087 temporary real.
LONG__INTEGER	Define the memory format of the next instruction to be 8087 long integer.
BCD	Define the memory format of the next instruction to be 8087 BCD.
ST	Store a value in "tipe" indicating a 8087 register is being used.
set-st	Set the register number field in the addressing mode byte being formed, and store the addressing mode byte.

st?	Determine whether a 8087 register has been specified.
1cell	Define single-word 8087 opcodes.
FCOMPP	Assemble the 8087 compare and pop twice instruction.
FTST	Assemble the 8087 test top stack value instruction.
FXAM	Assemble the 8087 examine the top stack value instruction.
ZFLD	Assemble the 8087 load-zero instruction.
1FLD	Assemble the 8087 load-one instruction.
PIFLD	Assemble the 8087 load-pi instruction.
L2TFLD	Assemble the 8087 log 2 (10) instruction.
L2EFLD	Assemble the 8087 log 2 (e) instruction.
LG2FLD	Assemble the 8087 log 10 (2) instruction.
LN2FLD	Assemble the 8087 log e (2) instruction.
FSQRT	Assemble the 8087 square root instruction.
FSCALE	Assemble the 8087 scale instruction.
FPREM	Assemble the 8087 remainder instruction.
FRNDINT	Assemble the 8087 round to integer instruction.
FXTRACT	Assemble the 8087 extract mantissa and exponent instruction.
FABS	Assemble the 8087 absolute value instruction.
FCHS	Assemble the 8087 change sign instruction.
FPTAN	Assemble the 8087 tangent instruction.
FPATAN	Assemble the 8087 arctangent instruction.
F2XM1	Assemble the 8087 2^ST(0)−1 instruction.
FYL2X	Assemble the 8087 ST(1)*log 2(St(0)) instruction.
FYL2XPI	Assemble the 8087 ST(1)*Log 2(ST(0)+1) instruction.
FINIT	Assemble the 8087 initialize instruction.
FENI	Assemble the 8087 enable interrupts instruction.
FDISI	Assemble the 8087 disable interrupts instruction.
FCLEX	Assemble the 8087 clear exceptions instruction.
FINCSTP	Assemble the 8087 increment stack pointer instruction.
FDECSTP	Assemble the 8087 decrement stack pointer instruction.
FNOP	Assemble the 8087 no operation instruction.
START8087	Initialize the 8087 and clear all interrupts.

F@	Fetch a floating-point value.
F!	Store a floating-point value.
FVARIABLE	Define and initialize a floating-point variable.
FCONSTANT	Define a floating-point constant.
F+	Add two floating-point numbers.
F-	Subtract two floating-point numbers.
F*	Multiply two floating-point numbers.
F/	Divide two floating-point numbers.
FNEGATE	Reverse the sign of a floating-point number.
FP->INT	Convert a floating-point number to an integer.
INT->FP	Convert an integer to a floating-point number.
FABS	Find the absolute value of a floating-point number.
F*10	Multiply a floating-point number by 10.
F/10	Divide a floating-point number by 10.
F=	Compare two floating-point numbers, checking for equality.
F0=	Compare a floating-point number to zero.
F<	Compare two floating-point numbers, checking for a less than condition.
F2DUP	Duplicate the top two floating-point numbers on the stack.
F>	Compare two floating-point numbers, checking for a greater than condition.
F<=	Compare two floating-point numbers, checking for a less than or equal condition.
F<>	Compare two floating-point numbers, checking for inequality.
FP->DINT	Convert a floating-point number to a double-length integer.
DINT->FP	Convert a double-length integer to a floating-point number.
FTRUNC	Truncate a floating-point number.
F.R	Print a floating-point number in normal form within a specified field.
F.	Print a floating-point number in normal form.
FE.R	Print a floating-point number in scientific notation, within a specified field.
FE.	Print a floating-point number in scientific notation.
R.	Print a floating-point number.

FNUM	Convert a string to a floating-point number.
SQRT	Calculate the square root of a floating-point number.
LN	Calculate the natural logarithm of a floating-point number.
LOG	Calculate the base 10 logarithm of a floating-point number.
2^X	An assembler subroutine to calculate two to an arbitrary power.
EXP	Calculate the value of e raised to a floating-point number.
^	Calculate the exponentiation of one floating-point number to another.
[FPTAN]	An assembler subroutine to execute 8087 FPTAN instruction on an arbitrary number.
TAN	Calculate the tangent of a floating-point number.
2FLD	A macro to push a two onto the 8087 stack.
F/2	A macro to divide the top 8087 stack entry by two.
SIN	Calculate the sine of a floating-point number.
COS	Calculate the cosine of a floating-point number.
1/X	Calculate the multiplicative identity of a floating-point number.
COT	Calculate the cotangent of a floating-point number.
CSC	Calculate the cosecant of a floating-point number.
SEC	Calculate the secant of a floating-point number.
[FPATAN]	An assembler subroutine to execute the 8087 FPATAN instruction on an arbitrary number.
ATAN	Calculate the arctangent of a floating-point number.
ACOTN	Calculate the arccotangent of a floating-point number.
ASIN	Calculate the arcsine of a floating-point number.
ACOS	Calculate the arccosine of a floating-point number.
ASEC	Calculate the arcsecant of a floating-point number.
ACSC	Calculate the arccosecant of a floating-point number.

FSIGN	Calculate the sign of a floating-point number.
SINH	Calculate the hyperbolic sine of a floating-point number.
COSH	Calculate the hyperbolic cosine of a floating-point number.
TANH	Calculate the hyperbolic tangent of a floating-point number.
SECH	Calculate the hyperbolic secant of a floating-point number.
CSCH	Calculate the hyperbolic cosecant of a floating-point number.
COTNH	Calculate the hyperbolic cotangent of a floating-point number.
ASINH	Calculate the inverse hyperbolic sine of a floating-point number.
ACOSH	Calculate the inverse hyperbolic cosine of a floating-point number.
ATANH	Calculate the inverse hyperbolic tangent of a floating-point number.
ASECH	Calculate the inverse hyperbolic secant of a floating-point number.
ACSCH	Calculate the inverse hyperbolic cosecant of a floating-point number.
ACOTNH	Calculate the inverse hyperbolic cotangent of a floating-point number.

FLOATING-POINT NUMBERS

This chapter will present a complete set of Forth words to make use of the 8087 numerical coprocessor. This chip must be available for the floating-point routines described in this book to function. The words presented include almost every possible function, from simple addition to the esoteric inverse hyperbolic cosecant. The power of the 8087 will be made part of our Forth by these words.

The 8087 numeric coprocessor is an extremely powerful and complicated chip. It is designed to function with the 8088 CPU of our IBM-PC and extend its instruction set to deal with floating-point numbers. Because the 8087 can

only be accessed from the machine level, we will have to use the assembler we defined in the previous chapter for much of our work with the 8087. Our first step will be to extend the assembler defined in this chapter to include the 8087 instructions. From this base we will be able to construct our elementary floating-point functions, like addition and subtraction.

Once we have written the basic arithmetic instructions, we will be able to write floating-point input and output words. These will enable us to deal with floating-point numbers as part of the language. At this point, the rest of the floating-point package could be considered optional. The next set of words will be mathematical functions, including logarithms, exponentials, and hyperbolic and transcendental functions. These can be used as the needs of your particular application demand.

INSIDE THE 8087

The words in this chapter can be used without any knowledge of the 8087 chip. They are totally self-contained and handle all of the interaction with the numeric coprocessor. But knowing about the machine you are using is always beneficial. The 8087 has eight internal floating-point registers, each 80 bits long. The registers are organized into a stack, and most operations function by pushing and popping from the stack of registers. In Intel nomenclature, the top of the stack is known as ST(0) or just ST. The next number on the stack is referenced by ST(1), and so on up to ST(7).

The 8087 can handle seven types of numbers, or data formats. They are:

Name	Number of Bits	Range
Word Integer	16	−32768 −>32767
Short Integer	32	−2,147,483,648 −> 2,147,483,647
Long Integer	64	−9,223,372,032,759,841,344 −> 9,223,372,032,759,841,343
Packed Decimal	80	−999,999,999,999,999,999 −> 999,999,999,999,999,999
Short Real	32	−8.43 E−37 −> 3.37e38
Long Real	64	−4.19 E−307 −> 1.67 E308
Temporary Real	80	−3.4 E−4932 −> 1.2 e4932

As can be seen from the table, the 8087 can use a wide range of numbers.

All operations that take place on the chip itself are in temporary real format.

The 8087's instruction set is also wide ranging. It includes pops and pushes to the 8087 stack, basic math instructions like addition and subtraction, specialized math instructions for calculating common functions, and any number of processor control instructions. The 8087 has a control register that specifies how it deals with rounding, error conditions, and interrupts. The 8087 also has a status word, equivalent to the flag register on the 8088, that will hold the result of logical operations on numbers.

The 8088 and 8087 must operate in unison to be useful. The 8088 has a number of instructions that facilitate this interaction. The escape or ESC instruction is used to control the 8087. The 8088 WAIT instruction is used to make sure the two chips are synchronized. The ESC instruction has three unused bits. These bits, along with fields in the addressing mode byte, are used by the 8087 to decode its instructions. When the 8088 executes an ESC instruction, it does nothing. The 8087 is usually watching the 8088 execute its instructions. When the 8087 sees an ESC instruction, it takes over and decodes the instruction. After the 8087 decodes the instruction, it starts processing. While executing an instruction, the 8087 is not watching what the 8088 is doing. If another ESC instruction was encountered by the 8088 during this time, the 8087 would miss it. The 8088 WAIT instruction will make sure no instructions are missed, by causing the 8088 to stop processing until the 8087 is ready to perform another operation and is back watching the 8088 instruction fetch. The WAIT instruction is also necessary when the 8088 and 8087 are accessing the same memory location. The 8088 should WAIT until the 8087 has completed an instruction before using shared memory locations. Otherwise, the 8088 has no way of knowing when the 8087 is finished with the memory.

Additions to the 8088 Macro Assembler

The 8087 extensions to the assembler have the following new addressing modes:

Addressing Mode	Normal	Forth
Stack	FADD ST ST(3)	0 ST 3 ST FADD
Stack and pop	FADDP ST(3),ST	3 ST 0 ST FADD ANDPOP
Stack, reversed	FSUBR ST,ST(1)	0 ST 1 ST FSUB REVERSE

The following data identifiers are used in addressing memory:

Format	Identifier	Example
Short real	SHORT__REAL	SHORT__REAL [BX] FLD
Short integer	SHORT__INTEGER	SHORT__INTEGER 2 [BX+#] FADD
Long real	LONG__REAL	LONG__REAL 0 [BP+#] FSTP
Word integer	WORD__INTEGER	WORD__INTEGER [BX] FDIV REVERSE
Temporary real	TEMP__REAL	TEMP__REAL Holdme] FSTP
Long integer BCD	LONG__INTEGER BCD	LONG__INTEGER [SI] FLD BCD FinalTotal] FSTP

Suggested Extensions: The words in this chapter store all numbers in short real format. This gives six or seven digits of accuracy. If increased accuracy is desired, the words could be modified to use the long real or even temporary real formats for more precision. These words also have no provision for using the 8087 exceptions for overflow, underflow, divide by zero, etc. If these conditions are important in your application, the 8087 control register would need to be set to notify the 8088 of these conditions, and exception handling words would need to be written.

mftype (N –) (–)

Define a word to store a value in b/w.

Stack on Entry: (Compile Time) (N) – Value to store in b/w.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See words defined below.

Algorithm: At compile time, store the value in the dictionary. At run time, fetch the value and store it in b/w.

Suggested Extensions: None.

Definition:

```
ASSEMBLER DEFINITIONS

: mftype <BUILDS C, DOES> C@ b/w C! ;

( Define the types we need.)
2 mftype SHORT_REAL
3 mftype SHORT_INTEGER
4 mftype LONG_REAL
5 mftype WORD_INTEGER
6 mftype TEMP_REAL
7 mftype LONG_INTEGER
8 mftype BCD
```

mf (N –)

Build the memory format field in the opcode being assembled, and store the opcode.

Stack on Entry: (N) The opcode being assembled.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Take the memory format from b/w. Subtract two to get to the desired range. Shift it one bit to the left to place it in the proper position and OR it into the opcode. Store the opcode.

Suggested Extensions: None.

Definition:

```
: mf b/w C@ 2- 1 asl OR C, ;
```

ST (N –)

Store a value in "tipe," indicating an 8087 register is being used.

Stack on Entry: (N) The register being used.

Stack on Exit: Empty.

Example of Use:

```
... 1 ST FXCH ...
```

Assemble the 8087 instruction that exchanges the top two registers on the 8087 register stack.

Algorithm: Add 50 to the value and store it in the current position of "tipe"; increment the current position.

Suggested Extensions: Add error checking to make sure only values of zero to seven are specified.

Definition:

```
: ST 50 + tipe f/s+ ! next+ ;
```

set–st (N –)

Set the register number field in the addressing mode byte being formed, and store the addressing mode byte.

Stack on Entry: (N) The addressing mode byte being formed.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Fetch the value from "tipe" and subtract 50, OR the number into the addressing mode byte, and store it in the dictionary.

Suggested Extensions: None.

Definition:

```
: set–st tipe @ 50 – OR C, ;
```

st? (– B)

Determine whether an 8087 register has been specified.

Stack on Entry: Empty.

Stack on Exit: (B) Boolean flag, true if an 8087 register has been specified.

Example of Use: See words defined below.

Algorithm: Fetch the value from "tipe," and see if it lies in the range of valid registers (0–7).

Suggested Extensions: None.

Definition:

```
: st? tipe @ DUP 50 >= SWAP 57 <= AND ;
```

1cell (N –) (–)

Define a word to assemble one word opcodes.

Stack on Entry: (Compile Time) (N) – The opcode to assemble.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See words defined below.

Algorithm: At compile time, store the value in the dictionary. At run time, fetch the value and store it in the dictionary. Assemble a WAIT instruction before every opcode.

Suggested Extensions: None.

Definition:

```
: 1cell <BUILDS , DOES> WAIT @ , reset ;

( Define all the 8087 one word opcodes)

-9762 1cell FCOMPP          -6951 1cell FTST
-6695 1cell FXAM            -4391 1cell ZFLD
-5927 1cell 1FLD            -5159 1cell PIFLD
-5671 1cell L2TFLD          -5415 1cell L2EFLD
-4903 1cell LG2FLD          -4647 1cell LN2FLD
-1319 1cell FSQRT           -551 1cell FSCALE
-1831 1cell FPREM           -807 1cell FRNDINT
```

```
-2855 1cell FXTRACT      -7719 1cell FABS
-7975 1cell FCHS         -3367 1cell FPTAN
-3111 1cell FPATAN       -3879 1cell F2XM1
-3623 1cell FYL2X        -1575 1cell FYL2XPI
-7205 1cell FINIT        -7973 1cell FENI
-7717 1cell FDISI        -7461 1cell FCLEX
-2087 1cell FINCSTP      -2343 1cell FDECSTP
-12071 1cell FNOP
```

esc-only (N1 N2 -)(-)

Define a word to define words that will assemble opcodes that have normal addressing mode bytes.

Stack on Entry: (Compile Time) (N1) – The portion of the opcode in the ESC instruction.
(N2) – The portion of the opcode in the addressing mode byte.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See words defined below.

Algorithm: At compile time, store the values in the dictionary. At run time, fetch the values and form the ESC instruction. Form the addressing mode byte normally, adding in the portion of the opcode that belongs in it. Assemble a WAIT instruction before every opcode.

Suggested Extensions: None.

Definition:

```
216 CCONSTANT esc

: esc-only <BUILDS SWAP C, C, DOES>
   WAIT DUP C@ esc OR C,
   1+ C@ a-mode reset ;

1 40 esc-only FLDCW
1 56 esc-only FSTCW
5 56 esc-only FSTSTATUSW
1 48 esc-only FSTENV
```

```
1 32 esc-only FLDENV
5 48 esc-only FSAVE
5 32 esc-only FRSTOR
```

dset (N1 – N2)

Set the destination bit in the opcode being assembled.

Stack on Entry: (N1) The opcode being assembled.

Stack on Exit: (N2) The opcode with the destination bit set.

Example of Use: See words defined below.

Algorithm: Fetch the value from "tipe," and see if the destination is 8087 register zero. If it is, swap the values in "tipe". If it is not, set the direction bit, indicating a move to a register other than zero.

Suggested extensions: None.

Definition:

```
: dset tipe @ 50 = IF
     swap-dir
   ELSE
     4 OR
   ENDIF ;
```

ANDPOP (–)

Set the pop bit in the instruction last assembled.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
... 1 ST 0 ST FADD ANDPOP
```

This would assemble an 8087 instruction to add the top two 8087 registers, store the result in ST(1) and discard in ST(0).

Algorithm: Get the address of the last instruction from the variable rev-ad. Set bit two in the byte at that address, the pop bit in an 8087 instruction.

Suggested Extensions: None.

Definition:

```
0 VARIABLE rev-ad
: ANDPOP rev-ad @ DUP C@ 2 OR SWAP C! ;
```

REVERSE (-)

Set the reverse bit in the instruction last assembled.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
... 1 ST 0 ST FSUB ANDPOP REVERSE ...
```

This would assemble an 8087 instruction to subtract register zero from register one, store the result in ST(1) and discard in ST(0).

Algorithm: Get the address of the last instruction from the variable rev-ad. Set bit four in the byte at that address plus one, the reverse bit in an 8087 instruction.

Suggested Extensions: None.

Definition:

```
: REVERSE rev-ad @ 1+ DUP C@ 8 OR
   SWAP C! ;
```

FXCH (-)

Assemble the 8087 register exchange instruction.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
... 3 ST FXCH ...
```

This would assemble an 8087 instruction to exchange 8087 registers ST(3) and ST(0).

Algorithm: Set the opcode and then fill in the register field, store the result in the dictionary. Assemble a WAIT instruction before the opcode.

Suggested Extensions: Add error checking for invalid operands.

Definition:

```
: FXCH
   WAIT 217 C, 200 set-st reset ;
```

FST (-)

Assemble the 8087 store instruction.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
... SHORT_REAL TOTAL ] FST ...
```

This would assemble an 8087 instruction to move the value in ST(0) to the 32 bits at the memory address TOTAL, in short real format. The 8087 stack is undisturbed.

Algorithm: Determine if the instruction references the stack or memory and assemble accordingly. Assemble a WAIT instruction before the opcode.

Suggested Extensions: Add error checking for invalid operands.

Definition:

```
: FST
   WAIT st? IF
      221 C, 208 set-st
   ELSE
      217 mf 24 a-mode
   ENDIF reset ;
```

fkind (N1 N2 N3 –) (–)

Define a word to define words that will assemble the primary floating-point instructions.

Stack on Entry: (Compile Time) (N1) – One if the reverse flag is part of the instruction.
(N2) – The portion of the opcode in the ESC instruction.
(N3) – The portion of the opcode in the addressing mode byte.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See words defined below.

Algorithm: At compile time, store the values in the dictionary. At run time, fetch the values and form the ESC instruction. Form the addressing mode byte normally, adding in the portion of the opcode that belongs in it. If the reverse bit can be set, call dset. Assemble a WAIT instruction before every opcode.

Suggested Extensions: Add error checking for invalid operands.

Definition:

```
: fkind <BUILDS LROT SWAP C, C, C, DOES>
     WAIT HERE rev-ad ! st? IF
     216 OVER C@ IF
       dset
     ENDIF
     C, 1+ C@ set-st
   ELSE
     216 mf 2+ C@ a-mode
   ENDIF reset ;

( Define the primary 8087 instructions)
0 208 16 fkind FCOM
0 216 24 fkind FCOMP
1 192 0 fkind FADD
1 224 32 fkind FSUB
1 200 8 fkind FMUL
1 240 48 fkind FDIV
```

FSTP (–)

Assemble the 8087 store and pop instruction.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
... SHORT_REAL TOTAL ] FST ...
```

This would assemble an 8087 instruction to move the value in ST(0) to the 32 bits at the memory address TOTAL, in short real format. ST(0) would then be removed from the 8087 stack.

Algorithm: Determine if the instruction references the stack or memory and assemble accordingly. The special cases of the long integer, temporary real, and BCD formats must be handled. Assemble a WAIT instruction before the opcode.

Suggested Extensions: Add error checking for invalid operands.

Definition:

```
: FSTP
   WAIT st? IF
      221 C, 216 set-st
   ELSE
      b/w C@ 6 < IF
         217 mf 24 a-mode
      ELSE
         b/w C@ 7 = IF
            223 C, 56 a-mode
         ELSE
            b/w C@ 6 = IF
               219 C, 40 a-mode
            ELSE
               223 C, 48 a-mode
   ENDIF ENDIF ENDIF ENDIF reset ;
```

FLD (–)

Assemble the 8087 load instruction.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
... SHORT_REAL TOTAL ] FLD ...
```

This would assemble an 8087 instruction to push the value at the 32 bits at the memory address TOTAL, in short real format, to the 8087 stack.

Algorithm: Determine if the instruction references the stack or memory and assemble accordingly. The special cases of the long integer, temporary real, and BCD formats must be handled. Assemble a WAIT instruction before the opcode.

Suggested Extensions: Add error checking for invalid operands.

Definition:

```
: FLD
   WAIT st? IF
      217 C, 192 set-st
   ELSE
      b/w C@ 6 < IF
         217 mf 0 a-mode
      ELSE
         b/w C@ 7 = IF
            223 C, 40 a-mode
         ELSE
            b/w C@ 6 = IF
               219 C, 32 a-mode
            ELSE
               223 C, 32 a-mode
   ENDIF ENDIF ENDIF ENDIF reset ;
```

START8087 (-)

Initialize the 8087.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

CODE PROGRAM START8087 ...

This would initialize the 8087 and clear its stack. Interrupts are disabled.

Algorithm: Assemble the FINIT, and FDISI instructions.

Suggested Extensions: None.

Definition:

```
CODE START8087 FINIT FDISI NEXT
```

F@ (A – F)

Fetch a floating-point variable.

Stack on Entry: (A) The address of the floating-point value.

Stack on Exit: (F) The floating-point number at A.

Example of Use:

```
... balance F@ R. ...
```

This code fragment would fetch the value of the variable balance and print it on the display.

Algorithm: floating-point numbers are 32 bits long. Use two 16-bit fetches to place the number on the stack.

Suggested Extensions: None.

Definition:

```
: F@ DUP @ SWAP 2+ @ SWAP ;
```

F! (F A –)

Store a floating-point variable.

Stack on Entry: (F) A floating-point number.
(A) The address of the floating-point value.

Stack on Exit: (F) The floating-point number at A.

Example of Use:

```
... Sum F@ 1.5 Sum F! ...
```

This code fragment would add 1.5 to the value held in Sum.

Algorithm: Floating-point numbers are 32-bits long. Use two 16-bit stores to place the number in memory.

Suggested Extensions: None.

Definition:

```
: F! DUP LROT ! 2+ ! ;
```

FVARIABLE (F –) (F – A)

Define a floating-point variable.

Stack on Entry: (Define Time) (F) The initial value of the variable.
(Run Time) Empty.

Stack on Exit: (Define Time) Empty.
(A) The address of the variable.

Example of Use:

```
... Sum F@ 1.5 F+ Sum F! ...
```

This code fragment would add 1.5 to the value held in Sum.

Algorithm: Allocate 4 bytes at define time and store the value found on the stack. At run time, leave the address of the variable.

Suggested Extensions: None.

Definition:

```
: FVARIABLE <BUILDS HERE 4 ALLOT F!
  DOES> ;
```

FCONSTANT (F –) (– F)

Define a floating-point constant.

Stack on Entry: (Define Time) (F) The value of the constant.
(Run Time) Empty.

Stack on Exit: (Define Time) Empty.
(F) The value of the constant.

Example of Use:

```
3.14157 FCONSTANT PI
```

The constant pi.

Algorithm: Allocate 4 bytes at define time and store the value found on the stack. At run time, fetch the value stored in memory.

Suggested Extensions: None.

Definition:

```
: FCONSTANT <BUILDS HERE 4 ALLOT F!
  DOES> F@ ;
```

F+ (F1 F2 – F3)

Leave F3, the floating-point sum of F1 and F2.

Stack on Entry: (F1) A floating-point number.
(F2) A floating-point number.

Stack on Exit: (F3) The sum of F1 and F2.

Example of Use:

```
... Sum F@ 1.5 F+ Sum F! ...
```

This code fragment would add 1.5 to the value held in Sum.

Algorithm: Move both numbers from the 8088 data stack to the 8087. Decrement the stack with the pops. Add the numbers on the 8087, then move the result back to the 8088 data stack.

Suggested Extensions: None.

Definition: