Questions, Projects, and Labs Chapter Thirteen

13.1 Questions

1) What is the purpose of the INTO instruction?
2) What was the main purpose for the INTMUL instruction in this volume? What was it introduced before the chapter on integer arithmetic?
3) Describe how to declare byte variables. Give several examples. What would you normally use byte variables for in a program?
4) Describe how to declare word variables. Give several examples. Describe what you would use them for in a program.
5) Repeat question 4 for double word variables.
6) What are qword and tbyte objects?
7) How does HLA differentiate string and character constants?
8) Explain the purpose of the TYPE section. Give several examples of its use.
9) What is a pointer variable?
10) How do you declare pointer variables in a program?
11) How do you access the object pointed at by a pointer. Give an example using 80x86 instructions.
12) What is the difference between a CONST object and a VAL object?
13) What is the "?" statement used for?
14) What is an enumerated data type?
15) Given the two literal string constants "Hello " and "World", provide two fundamentally different ways to concatenate these string literal constants into a single string within an HLA source file.
16) What is the difference between a STRING constant and a TEXT constant?
17) What is a pointer constant? What is the limitation on a pointer constant?
18) What is a pointer expression? What are the limitations on pointer expressions?
19) What is a dangling pointer? How do you avoid the problem of dangling pointers in your programs?
20) What is a memory leak? What causes it? How can you avoid memory leaks in y our programs?
21) What is likely to happen if you use an uninitialized pointer?
22) What is a composite data type?
23) If you have a variable of type STRING, how many bytes in memory does this variable consume? What value(s) appear in this variable?
24) What is the data format for HLA string data?
25) What is the difference between a length-prefixed and a zero terminated string? What are the advantages and disadvantages of each?
26) Are HLA string zero terminated or length prefixed?
27) How do you directly obtain the length of an HLA string (i.e., w/o using the str.length function)?
28) What are the two pieces of length data maintained for each HLA string? What is the difference between them?
29) Why would you want to use the *str.cpy* routine (which is slow) rather than simply copying the pointer data from one string to another (which is fast)?

30) Many string functions contain the prefix "a_" in their name (e.g., str.cpy vs. str.a_cpy). What is the difference between the functions containing the "a_" prefix and those that do not contain this prefix?

31) Explain how to access the individual characters in an HLA string variable.

32) What is the purpose of string concatenation?

33) What is the difference between the str.cat and str.insert? Which is the more general of the two routines? How could you use the more general routine to simulate the other?

34) How can you perform a case-insensitive string comparison using the HLA Standard Library Strings module?

35) Explain how to convert an integer to its string representation using functions from the HLA Standard Library's String module.

36) How does HLA implement character sets? How many bytes does a character set variable consume?

37) If *CST* is a character set constant, what does "-CST" produce?

38) Explain the purpose of the character classification routines in the HLA Standard Library chars.hhf module.

39) How do you compute the intersection, union, and difference of two character set constants in an HLA constant expression?

40) How do you compute the interesection, union, and difference of two character set variables in an HLA program at run-time?

41) What is the difference between a subset and a proper subset?

42) Explain how you can use a character set to validate user input to a program.

43) How do you declare arrays in assembly language? Give the code for the following arrays:

a) A two dimensional 4x4 array of bytes

b) An array containing 128 double words

c) An array containing 16 words

d) A 4x5x6 three dimensional array of words

44) Describe how you would access a single element of each of the above arrays. Provide the necessary formulae and 80x86 code to access said element (assume variable I is the index into single dimension arrays, I & J provide the index into two dimension arrays, and I, J, & K provide the index into the three dimensional array). Assume row major ordering, where appropriate.

45) Repeat question (44) using column major ordering.

46) Explain the difference between row major and column major array ordering.

47) Suppose you have a two-dimensional array of bytes whose values you want to initialize as follows:

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

Provide the variable declaration to accomplish this. Note: Do not use 80x86 machine instructions to initialize the array. Initialize the array in your STATIC section.

Questions (48)-(52) use these HLA declarations:

```
type

    Date:  Record
                   Month:int8;
                   Day:int8;
                   Year:int8;
           endrecord;
```



```
    Time=  record
                  Hours:int8;
                  Minutes:int8;
                  Seconds:int8;
           endrecord;

    VideoTape:    record
                      Title:string;
                      ReleaseDate:Date;
                      Price:Real32;
                      Length: Time;
                      Rating:char;
                  endrecord;

var

    TapeLibrary : VideoTape[128];
```

48) Suppose EBX points at an object of type VideoTape. What is the instruction that properly loads the *Rating* field into AL?

49) Provide an example of a STATIC *VideoTape* variable initialized with an appropriate constant (should be legal in the HLA STATIC section).

50) This data structure (Date) suffers from the good old "Y2K" bug. Provide a correction for this problem (See Chapter Six in this volume if you have any questions about "Y2K" or the solution to this problem).

51) A one-character rating is probably not sufficient since there are ratings like "PG" and "PG-13" on movies. On the other hand, the longest rating string is five characters, so a string variable (minimum of 16 characters between the pointer and the resident string data) is probably overkill. Describe a better way to handle this using a one-byte variable/data type.

52) Provide a variable declaration for a pointer to a *VideoTape* object.

53) Provide an example of an HLA array constant containing eight *uns8* values.

54) How many bytes of memory does a 4x5 array of *dwords* require?

55) What is the difference between an array of characters and a string?

56) What is the difference between the indexes you would use to access an HLA array constant and the indexes you would use to access an array element in memory at run-time?

57) Why is the bubble sort generally a poor choice of algorithm when sorting data? Under what circumstances is the bubble sort a good choice?

58) What is a dynamically allocated array? How do you create one?

59) Explain the use of the @size compile-time function when indexing into an array at run-time.

60)

61) How does HLA store the fields of a record in memory?

62) Provide an example of a *student* record constant (see "Records" on page 483 for a description of the *student* record).

63) If you have an array of records and you want to compute an index into the array, describe how HLA can compute the size of each element of the array (i.e., the record size) for you.

64) Given the following declaration, provide the code to access element a[i].b[j] of this array:

```
type
    bType: uns32[16];
    rType:
        record
            i:int32;
```



```
            u:uns32;
            b:bType;
        endrecord;

    pType:pointer to rType

static
    a:rType[32];
```

65) Given the definitions in question (64), explain how you would access the object the j^{th} element of the *b* field of the record where *p* points in assembly language ("p->b[j]" using C syntax, "p^.b[j]" for Pascal programmers).

66) Explain how to set the offset of the first field in a record. Give an example of why you would want to do this.

67) Explain how to align a particular field in a record to a given boundary.

68) Describe how to pad a record so that its size is an even multiple of some number of bytes.

69) What is the difference between a record and a union?

70) What is an anonymous union? What would you use it for?

71) What is a variant type? What data structure(s) would you use to create a variant object?

72) What is a namespace? What is its purpose?

73) What is the difference between a namespace and a record?

74) What type of declaration may not appear in a namespace?

75) What is namespace pollution? How do namespaces solve this problem?

76) What is the data structure for an HLA Standard Library DATE data type?

77) What is the data structure for an HLA Standard Library TIME data type?

78) What is the rule for computing whether a given year is a leap year?

79) What Standard Library routines would you call to get the current date and current time?

80) What is a Julian Day Number? Why are such dates interesting to us?

81) What routines does the HLA Standard Library provide to perform date arithmetic?

82) What is the difference between a random access file and a sequential file?

83) What is the difference between a binary and a text file?

84) What is the difference between a variable-length record and a fixed length record? Which would you use in a random access file? Why?

85) What is an ISAM file? What is the purpose of an ISAM list?

86) Explain how to copy one file to another using the HLA *fileio* routines.

87) How does computing the file size help you determine the number of records in a random access file?

88) What is the syntax for an HLA procedure declaration?

89) What is the syntax for an HLA procedure invocation (call)?

90) Why is it important to save the machine state in a procedure?

91) What is lexical scope?

92) What is the lifetime of a variable?

93) What types of variables use automatic storage allocation?

94) What types of variables maintain their values across a procedure call (i.e., still have the value from the last call during the current call)?

95) What is the lifetime of a global variable?

96) What is the lifetime of a local variable?

97) What are the six different ways HLA lets you pass parameters?

98) What are two different ways to declare pass by value parameters in an HLA procedure?

99) How do you declare pass by reference parameters in an HLA procedure?

100) What are the advantages and disadvantages of callee-preservation and caller-preservation when saving registers inside a procedure?

101) How do pass by value parameters work?

102) How do pass by reference parameters work?

103) How would you load the value associated with a pass by value parameter, V, into the EAX register?

104) How would you load the value associated with a pass by reference parameter, R, into the EAX register?

105) What is the difference between a function and a procedure?

106) What is the syntactical difference between a function and a procedure in HLA?

107) Where do you typically return function results?

108) What is instruction composition?

109) What is the purpose of the RETURNS option in a procedure declaration?

110) What is a side effect?

111) What is wrong with side effects in your programs?

112) What is recursion?

113) What is the FORWARD procedure option used for?

114) What is the one time that a forward procedure declaration is absolutely necessary?

115) Give an example of when it would be convenient to use a forward declaration even if it was specifically required.

116) Explain how to return from a procedure in the middle of that procedure (i.e., without "running off the end" of the procedure).

117) What does the #INCLUDE directive do?

118) What do you normally use the #INCLUDE directive for?

119) What is the difference between the #INCLUDE and the #INCLUDEONCE directives?

120) What is the syntax for an HLA unit?

121) What is the purpose of an HLA unit?

122) What is the purpose of separate compilation?

123) How do you declare EXTERNAL procedures? Variables?

124) What variable types cannot be EXTERNAL?

125) How do you declare a public symbol in HLA?

126) What is a header file?

127) What is a file dependency?

128) What is the basic syntax for a makefile?

129) What is a recursive file inclusion? How can you prevent this?

130) What are the benefits of code reuse?

131) What is a library?

132) Why would you not want to compile all your library routines into a single OBJ file?

133) What types of files are merged together to form a .LIB file?

134) What program would you use to create a library file?

135) How does a library contribute to name space pollution? How can you solve this problem?

136) What are the differences between the INTMUL and the IMUL instruction?

137) What must you do before executing the DIV and IDIV instructions?

138) How do you compute the remainder after the division of two operands?

139) Assume that VAR1 and VAR2 are 32 bit variables declared with the DWORD type. Write code sequences that will compute the boolean result (true/false) of each of these, leaving the result in BL:

a) VAR1 = VAR2

b) VAR1 <> VAR2

c) VAR1 < VAR2 (Unsigned and signed versions

d) VAR1 <= VAR2 for each of these)

e) VAR1 > VAR2

f) VAR1 >= VAR2

140) Provide the code sequence that will compute the result of the following expressions. Assume all variables are UNS32 values and you are computing unsigned results:

a) A = (X-2) * (Y + 3);

b) B = (A+2)/(A - B);

c) C = A/X * B - C - D;

d) D = (A + B)/12 * D;

e) E = (X * Y) / (X-Y) / C+D;

f) F = (A <= B) && (C != D) || (E > F);

(note to Pascal users, the above is “F := (A <= B) and (C <> D) or (E > F);”)

141) Repeat question (140) assuming all variables are INT32 objects and you are using signed integer arithmetic.

142) Repeat question (140) assuming all variables are REAL64 objects and you are using floating point arithmetic. Also assume that an acceptable error value for floating point comparisons is 1.0e-100.

143) Convert the following expressions into assembly language code employing shifts, additions, and subtractions in place of the multiplication:

a) EAX*15

b) EAX*129

c) EAX*1024

d) EAX*20000

144) How could you use the TEST instruction (or a sequence of TEST instructions) to see if bits zero and four in the AL register are both set to one? How would the TEST instruction be used to see if either bit is set? How could the TEST instruction be used to see if neither bit is set?

145) Why are commutative operators easier to work with when converting expressions to assembly language (as opposed to non-commutative operators)?

146) Provide a single LEA instruction that will multiply the value in EAX by five.

147) What happens if INTMUL produces a result that is too large for the destination register?

148) What happens if IMUL or MUL produces a result that is too large for EAX?

149) Besides divide by zero, what is the other division error that can occur when executing IDIV or DIV?

150) What is the difference between the MOD instruction and the DIV instruction?

151) After a CMP instruction, how do the condition code bits indicate that one signed operand is less than another signed operand?

152) What instruction is CMP most similar to?

153) What instruction is TEST most similar to?

154) How can you compute a pseudo-random number between 1 and 10 using the HLA Standard Library rand.hhf module?

155) Explain how to copy the floating point status register bits into the 80x86 FLAGs register so you can use the SETcc instructions after a floating point comparison.

156) Why is it not a good idea to compare two floating point values for equality?

157) Explain how to activate floating point exceptions on the FPU.

158) What are the purpose of the rounding control bits in the FPU control register?

159) Besides where they leave their results, what is the difference between the FIST instruction and the FRNDINT instruction?

160) Where does *stdin.getf()* leave the input value?

161) What is a mantissa?

162) Why does the IEEE floating point format mantissa represent values between 1.0 and 2.0?

163) How can you control the precision of floating point calculations on the FPU? Provide an example that shows how to set the precision to real32.

164) When performing floating point comparisons, you use unsigned comparisons rather than signed comparisons, even though floating point values are signed. Explain.

165) What is postfix notation?

166) Convert the expression in question (140) to postfix notation.

167) Why is postfix notation convenient when working with the FPU?

168) Explain how the XLAT instruction operates.

169) Suppose you had a *cipher* (code) that swaps various characters in the alphabet. Explain how you could use the XLAT instruction to decode a message encoded with this cipher. Explain how you could encode the message.

170) What is the purpose of *domain conditioning*?

171) What is the maximum set of values for the domain and range when using the XLAT instruction?

172) Explain the benefits of using one program to generate the lookup tables for another program.

13.2 Programming Projects

1) Write a procedure, PrintArray(var ary:int32; NumRows:uns32; NumCols:uns32), that will print a two-dimensional array in matrix form. Note that calls to the *PrintArray* function will need to coerce the actual array to an *int32*. Assume that the array is always an array of *int32* values. Write the procedure as part of a UNIT with an appropriate header file. Also write a sample main program to test the *PrintArray* function. Include a makefile that will compile and run the program. Here is an example of a typical call to *PrintArray*:

```
static
   MyArray: int32[4, 5];
       .
       .
       .
   PrintArray( (type int32 MyArray), 4, 5 );
```

2) Write a procedure that reads an unsigned integer value from the user and reprompts for the input if there is any type of error. The procedure should return the result in the EAX register. It shouldn't require any parameters. Compile this procedure in a UNIT. Provide an appropriate header file and a main program that will test the function. Include a makefile that will compile and run the program.

3) Extend the previous project (2) by writing a second routine for the UNIT that has two parameters: a minimum value and a maximum value. Not only should this routine handle all input problems, but it should also require that the input value fall between the two values passed as parameters.

4) Extend the unit in (3) by writing two addition procedures that provide the same facilities for signed integer values.

5) Write a program that reads a filename from the user and then counts the number of lines in the text file specified by the filename.

6) Write a program that reads a filename from the user and then counts the number of words in the specified text file. The program should display an accurate count of words. For our purposes, a "word" is defined as any sequence of non-whitespace characters. A whitespace character is the space (#$20), tab (#$9), carriage return (#$d), and the line feed (#$a). Read the text file a line at a time using fileio.gets or fileio.a_gets. Also remember that a zero byte (#0) marks the end of a string, so this clearly terminates a word as well.

7) Modify the CD database program in Laboratory Exercise 13.3.10 so that it reads the CD database from a text file. Your program should make two passes through the text file. On the first pass it should count the number of lines in the text file and divide this value by four (four lines per record) to determine how many records are in the database. Then it should dynamically allocate an array of CDs of the specified size. On the second pass, it should read the data from the text file into dynamically allocated array. Obviously, you must also modify the program so that it works with a variable number of CDs in the list rather than the fixed number currently specified by the NumCDs constant.

8) Modify program (7) to give the user the option of displaying all the records in the database sorted by title, artist, publisher, or date. Use the QuickSort algorithm appearing in this volume to do the sorting (note: the solution requiring the least amount of thought will require four different sort routines; a more intelligent version can get by with two versions of quicksort).

9) Write a "CD Database Input" program that reads the textfile database for program (7) into a fixed array in memory. Allow enough space for up to 500 records in the database (display an error and stop if the input exceeds this amount). Of course, the value 500 should be a symbolic constant that can easily be changed if you need a larger or smaller database. After reading the records into memory, the program should prompt the user to enter additional CDs for the database. When the user is through entering data, the program should write all the records back to the database file (including the new records).

10) Modify program (9) so that the user is given the option of deleting existing records (perhaps by specifying the index into the array) in addition to entering new entries into the database.

11) Write a MsgBoxInput procedure that has the following parameters:

```
procedure MsgBoxInput( prompt:string; row:word; col:word; result:string );
```

The procedure should do the following:

1) Save the rectangular region of the screen specified by (row, col) as the upper left hand corner and (row+4, col+length(prompt) +2) as the lower right hand corner. Stop the program with an error if this rectangle is outside the bounds (0,0) <-> (24,79). Use the console.a_getRect function to save this portion of the screen.

2) Fill the rectangular region saved above with spaces and a dark blue background. Set the foreground attribute to yellow. (console.fillRect).

3) Print the prompt message starting at position (row+1, col+1).

4) Position the cursor at (row+2, col+1).

5) Read a string from the user with no more than length(prompt) characters (see below).

7) Restore the attributes in the rectangular region to black and white. (console.fillRectAttr)

8) Redraw the text saved in step one above.

Note that you cannot use *stdin.gets* or *stdin.a_gets* to read the string from the user since these functions won't limit the number of input characters. Instead, you will have to write your own input routine using *stdin.getc* to read the data a character at a time. Don't forget to properly handle the backspace character. Also, *stdin.getc* does not echo the character, so you will have to handle this yourself.

Put these procedures in a unit and write a companion main program that you can use to test your dialog box input routine.

12) The Windows console device is a *memory mapped I/O device*. That is, the display adapter maps each character on the text display to a character in memory. The display is an 80x25 array of characters declared as follows:

```
display:char[25,80];
```

Display[0,0] corresponds to the upper left hand corner of the screen, display[0,79] is the upper right hand corner, display[24,0] is the lower left hand corner, and display[24,79] is the lower right hand corner of the display. Each array element contains the ASCII code of the character to appear on the screen.

The lab4_10_7.hla program demonstrates how to copy a matrix to a portion of the screen (or to the entire console, if you so desire). Currently, this program draws and erases a single asterisk over and over again on the screen to give the illusion of animation. Modify this program so that it draws a rectangular or triangular shape, based at the row/column address specified by the for-loops. Be sure you don't "draw" outside the bounds of the character array (*playingField*).

13) Create a program with a single dimension array of records. Place at least four fields (your choice) in the record. Write a code segment to access element *i* (*i* being a DWORD variable) in the array.

14) Modify the program above so that it contains two identical arrays of records. Have the program copy the data from the first array to the second array and then print the values in the second array to verify the copy operation.

15) Modify the program in (14) above to use a dynamically allocated array, rather than a statically allocated array, as the destination array.

16) Write a program which copies the data from a 4x4 array and stores the data into a second 4x4 array. For the first 4x4 array, store the data in row major order. For the second 4x4 array, store the data in column major order. Write the for loops to do this "transposition" yourself. Do not use the *array.transpose* routine in the HLA Standard library.

17) Modify program (16) so that a single constant controls the size of the matrices (they will always be square, having the same number of rows as columns, so a single constant will suffice).

18) Write a program that reads two lines of text from the user and displays all characters that appear on both lines, but does not display any characters that appear on only one of the two lines of text. (hint: use the character set functions.)

19) Modify program (18) so that it prints the characters common to both lines, the characters appearing only in the first string, and the characters appearing only in the second string. Print these three sets of characters on different lines with an appropriate message explaining their content.

20) Write a program that reads a line of text from the user and counts the number of occurrences of each character on the line. The program should display the character and count for each unique character found in the input line whose count is not zero (hint: create an array of 256 integers and use the ASCII code as an index into this array).

21) Modify program (20) so that it also displays the number of numeric characters, the number of alphabetic characters (upper and lower case), the number of uppercase alphabetic characters, the number of lower case alphabetic characters, the number of control characters (ASCII code <= $1f and $7f), and the number of punctuation characters (this is all characters other than the previously mentioned sets). Display these counts one per line before the output of the count for each character from problem (20).

22) Write a program that reads a line of text from the user and counts the number of words on that line. For our purposes, a "word" is defined as any sequence of non-whitespace characters. A whitespace character is the space (#$20), tab (#$9), carriage return (#$d), and the line feed (#$a). Do not use the *str.tokenize* or *str.tokenize2* routines to implement this program.

23) Look up the *str.tokenize2* routine in the HLA Standard Library strings module. Implement problem (22) using this procedure.

24) Write a program that reads a string from the user and checks the string to see if it takes the form "mm/dd/yy", or "mm-dd-yy" where *mm*, *dd*, and *yy* are all exactly two decimal digits. If the input string matches either of these patterns, extract these substrings into separate string variables and print them on separate lines with an appropriate description (i.e., month, day, and year).

25) Modify program (24) to verify that *mm* is in the range 01-12, *dd* is in the range 01-31, and *yy* is in the range 00-99. Report match/no match on the input depending upon whether the input string matches this format. Note: you will probably find this problem to be a whole lot less work if you look up the *conv.strTou8* procedure in the conversions module of the HLA Standard Library and do numeric comparisons rather than string comparisons).

26) Modify program (25) to allow single digit values for the month and day components when they fall in the range 1-10 (the year component, *yy*, still requires exactly two digits). Also allow leading or trailing whitespace in the string (hint: use the char.ispace function or the str.trim function). Display match/no match as before depending upon the validity of the string.

27) Modify program (26) so that it checks the validity of the date subject to the Gregorian Calendar. That is, the day value should not exceed the maximum number of days for each month. Note: you may use the date.IsLeapYear function but you may not use date.IsValid or date.validate for this project. Assume that the centuries portion of the date is in the range 2000..2099 (that is, add 2000 to the year value to obtain the correct year).

28) The Y2K problem): Modify program (27) as follows: (1) it reads a string from the user. (2) it searches for the pattern "mm/dd/yy" or "mm-dd-yy" anywhere in the string (there may be other text in the string in this program). If the program finds a string of the form "mm/dd/yy" or "mm-dd-yy" then it will replace that string with a string of the form "mm/dd/19yy" or "mm-dd-19yy" to make the string Y2K compatible (of course, a good programmer would use a string constant for "19" so it could be easily changed to "20" or any other two digits). If there is more than one date in the string that matches a valid Y2K date, you will need to find and convert that date as well (i.e., convert all possible dates found in the string). After the conversion process is complete, display the string which should be the original text typed by the user with these Y2K modifications. Do not translate any dates that don't exactly match one of these two patterns:

    Examples of strings containing good dates (ones you should translate):


```
01-01-01
02-29-96
End of the 1900s: 12/31/99
Today is '3/17/90'
Start of the 20th century: 1/1/01, end of the 20th century: 12/31/2000
(note: program must not convert 12/31/2000 above to 12/31/192000)
Yesterday was 7/4/76, tomorrow is 7/6/96.
(must convert both dates above.)
```

Examples of bad dates that your program must ignore:

```
01-01/01
02/29-96
End of the 1900s: 12/31/1999
Today is '123/17/90'
01/32/00
02/29/99
```

29) Write a program that reads a string of characters from the user and verifies that these characters are all decimal digits in the range '0'..'7'. Treat this string as an octal (base eight) number. Convert the string to an integer value. Note: values in the range 0..7 consume exactly three bits. You can easily construct the octal number using the SHL instruction; you will not need to use the INTMUL instruction for this assignment. Write a loop that reads the octal string from the user and, if the string is valid, converts the number to an integer and displays that integer value in decimal.

30) Modify the student database program in the laboratory exercises (see "Records, Arrays, and Pointers Laboratory Exercise" on page 698) to sort the students by their name prior to printing the result. Since the name field is a string, you will want to use the string comparison routines to compare them. You should also use the case insensitive string comparisons (e.g., *str.ile*) since upper and lower case are irrelevant to most people when viewing sorted lists of names.

31) Modify program (30) so that it uses two fields for the name (last name and first name). The user should still enter the full name. Under program control, extract the first and last names from the single input string and store these two name components into their respective fields in the record.

32) Write a program that computes the two solutions to the quadratic equation.

$$x_1 = \frac{-b-\sqrt{b^2-4ac}}{2a}$$

$$x_2 = \frac{-b+\sqrt{b^2-4ac}}{2a}$$

The program should verify that *a* is non-zero and that the value b^2-*4ac* is positive or zero before attempting to compute these roots.

33) Suppose you are given a line and two points on that line. Compute the point that is midway between the two points via the equations $X_{mid} = (x_1 + x_2)/2$ and $Y_{mid} = (y_1 + y_2)/2$. Read the X and Y coordinates for the two points from the user. Use a RECORD containing two fields, X and Y, to denote a single point in your program. Note: X and Y are real64 variables.

34) Write a program that computes the amount of interest earned on a savings account. The formula for computing interest is

$$DollarsEarned = InitialDeposit \times \left(1.0 + \frac{InterestRate}{100.0}\right)^{Years}$$

Hint: you will need to use routines from the "math.hhf" library module for this assignment.

35) Write a program that inputs a temperature in Celsius and converts it to degrees Fahrenheit. The translation is given by the following formula:

$$f = (9/5) \times c + 32$$

36) Solve the equation above (35) for c and write a program that also solves for Celsius given a temperature in Fahrenheit.

37) Write a program that inputs a set of grades for courses a student takes in a given quarter. The program should then compute the GPA for that student for the quarter. Assume the following grade points for each of the following possible letter grades:

- A+ 4.0
- A 4.0
- A- 3.7
- B+ 3.3
- B 3.0
- B- 2.7
- C+ 2.3
- C 2.0
- C- 1.7
- D+ 1.3
- D 1.0
- D- 0.7
- F 0

Display the GPA using the format X.XX.

38) Modify program (37) to handle courses where each course may have a different (integral) number of units. Multiply the grade points by the number of units for a given grade and then divide by the total number of units.

39) Write a program that accepts a dollars and cents amount as a floating point value and converts this to an integer representing the number of pennies in the entered amount (round the input to the nearest penny if

it is off a little bit). Translate this value into the minimum number of pennies, nickels, dimes, quarters, one dollar bills, five dollar bills, ten dollar bills, twenty dollar bills, and one hundred dollar bills that would be necessary to represent this result. Display your answer as follows:

```
The amount $xxx.xx requires no more than
        p pennies,
        n nickels,
        d dimes,
        q quarters,
        d $1 bills,
        f $5 bills,
        t $10 bills,
and     h $100 bills.
```

The italicized letters represent small integer constants (except *h* which can be a large integer constant). Do not display an entry if the count is zero (e.g., if there are zero $100 bills required, do not display the line "0 $100 bills.")

40) Modify program (39) so that it displays a singular noun if there is only one of a given item (e.g., display "1 nickel," rather than "1 nickels,")

41) Write a program that plots a sine curve on the console (see "Bonus Section: The HLA Standard Library CONSOLE Module" on page 192 for a discussion of the console support routines). Begin by clearing the screen with console.cls(). Then draw a row of dashes along line 12 on the screen to form the X-axis. Draw a column of vertical bars in column 0 to denote the Y-axis. Finally, plot the sine curve by using console.gotoxy to position the cursor prior to printing an asterisk ("*") to represent one point on the screen. Plot the curve between 0 and 4π radians. Each column on the screen should correspond to 4π/80 radians. Since the FSIN instruction only returns values between -1 and +1, you will need to scale the result by adding one to the value and multiplying it by 12. This will give the Y-coordinate of the point to plot. Increment through each of the X-coordinate positions on the screen when plotting the points (remember, each X-coordinate is 4π/80 radians greater than the previous X-coordinate).

42) Modify program (41) to simultaneously plot the COS curve at the same time you plot the SIN curve. Use at signs ("@") to plot the cosine points.

43) Write a program that accepts an integer value as a decimal (base 10) number and outputs the value in a user specified base between two and ten.

44) Modify program (43) so that it accepts an integer in one user-specified base and outputs the number in a second user-specified base (bases 2-10).

45) The factorial of a non-negative integer, *n!*, is the product of all integers between one and *n*. For example, 3! is 3*2*1 = 6. Write a function that computes n! using integer arithmetic. What is the largest value of *n* for which you can compute n! without overflow using 32-bit integers?

46) Since *n!* overflows so rapidly when using 32-bit integers, use the 64-bit integer capabilities of the FPU to compute *n!* (see problem 45). What is the maximum value of *n* for which you can compute n! when using 64-bit numbers?

47) You can estimate the value of the constant e using the following equation:

$$e = \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \ldots$$

Obviously, you cannot carry this out to an infinite number of terms, but the more terms you use the better the approximation. Write a program that will calculate *e* using this approximation that is accurate to at least 12 decimal places (e=2.71828182846).

48) You can compute the value of e^x using the following mathematical series:

$$e^x = \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots$$

Using the same number of terms required by problem (47), compute e^x with 12 digits of accuracy.

49) Write a program that reads the data from a text file and swaps the case of the alphabetic letters and writes the result to the console. Use a lookup table to perform the case swap (hint: it's probably easiest to have the program fill in the table using the first few instructions of the program rather than by typing the table by hand or write a separate program to generate the table). The program should prompt the user for the file name to translate.

50) Genokey Technologies manufactures portable keypads for laptop computers. They have boxes that will hold one, five, ten, twenty-five, and one hundred keypads. Write a program for the Genokey shipping clerk that will automatically choose the minimum number of boxes (and tell how many of each is required) that are necessary to ship *n* keypads to a distributor whenever that distributor orders *n* keypads.

51) Write a program that reads two eight-bit unsigned integer values from the user, zero-extends these integers to 16-bits, and then computes and prints their product as follows:

```
    123
x   241
------
    123
   492
  246
------
 29643
```

52) Extend the RPNcalculator program (found in the sample program section of the chapter on Real Arithmetic, see "Sample Program" on page 640) to add all the function which there are FPU instructions (e.g., COS, CHS, ABS, FPREM1, etc.).

53) Extend the calculator program above (52) to support the functions provided in the HLA "math.hhf" module.

54) Modify the RPN calculator program above (53) to support infix notation rather than postfix notation. For this assignment, remove the unary functions and operators.

13.3 Laboratory Exercises

Note: since this volume is rather long you should allow about twice as much time as normal to complete the following laboratory exercises.

Accompanying this text is a significant amount of software. The software can be found in the AoA_Software directory. Inside this directory is a set of directories with names like *Volume2* and *Volume3*, with the names obviously corresponding to the volumes in this textbook. All the source code to the example programs in this volume can be found in the subdirectories found in the Volume3 subdirectory. The Volume3\Ch13 subdirectory also contains the programs for this chapter's laboratory exercises. Please see this directory for more details.

13.3.1 Using the BOUND Instruction to Check Array Indices

The following program (Program 13.1) demonstrates how to use the BOUND instruction to check array indices at run time. This simple program reads an unsigned integer from the user and uses that value as an index into an array containing 10 characters then displays the character at the specified index. However, if the array index is out of bounds, this triggers an *ex.BoundInstr* exception and prints an error message (this program also handles a couple of other exceptions that *stdin.get* can raise).

```
// Program to demonstrate BOUND instruction.

program BoundLab;
#include( "stdlib.hhf" );

static

    index:          uns32;
    arrayBounds:    dword[2] := [ 0, 9 ];
    arrayOfChars:   char[ 10 ] :=
                        ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j' ];

    answer:         char;

begin BoundLab;

    repeat

        try

            // Read an integer index into "arrayOfChars" from the user:

            stdout.put( "Enter an integer index between 0 and 9: " );
            stdin.flushInput();
            stdin.get( index );

            // Verify that the input is in the range 0..9:

            mov( index, eax );
            bound( eax, arrayBounds );

            // If it was a good index, display the character at that
            // index:

            mov( arrayOfChars[ eax ], al );
            stdout.put
```



```
                (
                    "The character at that index is '",
                    (type char al),
                    "'"
                    nl
                );


            // Handle the exceptions possible in the stdin.get call as well
            // as the BOUND instruction exception.

            exception( ex.BoundInstr )

                stdout.put( "The index you entered is not in the range 0..9" nl );

            exception( ex.ConversionError )

                stdout.put( "You did not enter a valid unsigned integer" nl );

            exception( ex.ValueOutOfRange )

                stdout.put( "That integer value is way out of range!" nl );

            endtry;


            // Ask the user if they want to try again.  Force the user to
            // enter only a "Y" or an "N":
            repeat

                stdout.put( "Do you wish to try another index (Y/N)? " );
                stdin.flushInput();
                stdin.getc();

                // If the user enters a lower case character, convert it
                // to upper case.

                if( al in 'a'..'z' ) then

                    and( $5f, al );

                endif;
                mov( al, bl );  // Because cs.member wipes out AL.

            until( cs.member( al, {'Y', 'N'} ));

        until( bl = 'N' );


    end BoundLab;
```

Program 13.1 Using the BOUND Instruction to Check Array Indicies

Exercise A: Run this program and verify that it works in an acceptable fashion when the input is a legal integer in the range 0..9. Next, try entering a value greater than 10. Report the results of these experiments in your lab report.

Exercise B: At the end of the program, it asks you whether you want to rerun the program. The program uses the cs.member function to allow only a yes or no ("Y" or "N") response. Verify that this program will not accept any data other than "Y" or "N" when asking the user if they want to retry the operation.

Exercise C: As you can see, there is code that converts the character in the AL register to an upper case character if it is a lower case character. After the conversion, the character set membership test only checks for upper case characters. Verify that you can enter upper or lower case "Y" or "N" characters and the program still works.

Exercise D: The HLA Standard Library includes the CHARS module that supplies various character classification routines including functions like *chars.isLower* and *chars.isUpper*. These functions take a single character parameter and return true or false in the EAX register if the parameter is a lower case alphabetic or upper case alphabetic character (respectively). Modify the code in this sample program to test for lower case by making a call to the *chars.isLower* routine rather than using the boolean expression "(al in 'a'..'z')". Rerun the program and verify that it is still working correctly.[1] Include a copy of this converted program with your lab report.

Exercise E: In addition to character classification routines, the CHARS module also includes a couple of character conversion routines. The *chars.toUpper* routine will convert its single character parameter to upper case if it is lower case (it returns all other characters unchanged). Likewise, the *chars.toLower* routine converts the input parameter to a lower case character (if it was upper case to begin with). Both routines return the character in the AL register. They both return the original character if it was not alphabetic. Modify this program to use the *chars.toUpper* routine to do the upper case conversion rather than the IF statement it currently uses. Test the resulting program. Include a copy of this converted program with your lab report.

Exercise F: HLA supports a special, extended, syntax for the BOUND instruction that has three parameters. The second and third parameters of BOUND are integer constants specifying the lower and upper bounds to check. This form is often more convenient than the BOUND instruction appearing in Program 13.1 because you don't have to declare a two-element initialized array for use as the BOUND parameter. Modify your current program by eliminating the *arrayBounds* variable and using this form of the BOUND instruction. Verify that your program still allows indices in the range 0..9. Include the program listing of your modified program in your lab report.

Exercise G: Program 13.1 uses the literal constants nine and ten to denote the upper bounds on the array as well as the number of elements in the array. Using literal constants like this makes programs much more difficult to read and maintain. A better solution is to use a symbolic constant that you define once at the top of the program and reference throughout the code. By doing so, you can change the size of the array by changing only one or two statements in your program[2]. Modify this program so that it contains a single constant specifying the number of array elements. Use this constant wherever the number of array elements or the array's upper bound appears in the program (hint: use constant expressions to compute the value nine from this constant). Verify that the program still works correctly and include the source code with your lab report.

Exercise H: HLA predefines a special boolean VAL constant, @bound, that controls the compilation of the BOUND instruction. If @bound is true (the default) then HLA will compile BOUND instructions found in your program. If @bound is false, then HLA ignores all BOUND instructions it encounters. You can change the value of the @bound constant by using a statement of the form:

```
?@bound := false;
```

Modify Program 13.1 (the original version) and set the *arrayBounds* value so that it only allows array indices in the range 0..5. Run the program and verify this new operation. Next, insert "?@bound:=false;" immediately after the "begin pgm4_17;" statement in the program. Recompile and run the program and ver-

1. Note that using the IN operator is much more efficient than calling the *chars.isLower* function. Don't get the impression from this exercise that *chars.isLower* is the best way to check for a lower case character. The *chars.isLower* routine is appropriate in many cases, but this probably isn't a good example of where you would want to use this function. The only reason this exercise uses this function is to introduce it to you.

2. Don't forget, if you change the size of the character array in Program 13.1 then you will need to change the number of array elements in the initializer for that array as well.

ify that the BOUND instruction is no longer active by entering an index in the range 6..9. Describe the results in your laboratory report.

13.3.2 Using TEXT Constants in Your Programs

HLA's TEXT constants let you replace long repetitive sequences in your program with a single identifier, similar to the #define statement in C/C++[3]. This laboratory exercise demonstrates how you can use TEXT constants to save considerable typing in your programs.

Consider Program 13.2. This program uses TEXT constants to compress often-used text in three main areas: first, it reduces the strings "stdout.put" and "stdin.get" to the single identifiers *put* and *get* saving a bit of typing (since these calls occur frequently in HLA programs). Second, it replaces the string "(type uns32 i)" with the single identifier *ui*. Although this program only uses *ui* once, in a real program you might wind up using a single string like "(type uns32 i)" on several occasions. Finally, this program uses TEXT constants to combine several common exceptions into the *endOfTry* constant, saving considerable typing in each TRY..ENDTRY block.

```
// Demonstration of TEXT constants in a program.

program TextLab;
#include( "stdlib.hhf" );

const
    put: text := "stdout.put";
    get: text := "stdin.get";
    ui:  text := "(type uns32 i)";

    VOR: string := "Value out of range";
    CE:  string := "Conversion error";

    exRange: text :=
        "exception( ex.ValueOutOfRange ); "
            "put( VOR, nl );";

    exConv:  text :=
        "exception( ex.ConversionError ); "
            "put( CE, nl );";

    endOfTry:text :=
        "exRange; exConv; endtry";



static

    i:      int32;
    range:  dword[2] := [1,10];

begin TextLab;

    try

        put( "Enter an integer value: " );
```

3. HLA's TEXT constants don't support parameters making them weaker than C/C++'s #define macro capability. Fear not, however, HLA has it's own macro facility that is much more powerful than C/C++'s. You will learn about HLA's macro facilities in the chapter on macros and the compile-time language.


```
        get( i );
        put( "The value you entered was ", i, nl );

    endOfTry;


    try

        repeat

            put( "Now enter a negative integer: " );
            get( i );

        until( i < 0 );
        put( "As an unsigned integer, 'i' is ", ui, nl );

    endOfTry;


    try

        repeat

            put( "Now enter an integer between one and ten: " );
            get( i );
            mov( i, eax );
            bound( eax, range );

        until( i < 0 );
        put( "The value you entered was ", i, nl );

      exception( ex.BoundInstr );

        put( "The value was not in the range 1..10", nl );

    endOfTry;


end TextLab;
```

Program 13.2 TEXT Constant Laboratory Exercise Code

Exercise A: Compile and run this program. For each TRY..ENDTRY block, enter values that raise the conversion error exception and the value out of range exception[4]. For the last TRY..ENDTRY block, enter a value outside the range 1..10 to raise an *ex.BoundsInstr* exception. Describe what happens in your lab report.

Exercise B: Add a TEXT constant declaration for the *exBound* symbol. Set the value of the TEXT object equal to the string associated with the BOUND instruction exception (see the examples for *ex.ConversionError* and *ex.ValueOutOfRange* to figure out how to do this). Modify the last TRY..ENDTRY statement to use this TEXT constant in place of the associated exception handling section. Run the program to verify its operation. Include the modified program with your laboratory report.

4. Entering a 12-digit integer value will raise the *ex.ValueOutOfRange* exception. Typing non-numeric characters will raise the *ex.ConversionError* exception.

Exercise C: Modify the first TRY..ENDTRY block above to read and display an unsigned integer value using the *ui* TEXT constant in place of the *i* variable. Verify, by running the program, that it will no longer let you enter negative values. Include the modified program with your lab report.

Exercise D: In the *exRange* and *exConv* constants, the *put* statement uses the symbolic string constants *CE* and *VOR* to display the error message. Replace these symbolic identifiers with their corresponding literal string constants (Hint: you need to remember how to embed quotation marks inside a string). Run and test the program. Include the source code to this modification in your lab report and comment on the wisdom of using string constants rather than string literals in your programs.

13.3.3 Constant Expressions Lab Exercise

This laboratory exercise demonstrates the use of constant expressions in an HLA program. As you are reading through the following program, keep in mind that many of the constant expressions were added to this program simply to demonstrate various operators; you wouldn't normally use these operators in a trivial program such as this one. However, to write a complex program that fully uses constant expressions is beyond the scope of this laboratory exercise. Part of this laboratory exercise is to undo some of the excessive use of constant expressions in the program. So bear with this example.

```
// Demonstration of constant expressions in a program.

program ConstExprLab;
#include( "stdlib.hhf" );

const
    ElementsInArray := 10;

    sin:    string  := "stdin.";
    sout:   string  := "stdout.";

    put:    text    := sout + "put";
    get:    text    := sin  + "get";
    flush:  text    := sin  + "flushInput";


var
    input:          uns32;
    GoodInput:      boolean;
    InputValues:    int32[ ElementsInArray ];

readonly

    IV_bounds:  dword[2] := [0, ElementsInArray-1 ];


begin ConstExprLab;

    for( mov( 0, ebx ); ebx < ElementsInArray; inc( ebx )) do

        repeat

            mov( false, GoodInput );
            try

                put( "Enter integer #", (type uns32 ebx), ": " );
                flush();
                get( input );
```



```
                bound( ebx, IV_bounds );
                mov( input, InputValues[ ebx*4 ] );
                mov( true, GoodInput );

              exception( ex.ConversionError )

                put( "Number contains illegal characters", nl );

              exception( ex.ValueOutOfRange )

                put( "The number is way too large", nl );

              exception( ex.BoundInstr )

                put( "Array index is out of bounds", nl );

            endtry;

        until( GoodInput );

    endfor;

    // Now print the values entered by the user in the reverse order
    // they were entered. Note: this code doesn't use the scaled (*4)
    // indexed addressing mode.  So it initializes EBX with four times
    // the index of the last element in the array.

    mov( ( ElementsInArray - 1 ) * 4, ebx );
    while( (type int32 ebx) >= 0 ) do

        put
        (
            "Value at offset ",
            (type uns32 ebx):2,
            " was ",
            InputValues[ebx],
            nl
        );

        // Since we're not using the scaled (*4) index addressing mode,
        // drop the count by four for each array element since it's an
        // array of dwords.

        sub( 4, ebx );

    endwhile;

end ConstExprLab;
```

Program 13.3 Constant Expressions Laboratory Exercise

Exercise A: Compile this program and run it. Describe the operation of the program in your laboratory report.

Exercise B: One extreme use of constant expressions in this program appears in the following code:

```
sin:    string  := "stdin.";
sout:   string  := "stdout.";
```



```
    put:    text    := sout + "put";
    get:    text    := sin  + "get";
    flush:  text    := sin  + "FlushInput";
```

Modify the program to eliminate the *sin* and *sout* constants from the code. The program should be easier to read once you have completed this task. Compile and run the program and verify that it produces the same results as before the change. Include the source code to the modified program with your lab report. Briefly explain why you think that the use of the string concatenation operator wasn't worth much in this code.

Exercise C: This program provides the *ElementsInArray* constant that lets you easily change the size of the array that this program works with. Change the value of this constant to five and recompile and run the program. Verify that it works correctly for five array elements (versus ten in the previous version). Identify all locations in the program that modifying the *ElementsInArray* constant affects (use a hi-liter or other such tool to make these points visible in the listing appearing in your lab report). Describe in the lab report write-up the effort that would be needed to change the size of the array had this program not used the *ElementsInArray* constant.

Exercise D: The *get* and *put* constants save considerable typing in this program. Can you think of a reason why it might not be such a good idea to use TEXT constants like these to save typing in your program? Justify your answer in your laboratory report.

13.3.4 Pointers and Pointer Constants Exercises

Pointers are especially important in assembly language programs. This laboratory exercise demonstrates that HLA programs often use pointers to access elements of an array. This program also demonstrates how to use the BOUND instruction to check to see if a pointer contains an address between two extremes.

Conceptually, this program is very simple. It asks the user for an integer and then prints that many characters out of an array (checking, of course, to verify that the program doesn't exceed array bounds). To print those characters, this program loads the base address of an array into a register and then increments that register to step through the array. Contrast this with examples appearing later in these lab exercises where the program leaves the register containing the base address unmodified and increments a separate index register.

```
// Demonstration of pointers and pointer constants in a program.

program PointerLab;
#include( "stdlib.hhf" );

const
    NumberOfChars := 8;

type
    CharPtr: pointer to char;

static

    CharArray: char[ NumberOfChars ] :=
                    ['1', '2', '3', '4', '5', '6', '7', '8' ];

    CA_bounds: CharPtr[2] := [&CharArray, &CharArray + (NumberOfChars-1)];


begin PointerLab;

    // Get the number of characters to print into EAX:
```



```
    stdout.put( "How many characters would you like to print? " );
    stdin.getu32();

    // Get address of CharArray into EBX:

    lea( ebx, CharArray );

    // Print the specified number of characters:

    while( eax > 0 ) do

        bound( ebx, CA_bounds );
        stdout.putc( [ebx] );
        inc( ebx );
        dec( eax );

    endwhile;
    stdout.newln();

end PointerLab;
```

Program 13.4 Pointers and Pointer Constants Laboratory Exercise Code

Exercise A: Compile and run this program. Describe its operation in your laboratory report.

Exercise B: This program contains a constant *NumberOfChars*. Change this constant from eight to ten and recompile the program. Explain why this one change is not sufficient to change the size of the array from eight to ten elements. Make an appropriate change to this program so that it will compile and run correctly. Compare the number of changes needed in the program to get it to compile and run with the number of changes that would be necessary had this program not used the *NumberOfChars* constant. Describe the benefits to this program of using the *NumberOfChars* constant.

Exercise C: This program does not contain a TRY..ENDTRY block to handle the BOUND instruction exception. Add such a statement to the program that will print a nice error message. If the exception occurs, the program should quit.

13.3.5 String Exercises

In this laboratory exercise you will work with statically and dynamically allocated strings. The first new feature found in this program is the *str.strvar* declaration. The *str.strvar* data type lets you statically allocate storage for a string variable in the STATIC section of your program[5]. This data type takes a single parameter that specifies the maximum number of legal characters in the string variable. This data type sets aside the appropriate amount of storage for the string, initializes the maximum length field to the value passed as a parameter, and then initializes the string variable (*s1* in the example below) with the address of this newly allocated string. The *str.strvar* data type is very useful for declaring small strings in your STATIC section because it provides both declaration and allocation in a single statement.

This program uses *str.strvar* to allocate storage for a ten character string. When the program starts running, it requests that the user enter a string longer than ten characters to demonstrate the *ex.StringOverflow* exception. Finally, this program concludes by demonstrating how to access the individual characters in a string.

5. Note that *str.strvar* is legal *only* in the STATIC section. You can not use it in any of the other HLA variable declaration sections.


```
// Demonstration of strings in a program.

program StringLab;
#include( "stdlib.hhf" );

static

    s1: str.strvar( 10 );
    s2: string;

begin StringLab;

    try

        stdout.put( "Enter a string longer than 10 characters: " );
        stdin.gets( s1 );
        str.length( s1 );
        stdout.put
        (
            "The string you entered was only ",
            (type uns32 eax ),
            " characters long, "
            nl
            "try a longer string next time"
            nl
        );

      exception( ex.StringOverflow )

        stdout.put( "stdin.gets raised the ex.StringOverflow exception" nl );

    endtry;


    // Use stdin.a_gets to read a string and allocate storage for it.
    // Print the characters in the string one at a time after reading
    // the string.
    //
    // There is something wrong with this code, can you fix it?

    stdout.put( nl "Enter a string of any length: " );
    stdin.a_gets();
    mov( eax, s2 );
    mov( eax, ebx );
    str.length( eax );
    stdout.put
    (
        "The string length was ",
        (type uns32 eax),
        nl,
        "The string you entered was",
        nl
    );

    while( eax > 0 ) do

        stdout.put( (type char [ebx]), nl );
        dec( eax );
        inc( ebx );
```



```
    endwhile;
    stdout.newln();

end StringLab;
```

Program 13.5 String Exercises Code

Exercise A: Compile this program and run it. Be sure to follow all directions provided in the program. Describe what the program does in your lab report.

Exercise B: There is a defect in the way this program handles dynamically allocated strings. Locate the defect and correct it. Include a copy of the corrected program with your lab report. Note that this program runs correctly; you will have to analyze the source code to find the defect.

13.3.6 String and Character Set Exercises

In this laboratory exercise you'll be working with a program that disassembles an input string. The program begins by reading the user's full name (first and last name). It then extracts the first and last names into separate strings. Although this is a relatively simple program, it demonstrates a very important concept in string manipulation. Most complex programs that manipulate strings need to break those strings down into their constituent parts. The techniques this program uses are the basis for much larger and more complex programs.

This program also demonstrates the use of character sets and character set expressions. In particular, note the following constant declaration in the program:

```
NonAlphaChars: cset := -AlphabeticChars - {#0};
```

The "-" operator, prefacing a character set, takes the *complement* of that set. The complement of a set contains all the characters that are *not* in that set. So "-AlphabeticChars" returns the set of all characters that are not alphabetic. The "- {#0}" component of the expression above removes the NUL character from the set of nonalphabetic characters. (Note that in this case the "-" operator is a binary operator denoting set difference.) This operation removes the NUL character from the set, not because it's an alphabetic character, but because the program uses the *NonAlphaChars* set to skip over characters while looking for the last name. If a last name is not present, this code needs to stop scanning once it encounters the end of the string. The #0 character always marks the end of an HLA string. By removing #0 from the *NonAlphaChars* set, the code that skips non-alphabetic characters will automatically stop if it encounters the end of the string.

```
// Demonstration of strings and character setsin a program.

program strcsetLab;
#include( "stdlib.hhf" );

const
    AlphabeticChars:    cset := {'a'..'z', 'A'..'Z' };
    NonAlphaChars:      cset := -AlphabeticChars - {#0};

var
    StartOfFirstName:   uns32;
    LengthOfFirstName:  uns32;
    StartOfLastName:    uns32;
    LengthOfLastName:   uns32;
```



```
    FullName:           string;
    FirstName:          string;
    LastName:           string;

begin strcsetLab;

    // Read the full name from the user:

    stralloc( 256 );        // Allocate storage for the string.
    mov( eax, FullName );

    stdout.put( "Enter your full name (first, then last): " );
    stdin.get( FullName );

    // Skip over any leading spaces in the name:

    mov( FullName, ebx );
    mov( 0, esi );
    while( (type char [ebx+esi]) = ' ' ) do

        inc( esi );

    endwhile;

    // Okay, presumably we're at the beginning of the first
    // name.  Find the end of it here:

    mov( esi, StartOfFirstName );
    mov( esi, LengthOfFirstName );
    while( cs.member( (type char [ebx+esi]), AlphabeticChars )) do

        inc( esi );

    endwhile;
    sub( esi, LengthOfFirstName );
    neg( LengthOfFirstName );

    // Skip over any non-alphabetic characters

    while( cs.member( (type char [ebx+esi]), NonAlphaChars )) do

        inc( esi );

    endwhile;

    // Okay, find the end of the last name:

    mov( esi, StartOfLastName );
    while( cs.member( (type char [ebx+esi]), AlphabeticChars )) do

        inc( esi );

    endwhile;
    sub( StartOfLastName, esi );
    mov( esi, LengthOfLastName );


    // Extract the first and last names from the string:

    str.a_substr( FullName, StartOfFirstName, LengthOfFirstName );
```



```
    mov( eax, FirstName );

    str.a_substr( FullName, StartOfLastName, LengthOfLastName );
    mov( eax, LastName );

    // Display the results:

    stdout.put( "First: ", FirstName, nl );
    stdout.put( "Last:  ", LastName, nl );

    // Clean up the allocated storage:

    strfree( FullName );
    strfree( FirstName );
    strfree( LastName );

end strcsetLab;
```

Program 13.6 String and Character Set Exercises Code

Exercise A: Compile and run this program. Describe the operation of the program and its output in your laboratory report.

Exercise B: This program uses statements like "while(cs.member((type char [ebx+esi]), Alphabetic-Chars)) do" to loop while the current character in the string is a member of some character set. Explain why you cannot use the "IN" operator described in this chapter in place of the *cs.member* function. (Hint: if the reason isn't obvious, trying changing the program and compiling it.)

The following sample program also demonstrates the synergy between strings and character sets in a program. This program reads a string from the user and converts that string into a character set (effectively removing duplicate characters from the string). It then displays the character set, the vowels, the consonants, and non-alphabetic characters found in the string. This program demonstrates some run-time character set functions (like intersection) as well as compile-time constant expressions. This program also demonstrates the new form of the IN operator in boolean expressions that this chapter introduces.

```
// Demonstration of character sets and strings in a program.

program strcsetLab2;
#include( "stdlib.hhf" );

const

    AlphabeticChars:    cset := {'a'..'z', 'A'..'Z' };

    Vowels:             cset := {
                                    'a', 'e', 'i', 'o', 'u', 'y', 'w',
                                    'A', 'E', 'I', 'O', 'U', 'Y', 'W'
                                };

    Consonants:         cset := AlphabeticChars - Vowels;


var

    InputLine:      string;
```



```
        InputSet:       cset;
        VowelSet:       cset;
        ConsonantSet:   cset;
        NonAlphaSet:    cset;

    begin strcsetLab2;

        repeat

            stdout.put( "Enter a line of text: " );
            stdin.flushInput();
            stdin.a_gets();
            mov( eax, InputLine );
            stdout.put( nl, "The input line: '", InputLine, "'" nl nl );

            cs.strToCset( InputLine, InputSet );
            stdout.put( "Characters on this line: '", InputSet, "'" nl );

            cs.cpy( InputSet, VowelSet );
            cs.intersection( Vowels, VowelSet );
            stdout.put( "Vowels on this line: '", VowelSet, "'" nl );

            cs.cpy( InputSet, ConsonantSet );
            cs.intersection( Consonants, ConsonantSet );
            stdout.put( "Consonants on this line: '", ConsonantSet, "'" nl );

            cs.cpy( InputSet, NonAlphaSet );
            cs.intersection( -AlphabeticChars, NonAlphaSet );
            stdout.put
            (
                "Nonalphabetic characters on this line: '",
                NonAlphaSet,
                "'"
                nl
            );

            repeat

                stdout.put
                (
                    "Would you like to try another line of text? (Y/N): "
                );
                stdin.flushInput();
                stdin.getc();

            until( al in {'Y', 'y', 'N', 'n'} );

        until( al in {'n', 'N'} );


    end strcsetLab2;
```

Program 13.7 Character Classification Program

Exercise A: Compile and run this program. Describe its output and operation in your laboratory report.

Exercise B: Modify this program to display any decimal digit characters appearing in the input string. Compile and run the program to test its operation. Include the program listing in your laboratory report.

13.3.7 Console Array Exercise

Perhaps the most commonly encountered two-dimensional object a programmer encounters is the video display. The Windows console window, for example, is typically an 80x25 array of characters[6]. The HLA Standard Library console module provides various routines that let you manipulate the matrix of characters on the screen. In the previous chapter, for example, you saw how to use the *console.fillRect* routine to fill a rectangular region of the screen with a given character and attribute. This laboratory exercise demonstrates how to extract a rectangular region from the screen and how to redraw that text taken from the screen. This sample program also demonstrates how to write an arbitrary matrix of text to the screen.

This program calls the HLA Standard Library *console.getRect* routine to read the data from a rectangular portion of the screen into a local array of characters. This program uses *console.getRect* to save the screen display upon entry into the program. In addition to *console.getRect*, this program uses *console.putRect* to rapidly write the contents of a two-dimensional array of characters to the console display. For example, this program calls *console.putRect* to restore the original screen display when this program quits.

Between saving and restoring the original screen contents, this program moves a sequence of asterisks around in a local 80x25 array of characters and (after each movement) copies the characters to the video display. This is an extremely slow way to redraw the screen; however, today's machines are so fast that this inefficiency works to this program's advantage - it's a built-in governor that slows down the program so you can see it work.

```
// Demonstration of arrays in a program.
//
//  This code demonstrates how to access elements of
//  a two-dimensional array.  It also demonstrates how to
//  use the console.getRect and console.putRect routines.

program consoleArrayLab;
#include( "stdlib.hhf" );

const
    NumCols := 80;
    NumRows := 25;

type
    screen: char[ NumRows, NumCols ];

var

    column:     uns32;
    row:        uns32;
    SnapShot:   screen;


static

    playingField: screen := [ NumRows dup [ NumCols dup [' ']]];

begin consoleArrayLab;

    console.getRect(  0, 0, NumRows-1, NumCols-1, (type char SnapShot));
```

6. Actually, under Windows the console window can be made various different sizes. 80x25, however, is the default size.


```
            console.cls();

            // Fill an 80x24 region of the screen with a blue background.
            // Set the foreground attribute to yellow:

            console.fillRect
            (
                0,
                0,
                NumRows-1,
                NumCols-1,
                ' ',
                win.bgnd_Blue | win.fgnd_Yellow
            );

            // Play the "game".  Repeat the nested loops once for each
            // row on the screen.

            for( mov( 0, row ); row < NumRows; inc( row )) do

                // For each row, draw a row of asterisks across the screen
                // one character at a time by storing the asterisk into
                // the "playingField" array and then copying this array to
                // the screen.

                for( mov( 0, column ); column < NumCols; inc( column )) do

                    intmul( NumCols, row, ebx );
                    add( column, ebx );
                    mov( '*', playingField[ ebx ] );
                    console.putRect
                    (
                        0,
                        0,
                        NumRows-1,
                        NumCols-1,
                        (type byte playingField)
                    );

                endfor;

                // Now erase the row of asterisks one character at a time
                // by overwriting the asterisks in "playingField" and drawing
                // the matrix to the screen after blotting out each character.

                for( mov( 0, column ); column < NumCols; inc( column )) do

                    intmul( NumCols, row, ebx );
                    add( column, ebx );
                    mov( ' ', playingField[ ebx ] );
                    console.putRect
                    (
                        0,
                        0,
                        NumRows-1,
                        NumCols-1,
                        (type byte playingField)
                    );

                endfor;
```



```
    endfor;

    stdout.put( "Press Enter to continue: " );
    stdin.readLn();
    console.cls();
    console.putRect
    (
        0,
        0,
        NumRows-1,
        NumCols-1, (type byte SnapShot)
    );
    console.gotoxy( NumRows-1, 0 );


end consoleArrayLab;
```

Program 13.8 Console Matrix Program

Exercise A: Compile and run this program. Describe it's operation and output in your laboratory report.

Exercise B: Change the NumCols and NumRows constants so that the program operates in a 40x20 window rather than an 80x25 window. Compile and rerun the program and verify that it still works properly. Include the source code for the modified program in your lab report.

Exercise C: Modify the program so that it draws the string of asterisks from top to bottom, then left to right rather than from left to right, then top to bottom. Include the source code to the modified program in your lab report.

13.3.8 Multidimensional Array Exercises

The program for this exercise demonstrates an *outer product* computation. An outer product is an operation that takes an N-element array and an M-element array and produces an NxM matrix with cell [i,j] containing the value specified by some function f(Narray[i], Marray[j]) for all values 0 <= i < N and 0 <= j < M. The classic example of an outer product is the multiplication table you learned in elementary school (where the function is the multiplication operation). The program in this example computes an addition table by summing the elements of the two arrays together.

```
// Demonstration of multidimensional arrays in a program.

program multidimArrayLab;
#include( "stdlib.hhf" );

const
    NumRows := 10;
    NumCols := 10;

static

    RowStart:   uns32;
    ColStart:   uns32;

    SumArray:   uns32[ NumRows, NumCols];
    ColVals:    uns32[ NumCols ];
```



```
        RowVals:    uns32[ NumRows ];

        GoodInput:  boolean;

    begin multidimArrayLab;

        stdout.put
        (
            “Addition table generator” nl
            “------------------------” nl
            nl
        );

        // Get the starting value for the rows from the user:

        repeat

            try

                mov( false, GoodInput );
                stdout.put( “Enter a starting value for the rows: “ );
                stdin.get( RowStart );
                mov( true, GoodInput );

              exception( ex.ConversionError )

                stdout.put( “Bad characters in number, please re-enter”, nl );

              exception( ex.ValueOutOfRange )

                stdout.put( “The value is out of range, please re-enter”, nl );

            endtry;

        until( GoodInput );

        // Fill in the RowVals array:

        mov( RowStart, eax );
        for( mov( 0, ebx ); ebx < NumRows; inc( ebx )) do

            mov( eax, RowVals[ ebx*4 ] );
            inc( eax );

        endfor;




        // Get the starting value for the columns from the user:

        repeat

            try

                mov( false, GoodInput );
                stdout.put( “Enter a starting value for the columns: “ );
                stdin.get( ColStart );
                mov( true, GoodInput );
```



```
      exception( ex.ConversionError )

        stdout.put( "Bad characters in number, please re-enter", nl );

      exception( ex.ValueOutOfRange )

        stdout.put( "The value is out of range, please re-enter", nl );

    endtry;

until( GoodInput );

// Fill in the ColVals array:

mov( ColStart, eax );
for( mov( 0, ebx ); ebx < NumCols; inc( ebx )) do

    mov( eax, ColVals[ ebx*4 ] );
    inc( eax );

endfor;

// Now generate the addition table:

for( mov( 0, ebx ); ebx < NumRows; inc( ebx )) do

    for( mov( 0, ecx ); ecx < NumCols; inc( ecx )) do

        intmul( NumCols, ebx, edx );    // Compute index into
        add( ecx, edx );                // array as ebx*NumCols+ecx.
        mov( ColVals[ ecx*4 ], eax );   // Compute sum for this cell.
        add( RowVals[ ebx*4 ], eax );
        mov( eax, SumArray[ edx*4 ]  );

    endfor;

endfor;


// Now print the addition table.
//
// Begin by printing the Column values above the matrix

stdout.put( nl " add |" );
for( mov( 0, ebx ); ebx < NumCols; inc( ebx )) do

    stdout.put( ColVals[ ebx*4 ]:4, " | " );

endfor;

// Print a line below the column values.

stdout.put( nl "-----+" );
for( mov( 0, ebx ); ebx < NumCols-1; inc( ebx )) do

    stdout.put( "-----+-" );

endfor;
stdout.put( "-----|" nl );
```



```
        // Now print the body of the matrix.  This also prints Row
        // values in the left hand column of the matrix.

        for( mov( 0, ebx ); ebx < NumRows; inc( ebx )) do

            // Print the current row value.

            stdout.put( RowVals[ ebx*4 ]:4, " :" );

            // Now print the values for this row.

            for( mov( 0, ecx ); ecx < NumCols; inc( ecx )) do

                intmul( NumCols, ebx, edx );    // Compute index into
                add( ecx, edx );                // array as ebx*NumCols+ecx.
                stdout.put( SumArray[ edx*4 ]:4, " | " );

            endfor;

            // Print a line below each row of numbers

            stdout.put( nl "-----+" );
            for( mov( 0, ecx ); ecx < NumCols-1; inc( ecx )) do

                stdout.put( "-----+-" );

            endfor;
            stdout.put( "-----|" nl );

        endfor;


    end multidimArrayLab;
```

Program 13.9 Addition Table Generator

Exercise A: Compile and run this program. Describe its output in your lab report.

Exercise B: Modify the NumRows and NumCols constants in this program to generate a 5x6 array rather than a 10x10 array. Recompile and run the program. Describe the output in your lab report.

Exercise C: (optional) Modify the program to use the line drawing graphic characters (see appendix B) rather than the ASCII dash, vertical bar, and hyphen characters.

13.3.9 Console Attributes Laboratory Exercise

The program in this laboratory exercise demonstrates the *console.info* call that returns various bits of interesting information about the Windows console device. This information comes in handy when writing console applications that work regardless of the size of the console window. For more information about the information that *console.info* returns, please see the definition of the windows console screen buffer info record in the "WIN32.HHF" header file.

```
// Demonstration of records and the console.info routine
// (and the win.CONSOLE_SCREEN_BUFFER_INFO data type).
```



```
program ConsoleAttrLab;
#include( "stdlib.hhf" );

static

    csbi:       win.CONSOLE_SCREEN_BUFFER_INFO;

begin ConsoleAttrLab;

    // Grab the screen metric information:

    console.info( csbi );


    // Clear the screen.

    console.cls();

    // Set the whole screen to blue:

    movzx( csbi.dwSize.Y, eax );
    movzx( csbi.dwSize.X, ebx );
    dec( eax );
    dec( ebx );
    console.fillRectAttr( 0, 0, ax, bx, win.bgnd_Blue );
    console.setOutputAttr( win.bgnd_Blue | win.fgnd_Yellow );

    // Read and display the current console window title:

    console.a_getTitle();
    stdout.put
    (
        "Previous console window title was: '",
        (type string eax),
        "'" nl
    );
    strfree( eax );

    // Set the console window title:

    console.setTitle( "Art of Assembly Program 4.26" );

    // Display the screen metric information:

    stdout.put
    (
        "Size X=", csbi.dwSize.X, nl,
        "Size Y=", csbi.dwSize.Y, nl,
        "Cursor X=", csbi.dwCursorPosition.X, nl,
        "Cursor Y=", csbi.dwCursorPosition.Y, nl,
        "Window Position Left=", csbi.srWindow.left, nl,
        "Window Position Right=", csbi.srWindow.right, nl,
        "Window Position Top=", csbi.srWindow.top, nl,
        "Window Position Bottom=", csbi.srWindow.bottom, nl,
        "Character display attribute=$", csbi.wAttributes, nl,
        nl
        "Maximum window height=", csbi.dwMaximumWindowSize.Y, nl,
        "Maximum window width =", csbi.dwMaximumWindowSize.X, nl,
        nl
    );
```



```
    // Let the user see everything before we clean up and quit.

    stdout.put( "Press ENTER to continue: " );
    stdin.readLn();

    // Be nice and set the console window back to black & white:

    console.setOutputAttr( csbi.wAttributes );
    console.cls();


end ConsoleAttrLab;
```

Program 13.10 Displaying Console Attributes

Exercise A: Compile and run this program. Describe the console metrics for your particular system in your laboratory report.

Exercise B: Modify this program to save the screen text upon entry into the program and restore the screen when this program exits. Use the cursor position in the *csbi* record to restore the cursor to its original position upon program exit. Include the modified source code with your lab report.

13.3.10 Records, Arrays, and Pointers Laboratory Exercise

The following program demonstrates dynamic allocation of an array of records. It also demonstrates enumerated data types and record constants. This program asks the user to specify some number of students and then it inputs the data for the specified students and stores the data into the dynamic array it creates.

```
// Demonstration of a pointer to an array of records.
// This is a small student database program the lets the
// user input a bunch of student names and it prints a report
// after the data is entered.

program StudentDatabase;
#include( "stdlib.hhf" );

const

    // The following text string appears frequently in this code.

    CurStudent: text := "(type EngStudent [edi+ecx])";

type
    EngMajor:   enum{ ElecEng, CompSci, MechEng, BioChemEng };
    Standing:   enum{ Freshman, Sophomore, Junior, Senior, Grad };

    // Here's the basic data type for our student database:

    EngStudent:
        record

            StudentName:    string;
            ID:             string;
```



```
            major:          EngMajor;
            Year:           Standing;

        endrecord;

    ESPtr:  pointer to EngStudent;


readonly

    // The following string make it easy to output
    // the EngMajor and Standing enumerated data types.

    MajorStrs:      string[4] :=
                        [
                            “Electrical Engineering “,
                            “Computer Science       “,
                            “Mechanical Engineering “,
                            “Biochemical Engineering”
                        ];

    YearStrs:       string[4] :=
                        [
                            “Freshman “,
                            “Sophomore”,
                            “Junior   “,
                            “Senior   “
                        ];

    // An extra student to copy into the list.
    // This exists here mainly to demonstrate record constants.

    JohnSmith:      EngStudent :=
                        EngStudent:
                        [
                            “John Smith”,
                            “555-55-5555”,
                            CompSci,
                            Junior
                        ];


static

    NumStudents:    uns32;
    StudentArray:   ESPtr;
    GoodInput:      boolean;




begin StudentDatabase;

    stdout.put( “Engineering Student Database Program” nl nl );

    // Get the number of students the user wants to enter into
    // the database.  Because this is just a demo, ask them if
    // they really want to enter more than 10 students if they
    // enter a large number.

    repeat
```



```
        try

            mov( false, GoodInput );
            stdout.put
            (
                nl
                "How many students would you like to enter? "
            );
            stdin.get( NumStudents );
            mov( true, GoodInput );

            // Allow 0..10 students with no questions asked.
            // But if they specify more than 10...

            if( NumStudents > 10 ) then

                stdout.put
                (
                    "Are you sure you want to enter ",
                    NumStudents,
                    " students into the database? (Y/N):"
                );
                stdin.flushInput();
                stdin.getc();

                // Set GoodInput to true if they answer yes.

                cs.member( al, {'y', 'Y'} );
                mov( al, GoodInput );

            endif;

          exception( ex.ConversionError )

            stdout.put( "Illegal characters in number, please re-enter" nl );

          exception( ex.ValueOutOfRange )

            stdout.put( "Value is too big, please re-enter" nl );

        endtry;

    until( GoodInput );


    // Okay, we've got the number of students.  Bump in by one so we
    // can sneak John Smith's record into the data base.  Then allocate
    // storage for a dynamic array of records.

    mov( NumStudents, eax );    // Get the user-specified count.
    inc( eax );                 // Make room for John Smith.

    // Since each student consumes "@size( EngStudent )" bytes, we need
    // to multiply our student count by this value to compute the number
    // of bytes we will need.

    intmul( @size( EngStudent ), eax );

    // Allocate storage for the array of students.
```



```
malloc( eax );
mov( eax, StudentArray );


// Copy the data for John Smith to the first element of
// our array.  Note: This code just copies the pointers
// to the StudentName and ID fields because no modifications
// are ever made to the JohnSmith record in this program.

mov( JohnSmith.StudentName, ecx );
mov( ecx, (type EngStudent [eax]).StudentName );

mov( JohnSmith.ID, ecx );
mov( ecx, (type EngStudent [eax]).ID );

mov( JohnSmith.major, cl );
mov( cl, (type EngStudent [eax]).major );

mov( JohnSmith.Year, cl );
mov( cl, (type EngStudent [eax]).Year );


// Okay, now input each of the remaining students from the user:

mov( StudentArray, edi );   // Get base address of array.

for( mov( 1, ebx ); ebx <= NumStudents; inc( ebx )) do

    intmul( @size( EngStudent ), ebx, ecx );    // Index into array.

    stdout.put( "Enter student's name: " );
    stdin.flushInput();
    stdin.a_gets();
    mov( eax, CurStudent.StudentName );

    stdout.put( "Enter the student ID: " );
    stdin.a_gets();
    mov( eax, CurStudent.ID );

    // Get (and verify) F, S, J, or N from the user to select
    // this student's standing in the college.

    repeat

        stdout.put
        (
            "Is this student a "
            "F)reshman, S)ophomore, J)unior, or seN)ior? "
        );
        stdin.flushInput();
        stdin.getc();
        chars.toUpper( al );

    until( al in {'F', 'S', 'J', 'N'} );
    if( al = 'F' ) then

        mov( Freshman, CurStudent.Year );

    elseif( al = 'S' ) then

        mov( Sophomore, CurStudent.Year );
```



```
elseif( al = 'J' ) then

    mov( Junior, CurStudent.Year );

else

    mov( Senior, CurStudent.Year );

endif;


// Get the student's major from the user.  Again,
// verify that the input is correct before proceeding.

repeat

    stdout.put
    (
        "What is this student's major?" nl
        "  B)iochemical Engineering" nl
        "  C)omputer Science" nl
        "  E)lectrical Engineering" nl
        "  M)echanical Engineering" nl
        nl
        "(B,C,E,M)? "
    );
    stdin.flushInput();
    stdin.getc();
    chars.toLower( al );

until( al in {'b', 'c', 'e', 'm'} );
if( al = 'b' ) then

    mov( BioChemEng, CurStudent.major );

elseif( al = 'c' ) then

    mov( CompSci, CurStudent.major );

elseif( al = 'e' ) then

    mov( ElecEng, CurStudent.major );

else

    mov( MechEng, CurStudent.Year );

endif;

// Okay, store away the CurStudent variable data into
// the next available slot in the ESPtr array.

mov( StudentArray, eax );

// Compute index into array:

intmul( @size( EngStudent ), ebx, ecx );

mov( CurStudent.StudentName, edx );
mov( edx, (type EngStudent [eax+ecx]).StudentName );
```



```
            mov( CurStudent.ID, edx );
            mov( edx, (type EngStudent [eax+ecx]).ID );

            mov( CurStudent.major, dl );
            mov( dl, (type EngStudent [eax+ecx]).major );

            mov( CurStudent.Year, dl );
            mov( dl, (type EngStudent [eax+ecx]).Year );



        endfor;

        // Okay, now that we've got all the students into the system,
        // print a brief report.

        stdout.put
        (
            "-----Name---------- "
            "-----ID----- "
            "-------Major---------- "
            "--Year---"
            nl
            nl
        );

        mov( StudentArray, edi );   // Not really needed, but just to be sure.

        for( mov( 0, ebx ); ebx <= NumStudents; inc( ebx )) do

            intmul( @size( EngStudent ), ebx, ecx );    // Index into array.
            movzx( CurStudent.major, edx );
            movzx( CurStudent.Year, esi );
            stdout.put
            (
                CurStudent.StudentName:-20, " ",
                CurStudent.ID:-12, " ",
                MajorStrs[ edx*4 ]:-23, " ",
                YearStrs[ esi*4 ],
                nl
            );

        endfor;
        stdout.newln();



    end StudentDatabase;
```

Program 13.11 Student Database Program

Exercise A: Compile and run this program. Describe its output in your lab report.

Exercise B: Modify this program to let the user specify that only those students in a single major be printed (by specifying a single letter to select the major- see the body of the code for details on this). Provide an "A)ll" option that prints the full report as the program currently does.

13.3.11Separate Compilation Exercises

In this laboratory exercise you will compile a multipart program and link it into a single executable file. You will also construct a makefile for the project and test the workings of the makefile by making minor modifications to the program. Finally, you will split the main program into two pieces and move one section into its own module.

The following listings provide the necessary source code for this program. Note that these modules can be found in the CH05 subdirectory of the accompanying CD. The main program is lab5_1.hla; the header file is lab5_1.hhf; the first unit can be found in lab5_1db.hla; the second unit is in the file lab5_1unita.hla.

```
program cor_mainPgm;
#include( "stdlib.hhf" );
#include( "sepCompDemo.hhf" );

    // GetYorN-
    //
    //  This procedure prints a prompt that requires a yes/no
    //  answer.  It continually prompts the user to input Y or N
    //  until they enter one of these characters.  It returns
    //  true in AL if the user entered "Y" or "y".  It returns
    //  false if the user entered "n" or "N".

    procedure GetYorN( Prompt:string ); returns( "AL" );
    begin GetYorN;

        repeat

            stdout.put( Prompt, " (Y/N):" );
            stdin.flushInput();
            stdin.getc();

        until( al in {'y', 'n', 'Y', 'N' });
        if( al in {'y', 'Y'} ) then

            mov( true, al );

        else

            mov( false, al );

        endif;

    end GetYorN;


    // GetCinSet-
    //
    //  This is another procedure, similar to the above, that
    //  repeatedly requests input from the user until the user
    //  enters a valid character.  This procedure returns once
    //  the user enters a character that is a member of the
    //  cset passed as a parameter.

    procedure GetCinSet( legalInput:cset ); returns( "AL" );
    begin GetCinSet;
```



```
        repeat

            stdout.put( " {", legalInput, "}:" );
            stdin.flushInput();
            stdin.getc();

        until( al in legalInput );

    end GetCinSet;


var
    artist: string;
    title:  string;


begin cor_mainPgm;

    stdout.put
    (
        "Welcome to the CD database program" nl
        "-----------------------------------" nl
        nl
    );
    repeat

        stdout.put( "Select CD by A)rtist or by T)itle? " );
        GetCinSet( {'a', 'A', 't', 'T'} );
        chars.toUpper( al );
        if( al = 'A' ) then

            stdout.put( "Input artist's name: " );
            stdin.flushInput();
            stdin.a_gets();
            mov( eax, artist );
            SearchByArtist( artist, MyCDCollection );
            strfree( artist );

        else

            stdout.put( "Input CD's title: " );
            stdin.flushInput();
            stdin.a_gets();
            mov( eax, title );
            SearchByTitle( title, MyCDCollection );
            strfree( title );

        endif;

        stdout.newln();
        GetYorN( "Would you like to search for another CD?" );

    until( !al );


end cor_mainPgm;
```

Program 13.12 The MAIN Program for this Laboratory Exercise (mainPgm.hla)


```
const
    NumCDs := 10;

type
    CD: record

            Title:     string;
            Artist:    string;
            Publisher:string;
            CopyrightYear:uns32;

        endrecord;

    CDs: CD[ NumCDs ];


static
    MyCDCollection: CDs; external;



    procedure SearchByTitle( Title:string; var CDList:CDs ); external;
    procedure SearchByArtist( Artist:string; var CDList:CDs ); external;
```

Program 13.13 The sepCompDemo.hhf Header File for this Laboratory Exercise

```
unit Unitdb;
#include( "stdlib.hhf" );
#include( "sepCompDemo.hhf" );


static
    MyCDCollection: CDs :=
        [
            CD:[ "The Cars Greatest Hits", "The Cars", "Elektra", 1985 ],
            CD:[ "Boston", "Boston", "Epic", 1976 ],
            CD:[ "Imaginos", "Blue Oyster Cult", "Columbia", 1988 ],
            CD:[ "Greatest Hits", "Dan Fogelberg", "Epic", 1982 ],
            CD:[ "Walk On", "Boston", "MCA", 1994 ],
            CD:[ "Big Ones", "Loverboy", "Columbia", 1989 ],
            CD:[ "James Bond 30th Anniversary", "various", "EMI", 1992 ],
            CD:[ "Give Thanks", "Don Moen", "Hosanna! Music", 1991 ],
            CD:[ "Star Tracks", "Erich Kunzel", "Telarc", 1984 ],
            CD:[ "Tones", "Eric Johnson", "Warner Bros", 1986 ]
        ];



end Unitdb;
```


Program 13.14 The First UNIT Associated with this Lab (the Database File)

```
unit DBunit;
#include( "stdlib.hhf" );
#include( "sepCompDemo.hhf" );


// PrintInfo
//
//  This is a private procedure (local to this unit)
//  that displays the information about the CD passed
//  as a parameter.

procedure PrintInfo( var theCD:CD );
const
    cdt: text := "(type CDs [ebx])";

begin PrintInfo;

    push( ebx );
    mov( theCD, ebx );
    stdout.put( "Title:    ", cdt.Title, nl );
    stdout.put( "Artist:   ", cdt.Artist, nl );
    stdout.put( "Publisher:", cdt.Publisher, nl );
    stdout.put( "Year:     ", cdt.CopyrightYear, nl nl );
    pop( ebx );

end PrintInfo;


// SearchByTitle
//
//  This procedure searches through a CD database for
//  all CDs with a given title.  The title of the CD
//  to search for and the CD database array are the
//  parameters for this procedure.

procedure SearchByTitle( Title:string; var CDList:CDs );
const

    // Because CDList is passed by reference, we access
    // it indirectly using a 32-bit register.  The
    // following text equate lets us use a more meaningful
    // name for this pointer than "[ebx]".

    cdArray: text := "(type CDs [ebx])";


var
    FoundCD: boolean;   // Set to true if we find at least
                        // one CD with the given title.

begin SearchByTitle;

    push( ebx );
```



```
        push( esi );

        mov( CDList, ebx ); // Put address of array in EBX so we can
                            // use the "cdArray" symbol to reference
                            // the array passed by reference

        mov( false, FoundCD );      // Assume we didn't find a CD.
        mov( 0, esi );              // Index into cdArray.
        repeat

            // See if the parameter Title matches the current
            // title in cdArray.  Note that str.ieq does as
            // case insenstive comparison.

            if( str.ieq( Title, cdArray.Title[ esi ] )) then

                // If we found it, print the information
                // associated with this CD.

                PrintInfo( (type CD cdArray[ esi ]) );
                mov( true, FoundCD );

            endif;

            // Move on to the next element of the CDList array.
            // Note that rather than multiplying the index into
            // cdArray by @size( CD ) on each access, this code
            // simply bumps ESI by the size of a cdArray element
            // each time through the loop.  Therefore, ESI will
            // contain the index of each successive array element
            // on each pass through the loop.

            add( @size( CD ), esi );

            // We're done when ESI moves off the end of the array.
            // The constant "@size(CD)*NumCDs" is the index of
            // the first memory location beyond the end of the array.

        until( esi >= @size( CD ) * NumCDs );

        // Print an appropriate message if the lookup operation did
        // not find a CD with the specified title.

        if( !FoundCD ) then

            stdout.put( "That title does not exist in the database" nl );

        endif;
        pop( esi );
        pop( ebx );

    end SearchByTitle;




    // SearchByArtist-
    //
    //  Searches for all CDs by a given artist name.
    //  This code is nearly identical to SearchByTitle.
    //  See the comments in SearchByTitle for additional
```



```
//  comments on the operation of this code.

procedure SearchByArtist( Artist:string; var CDList:CDs );
const
    cdArray: text := "(type CDs [ebx])";

var
    FoundCD: boolean;

begin SearchByArtist;

    push( ebx );
    push( esi );

    mov( CDList, ebx ); // Put address of array in EBX so we can
                        // use the "cdArray" symbol to reference
                        // the array passed by reference

    mov( false, FoundCD );      // Assume we didn't find a CD.
    mov( 0, esi );
    repeat

        if( str.ieq( Artist, cdArray.Artist[ esi ] )) then

            PrintInfo( (type CD cdArray[ esi ]) );
            mov( true, FoundCD );

        endif;

        // Move on to the next element of the CDList array.

        add( @size( CD ), esi );

    until( esi >= @size( CD ) * NumCDs );
    if( !FoundCD ) then

        stdout.put( "That artist does not exist in the database" nl );

    endif;
    pop( esi );
    pop( ebx );

end SearchByArtist;

end DBunit;
```

Program 13.15 The Second Unit Need by this Lab Exercise (DBunit.hla)

Exercise A: Compile this program with the following HLA command.

```
hla mainPgm lnitb DBunit
```

Run the program a few times and learn how to use it. This is a small (trivial) little database program that stores information about a private CD collection (for brevity, it only has 10 CDs in the database). When the user runs the program it will ask if they want to look up a CD by artist or by title. As you can see in the sample database, "Boston" is both a title and an artist. "Imaginos" is an example of a title. Run the program and

specify these values for artist and title to familiarize yourself with this code. If possible, include a printout of the program's operation with your lab report, otherwise describe the output in your lab report.

Exercise B: Determine the dependencies between these files. Write a makefile for this code. Include the makefile with your lab report. Verify that the makefile works properly by changing each of the files and then running the make program after each modification.

Exercise C: The *GetYorN* and *GetCinSet* procedures in the main program really belong in their own unit. Move these subroutines to the unit *InputUnit* in the file inputunit.hla. Create a header file named InputUnit.hhf to contain the external declarations. Modify your makefile to accommodate these changes. Recompile the system and verify that it still works. Include the modified source code and makefile in your lab report.

13.3.12The HLA (Pseudo) Random Number Unit

In this laboratory exercise, you will test the quality of the two HLA random number generators. The following program code provides a simple test by plotting asterisks at random positions on the screen. This program works by choosing two random numbers, one between zero and 79, the other between zero and 23. Then the program uses the *console.puts* function to print a single asterisk at the (X,Y) coordinate on the screen specified by these two random numbers. After 10,000 iterations of this process the program stops and lets you observe the result. **Note**: since random number generators generate random numbers, you should not expect this program to fill the entire screen with asterisks in only 10,000 iterations.

```
program testRandom;
#include( "stdlib.hhf" );

begin testRandom;

    console.cls();
    mov( 10_000, ecx );
    repeat

        // Generate a random X-coordinate
        // using rand.range.

        rand.range( 0, 79 );
        mov( eax, ebx );            // Save the X-coordinate for now.

        // Generate a random Y-coordinate
        // using rand.urange.

        rand.urange( 0, 23 );

        // Print an asterisk at
        // the specified coordinate on the screen.

        console.puts( ax, bx, "*" );

        // Repeat this 10,000 times to get
        // a good distribution of values.

        dec( ecx );

    until( @z );

    // Position the cursor at the bottom of the
    // screen so we can observe the results.
```



```
    console.gotoxy( 24, 0 );

end testRandom;
```

Program 13.16 Screen Plot Test of the HLA Random Number Generators

Exercise A: Run this program and note how many "holes" are left unfilled on the screen. If possible, make a print-out of the screen to include with your lab manual. Compare the result of your output with another student's output for this exercise. Did they get the same output as you? Explain why or why not in your lab report.

Exercise B: Add the statement "rand.randomize()" as the first statement in this program. Recompile and run the program. Compare this output to the output from Exercise A. Compare this output to that of another student in your class for this exercise. Did they get the same output as you? Explain why or why not in your lab report.

Exercise C: Remove the call to *rand.randomize* that you inserted in Exercise B. Bump up the number of iterations to 25,000. Does the program fill the screen with asterisks? Determine to the nearest 100, the minimum number of iterations it takes to completely fill the screen with asterisks.

13.3.13File I/O in HLA

The HLA Standard Library provides a file input/output module that makes writing data to a file and reading data from a file almost as easy as working with the standard output and standard input devices. In this laboratory exercise you will discover how easy it is to read and write file data in HLA.

The following program code opens a new file, writes some data to it, closes that file, reopens it for reading, reads the data from the file, displays the data to the standard output, and finally closes the file.

```
program fileDemo;
#include( "stdlib.hhf" );

var
    fileHandle: dword;
    s:          string;

begin fileDemo;

    // Note the use of instruction composition to
    // store the return value from stralloc (which
    // is in EAX) directly into s.

    mov( stralloc( 256 ), s );

    // Open a brand new file and write some
    // data to it.  Note that the filename "x.txt"
    // must not already exist or the fileio.OpenNew
    // call will raise an exception.

    mov( fileio.openNew( "x.txt" ), fileHandle);

    fileio.put
    (
        fileHandle,
```



```
            "Data Written to a text file" nl
            "--------------------------" nl
            nl
        );

        for( mov( -5, ebx ); (type int32 ebx) <= 5; inc( ebx )) do

            fileio.put( fileHandle, "ebx=", (type int32 ebx ), nl );

        endfor;

        // Done writing the data to the file, close it.

        fileio.close( fileHandle );


        // Reopen the file, this time for reading:

        mov( fileio.open( "x.txt", fileio.r ), fileHandle);

        // Read the data from the file and write it to the
        // standard output device:

        while( !fileio.eof( fileHandle )) do

            fileio.gets( fileHandle, s );
            stdout.put( s, nl );

        endwhile;
        fileio.close( fileHandle );
        strfree( s );

    end fileDemo;
```

Program 13.17 HLA File I/O Demonstration

Exercise A: Compile and run this program. Note that this program creates a text file "x.txt" on your disk. Include a printout of this file with your laboratory report.

Exercise B: Modify this program so that it reads the filename from the user and opens the specified file. Compile and test the modified program. Include a printout of the program with your laboratory report. **Warning: when running this program, be careful not to specify the name of an existing file on your disk.** Remember, this program will delete any pre-existing data in the file you specify.

Exercise C: Split this program into two separate programs. The first program should write the data to the text file, the second program should read and display the data in the text file. Compile and run the programs to verify proper operation. Include the source listings of these two programs with your lab reports.

13.3.14 Timing Various Arithmetic Instructions

In this laboratory exercise you will run several sections of code that execute various arithmetic instructions one billion times. Using a stopwatch, you will measure the time required by each computation and compare the execution times of these various instructions.

```
program TimeArithmetic;
```



```
#include( "stdlib.hhf" );


begin TimeArithmetic;


    // The following code does one billion additions to
    // provide a baseline for further instruction timing
    // comparisons.

    stdout.put
    (
        "Multiplication timing test:" nl
        nl
        "This program will compute EAX+12 1,000,000,000 times" nl
        "using the ADD instruction." nl
        nl
        "You should time this with a stop watch" nl
        nl
        "Press ENTER to begin the additions:"
    );
    stdin.readLn();
    for( mov( 1_000_000_000, ecx ); ecx > 0; dec( ecx )) do

        mov( 123_456_789, eax );
        add( 12, eax );

    endfor;
    stdout.put( stdio.bell, "Okay, stop timing the code" nl );




    // The following code demonstrates the expense of
    // the INTMUL instruction relative to addition:

    stdout.put
    (
        "Multiplication timing test:" nl
        nl
        "This program will compute EAX*12 1,000,000,000 times" nl
        "using the INTMUL instruction." nl
        nl
        "You should time this with a stop watch" nl
        nl
        "Press ENTER to begin the multiplications:"
    );
    stdin.readLn();
    for( mov( 1_000_000_000, ecx ); ecx > 0; dec( ecx )) do

        mov( 123_456_789, eax );
        intmul( 12, eax );

    endfor;
    stdout.put( stdio.bell, "Okay, stop timing the code" nl );



    // The following code demonstrates the expense of IMUL
    // relative to addition:
```



```
stdout.put
(
    “Now this program will compute EAX*12 1,000,000,000 times” nl
    “using the IMUL instruction.” nl
    nl
    “You should time this with a stop watch” nl
    nl
    “Press ENTER to begin the multiplications:”
);
stdin.readLn();
for( mov( 1_000_000_000, ecx ); ecx > 0; dec( ecx )) do

    mov( 123_456_789, eax );
    imul( 12, eax );

endfor;
stdout.put( stdio.bell, “Okay, stop timing the code” nl );


// The following code demonstrates the cost of multiplication
// via shifts and adds:

stdout.put
(
    “Now this program will compute EAX*12 1,000,000,000 times” nl
    “using the shifts and adds instruction.” nl
    nl
    “You should time this with a stop watch” nl
    nl
    “Press ENTER to begin the multiplications:”
);
stdin.readLn();
for( mov( 1_000_000_000, ecx ); ecx > 0; dec( ecx )) do

    mov( 123_456_789, eax );
    mov( 123_456_789, ebx );
    shl( 1, ebx );
    shl( 3, eax );
    add( ebx, eax );

endfor;
stdout.put( stdio.bell, “Okay, stop timing the code” nl );



// The following code demonstrates the cost of the IDIV
// instruction relative to addition (warning: IDIV is slow!):

stdout.put
(
    “Now this program will compute EAX/12 1,000,000,000 times” nl
    “using the IDIV instruction.” nl
    nl
    “You should time this with a stop watch” nl
    nl
    “Press ENTER to begin the divisions:”
);
stdin.readLn();
for( mov( 1_000_000_000, ecx ); ecx > 0; dec( ecx )) do
```



```
        mov( 123_456_789, eax );
        cdq();
        idiv( 12, edx:eax );

    endfor;
    stdout.put( stdio.bell, "Okay, stop timing the code" nl );


end TimeArithmetic;
```

Program 13.18 Instruction Timing Code (TimeArithmetic.hla)

Exercise A: Compile and run this program. Measure the times required by each instruction sequence in the program. Describe the differences in timing in your laboratory report.

Exercise B: Simply dividing the time measured for each section doesn't accurately compute the time for each instruction because you're also measuring the time required by the FOR loop as well as the MOV instruction and any other instructions appearing in the loop (that aren't the ADD, INTMUL, IMUL, IDIV, or SHL instructions). To get a better result, you should time an empty loop and subtract the time for a loop without the specified instruction (i.e., remove the ADD instruction from the first code section in Program 13.18 above and use the resulting code as your base code).

Exercise C: Modify the program so that it computes the time required by an empty loop. Run this program and report the results in your lab report. Subtract this time from your other results to get results that are closer to reality.

13.3.15 Using the RDTSC Instruction to Time a Code Sequence

One problem with timing statements as done in the previous section is that the time you obtain is extremely inaccurate. This is because the operating system periodically interrupts the code, effectively freezing it in its tracks for a few moments, while other activities take place. Since your stopwatch is still running while the program is frozen, your timings will not accurate reflect the time it takes to execute these instructions.

On Pentium and later processors, Intel added the RDTSC (ReaD Time Stamp Counter) instruction to help software engineers better time certain instruction sequences. The Time Stamp Counter is a 64-bit internal register that the CPU increments by one on each clock cycle. Executing the RDTSC instruction copies this 64-bit value into the EDX:EAX register pair. By reading the value of the Time Stamp Counter before and after an instruction sequence, then computing the difference between these two values, you can effectively measure the time of some instructions in clock cycles.

There is one major problem with the RDTSC instruction: not all CPUs support this instruction. Intel CPUs earlier than the Pentium do not support this instruction. Furthermore, certain non-Intel CPUs do not support this instruction either. If you happen to have such a CPU, executing the RDTSC instruction will raise an "invalid instruction" exception. If this is the case for your machine, then simply skip this lab exercise.

There are two additional, but minor, problems with the RDTSC instruction. First, it produces a 64-bit result. Since we've only seen how to do 32-bit arithmetic on the 80x86, computing the difference of the two 64-bit values will prove to be difficult[7]. We can solve this problem by simply ignoring the H.O. DWORD of the 64-bit result that RDTSC returns. Once in a while we will get a nonsensical result (generally negative), but as long as we're willing to ignore such results and re-run the program to get correct results, we'll do fine.

7. Actually, we could use the FILD and FIST instructions on the FPU, but that won't prove to be necessary.

Note that the chapter on Advanced Arithmetic discusses extended precision arithmetic, for those who want to learn how to manipulate 64-bit (or larger) integer values.

The second problem with using RDTSC is that it is still possible for an interrupt to occur in the middle of the instruction sequence we're timing. This will produce inaccurate results. Fortunately, it is very rare for an interrupt to occur in the middle of a short instruction sequence, so it is unlikely that this will affect the results you get. To be sure this problem doesn't occur, all you've got to do is execute your program twice and verify that the results you obtain are similar; if one value is off by an order of magnitude, or more, then you should re-run the program a few more times.

```
program RDTSCdemo;
#include( "stdlib.hhf" );


begin RDTSCdemo;



    stdout.put
    (
        "Instruction timing test:" nl
        nl
    );

    // Note: repeat the following code twice in order
    // to ensure that all code comes out of the cache.
    // Compute the timings on the second execution of the loop.

    for( mov( 2, edi ); edi > 0; dec( edi )) do

        rdtsc();
        mov( eax, ecx );    // Save L.O. word in ECX

        // Do the instruction over 8 times to eliminate
        // certain problems inherent in RDTSC.

        mov( 123_456_789, ebx );
        add( 12, eax );
        mov( 123_456_789, ebx );
        add( 12, eax );
        mov( 123_456_789, ebx );
        add( 12, eax );
        mov( 123_456_789, ebx );
        add( 12, eax );
        mov( 123_456_789, ebx );
        add( 12, eax );
        mov( 123_456_789, ebx );
        add( 12, eax );
        mov( 123_456_789, ebx );
        add( 12, eax );
        mov( 123_456_789, ebx );
        add( 12, eax );

        rdtsc();            // Read time stamp counter after computations.

    endfor;
    sub( ecx, eax );        // Computer timer difference.
    stdout.put( "Cycles for ADD sequence: ", (type uns32 eax), nl );
```



```
// Time the INTMUL instruction:

for( mov( 2, edi ); edi > 0; dec( edi )) do

    rdtsc();
    mov( eax, ecx );    // Save L.O. word in ECX

    // Do the instruction over 8 times to eliminate
    // certain problems inherent in RDTSC.

    mov( 123_456_789, ebx );
    intmul( 12, eax );
    mov( 123_456_789, ebx );
    intmul( 12, eax );
    mov( 123_456_789, ebx );
    intmul( 12, eax );
    mov( 123_456_789, ebx );
    intmul( 12, eax );
    mov( 123_456_789, ebx );
    intmul( 12, eax );
    mov( 123_456_789, ebx );
    intmul( 12, eax );
    mov( 123_456_789, ebx );
    intmul( 12, eax );
    mov( 123_456_789, ebx );
    intmul( 12, eax );

    rdtsc();            // Read time stamp counter after computations.

endfor;
sub( ecx, eax );        // Computer timer difference.
stdout.put( "Cycles for INTMUL sequence: ", (type uns32 eax), nl );



// Time the IMUL instruction:

for( mov( 2, edi ); edi > 0; dec( edi )) do

    rdtsc();
    mov( eax, ecx );    // Save L.O. word in ECX

    // Do the instruction over 8 times to eliminate
    // certain problems inherent in RDTSC.

    mov( 123_456_789, ebx );
    imul( 12, eax );
    mov( 123_456_789, ebx );
    imul( 12, eax );
    mov( 123_456_789, ebx );
    imul( 12, eax );
    mov( 123_456_789, ebx );
    imul( 12, eax );
    mov( 123_456_789, ebx );
    imul( 12, eax );
    mov( 123_456_789, ebx );
    imul( 12, eax );
    mov( 123_456_789, ebx );
    imul( 12, eax );
    mov( 123_456_789, ebx );
    imul( 12, eax );
    mov( 123_456_789, ebx );
    imul( 12, eax );
```



```
            rdtsc();            // Read time stamp counter after computations.

        endfor;
        sub( ecx, eax );        // Computer timer difference.
        stdout.put( "Cycles for IMUL sequence: ", (type uns32 eax), nl );



        // Time the IDIV instruction:

        for( mov( 2, edi ); edi > 0; dec( edi )) do

            rdtsc();
            mov( eax, ecx );    // Save L.O. word in ECX

            // Do the instruction over 8 times to eliminate
            // certain problems inherent in RDTSC.

            mov( 123_456_789, ebx );
            cdq();
            idiv( 12, edx:eax );
            mov( 123_456_789, ebx );
            cdq();
            idiv( 12, edx:eax );
            mov( 123_456_789, ebx );
            cdq();
            idiv( 12, edx:eax );
            mov( 123_456_789, ebx );
            cdq();
            idiv( 12, edx:eax );
            mov( 123_456_789, ebx );
            cdq();
            idiv( 12, edx:eax );
            mov( 123_456_789, ebx );
            cdq();
            idiv( 12, edx:eax );
            mov( 123_456_789, ebx );
            cdq();
            idiv( 12, edx:eax );

            rdtsc();            // Read time stamp counter after computations.

        endfor;
        sub( ecx, eax );        // Computer timer difference.
        stdout.put( "Cycles for IDIV sequence: ", (type uns32 eax), nl );


    end RDTSCdemo;
```

Program 13.19 Timing Code Using RDTSC

Exercise A: Compile and run this program. Compare the times produced by this program against the previous exercise (to compute time from clock cycles, multiply by one billion and divide by the clock frequency of your system). Describe the differences in timing in your laboratory report.

Exercise B: The RDTSC instruction is quite slow compared to many other instructions. Devise a short program to test the execution time of the RDTSC instruction. Report the result of this timing in your lab report.

13.3.16Timing Floating Point Instructions

Floating point instructions tend to execute more slowly than their integer counterparts. In this laboratory exercise you will compute the running time of the FADD, FSUB, FMUL, and FDIV instructions. This code uses the RDTSC instruction. If your CPU does not support this instruction, modify the code to use the timing loops in the first lab exercise in this chapter and time your code that way.

```
program FPTiming;
#include( "stdlib.hhf" );


begin FPTiming;


    stdout.put
    (
        "Instruction timing test:" nl
        nl
    );

    // Note: repeat the following code twice in order
    // to ensure that all code comes out of the cache.
    // Compute the timings on the second execution of the loop.

    for( mov( 2, edi ); edi > 0; dec( edi )) do

        rdtsc();
        mov( eax, ecx );    // Save L.O. word in ECX

        // Do the instruction over 8 times to eliminate
        // certain problems inherent in RDTSC.

        fld( 123_456_789e0 );
        fld( 12.0 );
        fadd();
        fld( 12.0 );
        fadd();
        fld( 12.0 );
        fadd();
        fld( 12.0 );
        fadd();
        fld( 12.0 );
        fadd();
        fld( 12.0 );
        fadd();
        fld( 12.0 );
        fadd();
        fld( 12.0 );
        fadd();
        fld( 12.0 );
        fadd();
```



```
    rdtsc();            // Read time stamp counter after computations.

endfor;
sub( ecx, eax );        // Computer timer difference.
stdout.put( "Cycles for FADD sequence: ", (type uns32 eax), nl );


// Time the FSUB instruction:

for( mov( 2, edi ); edi > 0; dec( edi )) do

    rdtsc();
    mov( eax, ecx );    // Save L.O. word in ECX

    // Do the instruction over 8 times to eliminate
    // certain problems inherent in RDTSC.

    fld( 123_456_789e0 );
    fld( 12.0 );
    fsub();
    fld( 12.0 );
    fsub();
    fld( 12.0 );
    fsub();
    fld( 12.0 );
    fsub();
    fld( 12.0 );
    fsub();
    fld( 12.0 );
    fsub();
    fld( 12.0 );
    fsub();
    fld( 12.0 );
    fsub();


    rdtsc();            // Read time stamp counter after computations.

endfor;
sub( ecx, eax );        // Computer timer difference.
stdout.put( "Cycles for FSUB sequence: ", (type uns32 eax), nl );




// Time the FMUL instruction:

for( mov( 2, edi ); edi > 0; dec( edi )) do

    rdtsc();
    mov( eax, ecx );    // Save L.O. word in ECX

    // Do the instruction over 8 times to eliminate
    // certain problems inherent in RDTSC.

    fld( 123_456_789e0 );
    fld( 12.0 );
    fmul();
    fld( 12.0 );
```



```
            fmul();
            fld( 12.0 );
            fmul();
            fld( 12.0 );
            fmul();
            fld( 12.0 );
            fmul();
            fld( 12.0 );
            fmul();
            fld( 12.0 );
            fmul();
            fld( 12.0 );
            fmul();


            rdtsc();            // Read time stamp counter after computations.

        endfor;
        sub( ecx, eax );        // Computer timer difference.
        stdout.put( "Cycles for FMUL sequence: ", (type uns32 eax), nl );



        // Time the FDIV instruction:

        for( mov( 2, edi ); edi > 0; dec( edi )) do

            rdtsc();
            mov( eax, ecx );    // Save L.O. word in ECX

            // Do the instruction over 8 times to eliminate
            // certain problems inherent in RDTSC.

            fld( 123_456_789e0 );
            fld( 12.0 );
            fdiv();
            fld( 12.0 );
            fdiv();
            fld( 12.0 );
            fdiv();
            fld( 12.0 );
            fdiv();
            fld( 12.0 );
            fdiv();
            fld( 12.0 );
            fdiv();
            fld( 12.0 );
            fdiv();
            fld( 12.0 );
            fdiv();


            rdtsc();            // Read time stamp counter after computations.

        endfor;
        sub( ecx, eax );        // Computer timer difference.
        stdout.put( "Cycles for FDIV sequence: ", (type uns32 eax), nl );


    end FPTiming;
```


Program 13.20 Timing FADD, FSUB, FMUL, and FDIV

Exercise A: Compile and run this program. Compare the cycles counts (or times) produced by this program against the previous exercise. Describe the differences in timing in your laboratory report.

Exercise B: Floating point timings are very data dependent. Modify the program by changing the constants around. Retry the computations with constants that are close to one another and more widely separated than in this example. (Hint: create a CONST definition for the two constants this program uses in order to facilitate the changes; can you see the benefit of using CONST here?)

Exercise C: Modify the program so that the floating point control register specifies the 64-bit rounding mode (vs. 53 bits or 24 bits). Re-run the program and check the timings. Then set the round to 53 bits and 24 bits. Compare the results.

Exercise D: Modify the prorgram to factor out the time required by the RDTSC instruction. Report your results in your lab report.

13.3.17Table Lookup Exercise

The following laboratory exercise computes a sequence of sine values using table lookup and using the FPU FSIN instruction so you can see the difference in timing between the two techniques. Like the previous two exercises, this code uses the RDTSC instruction to time the sequences. If your CPU does not support the RDTSC instruction, rewrite the code to use surrounding loops as used in the first laboratory exercise of this chapter.

```
program TableLookup;
#include( "stdlib.hhf" );

static
    angle:      uns32;
    theSIN:     uns32;

readonly
    sines: int32[360] :=
            [

                  0,   17,   35,   52,   70,   87,  105,  122,
                139,  156,  174,  191,  208,  225,  242,  259,
                276,  292,  309,  326,  342,  358,  375,  391,
                407,  423,  438,  454,  469,  485,  500,  515,
                530,  545,  559,  574,  588,  602,  616,  629,
                643,  656,  669,  682,  695,  707,  719,  731,
                743,  755,  766,  777,  788,  799,  809,  819,
                829,  839,  848,  857,  866,  875,  883,  891,
                899,  906,  914,  921,  927,  934,  940,  946,
                951,  956,  961,  966,  970,  974,  978,  982,
                985,  988,  990,  993,  995,  996,  998,  999,
                999, 1000, 1000, 1000,  999,  999,  998,  996,
                995,  993,  990,  988,  985,  982,  978,  974,
                970,  966,  961,  956,  951,  946,  940,  934,
                927,  921,  914,  906,  899,  891,  883,  875,
                866,  857,  848,  839,  829,  819,  809,  799,
                788,  777,  766,  755,  743,  731,  719,  707,
                695,  682,  669,  656,  643,  629,  616,  602,
```



```
                    588,  574,  559,  545,  530,  515,  500,  485,
                    469,  454,  438,  423,  407,  391,  375,  358,
                    342,  326,  309,  292,  276,  259,  242,  225,
                    208,  191,  174,  156,  139,  122,  105,   87,
                     70,   52,   35,   17,    0,  -17,  -35,  -52,
                    -70,  -87, -105, -122, -139, -156, -174, -191,
                   -208, -225, -242, -259, -276, -292, -309, -326,
                   -342, -358, -375, -391, -407, -423, -438, -454,
                   -469, -485, -500, -515, -530, -545, -559, -574,
                   -588, -602, -616, -629, -643, -656, -669, -682,
                   -695, -707, -719, -731, -743, -755, -766, -777,
                   -788, -799, -809, -819, -829, -839, -848, -857,
                   -866, -875, -883, -891, -899, -906, -914, -921,
                   -927, -934, -940, -946, -951, -956, -961, -966,
                   -970, -974, -978, -982, -985, -988, -990, -993,
                   -995, -996, -998, -999, -999,-1000,-1000,-1000,
                   -999, -999, -998, -996, -995, -993, -990, -988,
                   -985, -982, -978, -974, -970, -966, -961, -956,
                   -951, -946, -940, -934, -927, -921, -914, -906,
                   -899, -891, -883, -875, -866, -857, -848, -839,
                   -829, -819, -809, -799, -788, -777, -766, -755,
                   -743, -731, -719, -707, -695, -682, -669, -656,
                   -643, -629, -616, -602, -588, -574, -559, -545,
                   -530, -515, -500, -485, -469, -454, -438, -423,
                   -407, -391, -375, -358, -342, -326, -309, -292,
                   -276, -259, -242, -225, -208, -191, -174, -156,
                   -139, -122, -105,  -87,  -70,  -52,  -35,  -17
               ];

    begin TableLookup;


        // The following touches all the items in
        // the sines array so that it is all in the cache.

        for( mov( 0, angle ); angle < 360; inc( angle )) do

            mov( angle, ebx );
            mov( sines[ ebx*4 ], eax );
            mov( eax, theSIN );

        endfor;

        // Compute the time required to look up the sines of the
        // angles 0..359:

        rdtsc();
        mov( eax, ecx );
        for( mov( 0, angle ); angle < 360; inc( angle )) do

            mov( angle, ebx );
            mov( sines[ ebx*4 ], eax );
            mov( eax, theSIN );

        endfor;
        rdtsc();
        sub( ecx, eax );
        stdout.put
        (
            “Time required to compute 360 sines via lookup:”,
            (type uns32 eax),
```



```
            nl
        );

        // Compute the time required to compute 360 sines using FSIN:


        rdtsc();
        mov( eax, ecx );
        for( mov( 0, angle ); angle < 360; inc( angle )) do

            fild( angle );
            fldpi();
            fmul();
            fld( 2.0 );
            fmul();
            fld( 180.0 );
            fdiv();
            fsin();
            fld( 1000.0 );
            fmul();
            fistp( theSIN );

        endfor;
        rdtsc();
        sub( ecx, eax );
        stdout.put
        (
            "Time required to compute 360 sines via FSIN:",
            (type uns32 eax),
            nl
        );


    end TableLookup;
```

Program 13.21 FSIN vs. Table Lookup

Exercise A: Compile and run this program. Compare the cycles counts between the two methods for computing SIN. Describe the differences in timing and comment on these differences in your laboratory report.

Exercise B: In one respect, this program cheats. The first look in the program "touches" all the memory locations in the *sines* table so that they are all loaded into the cache before the code actually times the execution of the lookup table code. Remove this initial loop and rerun the program. Comment on the difference in timing. Comment on the effectiveness of the cache. Is it worthwhile to use a lookup table if you rarely access that table (and it's data is not in cache)? Justify your answer.

