Problems

Part I: ARM

3.1 Assume the following register and memory contents in an ARM computer:

- Register R0 contains 1000.
- Register R1 contains 2000.
- Register R2 contains 1016.
- Register R6 contains 20.
- Register R7 contains 30.

The numbers 1, 2, 3, 4, 5, and 6, are stored in successive word locations starting at memory address 1000. What is the effect of executing each of the following three short instruction blocks, starting each time from the given initial values?

(a) LDR R8,[R0]
    LDR R9,[R0,#4]
    ADD R10,R8,R9
PROBLEMS 189

(b) STR R6,[R1,#−4]!
STR R7,[R1,#−4]!
LDR R8,[R1],#4
LDR R9,[R1],#4
SUB R10,R8,R9
(c) LDMIA R2!,{R4,R5}
ADD R4,R4,R5

3.2 Which of the following ARM instructions would cause the assembler to issue a syntax error message? Why?

(a) ADD R2,R2,R2
(b) SUB R0,R1,[R2,#4]
(c) MOV R0,#2_1010101
(d) MOV R0,#257
(e) ADD R0,R1,R11,LSL #8

3.3 When a byte is loaded from memory into an ARM processor register using the Load instruction, the high-order 24 bits are cleared to 0s. (See Section 3.1.2.) If the loaded byte represents an 8-bit signed integer in 2’s-complement representation, it must be sign-extended to 32 bits in the register before it can be used in arithmetic operations. Assuming such a byte has been loaded into register R0, write a short routine to sign-extend it to the 32-bit register length. (Hint: Use MOV instructions to move the contents of R0 back into R0 after appropriate shifts from the possibilities LSL, LSR, and ASR, as described in Section 2.10.2 and shown in Figure 2.30.)

3.4 Write an ARM program to reverse the order of bits in register R2. For example, if the starting pattern in R2 is 1110 ... 0100, the result left in R2 should be 0010 ... 0111. (Hint: Use shift and rotate operations.)

3.5 A program trace is a listing of the contents of certain registers and memory locations at different times during the execution of a program. List the contents of registers R0, R1, and R2 after each of the first three executions of the BGT instruction in the program in Figure 3.7. Present the results in a table that has the three registers as column headers. Use three rows to list the contents of the registers after each execution of the BGT instruction. The program data are as given in Figure 3.8.

3.6 Write an ARM program that compares the corresponding bytes of two lists of bytes and places the larger byte in a third list. The two lists start at byte locations X and Y, and the larger-byte list starts at LARGER. The length of the lists is stored in memory location N.

3.7 An ARM program is required for the following character manipulation task: A string of n characters is stored in the memory in consecutive byte locations, beginning at location STRING. Another shorter string of m characters is stored in consecutive byte locations, beginning at location SUBSTRING. The program must search the string that begins at
STRING to determine whether or not it contains a contiguous substring identical to the
string that begins at SUBSTRING. The length parameters \( n \) and \( m \), where \( n > m \), are
stored in memory locations \( N \) and \( M \), respectively. If a matching substring is found,
the address of its first byte is to be stored in register \( R0 \); otherwise, the contents of \( R0 \)
are to be cleared to 0. The program does not need to determine multiple occurrences of
the substring. Only the address of the first matching substring is required.

3.8 Write an ARM program that generates the first \( n \) numbers of the Fibonacci series. In
this series, the first two numbers are 0 and 1, and each subsequent number is generated
by adding the preceding two numbers. For example, for \( n = 8 \), the series is

\[
0, 1, 1, 2, 3, 5, 8, 13
\]

Your program should store the numbers in successive memory word locations starting
at MEMLOC. Assume that the value \( n \) is stored in location \( N \).

3.9 Write an ARM program to convert a word of text from lowercase to uppercase. The
word consists of ASCII characters stored in successive byte locations in the memory,
starting at location WORD and ending with a space character. (See Appendix E for the
ASCII code.)

3.10 The list of student marks shown in Figure 2.14 is changed to contain \( j \) test scores for
each student. Assume that there are \( n \) students. Write an ARM program for computing
the sums of the scores on each test and store these sums in the memory word locations
at addresses \( \text{SUM} \), \( \text{SUM} + 4 \), \( \text{SUM} + 8 \), \ldots. The number of tests, \( j \), is larger than the
number of registers in the processor, so the type of program shown in Figure 2.15 for
the 3-test case cannot be used. Use two nested loops, as suggested in Section 2.5.3. The
inner loop should accumulate the sum for a particular test, and the outer loop should
run over the number of tests, \( j \). Assume that \( j \) is stored in memory location \( J \), placed
ahead of location \( N \).

3.11 Consider an array of numbers \( A(i, j) \), where \( i = 0 \) through \( n - 1 \) is the row index and
\( j = 0 \) through \( m - 1 \) is the column index. The array is stored in the memory of an ARM
computer one row after another, with elements of each row occupying \( m \) successive
word locations. Write an ARM subroutine for adding column \( x \) to column \( y \), element
by element, leaving the sum elements in column \( y \). The indices \( x \) and \( y \) are passed to the
subroutine in registers \( R1 \) and \( R2 \). The parameters \( n \) and \( m \) are passed to the subroutine
in registers \( R3 \) and \( R4 \), and the address of element \( A(0,0) \) is passed in register \( R0 \). Any
of the addressing modes in Table 3.1 can be used.

3.12 Write an ARM program that reads \( n \) characters from a keyboard and echoes them back
to a display after pushing them onto a user stack as they are read. Use register \( R6 \) as
the stack pointer. The count value \( n \) is contained in memory word location \( N \).

3.13 Assume that the average time taken to fetch and execute an instruction in the
program in Figure 3.9 is 20 nanoseconds. If keyboard characters are entered at the
rate of 10 per second, approximately how many times is the BEQ READ in-
sertion executed per character entered? Assume that the time taken to display each
caracter is much less than the time between the entry of successive characters at the
keyboard.
3.14 In the ARM program in Figure 3.9, “in-line” code is used to read a line of characters and display them. Rewrite this program in the form of a main program that calls a subroutine named GETCHAR to read a single character and calls another subroutine named PUTCHAR to display a single character. The address INSTATUS is passed to GETCHAR in register R1, and the main program expects to get the character passed back in register R3. The address OUTSTATUS and the character to be displayed are passed to PUTCHAR in registers R2 and R3, respectively. Any other registers used by either subroutine must be saved and restored by the subroutine using a stack whose pointer is register R13. Storing the characters in memory and checking for the end-of-line character CR is to be done in the main program.

3.15 Repeat Problem 3.14 using the stack to pass parameters.

3.16 Write an ARM program to accept three decimal digits from a keyboard. Each digit is represented in the ASCII code (see Appendix E). Assume that these three digits represent a decimal integer in the range 0 to 999 and convert the integer into a binary number representation. The high-order digit is received first. To aid in this conversion, two tables of words are stored in the memory. Each table has 10 entries. The first table, starting at word location TENS, contains the binary representations for the decimal values 0, 10, 20, ..., 90. The second table starts at word location HUNDREDS and contains the decimal values 0, 100, 200, ..., 900 in binary representation.

3.17 The decimal-to-binary conversion program of Problem 3.16 is to be implemented using two nested subroutines. The main program that calls the first subroutine passes two parameters by pushing them onto the stack whose pointer register is R13. The first parameter is the address of a 3-byte memory buffer area for storing the input decimal-digit characters. The second parameter is the address of the location where the converted binary value is to be stored. The first subroutine reads in the three characters from the keyboard and then calls the second subroutine to perform the conversion. The necessary parameters are passed to this subroutine via the processor registers. Both subroutines must save the contents of any registers that they use on the stack.

(a) Write the two subroutines for the ARM processor.
(b) Give the contents of the stack immediately after the execution of the instruction that calls the second subroutine.

3.18 Consider the queue structure described in Problem 2.18. Write ARM APPEND and REMOVE routines that transfer data between a processor register and the queue. Be careful to inspect and update the state of the queue and the pointers each time an operation is attempted and performed.

3.19 Using the format for presenting results that is described in Problem 3.5, give a program trace for the byte-sorting program in Figure 3.15b. Show the contents of registers R0, R2, and R3, and list byte locations LIST, LIST + 1, ..., LIST + 4 for a 5-byte list after each execution of the last instruction in the program. Assume that LIST = 1000 and that the initial list of byte values is 120, 13, 106, 45, and 67, where [LIST] = 120.
3.20 Rewrite the byte-sorting program in Figure 3.15b as a subroutine that sorts a list of 32-bit positive integers. The calling program should pass the list address to the subroutine. The first 32-bit quantity at that location is the number of entries in the list, followed by the numbers to be sorted.

3.21 Consider the byte-sorting program of Figure 3.15b. During each pass through a sublist, LIST(j) through LIST(0), list entries are swapped whenever LIST(k) > LIST(j). An alternative strategy is to keep track of the address of the largest value in the sublist and to perform, at most, one swap at the end of the sublist search. Rewrite the program using this approach. What is the advantage of this approach?

3.22 Assume that the list of student test scores shown in Figure 2.14 is stored in the memory as a linked list as shown in Figure 2.36. Write an ARM program that accomplishes the same thing as the program in Figure 2.15. The head record is stored at memory location 1000.

3.23 The linked-list insertion subroutine in Figure 3.16 does not check if the ID of the new record matches that of a record already in the list. What happens in the execution of the subroutine if this is the case? Modify the subroutine to return the address of the matching record in register R10 if this occurs or to return a zero if the insertion is successful.

3.24 The linked-list deletion subroutine in Figure 3.17 assumes that a record with the ID contained in register RIDNUM is in the list. What happens in the execution of the subroutine if there is no record with this ID? Modify the subroutine to return a zero in RIDNUM if deletion is successful, or leave RIDNUM unchanged if the record is not in the list.

PART II: 68000

3.25 Consider the following state of the 68000 processor:

- Register D0 contains $1000.
- Register A0 contains $2000.
- Register A1 contains $1000.
- Memory location $1000 contains the long word $2000.
- Memory location $2000 contains the long word $3000.

What is the effect of executing each of the following three instructions, starting each time from this initial state? How many bytes does each instruction occupy? How many memory accesses does the fetching and execution of each instruction require?

(a) ADD.L D0,(A0)
(b) ADD.L (A1,D0),D0
(c) ADD.L #$2000,(A0)
3.26 Find the syntax errors in the following 68000 instructions:

(a) ADDX $\text{-(A2),D3}$
(b) LSR.L $\#9,D2$
(c) MOVE.B $520(A2,D2)$
(d) SUBA.L $12(A2,PC),A0$
(e) CMP.B $\text{#254,$12(A2,D1.B)}$

3.27 A program trace is a listing of the contents of certain registers and memory locations at different times during the execution of a program. List the contents of registers D0, D1, and A2 and memory locations N, NUM1, and SUM after each of five executions of the ADD.W instruction and after execution of the last MOVE.L instruction in the program in Figure 3.25. Present the results in a table that has the registers and memory locations as column headers. Use six rows to list the contents of the registers and memory locations after execution of each of the instructions. Assume the following initial values: [SUM] = 0, [N] = 5, NUM1 = 2400, and the five numbers are 83, 45, 156, −250, and 100.

3.28 Consider the following 68000 program:

```assembly
MOVEA.L MEM1,A0
MOVEA.L MEM2,A2
ADDA.L A0,A1
MOVEA.L A0,A2
MOVE.B (A0)+,D0
LOOP CMP.B (A0)+,D0
BLE NXT
LEA −1(A0),A2
MOVE.B (A2),D0
NXT CMPA.L A0,A1
BGT LOOP
MOVE.L A2,DESIRED
```

(a) What does this program do?
(b) How many 16-bit words are needed to store this program in the memory?
(c) Give an expression for the number of memory accesses required. The expression should be of the form $T = a + bn + cm$, where $n$ is the number of times the loop is executed, $m$ is the number of times the branch to NXT is not taken, and $a$, $b$, and $c$ are constants.

3.29 Consider the two 68000 programs given in Figure P3.1.

(a) Do these programs leave the same value in location RSLT?
(b) What task(s) do they accomplish?
Program 1               Program 2
CLR.L                MOVE.W  #$FFFF,D0
MOVEA.L  #LIST,A0    MOVEA.L  #LIST,A0
LOOP                  LOOP
MOVE.W  (A0)+,D1     LSL.W  (A0)+
BGE LOOP             BCC LOOP
ADDQ.L  #1,D0        LSL.W  #1,D0
CMPF D0              BCS LOOP
BLT LOOP             -2(A0),RSLT
MOVE.W  −2(A0),RSLT   MOVE.W  −2(A0),RSLT

Figure P3.1 Two 68000 programs for Problem 3.29.

(c) How many bytes of memory are needed to store each program?
(d) Which program requires the larger number of memory accesses?
(e) What are the advantages and disadvantages of these programs?

3.30 Write a 68000 program that compares the corresponding bytes of two lists of bytes and places the larger byte in a third list. The two lists start at byte locations X and Y, and the larger-byte list starts at LARGER. The length of the lists is stored in memory location N.

3.31 A 68000 program is required for the following character manipulation task: A string of n characters is stored in the memory in consecutive byte locations, beginning at location STRING. Another, shorter string of m characters is stored in consecutive byte locations, beginning at location SUBSTRING. The program must search the string that begins at STRING to determine whether or not it contains a contiguous substring identical to the string that begins at SUBSTRING. The length parameters n and m, where n > m, are stored in memory locations N and M, respectively. If a matching substring is found, the address of its first byte is to be stored in register D0; otherwise, the contents of D0 are to be cleared to 0. The program does not need to determine multiple occurrences of the substring. Only the address of the first matching substring is required.

3.32 Write a 68000 program that generates the first n numbers of the Fibonacci series. In this series, the first two numbers are 0 and 1, and each subsequent number is generated by adding the preceding two numbers. For example, for n = 8, the series is

0, 1, 1, 2, 3, 5, 8, 13

Your program should store the numbers in memory byte locations starting at MEMLOC. Assume that the value n is stored in location N. What is the largest n that your program can handle?

3.33 Write a 68000 program to convert a word of text from lowercase to uppercase. The word consists of ASCII characters stored in successive byte locations in the memory, starting at location WORD and ending with a space character. (See Appendix E for the ASCII code.)
3.34 The list of student marks shown in Figure 2.14 is changed to contain \( j \) test scores for each student. Each entry in the list is a 16-bit word, so the increments on LIST are by 2. Assume that there are \( n \) students. Write a 68000 program for computing the sums of the scores on each test and store these sums in the memory word locations at addresses SUM, SUM + 2, SUM + 4, \ldots . The number of tests, \( j \), is larger than the number of registers in the processor, so the type of program shown in Figure 2.15 for the 3-test case cannot be used. Use two nested loops, as suggested in Section 2.5.3. The inner loop should accumulate the sum for a particular test, and the outer loop should run over the number of tests, \( j \). Assume that \( j \) is stored in memory location J, placed ahead of N.

3.35 Write a 68000 program that reads \( n \) characters from a keyboard and echoes them back to a display after pushing them onto a user stack as they are read. Use register A0 as the stack pointer. The count value \( n \) is stored in memory word location N.

3.36 Assume that the average time taken to fetch and execute an instruction in the program in Figure 3.27 is 20 nanoseconds. If keyboard characters are entered at the rate of 10 per second, approximately how many times is the BEQ READ instruction executed per character entered? Assume that the time taken to display each character is much less than the time between the entry of successive characters at the keyboard.

3.37 In the 68000 program in Figure 3.27, “in-line” code is used to read a line of characters and display them. Rewrite this program in the form of a main program that calls a subroutine named GETCHAR to read a single character and calls another subroutine named PUTCHAR to display a single character. The addresses INSTATUS and DATAIN are passed to GETCHAR in registers A0 and A1; and the main program expects to get the character passed back in register D0. The addresses OUTSTATUS and DATAOUT and the character to be displayed are passed to PUTCHAR in registers A2, A3, and D0, respectively. Any other registers used by either subroutine must be saved and restored by the subroutine using the processor stack whose pointer is register A7. Storing the characters in memory and checking for the end-of-line character CR is to be done in the main program.

3.38 Repeat problem 3.37 using the stack to pass parameters.

3.39 Consider the queue structure described in Problem 2.18. Write 68000 APPEND and REMOVE routines that transfer data between a processor register and the queue. Be careful to inspect and update the state of the queue and the pointers each time an operation is attempted and performed.

3.40 Write a 68000 program to accept three decimal digits from a keyboard. Each digit is represented in the ASCII code (see Appendix E). Assume that these three digits represent a decimal integer in the range 0 to 999 and convert the integer into a binary number representation. The high-order digit is received first. To aid in this conversion, two tables of words are stored in the memory. Each table has 10 entries. The first table, starting at word location TENS, contains the binary representations for the decimal values 0, 10, 20, \ldots , 90. The second table starts at word location HUNDREDS and contains the decimal values 0, 100, 200, \ldots , 900 in binary representation.
3.41 The decimal-to-binary conversion program of Problem 3.40 is to be implemented as two nested subroutines. The main program that calls the first subroutine passes two parameters by pushing them onto the processor stack. The first parameter is the address of a 3-byte memory buffer area for storing the input decimal-digit characters. The second parameter is the address of the location for the converted binary value. The first subroutine reads in the three characters from the keyboard and then calls the second subroutine to perform the conversion. The necessary parameters are passed to this subroutine via the processor registers. Both subroutines must save the contents of any registers that they use on the processor stack.

(a) Write the two subroutines for the 68000 processor.
(b) Give the contents of the processor stack immediately after the execution of the instruction that calls the second subroutine.

3.42 Consider an array of 16-bit numbers $A(i,j)$, where $i = 0$ through $n - 1$ is the row index and $j = 0$ through $m - 1$ is the column index. The array is stored in the memory of a 68000 computer one row after another, with elements of each row occupying $m$ successive word locations. Write a 68000 subroutine for adding column $x$ to column $y$, element by element, leaving the sum elements in column $y$. The indices $x$ and $y$ are passed to the subroutine in registers D1 and D2. The parameters $n$ and $m$ are passed to the subroutine in registers D3 and D4, and the address of element $A(0,0)$ is passed in register A0. Any of the addressing modes in Table 3.2 can be used.

3.43 Write a 68000 program to reverse the order of bits in register D2. For example, if the starting pattern in D2 is $1110 \ldots 0100$, the result left in D2 should be $0010 \ldots 0111$. (Hint: Use shift and rotate operations.)

3.44 How many bytes of memory are needed to store the program in Figure 3.32? How many memory accesses take place during execution of this program?

3.45 Using the format for presenting results that is described in Problem 3.27, give a program trace for the byte-sorting program in Figure 3.34b. Show the contents of registers D1, D2, and D3, and the list byte locations LIST, LIST + 1, \ldots, LIST + 4 for a 5-byte list after each execution of the last instruction in the program. Assume that LIST = 1000, and that the initial list of byte values is 120, 13, 106, 45, and 67, where [LIST] = 120.

3.46 Rewrite the byte-sorting program in Figure 3.34b as a subroutine that sorts a list of 16-bit positive integers. The calling program should pass the list address to the subroutine. The first 16-bit quantity at that location is the number of entries in the list, followed by the numbers to be sorted.

3.47 Consider the byte-sorting program of Figure 3.34b. During each pass through a sublist, $LIST(j)$ through $LIST(0)$, list entries are swapped whenever $LIST(k) > LIST(j)$. An alternative strategy is to keep track of the address of the largest value in the sublist and to perform, at most, one swap at the end of the sublist search. Rewrite the program using this approach. What is the advantage of this approach?

3.48 Assume that the list of student test scores shown in Figure 2.14 is stored in the memory as a linked list as shown in Figure 2.36. Write a 68000 program that accomplishes the
same thing as the program in Figure 2.15. The head record is stored at memory location 1000. Assume that all list entries are long words.

3.49 The linked-list insertion subroutine in Figure 3.35 does not check if the ID of the new record matches that of a record already in the list. What happens in the execution of the subroutine if this is the case? Modify the subroutine to return the address of the matching record in register A6 if this occurs, or return a zero if the insertion is successful.

3.50 The linked-list deletion subroutine in Figure 3.36 assumes that a record with the ID contained in register RIDNUM is in the list. What happens in the execution of the subroutine if there is no record with this ID? Modify the subroutine to return a zero in RIDNUM if deletion is successful, or leave RIDNUM unchanged if the record is not in the list.

PART III: Intel IA-32

3.51 Assume the following register and memory contents in an IA-32 computer:

- Register EBX contains 1000.
- Register ESI contains 2.
- The numbers 1, 2, 3, 4, 5, and 6, are stored in successive doubleword locations starting at memory address 1000.
- The address label LOC represents address 1008.

What is the effect of executing each of the following three short instruction blocks, starting each time from the given initial values?

(a) MOV EAX, 10
    ADD EAX, [EBX + ESI*4 + 8]
(b) PUSH 20
    PUSH 30
    POP EAX
    POP EBX
    SUB EAX, EBX
(c) LEA EAX, LOC
    MOV EBX, [EAX]

3.52 Which of the following IA-32 instructions would cause the assembler to issue a syntax error message? Why?

(a) ADD EAX, EAX
(b) ADD [EAX], [EBX + 4]
(c) SUB EAX, [EBX + ESI*4 + 20]
(d) SUB EAX, [EBX + ESI*10]
(e) ADD EAX, −31728542
(f) MOV 20, EAX
(g) MOV EAX, [EBP + ESP*4]
3.53 A program trace is a listing of the contents of certain registers and memory locations at different times during the execution of a program. List the contents of registers EAX, EBX, and ECX after each of the first three executions of the LOOP instruction in the program in Figure 3.40b. Present the results in a table that has the three registers as column headers. Use three rows to list the contents of the registers after each execution of the LOOP instruction. The program data is as given in Figure 3.42.

3.54 Write an IA-32 program that compares the corresponding bytes of two lists of bytes and places the larger byte in a third list. The two lists start at byte locations X and Y, and the larger-byte list starts at LARGER. The length of the lists is stored in memory location N.

3.55 An IA-32 program is required for the following character manipulation task: A string of n characters is stored in the memory in consecutive byte locations, beginning at location STRING. Another, shorter string of m characters is stored in consecutive byte locations, beginning at location SUBSTRING. The program must search the string that begins at STRING to determine whether or not it contains a contiguous substring identical to the string that begins at SUBSTRING. The length parameters n and m, where n > m, are stored in memory locations N and M, respectively. If a matching substring is found, the address of its first byte is to be stored in register EAX; otherwise, the contents of EAX are to be cleared to 0. The program does not need to determine multiple occurrences of the substring. Only the address of the first matching substring is required.

3.56 Write an IA-32 program that generates the first n numbers of the Fibonacci series. In this series, the first two numbers are 0 and 1, and each subsequent number is generated by adding the preceding two numbers. For example, for n = 8, the series is

\[0, 1, 1, 2, 3, 5, 8, 13\]

Your program should store the numbers in successive memory doubleword locations starting at MEMLOC. Assume that the value n is stored in location N.

3.57 Write an IA-32 program to convert a word of text from lowercase to uppercase. The word consists of ASCII characters stored in successive byte locations in the memory, starting at location WORD and ending with a space character. (See Appendix E for the ASCII code.)

3.58 The list of student marks shown in Figure 2.14 is changed to contain j test scores for each student. Assume that there are n students. Write an IA-32 program for computing the sums of the scores on each test and store these sums in the memory doubleword locations at addresses SUM, SUM + 4, SUM + 8, . . . . The number of tests, j, is larger than the number of registers in the processor, so the type of program shown in Figure 2.15 for the 3-test case cannot be used. Use two nested loops, as suggested in Section 2.5.3. The inner loop should accumulate the sum for a particular test, and the outer loop should run over the number of tests, j. Assume that j is stored in memory location J, placed ahead of location N.
3.59 Write an IA-32 program to reverse the order of bits in register EAX. For example, if the starting pattern in EAX is 1110 ... 0100, the result left in EAX should be 0010 ... 0111. (Hint: Use shift and rotate operations.)

3.60 Consider the queue structure described in Problem 2.18. Write IA-32 APPEND and REMOVE routines that transfer data between a processor register and the queue. Be careful to inspect and update the state of the queue and the pointers each time an operation is attempted and performed.

3.61 Write an IA-32 program that reads $n$ characters from a keyboard and echoes them back to a display after pushing them onto a user stack as they are read. Use register EBX as the stack pointer. The count value $n$ is stored in memory doubleword location N.

3.62 Assume that the average time taken to fetch and execute an instruction in the program in Figure 3.44 is 10 nanoseconds. If keyboard characters are entered at the rate of 10 per second, approximately how many times is the JNC READ instruction executed per character entered? Assume that the time taken to display each character is much less than the time between the entry of successive characters at the keyboard.

3.63 In the IA-32 program in Figure 3.44, “in-line” code is used to read a line of characters and display them. Rewrite this program in the form of a main program that calls a subroutine named GETCHAR to read a single character and calls another subroutine named PUTCHAR to display a single character. The addresses INSTATUS and DATAIN are passed to GETCHAR in registers EBX and EDX; and the main program expects to get the character passed back in register AL. The addresses OUTSTATUS and DATAOUT and the character to be displayed are passed to PUTCHAR in registers ESI, EDI, and AL, respectively. Any other registers used by either subroutine must be saved and restored by the subroutine using the processor stack, whose pointer is register ESP. Storing the characters in memory and checking for the end-of-line character CR is to be done in the main program.

3.64 Repeat problem 3.63, passing parameters on the processor stack.

3.65 Write an IA-32 program to accept three decimal digits from a keyboard. Each digit is represented in the ASCII code (see Appendix E). Assume that these three digits represent a decimal integer in the range 0 to 999 and convert the integer into a binary number representation. The high-order digit is received first. To aid in this conversion, two tables of doublewords are stored in the memory. Each table has 10 entries. The first table, starting at doubleword location TENS, contains the binary representations for the decimal values 0, 10, 20, ..., 90. The second table starts at doubleword location HUNDREDS and contains the decimal values 0, 100, 200, ..., 900 in binary representation.

3.66 The decimal-to-binary conversion program of Problem 3.65 is to be implemented using two nested subroutines. The main program that calls the first subroutine passes two parameters by pushing them onto the processor stack. The first parameter is the address of a 3-byte memory buffer area for storing the input decimal-digit characters. The second parameter is the address of the location for the converted binary value. The first subroutine reads in the three characters from the keyboard and then calls the
second subroutine to perform the conversion. The necessary parameters are passed to this subroutine via the processor registers. Both subroutines must save the contents of any registers that they use on the processor stack.

(a) Write the two subroutines for the IA-32 processor.

(b) Give the contents of the processor stack immediately after the execution of the instruction that calls the second subroutine.

3.67 Consider an array of numbers $A(i,j)$, where $i = 0$ through $n - 1$ is the row index and $j = 0$ through $m - 1$ is the column index. The array is stored in the memory of an IA-32 computer one row after another, with elements of each row occupying $m$ successive doubleword locations. Write an IA-32 subroutine for adding column $x$ to column $y$, element by element, leaving the sum elements in column $y$. The indices $x$ and $y$ are passed to the subroutine in registers ESI and EDI. The parameters $n$ and $m$ are passed to the subroutine in registers EAX and EBX, and the address of element $A(0,0)$ is passed in register EDX. Any of the addressing modes in Table 3.3 can be used.

3.68 Using the format for presenting results that is described in Problem 3.53, give a program trace for the byte-sorting program in Figure 3.50b. Show the contents of registers EDI, ECX, and DL, and list byte locations LIST, LIST + 1, . . . , LIST + 4 for a 5-byte list after each execution of the last instruction in the program. Assume that LIST = 1000 and that the initial list of byte values is 120, 13, 106, 45, and 67, where $[\text{LIST}] = 120$.

3.69 Rewrite the byte-sorting program in Figure 3.50b as a subroutine that sorts a list of 32-bit positive integers. The calling program should pass the list address to the subroutine. The first 32-bit quantity at that location is the number of entries in the list, followed by the numbers to be sorted.

3.70 Consider the byte-sorting program of Figure 3.50b. During each pass through a sublist, LIST($j$) through LIST(0), list entries are swapped whenever LIST($k$) > LIST($j$). An alternative strategy is to keep track of the address of the largest value in the sublist, and to perform, at most, one swap at the end of the sublist search. Rewrite the program using this approach. What is the advantage of this approach?

3.71 Assume that the list of student test scores shown in Figure 2.14 is stored in the memory as a linked list as shown in Figure 2.36. Write an IA-32 program that accomplishes the same thing as the program in Figure 2.15. The head record is stored at memory location 1000.

3.72 The linked-list insertion subroutine in Figure 3.51 does not check if the ID of the new record matches that of a record already in the list. What happens in the execution of the subroutine if this is the case? Modify the subroutine to return the address of the matching record in register EDX if this occurs or to return a zero if the insertion is successful.

3.73 The linked-list deletion subroutine in Figure 3.52 assumes that a record with the ID contained in register RIDNUM is in the list. What happens in the execution of the subroutine if there is no record with this ID? Modify the subroutine to return a zero in RIDNUM if deletion is successful, or leave RIDNUM unchanged if the record is not in the list.
Chapter 3

ARM, Motorola, and Intel Instruction Sets

PART I: ARM

3.1. (a) R8, R9, and R10, contain 1, 2, and 3, respectively.

(b) The values 20 and 30 are pushed onto a stack pointed to by R1 by the two Store instructions, and they occupy memory locations 1996 and 1992, respectively. They are then popped off the stack into R8 and R9. Finally, the Subtract instruction results in 10 (30 − 20) being stored in R10. The stack pointer R1 is returned to its original value of 2000.

(c) The numbers in memory locations 1016 and 1020 are loaded into R4 and R5, respectively. These two numbers are then added and the sum is placed in register R4. The final address value in R2 is 1024.

3.2. (b) A memory operand cannot be referenced in a Subtract instruction.

(d) The immediate value 257 is 100000001 in binary, and is thus too long to fit in the 8-bit immediate field. Note that it cannot be generated by the rotation of any 8-bit value.

3.3. The following two instructions perform the desired operation:

```assembly
MOV R0,R0,LSL #24
MOV R0,R0,ASR #24
```

3.4. Use register R0 as a counter register and R1 as a work register.

```assembly
MOV R0,#32  # Load R0 with count value 32.
MOV R1,#0   # Clear register R1 to zero.
LOOP MOV R2,R2,LSL #1  # Shift contents of R2 left one bit position, moving the high-order bit into the C flag.
MOV R1,R1,RRX # Rotate R1 right one bit position, including the C flag, as shown in Figure 2.32d.
SUBS R0,R0,#1 # Check if finished.
BGT LOOP
MOV R2,R1   # Load reversed pattern back into R2.
```

1
3.5. Program trace:

<table>
<thead>
<tr>
<th>TIME</th>
<th>R0</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>after 1st execution of BGT</td>
<td>3</td>
<td>4</td>
<td>NUM1 + 4</td>
</tr>
<tr>
<td>after 2nd execution of BGT</td>
<td>-14</td>
<td>3</td>
<td>NUM1 + 8</td>
</tr>
<tr>
<td>after 3rd execution of BGT</td>
<td>13</td>
<td>2</td>
<td>NUM1 + 12</td>
</tr>
</tbody>
</table>

3.6. Assume bytes are unsigned 8-bit values.

- LDR R0,N: R0 is list counter
- ADR R1,X: R1 points to X list
- ADR R2,Y: R2 points to Y list
- ADR R3,LARGER: R3 points to LARGER list
- LOOP LDRB R4,[R1],#1 Load X list byte into R4
- LOOP LDRB R5,[R2],#1 Load Y list byte into R5
- CMP R4,R5: Compare bytes
- STRHSB R4,[R3],#1 Store X byte if larger or same
- STRLOB R5,[R3],#1 Store Y byte if larger
- SUBS R0,R0,#1 Check if finished
- BGT LOOP

3.7. The inner loop checks for a match at each possible position.

- LDR R0,N: Compute outer loop count and store in R2.
- LDR R1,M:
- SUB R2,R0,R1:
- ADD R2,R2,#1:
- ADR R3,STRING: Use R3 and R4 as base pointers for each match.
- ADR R4,SUBSTRING:
- OUTER MOV R5,R3: Use R5 and R6 as running pointers for each match.
- MOV R6,R4:
- LDR R7,M: Initialize inner loop counter.
- INNER LDRB R0,[R5],#1: Compare bytes.
- LDRB R1,[R6],#1:
- CMP R0,R1:
- BNE NOMATCH: If not equal, go next.
- SUBS R7,R7,#1: Check if all bytes compared.
- BGT INNER:
- MOV R0,R3: If substring matches, load its position into R0 and exit.
- B NEXT:
- NOMATCH ADD R3,R3,#1: Go to next substring.
- SUBS R2,R2,#1: Check if all positions tried.
- BGT OUTER:
- MOV R0,#0: If yes, load zero into R0 and exit.
3.8. This solution assumes that the last number in the series of \( n \) numbers can be represented in a 32-bit word, and that \( n > 2 \).

\[
\begin{align*}
\text{MOV} & \quad \text{R0,N} \quad \text{Use R0 to count numbers} \\
\text{SUB} & \quad \text{R0,R0,#2} \quad \text{generated after 1.} \\
\text{ADR} & \quad \text{R1,MEMLOC} \quad \text{Use R1 as memory pointer.} \\
\text{MOV} & \quad \text{R2,#0} \quad \text{Store first two numbers,} \\
\text{STR} & \quad \text{R2,[R1],#4} \quad 0 \text{ and } 1, \text{ from R2} \\
\text{MOV} & \quad \text{R3,#1} \quad \text{and R3 into memory.} \\
\text{STR} & \quad \text{R3,[R1],#4} \\
\text{LOOP} & \quad \text{ADD} \quad \text{R3,R2,R3} \quad \text{Starting with number } i - 1 \\
\text{STR} & \quad \text{R3,[R1],#4} \quad \text{in R2 and } i \text{ in R3, compute} \\
& \quad \text{and place } i + 1 \text{ in R3} \\
& \quad \text{and store in memory.} \\
\text{SUB} & \quad \text{R2,R3,R2} \quad \text{Recover old } i \text{ and place} \\
\text{SUBS} & \quad \text{R0,R0,#1} \quad \text{Check if all numbers} \\
\text{BGT} & \quad \text{LOOP} \quad \text{have been computed.}
\end{align*}
\]

3.9. Let R0 point to the ASCII word beginning at location WORD. To change to uppercase, we need to change bit \( b_5 \) from 1 to 0.

\[
\begin{align*}
\text{NEXT} & \quad \text{LDRB} \quad \text{R1,[R0]} \quad \text{Get character.} \\
\text{CMP} & \quad \text{##&20,R1} \quad \text{Check if space character.} \\
\text{ANDNE} & \quad \text{##&DF,R1} \quad \text{If not space: clear} \\
\text{STRNEB} & \quad \text{R1,[R0],#1} \quad \text{bit 5, store} \\
\text{BNE} & \quad \text{NEXT} \quad \text{converted character,} \\
& \quad \text{get next character.}
\end{align*}
\]
3.10. Memory word location $J$ contains the number of tests, $j$, and memory word location $N$ contains the number of students, $n$. The list of student marks begins at memory word location LIST in the format shown in Figure 2.14. The parameter $\text{Stride} = 4(j + 1)$ is the distance in bytes between scores on a particular test for adjacent students in the list.

The Post-indexed addressing mode $[\text{R2}], \text{R3,LSL #2}$ is used to access the successive scores on a particular test in the inner loop. The value in register R2 before each entry to the inner loop is the address of the score on a particular test for the first student. Register R3 contains the value $j + 1$. Therefore, register R2 is incremented by the $\text{Stride}$ parameter on each pass through the inner loop.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR R3,J</td>
<td>Load $j + 1$ into R3 to be used as an address offset.</td>
</tr>
<tr>
<td>ADD R3,R3,#1</td>
<td>Initialize R4 to the sum location for test 1.</td>
</tr>
<tr>
<td>ADR R4,LIST</td>
<td>Load address of test 1 score for student 1 into R5.</td>
</tr>
<tr>
<td>ADD R5,R5,#4</td>
<td>Initialize outer loop counter R6 to $j$.</td>
</tr>
<tr>
<td>LDR R6,J</td>
<td>Initialize outer loop counter R7 to $n$.</td>
</tr>
<tr>
<td>MOV R2,R5</td>
<td>Initialize base register R2 to location of student 1 test score for next inner loop sum computation.</td>
</tr>
<tr>
<td>MOV R0,#0</td>
<td>Clear sum accumulator register R0.</td>
</tr>
<tr>
<td>LDR R1,[R2],R3,LSL #2</td>
<td>Load test score into R1 and increment R2 by $\text{Stride}$ to point to next test score.</td>
</tr>
<tr>
<td>ADD R0,R0,R1</td>
<td>Accumulate score into R0.</td>
</tr>
<tr>
<td>SUBS R7,R7,#1</td>
<td>Check if all student scores for current test are added.</td>
</tr>
<tr>
<td>BGT INNER</td>
<td>Store sum in memory.</td>
</tr>
<tr>
<td>STR R0,[R4],#4</td>
<td>Increment R5 to next test score for student 1.</td>
</tr>
<tr>
<td>ADD R5,R5,#4</td>
<td>Check if sums for all test scores have been accumulated.</td>
</tr>
<tr>
<td>SUBS R6,R6,#1</td>
<td></td>
</tr>
<tr>
<td>BGT OUTER</td>
<td></td>
</tr>
</tbody>
</table>
3.11. Assume that the subroutine can change the contents of any registers used to pass parameters.

```
STR R5,[R13,#4]!  Save [R5] on stack.
ADD R1,R0,R1,LSL #2  Load address of A(0,x) into R1.
ADD R2,R0,R2,LSL #2  Load address of A(0,y) into R2.
LOOP LDR R5,[R1],R4,LSL #2  Load [A(i,x)] into R5 and increment pointer R1 by Stride = 4m.
LDR R0,[R2]  Load [A(i,y)] into R0.
ADD R0,R0,R5  Add corresponding column entries.
STR R0,[R2],R4,LSL #2  Store sum in A(i,y) and increment pointer R2 by Stride.
SUBS R3,R3,#1  Repeat loop until all entries have been added.
BGT LOOP  Return.
```

3.12. This program is similar to Figure 3.9, and makes the same assumptions about register usage and status word bit locations.

```
LDR R0,N  Use R0 as the loop counter for reading n characters.
READ LDR R3,[R1]  Load [INSTATUS] and
TST R3,#8  wait for character.
BEQ READ
LDRB R3,[R1,#4]  Read character and push onto stack.
STRB R3,[R6,#-1]!
ECHO LDR R4,[R2]  Load [OUTSTATUS] and
TST R4,#8  wait for display.
BEQ ECHO
STRB R3,[R2,#4]  Send character to display.
SUBS R0,R0,#1  Repeat until n
BGT READ  characters read.
```

3.13. Assume that most of the time between successive characters being struck is spent in the three-instruction wait loop that starts at location READ. The BEQ READ instruction is executed once every 60 ns while this loop is being executed. There are \(10^9/10 = 10^8\) ns between successive characters. Therefore, the BEQ READ instruction is executed \(10^8/60 = 1.6666 \times 10^6\) times per character entered.
3.14. Main Program

READLINE  BL   GETCHAR  Call character read subroutine.
 STRB  R3,[R0],#1  Store character in memory.
 BL   PUTCHAR  Call character display subroutine.
 TEQ  R3,#CR  Check for end-of-line character.
 BNE  READLINE

Subroutine GETCHAR

 TST  R3,#8
 BEQ  GETCHAR
 LDRB  R3,[R1,#4]  Load character into R3.
 MOV  R15,R14  Return.

Subroutine PUTCHAR

PUTCHAR  STMFD  R13!,{R4,R14}  Save R4 and Link register.
 DISPLAY  LDR  R4,[R2]  Wait for display.
 TST  R4,#8
 BEQ  DISPLAY
 STRB  R3,[R2,#4]  Send character to display.
 LDMFD  R13!,{R4,R15}  Restore R4 and Return.
3.15. Address INSTATUS is passed to GETCHAR on the stack; the character read is passed back in the same stack position. The character to be displayed and the address OUTSTATUS are passed to PUTCHAR on the stack in that order. The stack frame structure shown in Figure 3.13 is used.

**Main Program**

```
READLINE  LDR  R1,POINTER1  Load address INSTATUS
          STR  R1,[SP,#−4]! contained in POINTER1 into R1 and push onto stack.
          BL GETCHAR  Call character read subroutine.
          LDRB R1,[SP]  Load character from top of stack.
          STRB R1,[R0],#1  stack and store in memory.
          LDR  R2,POINTER2  Load address OUTSTATUS
          STR  R2,[SP,#−4]! contained in POINTER2 into R2 and push onto stack.
          BL PUTCHAR  Call character display subroutine.
          ADD SP,SP,#8  Remove parameters from stack.
          TEQ R1,#CR  Check for end-of-line character.
          BNE READLINE
```

**Subroutine GETCHAR**

```
GETCHAR  STMFD SP!,{R1,R3,FP,LR}  Save registers.
          ADD FP,SP,#8  Load frame pointer.
          LDR R1[FP,#8]  Load address INSTATUS into R1.
          READ LDR R3,[R1]  Wait for character.
          TST R3,#8  BEQ READ
          LDRB R3,[R1,#4]  Load character into R3 and overwrite INSTATUS
          STRB R3,[FP,#8] on stack.
          LDMFD SP!,{R1,R3,FP,PC}  Restore registers and Return.
```

**Subroutine PUTCHAR**

```
PUTCHAR  STMFD SP!,{R2−R4,FP,LR}  Save registers.
          ADD FP,SP,#12  Load frame pointer.
          LDR R2[FP,#8]  Load address OUTSTATUS into R2.
          LDR R3,[FP,#12]  R2 and character into R3.
          DISPLAY LDR R4,[R2]  Wait for display.
          TST R4,#8  BEQ DISPLAY
          STRB R3,[R2,#4]  Send character to display.
          LDMFD SP!,{R2−R4,FP,PC}  Restore registers and Return.
```
3.16. The first program section reads the characters, stores them in a 3-byte area beginning at CHARSTR, and echoes them to a display. The second section does the conversion to binary and stores the result in BINARY. The I/O device addresses INSTATUS and OUTSTATUS are in registers R1 and R2.

ADR R0,CHARSTR
MOV R5,#3

READ
LDR R3,[R1]
TST R3,#8
BEQ READ
LDRB R3,[R1,#4]
STRB R3,[R0],#1

ECHO
LDR R4,[R2]
TST R4,#8
BEQ ECHO
STRB R3,[R2,#4]
SUBS R5,R5,#1

CONVERT
ADR R0,CHARSTR
ADR R1,HUNDREDS
ADR R2,TENS
LDRB R3,[R0],#1
AND R3,R3,#&F
LDR R4,[R1,R3,LSL #2]
ADD R4,R4,R3
LDRB R3,[R0],#1
AND R3,R3,#&F
LDR R3,[R2,R3,LSL #2]
ADD R4,R4,R3
STR R4,BINARY
3.17. (a) The names FP, SP, LR, and PC, are used for registers R12, R13, R14, and R15 (frame pointer, stack pointer, link register, and program counter). The 3-byte memory area for the characters begins at address CHARSTR; and the converted binary value is stored at BINARY.

The first subroutine, labeled READCHARS, is patterned after the program in Figure 3.9. It echoes the characters back to a display as well as reading them into memory. The second subroutine is labeled CONVERT.

The stack frame format used is like Figure 3.13.

A possible main program is:

**Main program**

```
ADR R10,CHARSTR Load parameters into
ADR R11,BINARY R10 and R11 and
STMFD SP!,{R10,R11} push onto stack.
BL READCHARS Branch to first subroutine.
RTNADDR ADD SP,SP,#8 Remove two parameters
....
```

**First subroutine READCHARS**

```
READCHARS STMFD SP!,{R0−R5,FP,LR} Save registers on stack.
ADD FP,SP,#28 Set up frame pointer.
LDR R0,[FP,#4] Load R0, R1,
ADR R1,INSTATUS and R2 with
ADR R2,OUTSTATUS parameters.
MOV R5,#3 Same code as
....
BGT READ Problem 3.16.
LDR R0,[FP,#8] Load R0,R1,R2
LDR R5,[FP,#12] and R5 with
ADR R1,HUNDREDS parameters.
ADR R2,TENS
BL CONVERT Call second subroutine.
LDMFD SP!,{R0−R5,FP,PC} Return to Main program.
```
Second subroutine CONVERT

CONVERT STMFD SP!,{R3,R4,FP,LR}  Save registers on stack.
ADD    FP,SP,#8     Set up frame pointer.
LDRB   R3,[R0],#1 Same code as in solution to
... ADD   R4,R4,R3 Problem 3.16.
STR    R4,[R5] Store binary number.
LDMFD  SP!,{R3,R4,FP,PC} Return to first subroutine.

(b) The contents of the top of the stack after the call to the CONVERT routine are:

```
| [R0] |
| [R1] |
| [R2] |
| [R3] |
| [R4] |
| [R5] |
| [FP] |

LR = RTNADDR
CHARSTR
BINARY
Original TOS
```
3.18. See the solution to Problem 2.18 for the procedures needed to perform the append and remove operations.

Register assignment:

\begin{align*}
R0 & \rightarrow \text{Data byte to append to or remove from queue} \\
R1 & \rightarrow \text{IN pointer} \\
R2 & \rightarrow \text{OUT pointer} \\
R3 & \rightarrow \text{Address of first queue byte location} \\
R4 & \rightarrow \text{Address of last queue byte location (}= [R3] + k - 1) \\
R5 & \rightarrow \text{Auxiliary register for address of next appended byte.}
\end{align*}

Initially, the queue is empty with \([R1] = [R2] = [R3]\)

**APPEND routine:**

\begin{align*}
\text{MOV} & \quad \text{R5,R1} \\
\text{ADD} & \quad \text{R1,R1,#1} \quad \text{Increment R1 Modulo } k. \\
\text{CMP} & \quad \text{R1,R4} \\
\text{MOVG} & \quad \text{R1,R3} \\
\text{CMP} & \quad \text{R1,R2} \quad \text{Check if queue is full.} \\
\text{MOVEQ} & \quad \text{R1,R5} \quad \text{If queue full, restore} \\
\text{BEQ} & \quad \text{QUEUEFULL} \quad \text{IN pointer and send message that queue is full.} \\
\text{STRB} & \quad \text{R0,[R5]} \quad \text{If queue not full, append byte and continue.}
\end{align*}

**REMOVE routine:**

\begin{align*}
\text{CMP} & \quad \text{R1,R2} \quad \text{Check if queue is empty.} \\
\text{BEQ} & \quad \text{QUEUEEMPTY} \quad \text{If empty, send message.} \\
\text{LDRB} & \quad \text{R0,[R2],#1} \quad \text{Otherwise, remove byte} \\
\text{CMP} & \quad \text{R2,R4} \quad \text{and increment R2} \\
\text{MOVG} & \quad \text{R2,R3} \quad \text{Modulo } k.
\end{align*}

3.19. Program trace:

<table>
<thead>
<tr>
<th>TIME</th>
<th>R0</th>
<th>R2</th>
<th>R3</th>
<th>LIST</th>
<th>LIST +1</th>
<th>LIST +2</th>
<th>LIST +3</th>
<th>LIST +4</th>
</tr>
</thead>
<tbody>
<tr>
<td>After 1st</td>
<td>120</td>
<td>1004</td>
<td>1000</td>
<td>106</td>
<td>13</td>
<td>67</td>
<td>45</td>
<td>120</td>
</tr>
<tr>
<td>After 2nd</td>
<td>106</td>
<td>1003</td>
<td>1000</td>
<td>67</td>
<td>13</td>
<td>45</td>
<td>106</td>
<td>120</td>
</tr>
<tr>
<td>After 3rd</td>
<td>67</td>
<td>1002</td>
<td>1000</td>
<td>45</td>
<td>13</td>
<td>67</td>
<td>106</td>
<td>120</td>
</tr>
<tr>
<td>After 4th</td>
<td>45</td>
<td>1001</td>
<td>1000</td>
<td>13</td>
<td>45</td>
<td>67</td>
<td>106</td>
<td>120</td>
</tr>
</tbody>
</table>
3.20. Calling program

ADR R4, LISTN  Pass parameter LISTN to subroutine in R4.
Assume LISTN + 4 = LIST.

BL SORT

Subroutine SORT

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STMFD R13!, {R0−R3,R5,R14}</td>
<td>Save registers.</td>
</tr>
<tr>
<td>LDR R0,[R4],#4</td>
<td>Initialize outer loop base register R2 to LIST + 4n.</td>
</tr>
<tr>
<td>ADD R2,R4,R0,LSL #2</td>
<td></td>
</tr>
<tr>
<td>ADD R5,R4,#4</td>
<td>Load LIST + 4 into register R5.</td>
</tr>
<tr>
<td>OUTER LDR R0,[R2,#−4]!</td>
<td>Comments similar as in Figure 3.15.</td>
</tr>
<tr>
<td>MOV R3,R2</td>
<td></td>
</tr>
<tr>
<td>INNER LDR R1,[R3,#−4]!</td>
<td></td>
</tr>
<tr>
<td>CMP R1,R0</td>
<td></td>
</tr>
<tr>
<td>STRGT R1,[R2]</td>
<td></td>
</tr>
<tr>
<td>STRGT R0,[R3]</td>
<td></td>
</tr>
<tr>
<td>MOVGT R0,R1</td>
<td></td>
</tr>
<tr>
<td>CMP R3,R4</td>
<td></td>
</tr>
<tr>
<td>BNE INNER</td>
<td></td>
</tr>
<tr>
<td>CMP R2,R5</td>
<td></td>
</tr>
<tr>
<td>BNE OUTER</td>
<td></td>
</tr>
<tr>
<td>LDMFD R13!, {R0−R3,R5,R15}</td>
<td>Restore registers and return.</td>
</tr>
</tbody>
</table>
3.21. The alternative program from the instruction labeled OUTER to the end is:

```
OUTER  LDRB  R0,[R2,#−1]! Load LIST(j) into R0.
      MOV   R3,R2 Initialize inner loop base register R3 to LIST + n − 1.
      MOV   R6,R2 Load address of initial largest element into R6.
      MOV   R7,R0 Load initial largest element into R7.
INNER  LDRB  R1,[R3,#−1]! Load LIST(k) into R1.
      CMP   R1,R7 Compare LIST(k) to current largest.
      MOVGT R6,R3 Update address and value of largest if LIST(k) larger.
      CMP   R3,R4 Check if inner loop completed.
      MOVGT R7,R1
      CMP   R3,R5
      BNE   INNER
      STRB  R0,[R6] Swap; correct code even if no larger element is found.
      CMP   R2,R5
      BNE   OUTER
```

The advantage of this approach is that the two MOVGT instructions in the inner loop of the alternative program execute faster than the three-instruction interchange code in Figure 3.15b.

3.22. The record pointer is register R0, and registers R1, R2, and R3, are used to accumulate the three sums, as in Figure 2.15. Assume that the list is not empty.

```
MOV   R0,#1000
MOV   R1,#0
MOV   R2,#0
MOV   R3,#0
LOOP  LDR  R5,[R0,#8]
      ADD  R1,R1,R5
      LDR  R5,[R0,#12]
      ADD  R2,R2,R5
      LDR  R5,[R0,#16]
      ADD  R3,R3,R5
      LDR  R0,[R0,#4]
      CMP  R0,#0
      BNE  LOOP
      STR  R1,SUM1
      STR  R2,SUM2
      STR  R3,SUM3
```
3.23. If the ID of the new record matches the ID of the Head record, the new record will become the new Head. If the ID matches that of a later record, it will be inserted immediately after that record, including the case where the matching record is the Tail.

Modify Figure 3.16 as follows:

- Add the following instruction as the first instruction of the subroutine:

  ```assembly
  INSERTION MOV R10,#0  Anticipate successful insertion of new record.
  ```

- After the second CMP instruction, insert the following two instructions:

  ```assembly
  MOVEQ R10, RHEAD  ID matches that of
  MOVEQ PC, R14   Head record.
  ```

- After the instruction labeled LOOP, insert the following four instructions:

  ```assembly
  LDR R0, [RNEXT]  
  CMP  R0, R1 
  MOVEQ R10, RNEXT 
  MOVEQ PC, R14
  ```

- Remove the instruction with the comment “Go further?” because it has already been done in the previous bullet.
3.24. If the list is empty, the result is unpredictable because the second instruction compares the new ID with the contents of memory location zero. If the list is not empty, the program continues until RCURRENT points to the Tail record. Then the instruction at LOOP loads zero into RNEXT and the result is unpredictable.

Replace Figure 3.17 with the following code:

```
DELETION
CMP    RHEAD, #0       If list is empty, return
MOVEQ  PC, R14
CHECKHEAD
LDR    R0, [RHEAD]     Check if Head record is
CMP    R0, RIDNUM      to be deleted. If yes,
LDREQ  RHEAD, [RHEAD,#4] delete it, and then return
MOVEQ  RIDNUM, #0     with zero in RIDNUM.
MOVEQ  PC, R14
MOV    RCURRENT, RHEAD Otherwise, continue search.
LOOP
LDR    RNEXT, [RCURRENT,#4] If all records checked, return
CMP    RNEXT, #0       with RIDNUM unchanged.
MOVEQ  PC, R14
LDR    R0, [RNEXT]     Is next record the one
CMP    R0, RIDNUM      to be deleted?
LDREQ  R0, [RNEXT,#4]  If yes, delete it, and
STREQ  R0, [RCURRENT,#4] return with zero
MOVEQ  RIDNUM, #0     in RIDNUM.
MOVEQ  PC, R14
MOV    RCURRENT, RNEXT Otherwise, loop back and
B      LOOP            continue to search.
```
PART II: 68000

3.25. (a) Location $2000 ← $1000 + $3000 = $4000
The instruction occupies two bytes. One memory access is needed to fetch the instruction and 4 to execute it.

(b) Effective Address = $1000 + $1000 = $2000,
D0 ← $3000 + $1000 = $4000
4 bytes; 2 accesses to fetch instruction and 2 to execute it.

(c) $2000 ← $2000 + $3000 = $5000
6 bytes; 3 accesses to fetch instruction and 4 to execute it.

3.26. (a) ADDX (A2),D3
In Add extended, both the destination and source operands must use the same addressing mode, either register or autodecrement.

(b) LSR.L #9,D2
The number of bits shifted must be less than 8.

(c) MOVE.B 520(A0,D2)
The offset value requires more than 8 bits. Also, no destination operand is specified.

(d) SUBA.L 12(A2,PC),A0
In relative full addressing mode the PC must be specified before the address register.

(e) CMP.B #254,$12(A2,D1.B)
The destination operand must be a data register. Also the source operand is outside the range of signed values that can be represented in 8 bits.

3.27. Program trace:

<table>
<thead>
<tr>
<th>TIME</th>
<th>D0</th>
<th>D1</th>
<th>A2</th>
<th>N</th>
<th>NUM1</th>
<th>SUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>after 1st ADD.W</td>
<td>83</td>
<td>5</td>
<td>2402</td>
<td>5</td>
<td>2400</td>
<td>0</td>
</tr>
<tr>
<td>after 2nd ADD.W</td>
<td>128</td>
<td>4</td>
<td>2404</td>
<td>5</td>
<td>2400</td>
<td>0</td>
</tr>
<tr>
<td>after 3rd ADD.W</td>
<td>284</td>
<td>3</td>
<td>2406</td>
<td>5</td>
<td>2400</td>
<td>0</td>
</tr>
<tr>
<td>after 4th ADD.W</td>
<td>34</td>
<td>2</td>
<td>2408</td>
<td>5</td>
<td>2400</td>
<td>0</td>
</tr>
<tr>
<td>after 5th ADD.W</td>
<td>134</td>
<td>1</td>
<td>2410</td>
<td>5</td>
<td>2400</td>
<td>0</td>
</tr>
<tr>
<td>after last MOVE.L</td>
<td>134</td>
<td>0</td>
<td>2410</td>
<td>5</td>
<td>2400</td>
<td>134</td>
</tr>
</tbody>
</table>
3.28. (a) This program finds the location of the smallest element in a list whose starting address is stored in MEM1, and size in MEM2. The smallest element is stored in location DESIRED.

(b) 16 words are required to store this program. We have assumed that the assembler uses short absolute addresses. (Long addresses are normally specified as MEM1.L, etc.) Otherwise, 3 more words would be needed.

(c) The expression for memory accesses is \( T = 16 + 5n + 4m \).

3.29. (a) They both leave the 17th negative word in RSLT.

(b) Both programs scan through the list to find the 17th negative number in the list.

(c) Program 1 takes 26 bytes of memory, while Program 2 requires 24.

(d) Let \( P \) be the number of non-negative entries encountered. Program 1 requires \( 9 + 7 \times 17 + 3 \times P \) and Program 2 requires \( 10 + 6 \times 17 + 4 \times P \) memory accesses.

(e) Program 1 requires slightly more memory, but has a clear speed advantage. Program 2 destroys the original list.

3.30. A 68000 program to compare two byte lists at locations X and Y, putting the larger byte at each position in a list starting at location LARGER, is:

```
MOVEA.L  #X,A0
MOVEA.L  #Y,A1
MOVEA.L  #LARGER,A2
MOVE.W  N,D0
SUBQ    #1,D0   Initialize D0 to [N]–1
LOOP   CMP.B  (A0)+,(A1)+   Compare lists and advance pointers
   BGT    LISTY
   MOVE.B  −1(A0),(A2)+  Copy item from list X
   BRA    NEXT  Check next item
LISTY  MOVE.B  −1(A1),(A2)+  Copy item from list Y
NEXT   DBRA  D0,LOOP  Continue if more entries
```

Introduzione all'architettura dei calcolatori 2/ed - Carl Hamacher, Zvonko Vranesic, Safwat Zaky
Copyright © 2006 - The McGraw-Hill Companies srl
3.31. A 68000 program for string matching:

```assembly
MOVEA.L #STRING,A0 ; Get location of STRING
MOVE.W N,D0 ; Load D0 with appropriate
MOVE.W M,D1 ; count for “match attempts”
SUB.W D1,D0

LOOP
MOVEA.L #SUBSTRING,A1 ; Get location of SUBSTRING
MOVE.W M,D1 ; Get size of SUBSTRING
MOVE.L A0,A2 ; Save location in STRING at which
              ; comparison will start

MATCHER  DBRA D1,SUCCESS
CMP.B (A0)+,(A1)+ ; Compare and advance pointers
BEQ MATCHER ; If same, check next character
MOVEA.L A2,A0 ; Match failed; advance starting
ADDQ.L #1,A0 ; character position in STRING
DBRA D0,LOOP ; Check if end of STRING
MOVE.L #0,D0 ; Substring not found
BRA NEXT ;

SUCCESS MOVEA.L A2,D0 ; Save location where match found

NEXT  Next instruction
```

Note that DBRA is used in two ways in this program, once at the beginning
and once at the end of a loop. In the first case, the counter is initialized
to [M], while in the second the corresponding counter is initialized to
[N]−[M]. This arrangement handles a substring of zero length correctly,
and stops the attempt to find a match at the proper position.
3.32. A 68000 program to generate the first \( n \) numbers of the Fibonacci series:

```asm
MOVEA.L #MEMLOC,A0 Starting address
MOVE.B N,D0 Number of entries
CLR D1 The first entry = 0
MOVE.B D1,(A0)+
MOVE M #1,D2 The second entry = 1
MOVE.B D2,(A0)+
SUBQ.B #3,D0 First two entries already saved
LOOP MOVE.B −2(A0),D1 Get second-last value
ADD.B D1,D2 Add to last value
MOVE.B D2,(A0)+ Store new value
DBRA D0,LOOP
```

The first 15 numbers in the Fibonacci sequence are: 0, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377. Therefore, the largest value of \( n \) that this program can handle is 14, because the largest number that can be stored in a byte is 255.

3.33. Let A0 point to the ASCII word. To change to uppercase, we need to change bit \( b_5 \) from 1 to 0.

```asm
NEXT MOVE.B (A0),D0 Get character
CMP.B #$20,D0 Check if space character
BEQ END
ANDI.B #$DF,D0 Clear bit 5
MOVE.B D0,(A0)+ Store converted character
BRA NEXT
END Next instruction
```
3.34. Let Stride = 2(j + 1), which is the distance in bytes between scores on a particular test for adjacent students in the list.

MOVE J,D3 Compute Stride = 2(j + 1)  
ADDQ #1,D3  
LSL #1,D3  
MOVEA.L #SUM,A4 Use A4 as pointer to the sums  
MOVEA.L #LIST,A5 Use A5 as pointer to scores  
ADDQ #2,A5 for student 1  
MOVE J,D6 Use D6 as outer loop counter  
SUBQ #1,D6 Adjust for use of DBRA instruction  

OUTER MOVE N,D7 Use D7 as inner loop counter  
SUBQ #1,D7 Adjust for use of DBRA instruction  
MOVE A5,A2 Use A2 as base for scanning test scores  
CLR D0 Use D0 as sum accumulator  

INNER ADD [A2],D0 Accumulate test scores  
ADD D3,A2 Point to next score  
DBRA D7,INNER Check if score for current test for all students have been added  
MOVE D0,[A4] Store sum in memory  
ADDQ #2,A5 Increment to next test  
ADDQ #2,A4 Point to next sum  
DBRA D6,OUTER Check if scores for all tests have been accumulated  

3.35. This program is similar to Figure 3.27, and makes the same assumptions about status word bit locations.

MOVE #N,D0 Initialize D0 to n − 1  
SUBQ.W #1,D0  
READ BTST.W #3,INSTATUS  
BEQ READ Wait for data ready  
MOVE.B DATAIN,D1 Get new character  
MOVE.B D1,−(A0) Push on user stack  
ECHO BTST.W #3,OUTSTATUS  
BEQ ECHO Wait for terminal ready  
MOVE.B D1,DATAOUT Output new character  
DBRA D0,READ Read next character
3.36. Assume that most of the time between successive characters being struck is spent in the two-instruction wait loop that starts at location READ. The BEQ READ instruction is executed once every 40 ns while this loop is being executed. There are $10^9/10 = 10^8$ ns between successive characters. Therefore, the BEQ READ instruction is executed $10^8/40 = 2.5 \times 10^6$ times per character entered.

3.37. Assume that register A4 is used as a memory pointer by the main program.

**Main Program**

```
READLINE BSR GETCHAR Call character read subroutine.
MOVE.B D0,(A4)+ Store character in memory.
BSR PUTCHAR Call character display subroutine.
CMPI.B #CR,D0 Check for end-of-line character.
BNE READLINE
```

**Subroutine GETCHAR**

```
GETCHAR BTST.W #3,(A0) Wait for character.
BEQ GETCHAR
MOVE.B (A1),D0 Load character into D0.
RTS Return.
```

**Subroutine PUTCHAR**

```
PUTCHAR BTST.W #3,(A2) Wait for display.
BEQ PUTCHAR
MOVE.B D0,(A3) Send character to display.
RTS Return.
```
3.38. Addresses INSTATUS and DATAIN are pushed onto the processor stack in that order by the main program as parameters for GETCHAR. The character read is passed back to the main program in the DATAIN position on the stack. The addresses OUTSTATUS and DATAOUT and the character to be displayed are pushed onto the processor stack in that order by the main program as parameters for PUTCHAR. A stack structure like that shown in Figure 3.29 is used.

GETCHAR uses registers A0, A1, and D0 to hold INSTATUS, DATAIN, and the character read.

PUTCHAR uses registers A0, A1, and D0 to hold OUTSTATUS, DATAOUT, and the character to be displayed.

The main program uses register A0 as a memory pointer, and uses register D0 to hold the character read.

**Main Program**

```
READLINE  MOVE.L  #INSTATUS,-(A7) ; Push address parameters onto the stack.
          MOVE.L  #DATAIN,-(A7)  ; Call character read subroutine.
          BSR       GETCHAR
          MOVE.L  (A7)+,D0       ; Pop long word containing character from top of stack into D0 and store character into memory.
          ADDI     #4,A7
          MOVE.L  #OUTSTATUS,-(A7) ; Push address parameters onto stack.
          MOVE.L  #DATAOUT,-(A7)  ; Push long word containing character onto stack.
          BSR       PUTCHAR
          ADDI     #12,A7         ; Remove three parameters from stack.
          CMPI.B   #CR,D0         ; Check for end-of-line character.
          BNE      READLINE
```

**Subroutine GETCHAR**

```
GETCHAR  MOVEM  D0/A0-A1,-(A7) ; Save registers.
          MOVE.L  20(A7),A0      ; Load address INSTATUS into A0.
          MOVE.L  16(A7),A1     ; Load address DATAIN into A1.
          BTST     #3,(A0)      ; Wait for character.
          BEQ      READ
          MOVE.B  (A1),D0      ; Load character into D0 and push onto the stack, overwriting DATAIN.
          MOVE.L  D0,16(A7)    ; Restore registers.
          MOVEM  (A7)+,D0/A0-A1 ; Return.
          RTS
```
Subroutine PUTCHAR

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUTCHAR</td>
<td>MOVEM D0/A0-A1,−(A7)</td>
</tr>
<tr>
<td></td>
<td>Save registers.</td>
</tr>
<tr>
<td>MOVE.L</td>
<td>24(A7),A0</td>
</tr>
<tr>
<td></td>
<td>Load address OUTSTATUS into A0.</td>
</tr>
<tr>
<td>MOVE.L</td>
<td>20(A7),A1</td>
</tr>
<tr>
<td></td>
<td>Load address DATAOUT into A1.</td>
</tr>
<tr>
<td>MOVE.L</td>
<td>16(A7),D0</td>
</tr>
<tr>
<td></td>
<td>Load long word containing</td>
</tr>
<tr>
<td></td>
<td>character into D0.</td>
</tr>
<tr>
<td>DISPLAY</td>
<td>BTST #3,(A0)</td>
</tr>
<tr>
<td></td>
<td>Wait for device ready.</td>
</tr>
<tr>
<td>BEQ</td>
<td>DISPLAY</td>
</tr>
<tr>
<td>MOVE.B</td>
<td>D0,(A1)</td>
</tr>
<tr>
<td></td>
<td>Send character to display.</td>
</tr>
<tr>
<td>MOVEM</td>
<td>(A7)+,D0/A0-A1</td>
</tr>
<tr>
<td></td>
<td>Restore registers.</td>
</tr>
<tr>
<td>RTS</td>
<td>Return.</td>
</tr>
</tbody>
</table>


3.39. See the solution to Problem 2.18 for the procedures needed to perform the append and remove operations.

Register assignment:

- **D0** — Data byte to append to or remove from queue
- **A1** — IN pointer
- **A2** — OUT pointer
- **A3** — Address of first queue byte location
- **A4** — Address of last queue byte location (= \([A3] + k - 1\))
- **A5** — Auxiliary register for address of next appended byte

Initially, the queue is empty with \([A1] = [A2] = [A3]\)

**APPEND routine:**

```
MOVEA.L A1,A5
ADDQ.L #1,A1 Increment A1 Modulo k.
CMPA.L A1,A4
BGE CHECK
MOVEA.L A3,A1
CHECK
CMPA.L A1,A2 Check if queue is full.
BNE APPEND If queue not full, append byte.
MOVEA.L A5,A1 Otherwise, restore
BRA QUEUEFULL IN pointer and send
message that queue is full.
```

**APPEND**

```
MOVE.B D0,[A5] Append byte.
```

**REMOVE routine:**

```
CMPA.L A1,A2 Check if queue is empty.
BEQ QUEUEEMPTY If empty, send message.
MOVE.B (A2)+,D0 Otherwise, remove byte
CMPA.L A2,A4 and increment A2
BGE NEXT Modulo k.
MOVEA.L A3,A2
NEXT . . .
```
3.40. Using the same assumptions as in Problem 3.35 and its solution, a 68000 program to convert 3 input decimal digits to a binary number is:

```
BSR READ Get first character
ASL #1,D0 Multiply by 2 for word offset
MOVE.W HUNDREDS(D0),D1 Get hundreds value
BSR READ Get second character
ASL #1,D0 Multiply by 2 for word offset
ADD.W TENS(D0),D1 Get tens value
BSR READ Get third character
ADD.W D0,D1 D1 contains value of binary number

READ BTST.W #3,INSTATUS
BEQ READ Wait for new character
MOVE.B DATAIN,D0 Get new character
AND.B #$0F,D0 Convert to equivalent binary value

RTS
```
3.41. (a) The subroutines convert 3 decimal digits to a binary value.

GETDECIMAL

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVEM.L D0/A0−A1,−(A7)</td>
<td>Save registers</td>
</tr>
<tr>
<td>MOVEA.L 20(A7),A0</td>
<td>Get string buffer address</td>
</tr>
<tr>
<td>MOVE.B #2,D0</td>
<td>Use D0 as character counter</td>
</tr>
</tbody>
</table>

READ

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTST.W #3,INSTATUS</td>
<td>Get and store character</td>
</tr>
<tr>
<td>BEQ READ</td>
<td>Repeat until all characters received</td>
</tr>
<tr>
<td>MOVE.B DATAIN,(A0)+</td>
<td>Get and store character</td>
</tr>
<tr>
<td>DBRA D0,READ</td>
<td>Repeat until all characters received</td>
</tr>
<tr>
<td>MOVE.L 16(A7),A1</td>
<td>Pointer to result</td>
</tr>
<tr>
<td>BSR CONVERSION</td>
<td>Restore registers</td>
</tr>
<tr>
<td>MOVEM.L (A7)+,D0−A0−A1</td>
<td>Restore registers</td>
</tr>
</tbody>
</table>

CONVERSION

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVEM.L D0−D1,−(A7)</td>
<td>Save registers</td>
</tr>
<tr>
<td>MOVE.B −(A0),D0</td>
<td>Get least sig. digit</td>
</tr>
<tr>
<td>AND.W #$0F,D0</td>
<td>Numeric value of digit</td>
</tr>
<tr>
<td>MOVE.B −(A0),D1</td>
<td>Get tens digit</td>
</tr>
<tr>
<td>AND.W #$0F,D1</td>
<td>Numeric value of digit</td>
</tr>
<tr>
<td>ASL #1,D1</td>
<td>Add tens value</td>
</tr>
<tr>
<td>ADD.W TENS(D1),D0</td>
<td>Add tens value</td>
</tr>
<tr>
<td>MOVE.B −(A0),D1</td>
<td>Get hundreds digit</td>
</tr>
<tr>
<td>AND.W #$0F,D1</td>
<td>Numeric value of digit</td>
</tr>
<tr>
<td>ASL #1,D1</td>
<td>Add hundreds value</td>
</tr>
<tr>
<td>ADD.W HUNDREDS(D1),D0</td>
<td>Add hundreds value</td>
</tr>
<tr>
<td>MOVE.W D0,(A1)</td>
<td>Store result</td>
</tr>
<tr>
<td>MOVEM.L (A7)+,D0−D1</td>
<td>Restore registers</td>
</tr>
</tbody>
</table>

(b) The contents of the top of the stack after the call to the CONVERSION routine are:

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0_MAIN</td>
</tr>
<tr>
<td>A1_MAIN</td>
</tr>
<tr>
<td>A0_MAIN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result address</td>
</tr>
<tr>
<td>Buffer address</td>
</tr>
<tr>
<td>ORIG TOS</td>
</tr>
</tbody>
</table>
3.42. Assume that the subroutine can change the contents of any registers used to pass parameters. Let Stride = 2m, which is the distance in bytes between successive word elements in a given column.

```
LSL  #1,D4       Set Stride in D4
SUB  D1,D2       Set D2 to contain
LSL  #1,D2       2(y - x)
LSL  #1,D1       Set A0 to address
ADDA D1,A0       A(0,x)
BRA  START
LOOP MOVE (A0),D1 Load [A(i,x)] into D1
ADD  D1,(A0,D2)  Add array elements
ADD  D4,A0       Move to next row
START DBRA D3,LOOP Repeat loop until all
          entries have been added
RTS          Return
```

Note that LOOP is entered by branching to the DBRA instruction. So DBRA decrements D3 to contain n - 1, which is the correct starting value when the DBRA instruction is used.

3.43. A 68000 program to reverse the order of bits in register D2:

```
MOVE  #15,D0     Use D0 as counter
CLR   D1         D1 will receive new value
LOOP  LSL  D2     Shift MSB of D2 into X bit
ROXR  D1         Shift X bit into MSB of D1
DBRA  D0,LOOP    Repeat until D0 reaches -1
MOVE  D1,D2      Put new value back in D2
```
3.44. Bytes/access

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Bytes/Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVEA.L #LOC,A0</td>
<td>6/3</td>
</tr>
<tr>
<td>MOVE.B (A0)+,D0</td>
<td>2/2</td>
</tr>
<tr>
<td>LSL.B #4,D0</td>
<td>2/1</td>
</tr>
<tr>
<td>MOVE.B (A0),D1</td>
<td>2/2</td>
</tr>
<tr>
<td>ANDI.B #$F,D1</td>
<td>4/2</td>
</tr>
<tr>
<td>OR.B D0,D1</td>
<td>2/1</td>
</tr>
<tr>
<td>MOVE.B D1,PACKED</td>
<td>4/3</td>
</tr>
</tbody>
</table>

Total size is 22 bytes and execution involves 14 memory access cycles.

3.45. The trace table is:

<table>
<thead>
<tr>
<th>TIME</th>
<th>1000</th>
<th>1001</th>
<th>1002</th>
<th>1003</th>
<th>1004</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
</tr>
</thead>
<tbody>
<tr>
<td>after 1st BGT OUTER</td>
<td>106</td>
<td>13</td>
<td>67</td>
<td>45</td>
<td>120</td>
<td>3</td>
<td>-1</td>
<td>120</td>
</tr>
<tr>
<td>after 2nd BGT OUTER</td>
<td>67</td>
<td>13</td>
<td>45</td>
<td>106</td>
<td>120</td>
<td>2</td>
<td>-1</td>
<td>106</td>
</tr>
<tr>
<td>after 3rd BGT OUTER</td>
<td>45</td>
<td>13</td>
<td>67</td>
<td>106</td>
<td>120</td>
<td>1</td>
<td>-1</td>
<td>67</td>
</tr>
<tr>
<td>after 4th BGT OUTER</td>
<td>13</td>
<td>45</td>
<td>67</td>
<td>106</td>
<td>120</td>
<td>0</td>
<td>-1</td>
<td>45</td>
</tr>
</tbody>
</table>

3.46. Assume the list address is passed to the subroutine in register A1. When the subroutine is entered, the number of list entries needs to be loaded into D1. Then A1 must be updated to point to the first entry in the list. Because addresses must be incremented or decremented by 2 to handle word quantities, the address mode (A1,D1) is no longer useful. Also, since the initial address points to the beginning of the list, we will scan the list forwards.

```assembly
MOVE (A1)+,D1 Load number of entries, n
SUBQ #2,D1 Outer loop counter ← n - 2 (j: 0 to n - 2)
OUTER MOVE A1,A2 Use A2 as a pointer in the inner loop
ADDQ #2,A2 k ← j + 1 (k: 1 to n - 1)
INNER MOVE (A1),D3 Current maximum value in D3
CMP (A2),D3 If LIST(j) ≤ LIST(k), go to next
BLE NEXT Interchange LIST(k)
MOVE (A2),(A1) and LIST(j).
MOVE D3,(A2)
NEXT ADDQ #2,A2 If not finished,
DBRA D2,INNER
ADDQ #2,A1
DBRA D1,OUTER return
RTS
```
3.47. Use D4 to keep track of the position of the largest element in the inner loop and D5 to record its value.

\begin{verbatim}
MOVEA.L #LIST,A1      Pointer to the start of the list
MOVE    N,D1          Initialize outer loop
SUBQ    #1,D1          index j in D1
OUTER   MOVE    D1,D2   Initialize inner loop
             #1,D2          index k in D2
             MOVE.L D1,D4 Index of largest element
             MOVE.B (A1,D1),D5 Value of largest element
INNER   MOVE.B (A1,D2),D3 Get new element, LIST(k)
             CMP.B D3,D5 Compare to current maximum
             BCC NEXT If lower go to next entry
             MOVE.L D2,D4 Update index of largest element
             MOVE.L D3,D5 Update largest value
NEXT     DBRA D2,INNER Inner loop control
             MOVE.B (A1,D1),(A1,D4) Swap LIST(j) and LIST(k);
             MOVE.B D5,(A1,D1) correct even if same
             SUBQ    #1,D1 Branch back
             BGT OUTER if not finished
\end{verbatim}

The potential advantage is that the inner loop of the new program should execute faster.

3.48. Assume that register A0 points to the first record. We will use registers D1, D2, and D3 to accumulate the three sums. Assume also that the list is not empty.

\begin{verbatim}
CLR       D1
CLR       D2
CLR       D3
LOOP     ADD.L 8(A0),D1 Accumulate scores for test 1
             ADD.L 12(A0),D2 Accumulate scores for test 2
             ADD.L 16(A0),D3 Accumulate scores for test 3
             MOVE.L 4(A0),D0 Get link
             MOVEA.L D0,A0 Load in pointer register
             BNE LOOP
             MOVE.L D1,SUM1
             MOVE.L D2,SUM2
             MOVE.L D3,SUM3
\end{verbatim}

Note that the MOVE instruction that reads the link value into register D0 sets the Z and N flags. The MOVEA instruction does not affect the condition code flags. Hence, the BNE instruction will test the correct values.
3.49. In the program of Figure 3.35, if the ID of the new record matches the ID of the Head record, the new record will become the new Head. If the ID matches that of a later record, it will be inserted immediately after that record, including the case where the matching record is the Tail.

Modify the program as follows.

Add the following as the first instruction

\[
\text{INSERTION MOVE.L } \#0,A6 \quad \text{Anticipate a successful insertion}
\]

After the instruction labeled HEAD insert

\[
\text{BEQ DUPLICATE1 New record matches head}
\]

After the BLT INSERT instruction insert

\[
\text{BEQ DUPLICATE2 New record matches a record other than head}
\]

Add the following instructions at the end

\[
\text{DUPLICATE1 MOVE.L A0,A6 \quad \text{Return the address of the head}}
\]
\[
\text{RTS}
\]
\[
\text{DUPLICATE2 MOVE.L A3,A6 \quad \text{Return address of matching record}}
\]
\[
\text{RTS}
\]

3.50. If the ID of the new record is less than that of the head, the program in Figure 3.36 will delete the head. If the list is empty, the result is unpredictable because the first instruction compares the new ID with the contents of memory location zero. If the list is not empty, the program continues until A2 points to the Tail record. Then the instruction at LOOP loads zero into A3 and the result is unpredictable.

To correct behavior, modify the program as follows.

After the first BGT instruction insert

\[
\text{BLT ERROR ID of new record less than head}
\]
\[
\text{MOVE.L } \#0,D1 \quad \text{Deletion successful}
\]

After the BEQ DELETE instruction insert

\[
\text{BGT ERROR ID of New record is less than that of the next record and greater than the current record}
\]

Add the following instruction after DELETE

\[
\text{MOVE.L } \#0,D1 \quad \text{Deletion successful}
\]

Add the following instruction at the end

\[
\text{ERROR RTS Record not in the list}
\]
PART III: Intel IA-32

3.51. Initial memory contents are:

\[
\begin{align*}
[1000] &= 1 \\
[1004] &= 2 \\
[1008] &= 3 \\
[1012] &= 4 \\
[1016] &= 5 \\
[1020] &= 6
\end{align*}
\]

(a) \([\text{EBX} + \text{ESI}\times4 + 8] = 1016\)
EAX ← 10 + 5 = 15

(b) The values 20 and 30 are pushed onto the processor stack, and then 30 is popped into EAX and 20 is popped into EBX. The Subtract instruction then performs 30 − 20, and places the result of 10 into EAX.

(c) The address value 1008 is loaded into EAX, and then 3 is loaded into EBX.

3.52. (a) OK
(b) ERROR: Only one operand can be in memory.
(c) OK
(d) ERROR: Scale factor can only be 1, 2, 4, or 8.
(e) OK
(f) ERROR: An immediate operand can not be a destination.
(g) ERROR: ESP cannot be used as an index register.

3.53. Program trace:

<table>
<thead>
<tr>
<th>TIME</th>
<th>EAX</th>
<th>EBX</th>
<th>ECX</th>
</tr>
</thead>
<tbody>
<tr>
<td>After 1st execution of LOOP</td>
<td>−113</td>
<td>NUM1 − 4</td>
<td>4</td>
</tr>
<tr>
<td>After 2nd execution of LOOP</td>
<td>129</td>
<td>NUM1 − 4</td>
<td>3</td>
</tr>
<tr>
<td>After 3rd execution of LOOP</td>
<td>78</td>
<td>NUM1 − 4</td>
<td>2</td>
</tr>
</tbody>
</table>
3.54. Assume bytes are unsigned 8-bit values.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV ECX,N</td>
<td>ECX is list counter.</td>
</tr>
<tr>
<td>LEA ESI,X</td>
<td>ESI points to X list.</td>
</tr>
<tr>
<td>SUB ESI,1</td>
<td></td>
</tr>
<tr>
<td>LEA EDI,Y</td>
<td>EDI points to Y list.</td>
</tr>
<tr>
<td>SUB EDI,1</td>
<td></td>
</tr>
<tr>
<td>LEA EDX,LARGER</td>
<td>EDX points to LARGER list.</td>
</tr>
<tr>
<td>SUB EDX,1</td>
<td></td>
</tr>
<tr>
<td>START: MOV AL,[ESI + ECX]</td>
<td>Load X byte into AL.</td>
</tr>
<tr>
<td>MOV BL,[EDI + ECX],</td>
<td>Load Y byte into BL.</td>
</tr>
<tr>
<td>CMP AL,BL</td>
<td>Compare bytes.</td>
</tr>
<tr>
<td>JAE XLARGER</td>
<td>Branch if X byte larger or same.</td>
</tr>
<tr>
<td>MOV [EDX + ECX],BL</td>
<td>Otherwise, store Y byte.</td>
</tr>
<tr>
<td>JMP CHECK</td>
<td></td>
</tr>
<tr>
<td>XLARGER MOV [EDX + ECX],AL</td>
<td>Store X byte.</td>
</tr>
<tr>
<td>CHECK LOOP START</td>
<td>Check if done.</td>
</tr>
</tbody>
</table>
3.55. The inner loop checks for a match at each possible position.

```
MOV EDX,N          Compute outer loop count
SUB EDX,M          and store in EDX.
INC EDX            
LEA EAX,STRING     Use EAX as a base
                   pointer for each match
                   attempt.

OUTER: MOV ESI,EAX Use ESI and EDI as
LEA EDI,STRING     running pointers for
                   each match attempt.
                   
INNER: MOV ECX,M   Initialize inner loop counter.
       MOV BL,[EDI]   Load next substring byte
       CMP BL,[ESI]   into BL and compare to
       JNE NOMATCH   corresponding string byte.
       INC ESI       If not equal, go to
                   next substring position.
       INC EDI       If equal, increment running
                   pointers to next byte
                   positions.
       LOOP INNER   Check if all substring
                   bytes compared.
       JMP NEXT     If a match is found,
                   exit with string position
                   in EAX.

NOMATCH: INC EAX   Increment EAX to next possible
           substring position.
           
DEC EDX           Check if all positions tried.
JG OUTER
MOV EAX,0         If yes, load zero into
                   EAX and exit.

NEXT:  ...          
```
3.56. This solution assumes that the last number in the series of \( n \) numbers can be represented in a 32-bit doubleword, and that \( n > 2 \).

\[
\begin{align*}
\text{MOV} & \quad \text{ECX}, N \quad \text{Use ECX to count numbers} \\
\text{SUB} & \quad \text{ECX}, 2 \quad \text{generated after 1.} \\
\text{LEA} & \quad \text{EDI}, \text{MEMLOC} \quad \text{Use EDI as a memory pointer.} \\
\text{MOV} & \quad \text{EAX}, 0 \quad \text{Store first two numbers} \\
\text{MOV} & \quad \text{[EDI], EAX} \quad \text{from EAX and EBX into memory.} \\
\text{MOV} & \quad \text{EBX}, 1 \\
\text{ADD} & \quad \text{EDI}, 4 \\
\text{MOV} & \quad \text{[EDI], EBX} \\
\text{LOOPSTART:} & \quad \text{ADD} \quad \text{EDI}, 4 \quad \text{Increment memory pointer.} \\
\text{MOV} & \quad \text{EAX}, [\text{EDI} - 8] \quad \text{Load second last value.} \\
\text{ADD} & \quad \text{EBX}, \text{EAX} \quad \text{Add to last value.} \\
\text{MOV} & \quad \text{[EDI], EBX} \quad \text{Store new value.} \\
\text{LOOP} & \quad \text{LOOPSTART} \quad \text{Check if all \( n \) numbers generated.}
\end{align*}
\]

3.57. Assume register EAX contains the address (WORD) of the first character. To change characters from lowercase to uppercase, change bit \( b_5 \) from 1 to 0.

\[
\begin{align*}
\text{NEXT:} & \quad \text{MOV} \quad \text{BL}, [\text{EAX}] \quad \text{Load next character into BL.} \\
\text{CMP} & \quad \text{BL}, 20H \quad \text{Check if space character.} \\
\text{JE} & \quad \text{END} \quad \text{If space, exit.} \\
\text{AND} & \quad \text{BL}, DFH \quad \text{Clear bit \( b_5 \).} \\
\text{MOV} & \quad \text{[EAX], BL} \quad \text{Store converted character.} \\
\text{INC} & \quad \text{EAX} \quad \text{Increment memory pointer.} \\
\text{JMP} & \quad \text{NEXT} \quad \text{Convert next character.} \\
\text{END:} & \quad \ldots
\end{align*}
\]
3.58. The parameter Stride = \((j + 1)\) is the distance in doublewords between scores on a particular test for adjacent students in the list.

\[
\begin{align*}
\text{MOV} & \quad \text{EDX, J} \quad \text{Load outer loop counter EDX.} \\
\text{INC} & \quad \text{J} \quad \text{Increment memory location J to contain Stride = } j + 1. \\
\text{LEA} & \quad \text{EBX, SUM} \quad \text{Load address SUM into EBX.} \\
\text{LEA} & \quad \text{EDI, LIST} \quad \text{Load address of test score 1 for student 1 into EDI.} \\
\text{ADD} & \quad \text{EDI, 4} \quad \text{for student 1 into EDI.} \\
\text{OUTER:} & \quad \text{MOV} \quad \text{ECX, N} \quad \text{Load inner loop counter ECX.} \\
& \quad \text{MOV} \quad \text{EAX, 0} \quad \text{Clear scores accumulator EAX.} \\
& \quad \text{MOV} \quad \text{ESI, 0} \quad \text{Clear index register ESI.} \\
\text{INNER:} & \quad \text{ADD} \quad \text{EAX, [EDI + ESI*4]} \quad \text{Add next test score.} \\
& \quad \text{ADD} \quad \text{ESI, J} \quad \text{Increment index register ESI by Stride value.} \\
& \quad \text{LOOP} \quad \text{INNER} \quad \text{Check if all } n \text{ scores have been added.} \\
& \quad \text{MOV} \quad \text{[EBX], EAX} \quad \text{Store current test sum.} \\
& \quad \text{ADD} \quad \text{EBX, 4} \quad \text{Increment sum location pointer.} \\
& \quad \text{ADD} \quad \text{EDI, 4} \quad \text{Increment base pointer to next test score for student 1.} \\
& \quad \text{DEC} \quad \text{EDX} \quad \text{Check if all test scores summed.} \\
& \quad \text{JG} \quad \text{OUTER}
\end{align*}
\]

This solution uses six of the IA-32 registers. It does not use registers EBP or ESP, which are normally reserved as pointers for the processor stack. Use of EBP to hold the parameter Stride would result in a somewhat more efficient inner loop.

3.59. Use register ECX as a counter register, and use EBX as a work register.

\[
\begin{align*}
\text{MOV} & \quad \text{ECX, 32} \quad \text{Load ECX with count value 32.} \\
\text{MOV} & \quad \text{EBX, 0} \quad \text{Clear work register EBX.} \\
\text{LOOPSTART:} & \quad \text{SHL} \quad \text{EAX, 1} \quad \text{Shift contents of EAX left one bit position, moving the high-order bit into the CF flag.} \\
& \quad \text{RCR} \quad \text{EBX, 1} \quad \text{Rotate EBX right one bit position, including the CF flag.} \\
& \quad \text{LOOP} \quad \text{LOOPSTART} \quad \text{Check if finished.} \\
& \quad \text{MOV} \quad \text{EAX, EBX} \quad \text{Load reversed pattern into EAX.}
\end{align*}
\]
3.60. See the solution to Problem 2.18 for the procedures needed to perform the append and remove operations.

Register assignment:

- **AL** – Data byte to append to or remove from the queue
- **ESI** – IN pointer
- **EDI** – OUT pointer
- **EBX** – Address of first queue byte location
- **ECX** – Address of last queue byte location (\([EBX] + k − 1\) )
- **EDX** – Auxiliary register for location of next appended byte

Initially, the queue is empty with \([ESI] = [EDI] = [EBX]\).

Append routine:

- **MOV EDX,ESI** – Save current value of IN pointer ESI in auxiliary register EDX.
- **INC ESI** – These four instructions increment ESI Modulo \(k\).
- **CMP ECX,ESI** – CHECK
- **JGE CHECK** – MOV ESI,EBX
- **MOV ESI,EDX** – Check if queue is full.
- **JNE APPEND** – Otherwise, restore IN pointer and send message that queue is full.
- **APPEND: MOV [EDX],AL** – Append byte.

Remove routine:

- **CMP EDI,ESI** – Check if queue is empty.
- **JE QUEUEEMPTY** – If empty, send message.
- **MOV AL,[EDI]** – Otherwise, remove byte and increment EDI Modulo \(k\).
- **INC EDI**
- **CMP ECX,EDI**
- **JGE NEXT**
- **MOV EDI,EBX**
- **NEXT: \ldots**
3.61. This program is similar to Figure 3.44; and it makes the same assumptions about status word bit locations.

MOV ECX,N  Use ECX as the loop counter.
READ: BT INSTATUS,3  Wait for the character.
JNC READ
MOV AL,DATAIN  Transfer character into AL.
DEC EBX  Push character onto user stack.
MOV [EBX],AL
ECHO: BT OUTSTATUS,3  Wait for the display.
JNC ECHO
MOV DATAOUT,AL  Send character to display.
LOOP READ  Check if all \( n \) characters read.

3.62. Assume that most of the time between successive characters being struck is spent in the two-instruction wait loop that starts at location READ. The JNC READ instruction is executed once every 20 ns while this loop is being executed. There are \( 10^9/10 = 10^8 \) ns between successive characters. Therefore, the JNC READ instruction is executed \( 10^8/20 = 5 \times 10^6 \) times per character entered.

3.63 Assume that register ECX is used as a memory pointer by the main program.

**Main Program**

READLINE: CALL GETCHAR
MOV [ECX],AL  Store character in memory.
INC ECX  Increment memory pointer.
CALL PUTCHAR
CMP AL,CR  Check for end-of-line.
JNE READLINE  Go back for more.

**Subroutine GETCHAR**

GETCHAR: BT DWORD PTR [EBX],3  Wait for character.
JNC GETCHAR
MOV AL,[EDX]  Load character into AL.
RET

**Subroutine PUTCHAR**

PUTCHAR: BT DWORD PTR [ESI],3  Wait for display.
JNC PUTCHAR
MOV [EDI],AL  Display character.
RET
3.64. Addresses INSTATUS and DATAIN are pushed onto the processor stack in that order by the main program as parameters for GETCHAR. The character read is passed back to the main program in the DATAIN position on the stack. The addresses OUTSTATUS and DATAOUT and the character to be displayed are pushed onto the processor stack in that order by the main program as parameters for PUTCHAR. A stack structure like that shown in Figure 3.46 is used.

GETCHAR uses registers EBX, EDX, and AL (EAX) to hold INSTATUS, DATAIN, and the character read.

PUTCHAR uses registers ESI, EDI, and AL (EAX) to hold OUTSTATUS, DATAOUT, and the character to be displayed.

Assume that register ECX is used as a memory pointer by the main program.

Main Program

```
READLINE:  PUSH OFFSET INSTATUS  Push address parameters
           PUSH OFFSET DATAIN    onto the stack.
           CALL GETCHAR
           POP EAX
           MOV [ECX],AL
           INC ECX
           ADD ESP,4
           PUSH OFFSET OUTSTATUS
           PUSH OFFSET DATAOUT
           PUSH EAX
           CALL PUTCHAR
           ADD ESP,12
           CMP AL,CR
           JNE READLINE
```


Subroutine GETCHAR

GETCHAR:  PUSH  EAX  
           PUSH  EBX  Save registers to be 
           PUSH  EDX  used in the subroutine. 
           MOV  EBX,[ESP + 20]  Load INSTATUS into EBX. 
           MOV  EDX,[ESP + 16]  Load DATAIN into EDX. 

READ:  BT  DWORD PTR [EBX],3  Wait for character. 
        JNC  READ  
        MOV  AL,[EDX]  Read character into AL. 
        MOV  [ESP + 16],EAX  Overwrite DATAIN in the 
                          stack with the doubleword 
                          containing the character read. 
        POP  EDX  Restore registers. 
        POP  EBX  
        POP  EAX  
        RET  

Subroutine PUTCHAR

PUTCHAR:  PUSH  EAX  Save registers to be 
           PUSH  ESI  used in the subroutine. 
           PUSH  EDI  
           MOV  ESI,[ESP + 24]  Load OUTSTATUS. 
           MOV  EDI,[ESP + 20]  Load DATAOUT. 
           MOV  EAX,[ESP + 16]  Load doubleword containing 
                          character to be displayed 
                          into register EAX. 

DISPLAY:  BT  DWORD PTR [ESI],3  Wait for the display. 
           JNC  DISPLAY  
           MOV  [EDI],AL  Display character. 
           POP  EDI  Restore registers. 
           POP  ESI  
           POP  EAX  
           RET  

39
3.65. Using the same assumptions as in Problem 3.61 and its solution, an IA-32 program to convert 3 input decimal digits to a binary number is:

```
CALL READ  Get first character
MOV EBX,[HUNDREDS + EAX * 4] Get hundreds value
CALL READ  Get second character
ADD EBX,[TENS + EAX * 4]  Add tens value
CALL READ  Get third character
ADD EBX,EAX EBX contains value of binary number

READ: BT INSTATUS,3  Wait for new character
JNC READ Get new character
MOV AL,DATAIN
AND AL,0FH  Convert to equivalent binary value
RET
```
3.66. (a) The subroutines convert 3 decimal digits to a binary value.

GETCHARS:  PUSH ECX  \hspace{1cm} \text{Save registers.}
            PUSH EBX
            PUSH EAX
            MOV ECX,3  \hspace{1cm} \text{Use ECX as character counter.}
            MOV EBX,[ESP + 20]  \hspace{1cm} \text{Load character buffer address into EBX.}

READ:      BT INSTATUS,3  \hspace{1cm} \text{Get and store character.}
            JNC READ  \hspace{1cm} \text{Increment buffer pointer.}
            MOV BYTE PTR [EBX],DATAIN  \hspace{1cm} \text{Repeat until all characters received.}
            INC EBX  \hspace{1cm} \text{Pointer to result.}
            LOOP READ
            MOV EAX,[ESP + 16]  \hspace{1cm} \text{Restore registers.}
            CALL CONVERT
            POP EAX
            POP EBX
            POP ECX
            RET

CONVERT:   PUSH ECX  \hspace{1cm} \text{Save registers.}
            PUSH EDX
            DEC EBX  \hspace{1cm} \text{Load low-order digit numerical value into EDX.}
            MOV DL,[EBX]
            AND DL,0FH
            DEC EBX  \hspace{1cm} \text{Load and add tens digit value into EDX.}
            MOV CL,[EBX]
            AND CL,0FH
            ADD EDX,[TENS + ECX * 4]
            DEC EBX  \hspace{1cm} \text{Load and add hundreds digit value into EDX.}
            MOV CL,[EBX]
            AND CL,0FH
            ADD EDX,[HUNDREDS + ECX * 4]
            MOV [EAX],EDX  \hspace{1cm} \text{Store result.}
            POP EDX
            POP ECX
            RET
(b) The contents of the top of the stack after the call to the CONVERT subroutine are:

<table>
<thead>
<tr>
<th>Return address to GETCHARS</th>
</tr>
</thead>
<tbody>
<tr>
<td>[EAX]</td>
</tr>
<tr>
<td>[EBX]</td>
</tr>
<tr>
<td>[ECX]</td>
</tr>
</tbody>
</table>

Return address to Main
Result address
Buffer address
ORIGINAL TOS

3.67. Assume that the subroutine can change the contents of any registers used to pass parameters. Let Stride = 4m, which is the distance in bytes between successive doubleword elements in a given column.

```
SHL EBX,2                   Set Stride in EBX.
SUB EDI,ESI                 Set EDI to y - x.
SHL ESI,2                   Set EDX to address A(0,x).
ADD EDX,ESI                 Add A(i,x) to A(i,y).
       ADD [EDX + EDI * 4],ESI
       ADD EDX,EBX
       DEC EAX
       JG LOOP
       RET
```

3.68. Program trace:

<table>
<thead>
<tr>
<th>TIME</th>
<th>EDI</th>
<th>ECX</th>
<th>DL</th>
<th>LIST</th>
<th>LIST</th>
<th>LIST</th>
<th>LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+1</td>
<td>+2</td>
<td>+3</td>
<td>+4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>After 1st</td>
<td>3</td>
<td>-1</td>
<td>120</td>
<td>106</td>
<td>13</td>
<td>67</td>
<td>45</td>
</tr>
<tr>
<td>After 2nd</td>
<td>2</td>
<td>-1</td>
<td>106</td>
<td>67</td>
<td>13</td>
<td>45</td>
<td>106</td>
</tr>
<tr>
<td>After 3rd</td>
<td>1</td>
<td>-1</td>
<td>67</td>
<td>45</td>
<td>13</td>
<td>67</td>
<td>106</td>
</tr>
<tr>
<td>After 4th</td>
<td>0</td>
<td>-1</td>
<td>45</td>
<td>13</td>
<td>45</td>
<td>67</td>
<td>106</td>
</tr>
</tbody>
</table>
3.69. Assume that the calling program passes the address LIST − 4 to the subroutine in register EAX.

Subroutine SORT

```
SORT:  PUSH EDI  Save registers.
    PUSH ECX
    PUSH EDX
    MOV EDI,[EAX] Initialize outer loop index
    DEC EDI register EDI to \( j = n - 1 \).
    ADD EAX,4 Set EAX to contain LIST.

OUTER: MOV ECX,EDI Initialize inner loop index
    DEC ECX register to \( k = j - 1 \).
    MOV EDX,[EAX + EDI * 4] Load LIST(\( j \)) into EDX.

INNER: CMP [EAX + ECX * 4],EDX Compare LIST(\( k \)) to LIST(\( j \)).
    JLE NEXT If LIST(\( k \)) \( \leq \) LIST(\( j \)),
        go to next \( k \) index entry;
    XCHG [EAX + ECX * 4],EDX Otherwise, interchange LIST(\( k \))
    MOV [EAX + EDI * 4],EDX and LIST(\( j \)), leaving
        (new) LIST(\( j \)) in EDX.

NEXT:  DEC ECX Decrement inner loop index \( k \).
    JGE INNER Repeat or terminate inner loop.
    DEC EDI Decrement outer loop index \( j \).
    JG OUTER Repeat or terminate outer loop.
    POP EDX Restore registers.
    POP ECX
    POP EDI
    RET
```
3.70. Use register ESI to keep track of the index position of the largest element in the inner loop, and use register EDX (DL) to record its value. Register EBX (BL) is used to hold sublist values to be compared to the current largest value.

```
LEA EAX, LIST
MOV EDI, N
DEC EDI
OUTER: MOV ECX, EDI
      DEC ECX
      MOV ESI, EDI Initial index of largest.
      MOV DL, [EAX + EDI] Initial value of largest.
      CMP BL, DL Compare to current largest.
      JLE NEXT If not larger, check next;
      MOV DL, BL Otherwise, update largest
      MOV ESI, ECX and update its index.
      NEXT: DEC ECX Repeat or terminate
      JGE INNER inner loop.
      XCHG [EAX + EDI], DL Interchange LIST(j)
      MOV [EAX + ESI], DL with LIST([ESI]).
      DEC EDI Repeat or terminate
      JG OUTER outer loop.
```

The potential advantage is that the inner loop should execute faster.

3.71. Assume that register ESI points to the first record, and use registers EAX, EBX, and ECX, to accumulate the three sums.

```
MOV EAX, 0
MOV EBX, 0
MOV ECX, 0
      CMP ESI, 0 Check if done.
      JNE LOOP
      MOV SUM1, EAX Store sums.
      MOV SUM2, EBX
      MOV SUM3, ECX
```
3.72. If the ID of the new record matches the ID of the Head record of the
current list, the new record will be inserted as the new Head. If the ID
of the new record matches the ID of a later record in the current list, the
new record will be inserted immediately after that record, including the
case where the matching record is the Tail record. In this latter case, the
new record becomes the new Tail record.

Modify Figure 3.51 as follows:

- Add the following instruction as the first instruction of the subrou-
tine:

  \[
  \text{INSERTION: MOV EDX, 0} \quad \text{Anticipate successful}
  \]
  \[
  \text{MOV RNEWID,}[RNEWREC] \quad \text{insertion of the new}
  \]
  \[
  \text{record.} \quad \text{(Existing instruction.)}
  \]

- After the second CMP instruction, insert the following three instruc-
tions:

  \[
  \text{JNE CONTINUE1 Three new instructions.}
  \]
  \[
  \text{MOV EDX,RHEAD}
  \]
  \[
  \text{RET}
  \]
  \[
  \text{CONTINUE1: JG SEARCH (Existing instruction.)}
  \]

- After the fourth CMP instruction, insert the following three instruc-
tions:

  \[
  \text{JNE CONTINUE2 Three new instructions.}
  \]
  \[
  \text{MOV EDX,RNEXT}
  \]
  \[
  \text{RET}
  \]
  \[
  \text{CONTINUE2: JL INSERT (Existing instruction.)}
  \]
3.73. If the list is empty, the result is unpredictable because the first instruction will compare the ID of the new record to the contents of memory location zero. If the list is not empty, the following happens. If the contents of RIDNUM are less than the ID number of the Head record, the Head record will be deleted. Otherwise, the routine loops until register RCURRENT points to the Tail record. Then RNEXT gets loaded with zero by the instruction at LOOPSTART, and the result is unpredictable.

Replace Figure 3.52 with the following code:

```
DELETION:  CMP RHEAD, 0 If the list is empty,  
            JNE CHECKHEAD return with RIDNUM  
            RET unchanged.

CHECKHEAD: CMP RIDNUM,[RHEAD] Check if Head record  
            JNE CONTINUE1 is to be deleted and  
            MOV RHEAD,[RHEAD + 4] perform deletion if it  
            MOV RIDNUM,0 is, returning with zero  
            RET in RIDNUM.

CONTINUE1: MOV RCURRENT,RHEAD Otherwise, continue  
            search.

LOOPSTART: MOV RNEXT,[RCURRENT + 4] If all records checked,  
            CMP RNEXT,0 return with IDNUM  
            JNE CHECKNEXT unchanged.  
            RET

CHECKNEXT: CMP RIDNUM,[RNEXT] Check if next record is  
            JNE CONTINUE2 to be deleted and  
            MOV RTEMP,[RNEXT + 4] perform deletion if  
            MOV [RCURRENT + 4],RTEMP it is, returning with  
            MOV RIDNUM,0 zero in RIDNUM.  
            RET

CONTINUE2: MOV RCURRENT,RNEXT Otherwise, continue  
            JMP LOOPSTART the search.
```