Appendix C

The ColdFire Processor

C.1. Instructions may include 32-bit constants, hence the operands need not be stored separately as data in memory.

```
        MOVE.L   #580, D0       Load 580.
        ADD.L    #6840, D0      Generate 580 + 6840.
        ADD.L    #80000, D0     Generate the final sum.
        MOVE.L   D0, SUM        Store the sum.
```

C.2. Assuming that the operands are to be interpreted as signed integers, the MULS instruction is used. Furthermore, it is assumed that all operands are 32 bits in size, and that the numerical values are such that the truncation of the product by the MULS instruction does not yield incorrect results.

```
           MOVE.L   A, D0        Load the operand A.
           MOVE.L   B, D1        Load the operand B.
           MULS.L   D1, D0       D0 set to D0 * D1 (A * B).
           MOVE.L   C, D2        Load the operand C.
           MOVE.L   D, D3        Load the operand D.
           MULS.L   D3, D2       D2 set to D2 * D3 (C * D).
           ADD.L    D2, D0       Generate sum of two products.
           MOVE.L   D0, ANSWER   Store the answer.

A:         .DC.L    100          Test data.
B:         .DC.L    50
C:         .DC.L    20
D:         .DC.L    400
ANSWER:    .DS.L    1            Space for the sum.
```

C.3. This program uses a loop and the Autoincrement addressing mode to count negative numbers that are found in the list. The TST

```
          MOVE.L    N, D0            Load the size of the list.
          CLR.L     D1               Initialize the counter to 0.
          MOVEA.L   #NUMBERS, A0     Load address of the first number.
LOOP:     MOVE.L    (A0)+, D2        Get the next number and increment pointer.
          BGE       NEXT             Test if number just loaded is negative.
          ADDQ.L    #1, D1           Increment the count.
NEXT:     SUBQ.L    #1, D0           Decrement the list counter.
          BGT       LOOP             Loop back if not finished.
          MOVE.L    D1, NEGNUM       Store the result.

NEGNUM:   .DS.L     1                Space for the result.
N:        .DC.L     6                Size of list.
NUMBERS:  .DC.L     23, −5, −128     Test data.
          .DC.L     44, −23, −9
```

C.4. In this program, three separate sums are maintained as the list of records is processed by the loop using a single pointer.

```
          MOVEA.L   #LIST, A0          Get the address LIST.
          CLR.L     D1
          CLR.L     D2
          CLR.L     D3
          MOVE.L    N, D7              Load the value n.
LOOP:     ADD.L     4(A0), D1          Add current student mark for Test 1.
          ADD.L     8(A0), D2          Add current student mark for Test 2.
          ADD.L     12(A0), D3         Add current student mark for Test 3.
          ADDA.L    #16, A0            Increment the pointer.
          SUBQ.L    #1, D7             Decrement the counter.
          BGT       LOOP               Loop back if not finished.
          MOVE.L    D1, SUM1           Store the total for Test 1.
          MOVE.L    D2, SUM2           Store the total for Test 2.
          MOVE.L    D3, SUM3           Store the total for Test 3.

SUM1:     .DS.L     1                  Space for SUM1.
SUM2:     .DS.L     1                  Space for SUM2.
SUM3:     .DS.L     1                  Space for SUM3.
N:        .DC.L     3                  Size of the list.
LIST:     .DC.L     1234, 62, 85, 75   Example records.
          .DC.L     1235, 90, 82, 88
          .DC.L     1236, 72, 65, 80
```

C.5. 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 2.10. The parameter Stride = $4(j+1)$ is the distance in bytes between scores on a particular test for adjacent students in the list. The program below processes the scores in reverse order so as to use the scaled value of the decrementing outer loop counter register for indexed addressing.

```
        MOVE.L    J, D0           Compute and place
        ADDQ.L    #1, D0            Stride = 4(j + 1)
        LSL.L     #2, D0            into register A0.
        MOVEA.L   D0, A0
        MOVEA.L   #LIST, A1       Initialize register A1 to the location
        ADDA.L    #4, A1            of the test 1 score for student 1.
        MOVEA.L   #SUM, A2        Initialize register A2 to the location
                                    of the sum for test 1.
        MOVE.L    J, D7           Initialize outer loop counter D7 to j.
OUTER:  CLR.L     D0              Clear the sum register D0.
        MOVEA.L   A1, A3          Use A3 as an index register.
        MOVE.L    D7, D5          Scale the outer loop counter by 4.
        LSL.L     #2, D5
        MOVE.L    N, D6           Initialize inner loop counter D6 to n.
INNER:  ADD.L     −4(A3,D5), D0   Accumulate the sum of test scores.
        ADDA.L    A0, A3          Increment index register by Stride value.
        SUBQ.L    #1, D6          Check if all student scores on current
        BGT       INNER             test have been accumulated.
        MOVE.L    D0, −4(A2,D5)   Store sum of current test scores.
        SUBQ.L    #1, D7          Check if the sums for all tests have
        BGT       OUTER             been computed.
        next instruction
```

C.6. To produce the correct list order, this program processes the list of byte-sized items from the end of the list to the beginning in the outer loop. The inner loop then works from the current position to beginning of the list to move items with lower value to the beginning of the list.

```
        MOVEA.L  #LIST, A0      Get the address LIST.
        MOVE.L   N, D0          Get the number of elements N.
        MOVEA.L  A0, A1         Initialize outer loop pointer
        ADDA.L   D0, A1           to LIST + n.
        CLR.L    D1             Clear registers used in byte comparisons.
        CLR.L    D2
OUTER:  ADDA.L   #−1, A1        Decrement the pointer.
        CMPA.L   A0, A1         Check if last entry.
        BLS      DONE
        MOVE.B   (A1), D1       Starting max value in sublist.
        MOVEA.L  A1, A2         Initialize inner loop pointer.
        ADDA.L   #−1, A2
INNER:  MOVE.B   (A2), D2       Check if the next entry
        CMP.L    D2, D1           is lower.
        BGE      NEXT
        MOVE.B   D2, (A1)       If yes, then swap
        MOVE.B   D1, (A2)         the entries and
        MOVE.B   D2, D1           update the max value.
NEXT:   ADDA.L   #−1, A2        Adjust the inner loop pointer.
        CMPA.L   A0, A2
        BHS      INNER
        JMP      OUTER
DONE:   next instruction

N:      .DC.L    10             Size of the list.
LIST:   .DC.B    'zZbB53kK24'   Test data.
```

C.7. The DISPLAY routine is invoked when a timer interrupt occurs. The main program prepares the timer registers appropriately, then proceeds to the COMPUTE task.

```
TIM_STATUS  .EQU      $4020
Interrupt-service routine
ILOC:       MOVE.L    D0, −(A7)          Save register.
            MOVE.B    TIM_STATUS, D0     Clear TIRQ and ZERO bits in status register.
            JSR       DISPLAY            Call the DISPLAY routine.
RTRN:       MOVE.L    (A7)+, D0          Restore register.
            RTE                          Return from interrupt.

Main program
START:      . . .                        Set up parameters for interrupts.
            MOVE.L    #$3B9ACA00, D0     Prepare the initial count value.
            MOVEA.L   #TIM_STATUS, A0
            MOVE.L    D0, 8(A0)          Set the initial count value.
            MOVEQ.L   #7, D0             Set the timer to free run and enable interrupts.
            MOVE.B    D0, 4(A0)
            MOVE.W    SR, D0             Read contents of status register into D0.
            ANDI.L    #$F8FF, D0         Clear bits for interrupt priority level mask,
                                           which effectively enables all interrupts.
            MOVE.W    D0, SR             Write value of D0 to status register.
COMPUTE:    next instruction
```

C.8. This program performs a lookup in a table of patterns to show a given decimal digit on a 7-segment display.

```
DIGIT    .EQU      $800                 Location of ASCII-encoded digit.
SEVEN    .EQU      $4030                Address of 7-segment display.

         MOVE.B    DIGIT, D0            Load the ASCII-encoded digit.
         MOVE.B    D0, D1
         AND.L     #$0F, D0             Extract the decimal number.
         AND.L     #$F0, D1             Extract high-order bits of ASCII.
         CMP.L     #$30, D1             Check if high-order bits of
         BEQ       HIGH3                  ASCII code are 0011.
         MOVEQ.L   #$0F, D0             Not a digit, display a blank.
HIGH3:   MOVEA.L   D0, A0               Prepare offset.
         MOVE.B    TABLE(A0), D1        Get the 7-segment pattern.
         MOVE.B    D1, SEVEN            Display the digit.

         .ORG      $1000
TABLE:   .DC.B     $7E,$30,$6D,$79      Table that contains
         .DC.B     $33,$5B,$5F,$70        the necessary
         .DC.B     $7F,$7B,$00,$00        7-segment patterns.
         .DC.B     $00,$00,$00,$00
```

C.9 The addressing modes of the ColdFire processor use 8-bit or 16-bit displacements. The address of $10100 for TABLE requires more than 16 bits to represent. It is therefore necessary to use a different instruction sequence than the one used in Problem C.8 for retrieving a 7-segment pattern from the table. The revised program is shown below.

```
DIGIT    .EQU      $800                 Location of ASCII-encoded digit.
SEVEN    .EQU      $4030                Address of 7-segment display.

         MOVE.B    DIGIT, D0            Load the ASCII-encoded digit.
         MOVE.B    D0, D1
         AND.L     #$0F, D0             Extract the decimal number.
         AND.L     #$F0, D1             Extract high-order bits of ASCII.
         CMP.L     #$30, D1             Check if high-order bits of
         BEQ       HIGH3                  ASCII code are 0011.
         MOVEQ.L   #$0F, D0             Not a digit, display a blank.
HIGH3:   MOVEA.L   #TABLE, A0           Prepare pointer and add to offset
         MOVE.B    (A0,D0), D1            to get the 7-segment pattern.
         MOVE.B    D1, SEVEN            Display the digit.

         .ORG      $10100
TABLE:   .DC.B     $7E,$30,$6D,$79      Table that contains
         .DC.B     $33,$5B,$5F,$70        the necessary
         .DC.B     $7F,$7B,$00,$00        7-segment patterns.
         .DC.B     $00,$00,$00,$00
```

C.10. The following program assumes that the display device interface has the registers shown in Figure 3.3.

```
         MOVEA.L   #LOC, A0          Get the address LOC.
         MOVEA.L   #DISP_DATA, A1    Get the address of display.
         MOVEA.L   #TABLE, A2        Get the table address.
         MOVEQ.L   #10, D0           Initialize the byte counter.
         CLR.L     D1                Clear all bits in register D1.
LOOP:    MOVE.B    (A0)+, D2         Load a byte and increment pointer.
         MOVE.B    D2, D1            Make a copy of the byte.
         AND.L     #$F0, D1          Select the high-order 4 bits.
         ASR.L     #4, D1            Shift right by 4 bit positions.
         MOVE.B    (A2,D1), D1       Get the character for display.
         JSR       DISPLAY
         MOVE.B    D2, D1            Make a copy of the original byte.
         AND.L     #$0F, D1          Select the low-order 4 bits.
         MOVE.B    (A2,D1), D1       Get the character for display.
         JSR       DISPLAY
         MOVEQ.L   #$20, D1          ASCII code for SPACE.
         JSR       DISPLAY
         SUBQ.L    #1, D0            Decrement the byte counter.
         BGT       LOOP              Branch back if not finished.
         next instruction

DISPLAY: MOVE.B    4(A1), D3
         AND.L     #4, D3            Check the DOUT flag.
         BEQ       DISPLAY
         MOVE.B    D1, (A1)          Send the character to display.
         RTS

TABLE:   .DC.B     $30,$31,$32,$33   Table that contains
         .DC.B     $34,$35,$36,$37     the necessary
         .DC.B     $38,$39,$41,$42     ASCII characters.
         .DC.B     $43,$44,$45,$46
```

C.11. The following program assumes that the display device interface has the registers shown in Figure 3.3.

```
            MOVE.L    BINARY, D0       Load the 16-bit pattern from the address BINARY.
            MOVEA.L   #DISP_DATA, A1   Get the address of display.
            MOVEQ.L   #16, D1          Initialize the bit counter.
            MOVE.L    #$8000, D2       Set bit 15 to 1 (and clear upper 16 bits).
LOOP:       MOVE.W    D0, D3           Make a copy of the pattern.
            AND.L     D2, D3           Test a bit.
            BEQ       ZERO             Check if 0 or 1, and
            MOVEQ.L   #$31, D4         set ASCII character value.
            JMP       CONT
ZERO:       MOVEQ.L   #$30, D4
CONT:       JSR       DISPLAY
            LSR.L     D2, 1            Shift to check the next bit.
            SUBQ.L    #1, D1           Decrement the bit counter.
            BGT       LOOP             Branch back if not finished.
            next instruction

DISPLAY:    MOVE.B    4(A1), D5
            AND.L     #4, D5           Check the DOUT flag.
            BEQ       DISPLAY
            MOVE.B    D4, (A1)         Send the character to display.
            RTS
```

C.12. The following program configures the timer count register for one-second intervals and uses polling to detect the timer expiry for each interval.

```
TIMER     .EQU      $4020              Location of ASCII-encoded digit.
SEVEN     .EQU      $4030              Address of 7-segment display.

          MOVEA.L   #TIMER, A0         Set pointer to timer status register.
          MOVE.L    #$5F5E100, D0      Prepare the count value for one-second intervals.
          MOVEA.L   #TABLE, A1
          MOVE.L    D0, 8(A0)          Set the timer count value.
          MOVE.B    #6, 4(A0)          Start the timer in the continuous mode.
          CLR.L     D0                 Clear the digit counter.
LOOP:     MOVE.B    (A0), D1           Wait for timer to reach the
          ANDI.L    #2, D1               end of the one-second interval.
          BEQ       LOOP
          MOVE.B    (A1,D0), D1        Load the ASCII-encoded digit.
          MOVE.B    D1, SEVEN          Display the digit.
          ADDQ.L    #1, D1             Increment the digit counter,
          CMPI.L    #10, D1              and check if > 9.
          BLT       LOOP
          CLR.L     D1                 Clear the digit counter.
          BRA       LOOP

          .ORG      $1000
TABLE:    .DC.B     $7E,$30,$6D,$79    Table that contains
          .DC.B     $33,$5B,$5F,$70      the necessary
          .DC.B     $7F,$7B,$00,$00      7-segment patterns.
          .DC.B     $00,$00,$00,$00
```

C.15. The subroutines for safe operations on the second stack are provided below. Register D0 holds the element that is to be pushed or popped. Register A0 is the pointer to the second stack. The ability to perform comparisons with 32-bit immediate values and the availability of the auto-increment/auto-decrement addressing modes enables the subroutines to have a small number of instructions.

```
SPUSH:  CMPA.L   #TOP, A0
        BLS      FULLERROR
        MOVE.L   D0, −(A0)
        RTS

SPOP:   CMPA.L   #BOTTOM, A0
        BHS      EMPTYERROR
        MOVE.L   (A0)+, D0
        RTS
```

C.16. There is no ISA-specific code to write for this problem. Instead, a brief description is provided as a solution for each part.

(*a*) When the end of the memory region has been reached as a result of adding a succession of items to the queue, it is necessary to *wrap around* to the beginning of the memory region for the next item to be added. This approach assumes that the location at the beginning of memory is not occupied by a valid data item, i.e., the OUT pointer has advanced to a higher address.

(*b*) Assume a queue of bytes in a dedicated memory region with locations numbered from 1 to k. Each of these locations also has an address that is used by memory access instructions, but the number of 1 to k is used in the discussion for this problem.

The IN pointer identifies the location where the next byte will be appended to the queue. The append operation can only be performed if this location is empty, i.e., the queue does not presently contain k valid data items.

The OUT pointer identifies to the location containing the next byte to be removed from the queue. The remove operation can only be performed if this location contains a valid byte, i.e., if the queue is not presently empty.

The initial state of the queue is empty, and the IN and OUT pointers both identify location 1 at the beginning of the dedicated memory region for the queue.

(*c*) The initial state described in part *b* for an empty has both IN and OUT pointers identifying the same location. If the append operation is performed k times in succession with no remove operations, then all k of the locations in the dedicated memory region will be occupied with valid data items. The OUT pointer will still identify location 1, and the IN pointer will have been incremented to the end of the memory region and wrapped around to location 1 again. Thus, the situation after k successive append operations appears identical to the initial situation with an empty queue.

(*d*) Although it is possible to supplement the IN and OUT pointers with a counter that reflects the number of items presently in the queue, the issue highlighted in part *c* can also be addressed without additional variables.

It is reasonable to retain the same condition for an empty queue. Therefore, the source of the difficulty becomes the situation of the IN and OUT pointers identifying the same location when the queue is full with k items. This difficulty can be avoided by not allowing the queue to contain k valid items, even though k locations are allocated in the dedicated memory region. If the maximum number of items is limited to $k-1$, then the "full" state occurs when $([\text{IN}]+1) \bmod k = [\text{OUT}]$. In other words, the queue always has at least one empty location.

(*e*) Using the solution proposed in part *d* above, the following procedure can be specified for the append operation. The original value of the IN pointer is restored if it is determined that the queue is full.

- TMP_PTR ← [IN]
- IN $\leftarrow ([\text{IN}]+1) \bmod k$
- if ([IN] = [OUT]) then
  IN ← [TMP_PTR]
  indicate failed append due to full queue
  else
  store new item in location at address TMP_PTR

The following procedure implements the remove operation.

- if ([IN] = [OUT]) then
  indicate failed remove operation due to empty queue
  else
  read item in location at address OUT
  $\text{OUT} \leftarrow ([\text{OUT}]+1) \bmod k$

C.17. For the implementation of the APPEND and REMOVE tasks described in Problem C.16, all of the necessary information for managing the queue can be maintained in registers. The use of registers for these tasks is summarized below.

A0: the IN pointer

A1: the OUT pointer

A2: address of beginning of queue area in memory (does not change)

A3: address of end of queue area in memory (does not change)

D0: data item to be appended to or removed from queue

A4: temporary storage for IN pointer before incrementing for APPEND

The initial empty state of the queue with pointers identifying the the beginning of the reserved area in memory is reflected by having registers A0 and A1 contain the same value as register A2.

The instructions for the necessary APPEND and REMOVE routines are provided below. The size of each item is assumed to be one byte. The size of the queue area is assumed to be k items, but the start and end addresses are used in the implementation to achieve the desired modulo behavior.

```
APPEND:    MOVEA.L  A0, A4        Set temporary register to current IN pointer.
           ADDA.L   #1, A0        Increment IN pointer (modulo k).
           CMPA.L   A0, A3        Compare against end address.
           BHS      CHECK         Continue if within bounds.
           MOVEA.L  A2, A0        Otherwise, reset IN to beginning address.
CHECK:     CMPA.L   A1, A0        Check if queue is full.
           BEQ      FULL
           MOVE.B   D0, (A4)      If queue not full, append item.
           BRA      CONTINUE
FULL:      MOVEA.L  A4, A0        Restore IN pointer and indicate that
           JSR      QUEUEFULL       queue is full.
CONTINUE:  next instruction
```

```
REMOVE:    CMPA.L   A0, A1        Check if queue is empty.
           BEQ      EMPTY
           MOVE.B   (A1)+, D0     Remove byte at end of queue and
                                    increment OUT pointer (modulo k).
           CMPA.L   A1, A3
           BHS      CONTINUE
           MOVEA.L  A2, A1        Reset OUT to beginning address.
           BRA      CONTINUE
EMPTY:     JSR      QUEUEEMPTY    Indicate that queue is empty.
CONTINUE:  next instruction
```

C.18. The values for successive elements of the OUT array representing the signal samples can be computed by using right-shift operations, which are denoted using syntax similar to the C language as “>> *amount*” in the expression below.

$$\text{OUT}(k) = \text{IN}(k){>>}3 + \text{IN}(k+1){>>}2 + \text{IN}(k+2){>>}\ 1$$

The following program uses the above expression in a loop to generate the elements in the OUT array.

```
        MOVE.L    N, D7        Get n for number of entries to generate.
        MOVEA.L   #IN, A0      Pointer to the IN list.
        MOVEA.L   #OUT, A1     Pointer to the OUT list.
LOOP:   MOVE.L    (A0), D0     Get the value IN(k) and
        ASR.L     #3, D0         divide it by 8.
        MOVE.L    4(A0), D1    Get the value IN(k+1) and
        ASR.L     #2, D1         divide it by 4.
        ADD.L     D1, D0
        MOVE.L    8(A0), D1    Get the value IN(k+2) and
        ASR.L     #1, D1         divide it by 2.
        ADD.L     D1, D0       Compute the sum and store it
        MOVE.L    D0, (A1)+      in OUT list (with pointer increment).
        ADDA.L    #4, A0       Increment the pointer to IN list.
        SUBQ.L    #1, D7       Continue until all values in
        BGT       LOOP           OUT list have been generated.
        next instruction
```

C.19. The copy subroutine is called with three parameters in registers. It copies items in the forward direction, unless the starting address of the second list falls within the region of memory occupied by the first list. In that special case, items are copied in the reverse direction.

```
            MOVEA.L   #FIRST, A0    Pointer to first list.
            MOVEA.L   #SECOND, A1   Pointer to second list.
            MOVE.L    N, D0         Load the length parameter into D0.
            JSR       MEMCPY
            next instruction

MEMCPY:     MOVE.L    A2, −(A7)     Save registers.
            MOVE.L    D1, −(A7)
            CMPA.L    A0, A1        Compare pointers for start of from list.
            BLO       LOOPF         If to < from, then copy in forward direction.
            MOVEA.L   A0, A2        Calculate end of from list.
            ADDA.L    D0, A2
            CMPA.L    A2, A1        Compare pointers for end of from list.
            BHS       LOOPF         If to ≥ from + length, then go forward.
            ADDA.L    D0, A0        Adjust to end of lists.
            ADDA.L    D0, A1
LOOPR:      MOVE.B    −(A0), D1     Load byte from source list, predecrement pointer.
            MOVE.B    D1, −(A1)     Store byte into destination list, predecrement pointer.
            SUBQ.L    #1, D0        Decrement count.
            BGT       LOOPR
            BRA       DONE
LOOPF:      MOVE.B    (A0)+, D1     Load byte from source list, postdecrement pointer.
            MOVE.B    D1, (A1)+     Store byte into destination list, postdecrement pointer.
            SUBQ.L    #1, D0
            BGT       LOOPF
DONE:       MOVE.L    (A7)+, D1     Restore registers.
            MOVEA.L   (A7)+, A2
            RTS
```

C.20. The comparison subroutine is called with three parameters in registers, and it returns the result in one of those registers.

```
          MOVEA.L  #FIRST, A0    Pointer to first list.
          MOVEA.L  #SECOND, A1   Pointer to second list.
          MOVE.L   N, D0         Load the length parameter into D0.
          JSR      MEMCMP
          next instruction

MEMCMP:   MOVE.L   D1, −(A7)     Save registers.
          MOVE.L   D2, −(A7)
          MOVE.L   D3, −(A7)
          CLR.L    D1            Clear the counter.
          CLR.L    D2            Clear registers used in byte comparisons.
          CLR.L    D3
LOOP:     MOVE.B   (A0)+, D2     Load the bytes that have
          MOVE.B   (A1)+, D3       to be compared
          CMP.L    D3, D2          (with pointer increment).
          BEQ      NEXT          If equal, do nothing,
          ADDQ.L   #1, D1          otherwise increment counter.
NEXT:     SUBQ.L   #1, D0        Branch back if the end of
          BGT      LOOP            lists is not reached.
          MOVE.L   D1, D0        Return the result via D0.
          MOVE.L   (A7)+, D3     Restore registers.
          MOVE.L   (A7)+, D2
          MOVE.L   (A7)+, D1
          RTS
```

C.21. The subroutine that replaces each period in a string with an exclamation mark can be called in the manner shown below.

```
          MOVEA.L  #STRING, A0   Pointer to the string.
          JSR      EXCLAIM
          next instruction

EXCLAIM:  MOVE.L   D0, −(A7)     Save registers.
          CLR.L    D0            Clear register for byte access.
LOOP:     MOVE.B   (A0), D0
          CMPI.L   #0, D0        Check if NUL.
          BEQ      DONE
          CMPI.L   #$2E, D0      If period, then replace
          BNE      NEXT            with exclamation mark.
          MOVEQ.L  #$21, D0
          MOVE.B   D0, (A0)
NEXT:     ADDA.L   #1, A0        Move to the next character.
          JMP      LOOP
DONE:     MOVE.L   (A7)+, D0     Restore registers.
          RTS
```

C.22. The subroutine that converts all lower-case characters in a string into upper-case characters is provided below.

```
          MOVEA.L   #STRING, A0   Pointer to the string.
          JSR       ALLCAPS
          next instruction

ALLCAPS:  MOVE.L    D0, −(A7)     Save registers.
          CLR.L     D0            Clear register for byte access.
LOOP:     MOVE.B    (A0), D0
          CMPI.L    #0, D0        Check if NUL.
          BEQ       DONE
          CMPI.L    #$61, D0      Compare with ASCII code for a.
          BLT       NEXT
          CMPI.L    #$7A, D0      Compare with ASCII code for z.
          BGT       NEXT
          ANDI.L    #$DF, D0      Create ASCII for the capital letter.
          MOVE.B    D0, (A0)      Store the capital letter.
NEXT:     ADDA.L    #1, A0        Move to the next character.
          JMP       LOOP
DONE:     MOVE.L    (A7)+, D0     Restore registers.
          RTS
```

C.23. The subroutine to count words checks for an empty string to be certain that the count is accurate. The subroutine does, however, assume that words are separated by one space, and that the last word is not followed by a space. To clearly distinguish address information from other data, the subroutine returns the count of words in register D0, rather than reusing the address register A0.

```
          MOVEA.L   #STRING, A0   Pointer to the string.
          JSR       WORDS         Result returned in D0.
          next instruction

WORDS:    MOVE.L    D1, −(A7)     Save registers.
          CLR.L     D0            Clear the word counter.
          CLR.L     D1            Clear register for byte access.
          MOVE.B    (A0), D1      Check for empty string.
          CMPI.L    #0, D1
          BEQ       DONE
          ADDQ.L    #1, D0        Otherwise, at least one word in string.
LOOP:     MOVE.B    (A0)+, D1     Get character and move to next one.
          CMPI.L    #0, D1        Check if NUL.
          BEQ       DONE
          CMPI.L    #$20, D1      Check if SPACE.
          BNE       NEXT
          ADDQ.L    #1, D0        Increment the word count.
NEXT:     JMP       LOOP
DONE:     MOVE.L    (A7)+, D1     Restore registers.
          RTS
```

C.24. The subroutine below searches for the proper insertion point for a new item in an existing list, and moves the items above the insertion point to create an open position for the new item. For modularity, values of any modified registers are saved.

```
            MOVEA.L   #LIST, A0     Pointer to the list.
            MOVE.L    N, D0         Number of elements in the list.
            MOVE.L    NEW, D1       New element to insert into the list.
            JSR       INSERT
            next instruction

INSERT:     MOVE.L    A1, −(A7)     Save registers.
            MOVE.L    A0, −(A7)
            MOVE.L    D2, −(A7)
            MOVE.L    D1, −(A7)
            MOVE.L    D0, −(A7)
            LSL.L     #2, D0        Multiply by 4.
            MOVEA.L   A0, A1
            ADDA.L    D0, A1        End of the list.
LOOP:       MOVE.L    (A0), D2      Check entries in the list
            CMP.L     D2, D1          until insertion point is reached.
            BLS       TRANSFER
            ADDA.L    #4, A0        Increment the list pointer.
            CMPA.L    A1, A0
            BLO       LOOP
            JMP       DONE
TRANSFER:   MOVE.L    (A0), D2      Insert the new entry and
            MOVE.L    D1, (A0)        move the rest of the entries
            MOVE.L    D2, D1          upwards in the list.
            ADDA.L    #4, A0        Increment the list pointer.
            CMPA.L    A1, A0
            BLO       TRANSFER
DONE:       MOVE.L    D1, (A0)      Store the last entry.
            MOVE.L    (A7)+, D0     Restore registers.
            MOVE.L    (A7)+, D1
            MOVE.L    (A7)+, D2
            MOVE.L    (A7)+, A0
            MOVE.L    (A7)+, A1
            RTS
```

C.25. INSERTSORT calls the INSERT subroutine described in Problem C.24, element by element, to construct the sorted new list from unsorted old list. The calling program calls INSERTSORT with the following registers providing the stated parameters:

- A0 contains the starting address of the unsorted (old) list
- D0 contains the number of elements in the unsorted (old) list
- A1 contains the starting address of the new list

The invocation of INSERTSORT from the calling program is shown below, before the definition of the INSERTSORT subroutine.

```
        MOVEA.L   #OLDLIST, A0    Pointer to the old list.
        MOVE.L    N, D0           Number of elements in the list.
        MOVEA.L   #NEWLIST, A1    Pointer to the new list.
        JSR       INSERTSORT
        next instruction
```

```
INSERTSORT:  MOVE.L    A0, −(A7)   Save registers.
             MOVE.L    A1, −(A7)
             MOVE.L    D0, −(A7)
             MOVE.L    D2, −(A7)
             MOVE.L    A2, −(A7)   (A2 used as initial old list pointer)
             MOVE.L    D1, −(A7)   (D1 used as total count of items)
             MOVEA.L   A0, A2      Point to start of old list.
             MOVE.L    D0, D2      Number of items.
             MOVEA.L   A1, A0      Point to start of new list.
             MOVE.L    (A2)+, D0   Move one element to new list, increment pointer.
             MOVE.L    D0, (A0)
             MOVEQ.L   #1, D0      Initialize count for new list.
SCAN:        MOVE.L    (A2)+, D1   Get next item to be inserted, increment pointer.
             JSR       INSERT      (A0 = list start, D0 = list length, D1 = item to insert)
             ADDQ.L    #1, D0      Increment the length of the new list.
             CMP.L     D2, D0
             BLO       SCAN
             MOVE.L    (A7)+, D1   Restore registers.
             MOVE.L    (A7)+, A2
             MOVE.L    (A7)+, D2
             MOVE.L    (A7)+, D0
             MOVE.L    (A7)+, A1
             MOVE.L    (A7)+, A0
             RTS
```

C.26. The program below prints a prompt “Type your name” before accepting characters entered by the user, then it prints a message “Your name reversed” followed by the entered characters for the name of the user in reverse order.

It includes code similar to that shown in Figure C.17 for accessing input/output interfaces such as those described in Chapter 3 for a keyboard and display, with a minor difference in that a more general solution based on an And instruction is used instead of a BitTest instruction to check status register contents.

```
KBD_DATA   EQU        $4000            Starting address of keyboard interface.
DISP_DATA  EQU        $4010            Starting address of display interface.

           CLR.L      D0               Clear register for byte accesses.
           MOVEA.L    #PROMPT, A0      Set pointer to location for prompt.
           MOVEA.L    #DISP_DATA, A6   Set pointer to display interface.
PLOOP:     MOVE.B     4(A6), D0        Read display status register.
           ANDI.L     #4, D0           Check the DOUT flag.
           BEQ        PLOOP
           MOVE.B     (A0)+, D0        Send a character of the prompt
           MOVE.B     D0, (A6)           to the display.
           CMPI.L     #$D, D0          Determine if at end of prompt.
           BNE        PLOOP

           MOVEA.L    #NAME, A0        Set pointer to location for name.
           MOVEA.L    #KBD_DATA, A5    Set pointer to keyboard interface.
READ:      MOVE.B     4(A5), D0        Read keyboard status register.
           ANDI.L     #2, D0           Check the KIN flag.
           BEQ        READ
           MOVE.B     (A5), D0         Read character from keyboard.
           MOVE.B     D1, (A0)+        Write character and increment the pointer.
ECHO:      MOVE.B     4(A6), D1        Read display status register.
           ANDI.L     #4, D1           Check the DOUT flag.
           BEQ        ECHO
           MOVE.B     D0, (A7)         Send the character to the display.
           CMPI.L     #$D, D0          Loop back if character is not CR.
           BNE        READ
           MOVEA.L    A0, A2           Save ending address for characters in name,
           SUBA.L     #1, A2             and adjust for proper initial position.

           MOVEA.L    #MSG, A0         Set pointer to location for message.
MLOOP:     MOVE.B     4(A6), D0        Read display status register.
           ANDI.L     #4, D0           Check the DOUT flag.
           BEQ        MLOOP
           MOVE.B     (A0)+, D0        Send a character of the message
           MOVE.B     D0, (A6)           to the display.
           CMPI.L     #$D, D0          Determine if at end of message.
           BNE        MLOOP

           MOVEA.L    #NAME, A0        Set pointer to location for name.
NLOOP:     MOVE.B     4(A6), D0        Read display status register.
           ANDI.L     #4, D0           Check the DOUT flag.
           BEQ        NLOOP
           MOVE.B     −(A2), D0        Get a character, decrement pointer,
           MOVE.B     D0, (A6)           and send character to the display.
           CMPA.L     A0, A2           Determine if more characters remain to print.
           BHI        NLOOP
           next instruction


PROMPT     .DC.B      $54, $79, $70, $65, $20, $79, $6F, $75
           .DC.B      $72, $20, $6E, $61, $6D, $65, $0D
MSG        .DC.B      $59, $6F, $75, $72, $20, $6E, $61, $6D, $65, $20
           .DC.B      $72, $65, $76, $65, $72, $73, $65, $64, $0D

NAME       .DS.B      100              Reserve 100 bytes for storing name.
```

C.27. The program below determines whether or not a word at location WORD in memory is a palindrome. The length of the word is stored at location LENGTH in memory. The result is placed in location RESULT in memory. The word is scanned in both directions with two pointers, checking for identical characters. It is possible to stop scanning when the pointers reach the middle of the word. For simplicity, this program continues until the pointers reach the opposite ends of the word.

```
        MOVEA.L   #WORD, A0     Set pointer to word.
        MOVEA.L   A0, A1        Prepare pointer
        ADDA.L    LENGTH, A1      to end of word.
        MOVEA.L   A1, A2        Save pointer to end of word.
        CLR.L     D0            Clear registers used for byte accesses.
        CLR.L     D1
DLOOP:  MOVE.B    (A0)+, D0     Compare the characters that are identified
        MOVE.B    −(A1), D1       by the two pointers.
        CMP.L     D0, D1
        BNE       NOTP          If not equal, the word is not a palindrome.

        CMPA.L    A2, A0        Determine if the forward pointer
        BLO       DLOOP           has passed the end of the word.
        MOVEQ.L   #1, D0        If this point is reached, the word is a palindrome.
        JMP       DONE
NOTP:   MOVEQ.L   #0, D0        Not a palindrome.
DONE:   MOVE.L    D0, RESULT    Store the result.
```

For input/output interfaces such as those described in Chapter 3, the inclusion of code similar to that in Figure C.17 would enable the above program to be extended with the ability to send the characters for prompt and result messages to a display, and to accept the characters for a candidate word from a keyboard.

C.28. The following program determines the size and position of the box to be printed around the characters beginning at location STRING in memory. A zero marks the end of the list of characters. The program then prints three lines of text based on the aforementioned size and position. The number of spaces to print at the beginning of each line for proper centering is determined from the expression $(80 - (\text{length} + 2))/2$ or $39 - \text{length}/2$.

```
        MOVEA.L  #STRING, A0   Load address of string.
        JSR      LENGTH        Compute length of string.
        CMPA.L   #78, A0       Check if length is greater than 78.
        BLS      CONT1         If not, continue to subsequent instructions,
        MOVEA.L  #78, A0          otherwise, truncate to 78.
CONT1:  MOVE.L   A0, D6        Save length in D6.
        MOVE.L   A0, D7        Use D7 to calculate and hold 39−length/2.
        ASR.L    #1, D7        Divide by 2,
        SUBI.L   #39, D7          subtract 39, and then change the sign
        NEG.L    D7               to obtain number of leading spaces.
        MOVE.L   D6, D0        Prepare arguments for call to subroutine
        MOVE.L   D7, D1           to display upper line of
        JSR      DISPA            bounding box with carriage return.
        MOVE.L   D7, D1        Initialize counter with number of leading spaces.
        MOVEQ.L  #$20, D0      Load space character into D0.
LOOP1:  JSR      WRITECHAR     Repeat this loop to display spaces
        SUBQ.L   #1, D1           until count has reached zero.
        BGT      LOOP1
DISP2:  MOVEQ.L  #$7C, D0      Load vertical bar character into D0.
        JSR      WRITECHAR     Display the character.
        MOVEA.L  #STRING, A1   Initialize pointer to string.
        MOVE.L   D6, D1        Initialize the counter for the string length.
LOOP2:  MOVE.B   (A1)+, D0     Get a character and advance the pointer.
        JSR      WRITECHAR     Display the character.
        SUBQ.L   #1, D1        Decrement the counter
        BGT      LOOP2            and repeat until all characters displayed.
        MOVEQ.L  #$7C, D0      Load the vertical bar character into D0.
        JSR      WRITECHAR     Display the character.
        MOVEQ.L  #$D, D0       Display a carriage return.
        JSR      WRITECHAR
        MOVE.L   D6, D0        Prepare arguments for call to subroutine
        MOVE.L   D7, D1           to display lower line of
        JSR      DISPA            bounding box with carriage return.
        next instruction
```

The subroutines called from the program above are shown below. Note the use of instructions to push and pop register values in order to make the subroutines modular in nature, i.e., all register values are unchanged upon return to the calling program with the exception of registers that are used to return values. The WRITECHAR subroutine follows the example of Figure C.17 closely; the available bit-testing instruction is used for checking the status register.

```
LENGTH:     MOVE.L    D0, -(A7)
            MOVE.L    D1, -(A7)
            CLR.L     D1                Initialize count of characters.
            CLR.L     D0                Clear register for byte accesses.
LEN_LOOP:   MOVE.B    (A0)+, D0         Loop until zero is detected.
            CMPI.L    #0, D0
            BEQ       LEN_DONE
            ADDQ.L    #1, D1            Increment count.
            JMP       LEN_LOOP
LEN_DONE:   MOVEA.L   D1, A0            Copy count to register for return value.
            MOVE.L    (A7)+, D1
            MOVE.L    (A7)+, D0
            RTS


WRITECHAR:  MOVE.L    A0, -(A7)
            MOVEA.L   #DISP_STATUS, A0  EQU directives are assumed for the I/O addresses.
WCLOOP:     BTST.B    #2, (A0)          Instruction set restrictions require use of indirect mode.
            BEQ       WCLOOP
            MOVE.B    D0, DISP_DATA
            MOVE.L    (A7)+, A0
            RTS


DISPA:      MOVE.L    D5, -(A7)
            MOVE.L    D1, -(A7)
            MOVE.L    D0, -(A7)
            MOVE.L    D0, D5            Save length.
            MOVEQ.L   #$20, D0          Load space character into D0.
SPLOOP:     JSR       WRITECHAR         Repeat this loop to display spaces
            SUBQ.L    #1, D1              until count has reached zero.
            BGT       SPLOOP
            MOVEQ.L   #$2B, D0          Display '+' character.
            JSR       WRITECHAR
            MOVEQ.L   #$2D, D0          Load '-' character into D0.
DSHLOOP:    JSR       WRITECHAR         Repeat this loop to display '-'
            SUBQ.L    #1, D5              until the other count has reached zero.
            BGT       DSHLOOP
            MOVEQ.L   #$2B, D0          Display '+' character.
            JSR       WRITECHAR
            MOVEQ.L   #$D, D0           Display carriage return.
            JSR       WRITECHAR
            MOVE.L    (A7)+, D0
            MOVE.L    (A7)+, D1
            MOVE.L    (A7)+, D5
            RTS
```

C.29. The following program scans the characters beginning at location TEXT. Assuming that no word is longer than 80 characters, the program maintains the count of available space in each line and scans forward without displaying any characters until it verifies that there is enough space to display a complete word. When there is insufficient space, the program first emits a carriage return to begin on a new line. In any case, a subroutine is then called to display a single word when the program determines it is appropriate to do so. The subroutine accepts as arguments the starting location for the word and the number of characters to display (as determined in the preceding scan). For the scanning of characters, the stated assumptions of no control characters other than the NUL character and a single space character between words are exploited to simplify the program.

```
           MOVEA.L  #TEXT, A1    Register A1 points to start of text.
           MOVEQ.L  #80, D2      Register D2 reflects space left on the current line.
           CLR.L    D1           Clear register used for byte access.
RESET:     CLR.L    D0           Clear count of characters in current word.
           MOVEA.L  A1, A0       Save the starting point of current word.
SCAN:      MOVE.B   (A1)+, D1    Read the next character and advance the pointer.
           CMPI.L   #$20, D1     Check for a control character or space,
           BLS      HAVEWORD       and process a complete word appropriately.
           ADDQ.L   #1, D0       Otherwise, increment count of characters,
           JMP      SCAN           and repeat inner loop for current word.
HAVEWORD:  MOVE.L   D2, D3       For complete word, use space left on current line
           SUB.L    D0, D3         and count of characters in current word
           BGE      DISP           to determine if word will fit.
           JSR      NEWLINE      Otherwise, move to a new line,
           MOVEQ.L  #80, D2        and reinitialize space left for new line.
DISP:      JSR      DISPLAY      Display word using A0 pointer and D0 count.
           SUB.L    D0, D2       Reduce space left on line using count.
           CMPI.L   #0, D3       If previous calculation indicated no space on line,
           BEQ      SKIP           skip printing a space after current word.
           MOVEQ.L  #$20, D0     Display a space (it is safe to use D0 here).
           JSR      WRITECHAR
           SUBQ.L   #1, D2       Reduce space on current line by one character.
SKIP:      CMPI.L   #0, D1       Finally, check if last character was NUL,
           BEQ      DONE           and end program.
           JMP      RESET        Otherwise, assume it was a space, and start new word.
DONE:      next instruction
```

The subroutines that are specific to the this program are shown below. The WRITECHAR subroutine of Problem C.28 is assumed to also be available. The subroutines are implemented in a modular fashion to allow the calling program to rely on register values being preserved.

```
NEWLINE:   MOVE.L    D0, −(A7)      Save register to use for character output.
           MOVEQ.L   #$D, D0        Send carriage return to move to new line.
           JSR       WRITECHAR
           MOVE.L    (A7)+, D0
           RTS


DISPLAY:   MOVE.L    A0, −(A7)      Save original word start for modularity.
           MOVE.L    D1, −(A7)      Save register to use for count.
           MOVE.L    D0, −(A7)      Save original character count for modularity.
           MOVE.L    D0, D1         Prepare counter for loop.
DLOOP:     MOVE.B    (A0)+, D0      Read next character and increment pointer.
           JSR       WRITECHAR      Display the character.
           SUBQ.L    #1, D1         Decrement the count.
           BGT       DLOOP          Repeat if not finished with current word.
           MOVE.L    (A7)+, D0      Restore registers.
           MOVE.L    (A7)+, D1
           MOVE.L    (A7)+, A0
           RTS
```

C.30. The subroutine below for approximating $\sin(x)$ accepts a pointer in register A0 to a double-precision floating-point value in memory as the input parameter x, and places the result in the same location. This particular implementation assumes that there are integer values for 1, 6, and 120 in the memory locations labelled ONE, SIX, and ONE_HUNDRED_TWENTY, and it uses the variant of the floating-point move instruction with automatic conversion to floating-point. The value x^2 is computed and retained in a floating-point register for reuse in completing the computation. For modularity, floating-point registers and address resgisters that are modified by the subroutine are saved and restored.

```
SIN:  FMOVE.D   FP0, −(A7)                   Save registers.
      FMOVE.D   FP1, −(A7)
      FMOVE.D   FP2, −(A7)
      FMOVE.D   FP3, −(A7)
      MOVE.L    A1, −(A7)
      MOVE.L    A2, −(A7)
      MOVE.L    A3, −(A7)
      MOVEA.L   #ONE, A1                     Set pointers to integer constants.
      MOVEA.L   #SIX, A2
      MOVEA.L   #ONE_HUNDRED_TWENTY, A3
      FMOVE.D   (A0), FP0                    Compute x^2.
      FMUL.D    FP0, FP0
      FMOVE.L   (A1), FP1                    Compute 1/120.
      FMOVE.L   (A3), FP2
      FDIV.D    FP2, FP1
      FMUL.D    FP0, FP1                     Compute x^2(1/120).
      FMOVE.L   (A1), FP2                    Compute 1/6.
      FMOVE.L   (A2), FP3
      FDIV.D    FP3, FP2
      FSUB.D    FP1, FP2                     Compute 1/6 − x^2(1/120).
      FMUL.D    FP0, FP2                     Compute x^2(1/6 − x^2(1/120)).
      FMOVE.L   (A1), FP1
      FSUB.D    FP2, FP1                     Compute 1 − x^2(1/6 − x^2(1/120)).
      FMOVE.D   (A0), FP0
      FMUL.D    FP1, FP0                     Compute x(1 − x^2(1/6 − x^2(1/120))).
      FMOVE.D   FP0, (A0)                    Replace input argument with result.
      MOVE.L    (A7)+, A3                    Restore registers.
      MOVE.L    (A7)+, A2
      MOVE.L    (A7)+, A1
      FMOVE.D   (A7)+, FP3
      FMOVE.D   (A7)+, FP2
      FMOVE.D   (A7)+, FP1
      FMOVE.D   (A7)+, FP0
      RTS
```