Appendix E

The Intel IA-32 Architecture

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

```
MOV    EAX, 580      Load 580.
ADD    EAX, 6840     Generate 580 + 6840.
ADD    EAX, 80000    Generate the final sum.
MOV    SUM, EAX      Store the sum.
```

E.2. Multiplication requires the use of the IMUL instruction whose destination operand is always EAX.

```
          .CODE
          MOV    EAX, A         Load the operand A.
          MOV    EDX, B         Load the operand B.
          IMUL   EDX            EAX set to EAX * EDX (A * B).
          MOV    EBX, EAX       Transfer first product to EBX.
          MOV    EAX, C         Load the operand C.
          MOV    EDX, D         Load the operand D.
          IMUL   EDX            EAX set to EAX * EDX (C * D).
          ADD    EAX, EBX       Generate sum of two products.
          MOV    ANSWER, EAX    Store the answer.

          .DATA
A         DD     100            Test data.
B         DD     50
C         DD     20
D         DD     400
ANSWER    DD     0              Space for the sum.
          END
```

E.3. This program uses a loop and the Register indirect addressing mode to count negative numbers that are found in the list.

```
          .CODE
          MOV     ECX, N                  Load the size of the list.
          MOV     EAX, 0                  Initialize the counter to 0.
          MOV     EBX, OFFSET NUMBERS     Load address of the first number.
LOOP:     MOV     EDX, [EBX]              Get the next number.
          CMP     EDX, 0                  Compare with zero.
          JGE     NEXT                    Test if number is negative.
          INC     EAX                     Increment the count.
NEXT:     ADD     EBX, 4                  Increment the pointer to list.
          DEC     ECX                     Decrement the list counter.
          JG      LOOP                    Loop back if not finished.
          MOV     NEGNUM, EAX             Store the result.

          .DATA
NEGNUM    DD      0                       Space for the result.
N         DD      6                       Size of list.
NUMBERS   DD      23, −5, −128            Test data.
          DD      44, −23, −9
          END
```

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

```
        .CODE
        MOV     EAX, OFFSET LIST    Get the address LIST.
        MOV     EBX, 0
        MOV     ECX, 0
        MOV     EDX, 0
        MOV     EDI, N              Load the value n.
LOOP:   ADD     EBX, [EAX + 4]      Add current student mark for Test 1.
        ADD     ECX, [EAX + 8]      Add current student mark for Test 2.
        ADD     EDX, [EAX + 12]     Add current student mark for Test 3.
        ADD     EAX, 16             Increment the pointer.
        DEC     EDI                 Decrement the counter.
        JG      LOOP                Loop back if not finished.
        MOV     SUM1, EBX           Store the total for Test 1.
        MOV     SUM2, ECX           Store the total for Test 2.
        MOV     SUM3, EDX           Store the total for Test 3.

        .DATA
SUM1    DD      0                   Space for SUM1.
SUM2    DD      0                   Space for SUM2.
SUM3    DD      0                   Space for SUM3.
N       DD      3                   Size of the list.
LIST    DD      1234, 62, 85, 75    Example records.
        DD      1235, 90, 82, 88
        DD      1236, 72, 65, 80
        END
```

E.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 in order to use the decrementing outer loop counter registers for indexed addressing.

```
        MOV   ESI, J                 Compute and place
        ADD   ESI, 1                   Stride = 4(j + 1)
        SHL   ESI, 2                   into register ESI.
        MOV   EDI, OFFSET LIST       Initialize register EDI to the location
        ADD   EDI, 4                   of the test 1 score for student 1.
        MOV   EBX, J                 Initialize outer loop counter EBX to j.
OUTER:  MOV   ECX, N                 Initialize inner loop counter ECX to n.
        MOV   EAX, 0                 Clear the sum register EAX.
        MOV   EDX, EDI               Use EDX as an index register.
INNER:  ADD   EAX, [EDX+EBX*4−4]     Accumulate the sum of test scores.
        ADD   EDX, ESI               Increment index register by Stride value.
        DEC   ECX                    Check if all student scores on current
        JG    INNER                    test have been accumulated.
        MOV   EDX, OFFSET SUM        Initialize register EDX to the location
                                       of the sum for test 1.
        MOV   [EDX+EBX*4−4], EAX     Store sum of current test scores.
        DEC   EBX                    Check if the sums for all tests have
        JG    OUTER                    been computed.
        next instruction
```

E.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.

```
        .CODE
        MOV   EDI, OFFSET LIST   Get the address LIST.
        MOV   EBX, N             Get the number of elements N.
        MOV   ESI, EDI           Initialize outer loop pointer
        ADD   ESI, EBX             to LIST + n.
OUTER:  DEC   ESI                Decrement the pointer.
        CMP   ESI, EDI           Check if last entry.
        JBE   DONE
        MOV   DL, [ESI]          Starting max value in sublist.
        MOV   ECX, ESI           Initialize inner loop pointer.
        DEC   ECX
INNER:  MOV   AL, [ECX]          Check if the next entry
        CMP   DL, AL               is lower.
        JGE   NEXT
        MOV   [ESI], AL          If yes, then swap
        MOV   [ECX], DL            the entries and
        MOV   DL, AL               update the max value.
NEXT:   DEC   ECX                Adjust the inner loop pointer.
        CMP   ECX, EDI
        JAE   INNER
        JMP   OUTER
DONE:   next instruction

        .DATA
N       DD    10                 Size of the list.
LIST    DB    'zZbB53kK24'       Test data.
        END
```

E.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.

```
TIMER     EQU   0x4020                 Location of timer status register.
```

Interrupt-service routine

```
ISR:      PUSH  EAX                    Save register for use within service routine.
          MOV   EAX, OFFSET TIMER      Set pointer to timer status register.
          MOV   AL, BYTE PTR [EAX]     Clear TIRQ and ZERO bits in status register.
          CALL  DISPLAY                Call the DISPLAY routine.
          POP   EAX                    Restore register.
          IRET                         Return from interrupt.
```

Main program

```
          MOV   EAX, OFFSET TIMER      Set pointer to timer status register.
          MOV   EDX, 0x3B9CA00         Prepare the count value for ten-second intervals.
          MOV   [EAX+8], EDX           Set the timer count value.
          MOV   BYTE PTR [EAX+4], 7    Start timer; continuous with interrupts.
          STI                          Set interrupt flag in processor register.
COMPUTE:  next instruction             Start of COMPUTE routine.
```

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

```
DIGIT    EQU    0x800                   Location of ASCII-encoded digit.
SEVEN    EQU    0x4030                  Address of 7-segment display.
         .CODE
         MOV    BL, DIGIT               Load the ASCII-encoded digit.
         MOV    BH, DIGIT
         AND    BL, 0x0F                Extract the decimal number.
         AND    BH, 0xF0                Extract high-order bits of ASCII.
         CMP    BH, 0x30                Check if high-order bits of
         JE     HIGH3                     ASCII code are 0011.
         MOV    BL, 0x0F                Not a digit, display a blank.
HIGH3:   AND    EBX, 0x0000000F         Clear upper bits.
         MOV    AL, [TABLE + EBX]       Get the 7-segment pattern.
         MOV    SEVEN, AL               Display the digit.

         .DATA
TABLE    DB     0x7E,0x30,0x6D,0x79     Table that contains
         DB     0x33,0x5B,0x5F,0x70       the necessary
         DB     0x7F,0x7B,0x00,0x00       7-segment patterns.
         DB     0x00,0x00,0x00,0x00
         END
```

E.9 The addressing modes of the Intel IA-32 architecture use 8-bit or 32-bit displacements. Hence, the address of 0x10100 for TABLE does not lead to a different instruction sequence than the one for the address of 0x1000 for table. Therefore, the solution to this problem is the same as the solution for Problem E.8, except that the address for TABLE is different.

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

```
          .CODE
          MOV     EAX, OFFSET LOC          Get the address LOC.
          MOV     EDI, OFFSET DISP_DATA    Get the address of display.
          MOV     ECX, 10                  Initialize the byte counter.
          MOV     EDX, 0                   Clear all bits in register EDX.
LOOP:     MOV     BL, [EAX]                Load a byte.
          MOV     DL, BL                   Make a copy of the byte.
          AND     DL, 0xF0                 Select the high-order 4 bits.
          SHR     DL, 4                    Shift right by 4 bit positions.
          MOV     DL, [EDX+TABLE]          Get the character for display.
          CALL    DISPLAY
          MOV     DL, BL                   Make a copy of the original byte.
          AND     DL, 0x0F                 Select the low-order 4 bits.
          MOV     DL, [EDX+TABLE]          Get the character for display.
          CALL    DISPLAY
          MOV     DL, 0x20                 ASCII code for SPACE.
          CALL    DISPLAY
          INC     EAX                      Increment the pointer.
          DEC     ECX                      Decrement the byte counter.
          JG      LOOP                     Branch back if not finished.
          next instruction

DISPLAY:  MOV     BH, [EDI+4]
          AND     BH, 4                    Check the DOUT flag.
          JE      DISPLAY
          MOV     [EDI], DL                Send the character to display.
          RET

          .DATA
TABLE     DB      0x30,0x31,0x32,0x33      Table that contains
          DB      0x34,0x35,0x36,0x37        the necessary
          DB      0x38,0x39,0x41,0x42        ASCII characters.
          DB      0x43,0x44,0x45,0x46
```

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

```
          MOV    EAX, BINARY              Load the 16-bit pattern from the address BINARY.
          MOV    EDI, OFFSET DISP_DATA    Get the address of display.
          MOV    ECX, 16                  Initialize the bit counter.
          MOV    EBX, 0x8000              Set bit 15 to 1.
LOOP:     MOV    EDI, EAX                 Make a copy of the pattern.
          AND    EDI, EBX                 Test a bit.
          JE     ZERO                     Check if 0 or 1, and
          MOV    DL, 0x31                 set ASCII character value.
          JMP    CONT
ZERO:     MOV    DL, 0x30
CONT:     CALL   DISPLAY
          SHR    EBX, 1                   Shift to check the next bit.
          DEC    ECX                      Decrement the bit counter.
          JG     LOOP                     Branch back if not finished.
          next instruction

DISPLAY:  MOV    DH, [EDI+4]
          AND    DH, 4                    Check the DOUT flag.
          JE     DISPLAY
          MOV    [EDI], DL                Send the character to display.
          RET
```

E.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    0x4020                   Location of timer status register.
SEVEN   EQU    0x4030                   Address of 7-segment display.
        .CODE
        MOV    EAX, OFFSET TIMER        Set pointer to timer status register.
        MOV    EDX, 0x5F5E100           Prepare the count value for one-second intervals.
        MOV    [EAX+8], EDX             Set the timer count value.
        MOV    BYTE PTR [EAX+4], 6      Start the timer in the continuous mode.
        CLR    EDX                      Clear the digit counter.
LOOP:   MOV    BL, [EAX]                Wait for timer to reach the
        AND    BL, 2                      end of the one-second interval.
        JE     LOOP
        MOV    BL, [EDX+TABLE]          Load the ASCII-encoded digit.
        MOV    SEVEN, BL                Display the digit.
        INC    EDX                      Increment the digit counter,
        CMP    EDX, 10                    and check if > 9.
        JL     LOOP
        CLR    EDX                      Clear the digit counter.
        J      LOOP

        .DATA
TABLE   DB     0x7E,0x30,0x6D,0x79      Table that contains
        DB     0x33,0x5B,0x5F,0x70        the necessary
        DB     0x7F,0x7B,0x00,0x00        7-segment patterns.
        DB     0x00,0x00,0x00,0x00
```

E.15. The subroutines for safe operations on the second stack are provided below. Register EAX holds the element that is to be pushed or popped. Register EBX is the pointer to the second stack.

```
SPUSH:  PUSH  ECX                   Save ECX on the processor stack.
        MOV   ECX, OFFSET TOP
        CMP   EBX, ECX
        JBE   FULLERROR
        SUB   EBX, 4
        MOV   [EBX], EAX
        POP   ECX                   Restore ECX.
        RET

SPOP:   PUSH  ECX                   Save ECX on the processor stack.
        MOV   ECX, OFFSET BOTTOM
        CMP   EBX, ECX
        JAE   EMPTYERROR
        MOV   EAX, [EBX]
        ADD   EBX, 4
        POP   ECX                   Restore ECX.
        RET
```

These subroutines could be somewhat shorter in length by performing the two comparisons directly with an immediate value, i.e.,

```
CMP   EBX, OFFSET TOP
```

and

```
CMP   EBX, OFFSET BOTTOM
```

In this manner, the use of register ECX and the related PUSH, MOV, and POP instructions would no longer be required in each subroutine. It is still necessary, however, to increment/decrement register EAX explicitly, as there are no auto-increment/auto-decrement addressing modes for the Intel IA-32 architecture.

E.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
  $\text{OUT} \leftarrow ([\text{OUT}]+1) \bmod k$

E.17. For the implementation of the APPEND and REMOVE tasks described in Problem E.16, all of the necessary information for managing the queue could be maintained in registers. The number of available registers, however, is rather modest for the Intel IA-32 architecture in comparison to other instruction sets. is the available registers are used in the manner indicated below.

EAX: the IN pointer

EBX: the OUT pointer

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

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

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

ESI: temporary storage for IN pointer before incrementing for APPEND

The initial empty state of the queue at the beginning of the area in memory is reflected by having registers EAX and EBX contain the same value as register ECX.

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.

```
APPEND:     MOV   ESI, EAX        Set temporary register to current IN pointer.
            INC   EAX             Increment IN pointer (modulo k).
            CMP   EDI, EAX        Compare against end address.
            JAE   CHECK           Continue if within bounds.
            MOV   EAX, ECX        Otherwise, reset IN to beginning address.
CHECK       CMP   EAX, EBX        Check if queue is full.
            JE    FULL
            MOV   [ESI], DL       If queue not full, append item.
            JMP   CONTINUE
FULL:       MOV   EAX, ESI        Restore IN pointer and indicate that
            CALL  QUEUEFULL         queue is full.
CONTINUE:   next instruction
```

```
REMOVE:     CMP   EAX, EBX        Check if queue is empty.
            JE    EMPTY
            MOV   DL, [EBX]       Remove byte at end of queue and
            INC   EBX               increment OUT pointer (modulo k).
            CMP   EDI, EBX
            JAE   CONTINUE
            MOV   EBX, ECX        Reset OUT to beginning address.
            JMP   CONTINUE
EMPTY:      CALL  QUEUEEMPTY      Indicate that queue is empty.
CONTINUE:   next instruction
```

E.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.

```
        MOV   ECX, N            Get n for number of entries to generate.
        MOV   EDI, OFFSET IN    Pointer to the IN list.
        MOV   ESI, OFFSET OUT   Pointer to the OUT list.
LOOP:   MOV   EAX, [EDI]        Get the value IN(k) and
        SAR   EAX, 3              divide it by 8.
        MOV   EBX, [EDI+4]      Get the value IN(k+1) and
        SAR   EBX, 2              divide it by 4.
        ADD   EAX, EBX
        MOV   EBX, [EDI+8]      Get the value IN(k+2) and
        SAR   EBX, 1              divide it by 2.
        ADD   EAX, EBX          Compute the sum and
        MOV   [ESI], EAX          store it in OUT list.
        ADD   EDI, 4            Increment the pointers
        ADD   ESI, 4              to IN and OUT lists.
        DEC   ECX               Continue until all values in
        JG    LOOP                OUT list have been generated.
        next instruction
```

E.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.

```
MEMCPY:  PUSH  ECX
         CMP   EBX, EAX    Compare pointers for start of from list.
         JB    LOOPF       If to < from, then copy in forward direction.
         MOV   ECX, EAX    Calculate end of from list.
         ADD   ECX, ESI
         CMP   EBX, ECX    Compare pointers for end of from list.
         JAE   LOOPF       If to ≥ from + length, then go forward.
         ADD   EAX, ESI    Adjust to end of lists.
         ADD   EBX, ESI
LOOPR:   DEC   EAX         Decrement pointers.
         DEC   EBX
         MOV   CL, [EAX]   Load byte from source list.
         MOV   [EBX], CL   Store byte into destination list.
         DEC   ESI         Decrement count.
         JG    LOOPR
         JMP   DONE
LOOPF:   MOV   CL, [EAX]   Load byte from source list.
         MOV   [EBX], CL   Store byte into destination list.
         INC   EAX         Increment pointers.
         INC   EBX
         DEC   ESI         Decrement count.
         JG    LOOPF
DONE:    POP   ECX
         RET
```

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

```
            MOV    EAX, OFFSET FIRST     Pointer to first list.
            MOV    EBX, OFFSET SECOND    Pointer to second list.
            MOV    ESI, N                Load the length parameter into ESI.
            CALL   MEMCMP
            next instruction

MEMCMP:     PUSH   EDI                   Save registers.
            PUSH   ECX
            PUSH   EDX
            CLR    EDI                   Clear the counter.
LOOP:       MOV    CL, [EAX]             Load the bytes that have
            MOV    DL, [EBX]               to be compared.
            CMP    CL, DL
            JE     NEXT
            INC    EDI                   Increment the counter.
NEXT:       INC    EAX                   Increment the pointers
            INC    EBX                     to the lists.
            DEC    ESI                   Branch back if the end of
            JG     LOOP                    lists is not reached.
            MOV    ESI, EDI              Return the result via ESI.
            POP    EDX                   Restore registers.
            POP    ECX
            POP    EDI
            RET
```

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

```
            MOV    EAX, OFFSET STRING    Pointer to the string.
            CALL   EXCLAIM
            next instruction

EXCLAIM:    PUSH   EBX                   Save registers.
LOOP:       MOV    BL, [EAX]
            CMP    BL, 0                 Check if NUL.
            JE     DONE
            CMP    BL, 0x2E              If period, then replace
            JNE    NEXT                    with exclamation mark.
            MOV    BL, 0x21
            MOV    [EAX], BL
NEXT:       INC    EAX                   Move to the next character.
            JMP    LOOP
DONE:       POP    EBX                   Restore registers.
            RET
```

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

```
          MOV    EAX, OFFSET STRING   Pointer to the string.
          CALL   ALLCAPS
          next instruction

ALLCAPS:  PUSH   EBX                  Save registers.
LOOP:     MOV    BL, [EAX]
          CMP    BL, 0                Check if NUL.
          JE     DONE
          CMP    BL, 0x61             Compare with ASCII code for a.
          JL     NEXT
          CMP    BL, 0x7A             Compare with ASCII code for z.
          JG     NEXT
          AND    BL, 0xDF             Create ASCII for the capital letter.
          MOV    [EAX], BL            Store the capital letter.
NEXT:     INC    EAX                  Move to the next character.
          JMP    LOOP
DONE:     POP    EBX                  Restore registers.
          RET
```

E.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.

```
          MOV    EAX, OFFSET STRING   Pointer to the string.
          CALL   WORDS
          next instruction

WORDS:    PUSH   EBX                  Save registers.
          PUSH   ECX
          CLR    ECX                  Clear the word counter.
          MOV    BL, [EAX]            Check for empty string.
          CMP    BL, 0
          JE     DONE
          INC    ECX                  Otherwise, at least one word in string.
LOOP:     MOV    BL, [EAX]
          CMP    BL, 0                Check if NUL.
          JE     DONE
          CMP    BL, 0x20             Check if SPACE.
          JNE    NEXT
          INC    ECX                  Increment the word count.
NEXT:     INC    EAX                  Move to the next character.
          JMP    LOOP
DONE:     MOV    EAX, ECX             Pass the result in EAX.
          POP    ECX                  Restore registers.
          POP    EBX
          RET
```

E.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.

```
            MOV    EAX, OFFSET LIST   Pointer to the list.
            MOV    EBX, N             Number of elements in the list.
            MOV    ECX, NEW           New element to insert into the list.
            CALL   INSERT
            next instruction

INSERT:     PUSH   EAX                Save registers.
            PUSH   EBX
            PUSH   ECX
            PUSH   EDX
            PUSH   EDI
            SHL    EBX, 2             Multiply by 4.
            MOV    EDX, EAX
            ADD    EDX, EBX           End of the list.
LOOP:       MOV    EDI, [EAX]         Check entries in the list
            CMP    ECX, EDI             until insertion point is reached.
            JBE    TRANSFER
            ADD    EAX, 4
            CMP    EAX, EDX
            JB     LOOP
            JMP    DONE
TRANSFER:   MOV    EDI, [EAX]         Insert the new entry and
            MOV    [EAX], ECX           move the rest of the entries
            MOV    ECX, EDI             upwards in the list.
            ADD    EAX, 4             Increment the list pointer.
            CMP    EAX, EDX
            JB     TRANSFER
DONE:       MOV    [EAX], ECX         Store the last entry.
            POP    EDI                Restore registers.
            POP    EDX
            POP    ECX
            POP    EBX
            POP    EAX
            RET
```

E.25. INSERTSORT calls the INSERT subroutine described in Problem E.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:

- EAX contains the starting address of the unsorted (old) list
- EBX contains the number of elements in the unsorted (old) list
- ECX 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.

```
            MOV    EAX, OFFSET OLDLIST   Pointer to the old list.
            MOV    EBX, N                Number of elements in the list.
            MOV    ECX, OFFSET NEWLIST   Pointer to the new list.
            CALL   INSERTSORT
            next instruction

INSERTSORT: PUSH   EAX          Save registers.
            PUSH   EBX
            PUSH   ECX
            PUSH   EDX          (EDX used as initial old list pointer)
            PUSH   EDI          (EDI used as total count of items)
            MOV    EDX, EAX     Point to start of old list.
            MOV    EDI, EBX     Number of items.
            MOV    EAX, ECX     Point to start of new list.
            MOV    EBX, [EDX]   Move one element from old list to new list.
            MOV    [EAX], EBX
            MOV    EBX, 1       Initialize count for new list.
SCAN:       ADD    EDX, 4       Increment pointer to old list.
            MOV    ECX, [EDX]   Get next item to be inserted.
            CALL   INSERT       (EAX = list start, EBX = list length, ECX = item to insert)
            INC    EBX          Increment the length of the new list.
            CMP    EBX, EDI
            JL     SCAN
            POP    EDI          Restore registers.
            POP    EDX
            POP    ECX
            POP    EBX
            POP    EAX
            RET
```

E.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 E.16 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    0x4000                     Starting address of keyboard interface.
DISP_DATA   EQU    0x4010                     Starting address of display interface.
            .CODE
            MOV    EAX, OFFSET PROMPT         Set pointer to location for prompt.
            MOV    EDI, OFFSET DISP_DATA      Set pointer to display interface.
PLOOP:      MOV    CL, [EDI+4]                Read display status register.
            AND    CL, 4                      Check the DOUT flag.
            JE     PLOOP
            MOV    DL, [EAX]                  Send a character of the prompt
            MOV    [EDI], DL                    to the display.
            INC    EAX
            CMP    DL, 0xD                    Determine if at end of prompt.
            JNE    PLOOP

            MOV    EAX, OFFSET NAME           Set pointer to location for name.
            MOV    ESI, OFFSET KBD_DATA       Set pointer to keyboard interface.
READ:       MOV    CL, [ESI+4]                Read keyboard status register.
            AND    CL, 2                      Check the KIN flag.
            JE     READ
            MOV    DL, [ESI]                  Read character from keyboard.
            MOV    [EAX], DL                  Write character into memory
            INC    EAX                          and increment the pointer.
ECHO:       MOV    CL, [EDI+4]                Read display status register.
            AND    CL, 4                      Check the DOUT flag.
            JE     ECHO
            MOV    [EDI], DL                  Send the character to the display.
            CMP    DL, 0xD                    Loop back if character is not CR.
            JNE    READ
            MOV    ESI, EAX                   Save ending address for characters in name,
            SUB    ESI, 2                       and adjust for proper initial position.

            MOV    EAX, OFFSET MSG            Set pointer to location for message.
MLOOP:      MOV    CL, [EDI+4]                Read display status register.
            AND    CL, 4                      Check the DOUT flag.
            JE     MLOOP
            MOV    DL, [EAX]                  Send a character of the message
            MOV    [EDI], DL                    to the display.
            INC    EAX
            CMP    DL, 0xD                    Determine if at end of message.
            JNE    MLOOP

            MOV    EAX, OFFSET NAME           Set pointer to location for name.
NLOOP:      MOV    CL, [EDI+4]                Read display status register.
            AND    CL, 4                      Check the DOUT flag.
            JE     NLOOP
            MOV    DL, [ESI]                  Send a character of the name
            MOV    [EDI], DL                    to the display.
            DEC    ESI                        Decrement pointer to move backward.
            CMP    ESI, EAX                   Determine if moved before start of name.
            JAE    NLOOP
            next instruction
```

```
          .DATA
PROMPT    DB      0x54, 0x79, 0x70, 0x65, 0x20, 0x79, 0x6F, 0x75
          DB      0x72, 0x20, 0x6E, 0x61, 0x6D, 0x65, 0x0D
MSG       DB      0x59, 0x6F, 0x75, 0x72, 0x20, 0x6E, 0x61, 0x6D, 0x65, 0x20
          DB      0x72, 0x65, 0x76, 0x65, 0x72, 0x73, 0x65, 0x64, 0x0D

NAME      DB      100 DUP(0)    Reserve 100 bytes for storing name.
```

E.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.

```
          MOV   EAX, OFFSET WORD    Set pointer to word.
          MOV   EBX, EAX            Prepare pointer
          ADD   EBX, LENGTH           to end
          DEC   EBX                   of word.
          MOV   ECX, EBX            Save pointer to end of word.
DLOOP:    MOV   DL, [EAX]           Compare the characters that are identified
          CMP   DL, [EBX]             by the two pointers.
          JNE   NOTP                If not equal, the word is not a palindrome.
          INC   EAX                 Increment pointer that is moving forward.
          DEC   EBX                 Decrement pointer that is moving backward.
          CMP   EAX, ECX            Determine if the forward pointer
          JBE   DLOOP                 has passed the end of the word.
          MOV   EAX, 1              If this point is reached, the word is a palindrome.
          JMP   DONE
NOTP:     MOV   EAX, 0              Not a palindrome.
DONE:     MOV   RESULT, EAX         Store the result.
```

For input/output interfaces such as those described in Chapter 3, the inclusion of code similar to that in Figure E.16 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.

E.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 $39 - \text{length}/2$.

```
        MOV   EAX, OFFSET STRING   Load address of string.
        CALL  LENGTH               Compute length of string.
        CMP   EAX, 78              Check if length is greater than 78.
        JLE   CONT1                If not, continue to subsequent instructions,
        MOV   EAX, 78                otherwise, truncate to 78.
CONT1:  MOV   ESI, EAX             Save length in ESI.
        MOV   EDI, EAX             Use EDI to calculate and hold 39-length/2.
        SAR   EDI, 1               Divide by 2,
        SUB   EDI, 39                subtract 39, and then change the sign
        NEG   EDI                    to obtain number of leading spaces.
        MOV   EAX, ESI             Prepare arguments for call to subroutine
        MOV   EBX, EDI               to display upper line of
        CALL  DISPA                  bounding box with carriage return.
        MOV   EBX, EDI             Initialize counter with number of leading spaces.
        MOV   AL, 0x20             Load space character into AL.
LOOP1:  CALL  WRITECHAR            Repeat this loop to display spaces
        DEC   EBX                    until count has reached zero.
        JG    LOOP1
DISP2:  MOV   AL, 0x7C             Load vertical bar character into AL.
        CALL  WRITECHAR            Display the character.
        MOV   ECX, OFFSET STRING   Initialize pointer to string.
        MOV   EBX, ESI             Initialize the counter for the string length.
LOOP2:  MOV   AL, [ECX]            Get a character from the string.
        INC   ECX                  Advance the pointer.
        CALL  WRITECHAR            Display the character.
        DEC   EBX                  Decrement the counter
        JG    LOOP2                  and repeat until all characters displayed.
        MOV   AL, 0x7C             Load the vertical bar character into AL.
        CALL  WRITECHAR            Display the character.
        MOV   AL, 0xD              Display a carriage return.
        CALL  WRITECHAR
        MOV   EAX, ESI             Prepare arguments for call to subroutine
        MOV   EBX, EDI               to display lower line of
        CALL  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 E.16 closely; the available bit-testing instruction is used for checking the status register.

```
LENGTH:      PUSH   EBX
             PUSH   ECX
             MOV    ECX, 0           Initialize count of characters.
LEN_LOOP:    MOV    BL, [EAX]        Loop until zero is detected.
             CMP    BL, 0
             JE     LEN_DONE
             INC    EAX              Increment pointer.
             INC    ECX              Increment count.
             JMP    LEN_LOOP
LEN_DONE:    MOV    EAX, ECX         Copy count to register for return value.
             POP    ECX
             POP    EBX
             RET


WRITECHAR:   BT     DISP_STATUS, 2   EQU directives are assumed for the I/O addresses.
             JNC    WRITECHAR
             MOV    DISP_DATA, AL
             RET


DISPA:       PUSH   ECX
             PUSH   EBX
             PUSH   EAX
             MOV    ECX, EAX         Save length.
             MOV    AL, 0x20         Load space character into AL.
SPLOOP:      CALL   WRITECHAR        Repeat this loop to display spaces
             DEC    EBX                until count has reached zero.
             JG     SPLOOP
             MOV    AL, 0x2B         Display '+' character.
             CALL   WRITECHAR
             MOV    AL, 0x2D         Load '-' character into AL.
DSHLOOP:     CALL   WRITECHAR        Repeat this loop to display '-'
             DEC    ECX                until the other count has reached zero.
             JG     DSHLOOP
             MOV    AL, 0x2B         Display '+' character.
             CALL   WRITECHAR
             MOV    AL, 0xD          Display carriage return.
             CALL   WRITECHAR
             POP    EAX
             POP    EBX
             POP    ECX
             RET
```

E.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.

```
           MOV   ECX, OFFSET TEXT   Register EBX points to start of text.
           MOV   EDX, 80            Register EDX reflects space left on the current line.
RESET:     MOV   ESI, 0             Clear count of characters in current word.
           MOV   EAX, ECX           Save the starting point of current word.
SCAN:      MOV   BL, [ECX]          Read the next character.
           INC   ECX                Advance the pointer.
           CMP   BL, 0x20           Check for a control character or space,
           JBE   HAVEWORD             and process a complete word appropriately.
           INC   ESI                Otherwise, increment count of characters,
           JMP   SCAN                 and repeat inner loop for current word.
HAVEWORD:  MOV   EDI, EDX           For complete word, use space left on current line
           SUB   EDI, ESI             and count of characters in current word
           JGE   DISP                 to determine if word will fit.
           CALL  NEWLINE            Otherwise, move to a new line,
           MOV   EDX, 80              and reinitialize space left for new line.
DISP:      CALL  DISPLAY            Display word using EAX pointer and ESI count.
           SUB   EDX, ESI           Reduce space left on line using count.
           CMP   EDI, 0             If previous calculation indicated no space on line,
           JE    SKIP                 skip printing a space after current word.
           MOV   AL, 0x20           Display a space (it is safe to use EAX here).
           CALL  WRITECHAR
           DEC   EDX                Reduce space on current line by one character.
SKIP:      CMP   BL, 0              Finally, check if last character was NUL,
           JE    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 E.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:    PUSH   EAX          Save register to use for character output.
            MOV    AL, 0xD      Send carriage return to move to new line.
            CALL   WRITECHAR
            POP    EAX
            RET


DISPLAY:    PUSH   EBX          Save register to use for loop pointer.
            PUSH   EAX          Save original word start for modularity.
            PUSH   ESI          Save original character count for modularity.
            MOV    EBX, EAX     Prepare pointer for loop.
DLOOP:      MOV    AL, [EBX]    Read next character for word.
            INC    EBX          Advance pointer.
            CALL   WRITECHAR    Display the character.
            DEC    ESI          Decrement the count.
            JG     DLOOP        Repeat if not finished with current word.
            POP    ESI          Restore registers.
            POP    EAX
            POP    EBX
            RET
```

E.30 The subroutine below for approximating $\sin(x)$ accepts a pointer in register EAX 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 6 and 120 in the memory locations labelled SIX and ONE_HUNDRED_TWENTY, and it uses the variant of the division instruction that perform automatic conversion to floating-point. The value x^2 is duplicated on the stack for reuse in completing the computation.

```
SIN:  PUSH    EBX
      FLD     QWORD PTR [EAX]                 Compute x^2.
      FMUL    QWORD PTR [EAX]
      FLDZ                                    Push zero on register stack.
      FADD    ST(0),ST(1)                     Duplicate x^2.
      FLD1                                    Push 1 on register stack.
      MOV     EBX, OFFSET ONE_HUNDRED_TWENTY  Point to integer 120 in memory,
      FIDIV   DWORD PTR [EBX]                   divide, replace ST(0) with result.
      FMULP   ST(1),ST(0)                     Compute x^2(1/120).
      FLD1                                    Push 1 on register stack.
      MOV     EBX, OFFSET SIX                 Point to integer 6 in memory,
      FIDIV   DWORD PTR [EBX]                   divide, replace ST(0) with result.
      FSUBRP  ST(1),ST(0)                     Compute 1/6 − x^2(1/120).
      FMULP   ST(1),ST(0)                     Compute x^2(1/6 − x^2(1/120)).
      FLD1                                    Push 1 on register stack.
      FSUBRP  ST(1),ST(0)                     Compute 1 − x^2(1/6 − x^2(1/120)).
      FMUL    QWORD PTR [EAX]                 Compute x(1 − x^2(1/6 − x^2(1/120))).
      FSTP    QWORD PTR [EAX]                 Replace input argument with result.
      POP     EBX
      RET
```