Chapter 3

ARM, Motorola, and Intel Instruction Sets

PART I: ARM

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

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

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

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

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

3.3. The following two instructions perform the desired operation:

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

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

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

3.5. Program trace:

TIME	R0	R1	R2
after 1st execution of BGT	3	4	NUM1 + 4
after 2nd execution of BGT	−14	3	NUM1 + 8
after 3rd execution of BGT	13	2	NUM1 + 12

3.6. Assume bytes are unsigned 8-bit values.

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

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

```
          LDR    R0,N            Compute outer loop count
          LDR    R1,M              and store in R2.
          SUB    R2,R0,R1
          ADD    R2,R2,#1
          ADR    R3,STRING       Use R3 and R4 as base
          ADR    R4,SUBSTRING      pointers for each match.
OUTER     MOV    R5,R3           Use R5 and R6 as running
          MOV    R6,R4             pointers for each match.
          LDR    R7,M            Initialize inner loop counter.
INNER     LDRB   R0,[R5],#1      Compare bytes.
          LDRB   R1,[R6],#1
          CMP    R0,R1
          BNE    NOMATCH         If not equal, go next.
          SUBS   R7,R7,#1        Check if all bytes compared.
          BGT    INNER
          MOV    R0,R3           If substring matches, load
          B      NEXT              its position into R0 and exit.
NOMATCH   ADD    R3,R3,#1        Go to next substring.
          SUBS   R2,R2,#1        Check if all positions tried.
          BGT    OUTER
          MOV    R0,#0           If yes, load zero into
NEXT      ...                      R0 and exit.
```

3.8. This solution assumes that the last number in the series of n numbers can be represented in a 32-bit word, and that $n > 2$.

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

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

```
NEXT  LDRB     R1,[R0]       Get character.
      CMP      #&20,R1       Check if space character.
      ANDNE    #&DF,R1       If not space: clear
      STRNEB   R1,[R0],#1      bit 5, store
      BNE      NEXT            converted character,
                               get next character.
```

3.10. Memory word location J contains the number of tests, j, and memory word location N contains the number of students, n. The list of student marks begins at memory word location LIST in the format shown in Figure 2.14. The parameter Stride = $4(j + 1)$ is the distance in bytes between scores on a particular test for adjacent students in the list.

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

```
        LDR   R3,J               Load j + 1 into R3 to
        ADD   R3,R3,#1             be used as an address offset.
        ADR   R4,SUM             Initialize R4 to the sum
                                   location for test 1.
        ADR   R5,LIST            Load address of test 1 score
        ADD   R5,R5,#4             for student 1 into R5.
        LDR   R6,J               Initialize outer loop counter
                                   R6 to j.
OUTER   LDR   R7,N               Initialize inner loop
                                   counter R7 to n.
        MOV   R2,R5              Initialize base register R2
                                   to location of student 1 test
                                   score for next inner loop
                                   sum computation.
        MOV   R0,#0              Clear sum accumulator
                                   register R0.
INNER   LDR   R1,[R2],R3,LSL #2  Load test score into R1
                                   and increment R2 by Stride to
                                   point to next test score.
        ADD   R0,R0,R1           Accumulate score into R0.
        SUBS  R7,R7,#1           Check if all student scores
        BGT   INNER                for current test are added.
        STR   R0,[R4],#4         Store sum in memory.
        ADD   R5,R5,#4           Increment R5 to next test
                                   score for student 1.
        SUBS  R6,R6,#1           Check if sums for all test
        BGT   OUTER                scores have been accumulated.
```

3.11. Assume that the subroutine can change the contents of any registers used to pass parameters.

```
       STR   R5,[R13,#4]!       Save [R5] on stack.
       ADD   R1,R0,R1,LSL #2    Load address of A(0,x) into R1.
       ADD   R2,R0,R2,LSL #2    Load address of A(0,y) into R2.
LOOP   LDR   R5,[R1],R4,LSL #2  Load [A(i,x)] into R5
                                  and increment pointer R1
                                  by Stride = 4m.
       LDR   R0,[R2]            Load [A(i,y)] into R0.
       ADD   R0,R0,R5           Add corresponding column entries.
       STR   R0,[R2],R4,LSL #2  Store sum in A(i,y) and
                                  increment pointer R2 by Stride.
       SUBS  R3,R3,#1           Repeat loop until all
       BGT   LOOP                 entries have been added.
       LDR   R5,[R13],#4        Restore [R5] from stack.
       MOV   R15,R14            Return.
```

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

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

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

3.14. **Main Program**

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

Subroutine GETCHAR

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

Subroutine PUTCHAR

```
PUTCHAR   STMFD  R13!,{R4,R14}  Save R4 and Link register.
DISPLAY   LDR    R4,[R2]        Wait for display.
          TST    R4,#8
          BEQ    DISPLAY
          STRB   R3,[R2,#4]     Send character to display.
          LDMFD  R13!,{R4,R15}  Restore R4 and Return.
```

3.15. Address INSTATUS is passed to GETCHAR on the stack; the character read is passed back in the same stack position. The character to be displayed and the address OUTSTATUS are passed to PUTCHAR on the stack in that order. The stack frame structure shown in Figure 3.13 is used.

Main Program

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

Subroutine GETCHAR

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

Subroutine PUTCHAR

```
PUTCHAR   STMFD  SP!,{R2−R4,FP,LR}   Save registers.
          ADD    FP,SP,#12           Load frame pointer.
          LDR    R2,[FP,#8]          Load address OUTSTATUS into
          LDR    R3,[FP,#12]           R2 and character into R3.
DISPLAY   LDR    R4,[R2]             Wait for display.
          TST    R4,#8
          BEQ    DISPLAY
          STRB   R3,[R2,#4]          Send character to display.
          LDMFD  SP!,{R2−R4,FP,PC}   Restore registers and Return.
```

3.16. The first program section reads the characters, stores them in a 3-byte area beginning at CHARSTR, and echoes them to a display. The second section does the conversion to binary and stores the result in BINARY. The I/O device addresses INSTATUS and OUTSTATUS are in registers R1 and R2.

```
          ADR    R0,CHARSTR          Initialize memory pointer
          MOV    R5,#3                 R0 and counter R5.
READ      LDR    R3,[R1]             Read a character and
          TST    R3,#8                 store it in memory.
          BEQ    READ
          LDRB   R3,[R1,#4]
          STRB   R3,[R0],#1
ECHO      LDR    R4,[R2]             Echo the character
          TST    R4,#8                 to the display.
          BEQ    ECHO
          STRB   R3,[R2,#4]
          SUBS   R5,R5,#1            Check if all three
          BGT    READ                  characters have been read.
CONVERT   ADR    R0,CHARSTR          Initialize memory pointers
          ADR    R1,HUNDREDS           R0, R1, and R2.
          ADR    R2,TENS
          LDRB   R3,[R0,]#1          Load high-order BCD digit
          AND    R3,R3,#&F             into R3.
          LDR    R4,[R1,R3,LSL #2]   Load binary value
                                       corresponding to decimal
                                       hundreds value into
                                       accumulator register R4.
          LDRB   R3,[R0],#1          Load middle BCD digit
          AND    R3,R3,#&F             into R3.
          LDR    R3,[R2,R3,LSL #2]   Load binary value
                                       corresponding to
                                       decimal tens value
                                       into register R3.
          ADD    R4,R4,R3            Accumulate into R4.
          LDRB   R3,[R0],#1          Load low-order BCD digit
          AND    R3,R3,#&F             into R3.
          ADD    R4,R4,R3            Accumulate into R4.
          STR    R4,BINARY           Store converted value
                                       into location BINARY.
```

3.17. (*a*) The names FP, SP, LR, and PC, are used for registers R12, R13, R14, and R15 (frame pointer, stack pointer, link register, and program counter). The 3-byte memory area for the characters begins at address CHARSTR; and the converted binary value is stored at BINARY.

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

The stack frame format used is like Figure 3.13.

A possible main program is:

Main program

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

First subroutine READCHARS

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

Second subroutine CONVERT

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

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

	[R0]
	[R1]
	[R2]
	[R3]
	[R4]
	[R5]
FP →	[FP]
	[LR] = RTNADDR
	CHARSTR
	BINARY
	Original TOS

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

Register assignment:

R0 − Data byte to append to or remove from queue

R1 − IN pointer

R2 − OUT pointer

R3 − Address of first queue byte location

R4 − Address of last queue byte location (= [R3] + k −1)

R5 − Auxiliary register for address of next appended byte.

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

APPEND routine:

```
MOV     R5,R1
ADD     R1,R1,#1      Increment R1 Modulo k.
CMP     R1,R4
MOVGT   R1,R3
CMP     R1,R2         Check if queue is full.
MOVEQ   R1,R5         If queue full, restore
BEQ     QUEUEFULL       IN pointer and send
                        message that queue is full.
STRB    R0,[R5]       If queue not full,
                        append byte and continue.
```

REMOVE routine:

```
CMP     R1,R2         Check if queue is empty.
BEQ     QUEUEEMPTY    If empty, send message.
LDRB    R0,[R2],#1    Otherwise, remove byte
CMP     R2,R4           and increment R2
MOVGT   R2,R3           Modulo k.
```

3.19. Program trace:

TIME	R0	R2	R3	LIST	LIST +1	LIST +2	LIST +3	LIST +4
After 1st	120	1004	1000	106	13	67	45	120
After 2nd	106	1003	1000	67	13	45	106	120
After 3rd	67	1002	1000	45	13	67	106	120
After 4th	45	1001	1000	13	45	67	106	120

3.20. **Calling program**

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

Subroutine SORT

```
SORT    STMFD   R13!,{R0−R3,R5,R14}   Save registers.
        LDR     R0,[R4],#4            Initialize outer loop
        ADD     R2,R4,R0,LSL #2         base register R2
                                        to LIST + 4n.
        ADD     R5,R4,#4              Load LIST + 4 into
                                        register R5.
OUTER   LDR     R0,[R2,#−4]!          Comments similar
        MOV     R3,R2                   as in Figure 3.15.
INNER   LDR     R1,[R3,#−4]!
        CMP     R1,R0
        STRGT   R1,[R2]
        STRGT   R0,[R3]
        MOVGT   R0,R1
        CMP     R3,R4
        BNE     INNER
        CMP     R2,R5
        BNE     OUTER
        LDMFD   R13!,{R0−R3,R5,R15}   Restore registers
                                        and return.
```

3.21. The alternative program from the instruction labeled OUTER to the end is:

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

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

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

```
        MOV   R0,#1000
        MOV   R1,#0
        MOV   R2,#0
        MOV   R3,#0
LOOP    LDR   R5,[R0,#8]
        ADD   R1,R1,R5
        LDR   R5,[R0,#12]
        ADD   R2,R2,R5
        LDR   R5,[R0,#16]
        ADD   R3,R3,R5
        LDR   R0,[R0,#4]
        CMP   R0,#0
        BNE   LOOP
        STR   R1,SUM1
        STR   R2,SUM2
        STR   R3,SUM3
```

3.23. If the ID of the new record matches the ID of the Head record, the new record will become the new Head. If the ID matches that of a later record, it will be inserted immediately after that record, including the case where the matching record is the Tail.

Modify Figure 3.16 as follows:

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

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

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

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

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

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

- Remove the instruction with the comment “Go further?” because it has already been done in the previous bullet.

3.24. If the list is empty, the result is unpredictable because the second instruction compares the new ID with the contents of memory location zero. If the list is not empty, the program continues until RCURRENT points to the Tail record. Then the instruction at LOOP loads zero into RNEXT and the result is unpredictable.

Replace Figure 3.17 with the following code:

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

PART II: 68000

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

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

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

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

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

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

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

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

3.27. Program trace:

TIME	D0	D1	A2	N	NUM1	SUM
after 1st ADD.W	83	5	2402	5	2400	0
after 2nd ADD.W	128	4	2404	5	2400	0
after 3rd ADD.W	284	3	2406	5	2400	0
after 4th ADD.W	34	2	2408	5	2400	0
after 5th ADD.W	134	1	2410	5	2400	0
after last MOVE.L	134	0	2410	5	2400	134

3.28. (*a*) This program finds the location of the smallest element in a list whose starting address is stored in MEM1, and size in MEM2. The smallest element is stored in location DESIRED.

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

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

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

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

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

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

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

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

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

3.31. A 68000 program for string matching:

```
         MOVEA.L  #STRING,A0      Get location of STRING
         MOVE.W   N,D0            Load D0 with appropriate
         MOVE.W   M,D1              count for "match attempts"
         SUB.W    D1,D0
LOOP     MOVEA.L  #SUBSTRING,A1   Get location of SUBSTRING
         MOVE.W   M,D1            Get size of SUBSTRING
         MOVE.L   A0,A2           Save location in STRING at which
                                    comparison will start
MATCHER  DBRA     D1,SUCCESS
         CMP.B    (A0)+,(A1)+     Compare and advance pointers
         BEQ      MATCHER         If same, check next character
         MOVEA.L  A2,A0           Match failed; advance starting
         ADDQ.L   #1,A0             character position in STRING
         DBRA     D0,LOOP         Check if end of STRING
         MOVE.L   #0,D0           Substring not found
         BRA      NEXT
SUCCESS  MOVEA.L  A2,D0           Save location where match found
NEXT     Next instruction
```

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

3.32. A 68000 program to generate the first n numbers of the Fibonacci series:

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

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

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

```
NEXT   MOVE.B   (A0),D0     Get character
       CMP.B    #$20,D0     Check if space character
       BEQ      END
       ANDI.B   #$DF,D0     Clear bit 5
       MOVE.B   D0,(A0)+    Store converted character
       BRA      NEXT
END    Next instruction
```

3.34. Let Stride = $2(j + 1)$, which is the distance in bytes between scores on a particular test for adjacent students in the list.

```
        MOVE      J,D3        Compute Stride = 2(j + 1)
        ADDQ      #1,D3
        LSL       #1,D3
        MOVEA.L   #SUM,A4     Use A4 as pointer to the sums
        MOVEA.L   #LIST,A5    Use A5 as pointer to scores
        ADDQ      #2,A5         for student 1
        MOVE      J,D6        Use D6 as outer loop counter
        SUBQ      #1,D6       Adjust for use of DBRA instruction
OUTER   MOVE      N,D7        Use D7 as inner loop counter
        SUBQ      #1,D7       Adjust for use of DBRA instruction
        MOVE      A5,A2       Use A2 as base for scanning test scores
        CLR       D0          Use D0 as sum accumulator
INNER   ADD       [A2],D0     Accumulate test scores
        ADD       D3,A2       Point to next score
        DBRA      D7,INNER    Check if score for current test
                                for all students have been added
        MOVE      D0,[A4]     Store sum in memory
        ADDQ      #2,A5       Increment to next test
        ADDQ      #2,A4       Point to next sum
        DBRA      D6,OUTER    Check if scores for all tests
                                have been accumulated
```

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

```
       MOVE     #N,D0
       SUBQ.W   #1,D0          Initialize D0 to n − 1
READ   BTST.W   #3,INSTATUS
       BEQ      READ           Wait for data ready
       MOVE.B   DATAIN,D1      Get new character
       MOVE.B   D1,−(A0)       Push on user stack
ECHO   BTST.W   #3,OUTSTATUS
       BEQ      ECHO           Wait for terminal ready
       MOVE.B   D1,DATAOUT     Output new character
       DBRA     D0,READ        Read next character
```

3.36. Assume that most of the time between successive characters being struck is spent in the two-instruction wait loop that starts at location READ. The BEQ READ instruction is executed once every 40 ns while this loop is being executed. There are $10^9/10 = 10^8$ ns between successive characters. Therefore, the BEQ READ instruction is executed $10^8/40 = 2.5 \times 10^6$ times per character entered.

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

Main Program

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

Subroutine GETCHAR

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

Subroutine PUTCHAR

```
PUTCHAR   BTST.W   #3,(A2)    Wait for display.
          BEQ      PUTCHAR
          MOVE.B   D0,(A3)    Send character to display.
          RTS                 Return.
```

3.38. Addresses INSTATUS and DATAIN are pushed onto the processor stack in that order by the main program as parameters for GETCHAR. The character read is passed back to the main program in the DATAIN position on the stack. The addresses OUTSTATUS and DATAOUT and the character to be displayed are pushed onto the processor stack in that order by the main program as parameters for PUTCHAR. A stack structure like that shown in Figure 3.29 is used.

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

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

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

Main Program

```
READLINE  MOVE.L  #INSTATUS,−(A7)    Push address parameters
          MOVE.L  #DATAIN,−(A7)        onto the stack.
          BSR     GETCHAR            Call character read subroutine.
          MOVE.L  (A7)+,D0           Pop long word containing
          MOVE.B  D0,(A0)+             character from top of
                                       stack into D0 and
                                       store character into memory.
          ADDI    #4,A7              Remove INSTATUS from stack.
          MOVE.L  #OUTSTATUS,−(A7)   Push address parameters
          MOVE.L  #DATAOUT,−(A7)       onto stack.
          MOVE.L  D0,−(A7)           Push long word containing
                                       character onto stack.
          BSR     PUTCHAR            Call character display subroutine.
          ADDI    #12,A7             Remove three parameters from stack.
          CMPI.B  #CR,D0             Check for end-of-line character.
          BNE     READLINE
```

Subroutine GETCHAR

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

Subroutine PUTCHAR

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

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

Register assignment:

D0 − Data byte to append to or remove from queue

A1 − IN pointer

A2 − OUT pointer

A3 − Address of first queue byte location

A4 − Address of last queue byte location (= [A3] + k −1)

A5 − Auxiliary register for address of next appended byte

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

APPEND routine:

```
         MOVEA.L  A1,A5
         ADDQ.L   #1,A1          Increment A1 Modulo k.
         CMPA.L   A1,A4
         BGE      CHECK
         MOVEA.L  A3,A1
CHECK    CMPA.L   A1,A2          Check if queue is full.
         BNE      APPEND         If queue not full, append byte.
         MOVEA.L  A5,A1          Otherwise, restore
         BRA      QUEUEFULL        IN pointer and send
                                   message that queue is full.
APPEND   MOVE.B   D0,[A5]        Append byte.
```

REMOVE routine:

```
         CMPA.L   A1,A2          Check if queue is empty.
         BEQ      QUEUEEMPTY     If empty, send message.
         MOVE.B   (A2)+,D0       Otherwise, remove byte
         CMPA.L   A2,A4            and increment A2
         BGE      NEXT             Modulo k.
         MOVEA.L  A3,A2
NEXT     ...
```

3.40. Using the same assumptions as in Problem 3.35 and its solution, a 68000 program to convert 3 input decimal digits to a binary number is:

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

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

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

```
GETDECIMAL   MOVEM.L   D0/A0−A1,−(A7)     Save registers
             MOVEA.L   20(A7),A0          Get string buffer address
             MOVE.B    #2,D0              Use D0 as character counter
READ         BTST.W    #3,INSTATUS
             BEQ       READ
             MOVE.B    DATAIN,(A0)+       Get and store character
             DBRA      D0,READ            Repeat until all characters
                                            received
             MOVE.L    16(A7),A1          Pointer to result
             BSR       CONVERSION
             MOVEM.L   (A7)+,D0/A0−A1     Restore registers
             RTS
CONVERSION   MOVEM.L   D0−D1,−(A7)        Save registers
             MOVE.B    −(A0),D0           Get least sig. digit
             AND.W     #$0F,D0            Numeric value of digit
             MOVE.B    −(A0),D1           Get tens digit
             AND.W     #$0F,D1            Numeric value of digit
             ASL       #1,D1
             ADD.W     TENS(D1),D0        Add tens value
             MOVE.B    −(A0),D1           Get hundreds digit
             AND.W     #$0F,D1            Numeric value of digit
             ASL       #1,D1
             ADD.W     HUNDREDS(D1),D0    Add hundreds value
             MOVE.W    D0,(A1)            Store result
             MOVEM.L   (A7)+,D0−D1        Restore registers
             RTS
```

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

Return address of CONVERSION
$D0_{MAIN}$
$A1_{MAIN}$
$A0_{MAIN}$
Return address of GETDECIMAL
Result address
Buffer address
ORIG TOS

3.42. Assume that the subroutine can change the contents of any registers used to pass parameters. Let Stride = $2m$, which is the distance in bytes between successive word elements in a given column.

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

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

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

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

3.44.

		Bytes/access
MOVEA.L	#LOC,A0	6/3
MOVE.B	(A0)+,D0	2/2
LSL.B	#4,D0	2/1
MOVE.B	(A0),D1	2/2
ANDI.B	#$F,D1	4/2
OR.B	D0,D1	2/1
MOVE.B	D1,PACKED	4/3

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

3.45. The trace table is:

TIME	1000	1001	1002	1003	1004	D1	D2	D3
after 1st BGT OUTER	106	13	67	45	120	3	−1	120
after 2nd BGT OUTER	67	13	45	106	120	2	−1	106
after 3rd BGT OUTER	45	13	67	106	120	1	−1	67
after 4th BGT OUTER	13	45	67	106	120	0	−1	45

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

```
        MOVE    (A1)+,D1     Load number of entries, n
        SUBQ    #2,D1        Outer loop counter ← n − 2 (j: 0 to n − 2)
OUTER   MOVE    D1,D2        Inner loop ← outer loop counter
        MOVEA   A1,A2        Use A2 as a pointer in the inner loop
        ADDQ    #2,A2        k ← j + 1 (k: 1 to n − 1)
INNER   MOVE    (A1),D3      Current maximum value in D3
        CMP     (A2),D3
        BLE     NEXT         If LIST(j) ≤ LIST(k), go to next
        MOVE    (A2),(A1)    Interchange LIST(k)
        MOVE    D3,(A2)        and LIST(j).
NEXT    ADDQ    #2,A2
        DBRA    D2,INNER
        ADDQ    #2,A1
        DBRA    D1,OUTER     If not finished,
        RTS                    return
```

3.47. Use D4 to keep track of the position of the largest element in the inner loop and D5 to record its value.

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

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

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

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

Note that the MOVE instruction that reads the link value into register D0 sets the Z and N flags. The MOVEA instruction does not affect the condition code flags. Hence, the BNE instruction will test the correct values.

3.49. In the program of Figure 3.35, if the ID of the new record matches the ID of the Head record, the new record will become the new Head. If the ID matches that of a later record, it will be inserted immediately after that record, including the case where the matching record is the Tail.

Modify the program as follows.

```
Add the following as the first instruction
INSERTION     MOVE.L   #0,A6         Anticipate a successful insertion
After the instruction labeled HEAD insert
              BEQ      DUPLICATE1    New record matches head
After the BLT INSERT instruction insert
              BEQ      DUPLICATE2    New record matches a record
                                       other than head
Add the following instructions at the end
DUPLICATE1    MOVE.L   A0,A6         Return the address of the head
              RTS
DUPLICATE2    MOVE.L   A3,A6         Return address of matching record
              RTS
```

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

To correct behavior, modify the program as follows.

```
After the first BGT instruction insert
        BLT      ERROR    ID of new record less than head
        MOVE.L   #0,D1    Deletion successful
After the BEQ DELETE instruction insert
        BGT      ERROR    ID of New record is less than
                            that of the next record and
                            greater than the current record
Add the following instruction after DELETE
        MOVE.L   #0,D1    Deletion successful
Add the following instruction at the end
ERROR   RTS               Record not in the list
```

PART III: Intel IA-32

3.51. Initial memory contents are:

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

(*a*) [EBX + ESI*4 + 8] = 1016
EAX ← 10 + 5 = 15

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

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

3.52. (*a*) OK

(*b*) ERROR: Only one operand can be in memory.

(*c*) OK

(*d*) ERROR: Scale factor can only be 1, 2, 4, or 8.

(*e*) OK

(*f*) ERROR: An immediate operand can not be a destination.

(*g*) ERROR: ESP cannot be used as an index register.

3.53. Program trace:

TIME	EAX	EBX	ECX
After 1st execution of LOOP	−113	NUM1 − 4	4
After 2nd execution of LOOP	129	NUM1 − 4	3
After 3rd execution of LOOP	78	NUM1 − 4	2

3.54. Assume bytes are unsigned 8-bit values.

```
            MOV    ECX,N               ECX is list counter.
            LEA    ESI,X               ESI points to X list.
            SUB    ESI,1
            LEA    EDI,Y               EDI points to Y list.
            SUB    EDI,1
            LEA    EDX,LARGER          EDX points to LARGER list.
            SUB    EDX,1
START:      MOV    AL,[ESI + ECX]      Load X byte into AL.
            MOV    BL,[EDI + ECX],     Load Y byte into BL.
            CMP    AL,BL               Compare bytes.
            JAE    XLARGER             Branch if X byte
                                         larger or same.
            MOV    [EDX + ECX],BL      Otherwise, store
                                         Y byte.
            JMP    CHECK
XLARGER     MOV    [EDX + ECX],AL      Store X byte.
CHECK       LOOP   START               Check if done.
```

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

```
          MOV   EDX,N          Compute outer loop count
          SUB   EDX,M           and store in EDX.
          INC   EDX
          LEA   EAX,STRING     Use EAX as a base
                                pointer for each match
                                attempt.
OUTER:    MOV   ESI,EAX        Use ESI and EDI as
          LEA   EDI,SUBSTRING   running pointers for
                                each match attempt.
          MOV   ECX,M          Initialize inner loop counter.
INNER:    MOV   BL,[EDI]       Load next substring byte
          CMP   BL,[ESI]        into BL and compare to
                                corresponding string byte.
          JNE   NOMATCH        If not equal, go to
                                next substring position.
          INC   ESI            If equal, increment running
          INC   EDI             pointers to next byte
                                positions.
          LOOP  INNER          Check if all substring
                                bytes compared.
          JMP   NEXT           If a match is found,
                                exit with string position
                                in EAX.
NOMATCH:  INC   EAX            Increment EAX to next possible
                                substring position.
          DEC   EDX            Check if all positions tried.
          JG    OUTER
          MOV   EAX,0          If yes, load zero into
                                EAX and exit.
NEXT:     ...
```

3.56. This solution assumes that the last number in the series of n numbers can be represented in a 32-bit doubleword, and that $n > 2$.

```
            MOV   ECX,N         Use ECX to count numbers
            SUB   ECX,2          generated after 1.
            LEA   EDI,MEMLOC    Use EDI as a memory
                                 pointer.
            MOV   EAX,0         Store first two numbers
            MOV   [EDI],EAX      from EAX and EBX into
            MOV   EBX,1          memory.
            ADD   EDI,4
            MOV   [EDI],EBX
LOOPSTART:  ADD   EDI,4         Increment memory pointer.
            MOV   EAX,[EDI − 8] Load second last value.
            ADD   EBX,EAX       Add to last value.
            MOV   [EDI],EBX     Store new value.
            LOOP  LOOPSTART     Check if all n numbers
                                 generated.
```

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

```
NEXT:  MOV  BL,[EAX]  Load next character into BL.
       CMP  BL,20H    Check if space character.
       JE   END       If space, exit.
       AND  BL,DFH    Clear bit b5.
       MOV  [EAX],BL  Store converted character.
       INC  EAX       Increment memory pointer.
       JMP  NEXT      Convert next character.
END:   ...
```

3.58. The parameter Stride = $(j + 1)$ is the distance in doublewords between scores on a particular test for adjacent students in the list.

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

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

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

```
            MOV   ECX,32      Load ECX with count value 32.
            MOV   EBX,0       Clear work register EBX.
LOOPSTART:  SHL   EAX,1       Shift contents of EAX left
                               one bit position, moving the
                               high-order bit into the CF flag.
            RCR   EBX,1       Rotate EBX right one bit
                               position, including the CF flag.
            LOOP  LOOPSTART   Check if finished.
            MOV   EAX,EBX     Load reversed pattern into EAX.
```

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

Register assignment:

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

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

Append routine:

```
            MOV   EDX,ESI       Save current value of IN
                                 pointer ESI in auxiliary
                                 register EDX.
            INC   ESI           These four instructions
            CMP   ECX,ESI        increment ESI Modulo k.
            JGE   CHECK
            MOV   ESI,EBX
CHECK:      CMP   EDI,ESI       Check if queue is full.
            JNE   APPEND        If not full, append byte.
            MOV   ESI,EDX       Otherwise, restore IN pointer
            JMP   QUEUEFULL      and send message that
                                 queue is full.
APPEND:     MOV   [EDX],AL      Append byte.
```

Remove routine:

```
            CMP   EDI,ESI       Check if queue is empty.
            JE    QUEUEEMPTY    If empty, send message.
            MOV   AL,[EDI]      Otherwise, remove byte and
            INC   EDI            increment EDI Modulo k.
            CMP   ECX,EDI
            JGE   NEXT
            MOV   EDI,EBX
NEXT:       ...
```

3.61. This program is similar to Figure 3.44; and it makes the same assumptions about status word bit locations.

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

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

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

Main Program

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

Subroutine GETCHAR

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

Subroutine PUTCHAR

```
PUTCHAR:  BT    DWORD PTR [ESI],3   Wait for display.
          JNC   PUTCHAR
          MOV   [EDI],AL            Display character.
          RET
```

3.64. Addresses INSTATUS and DATAIN are pushed onto the processor stack in that order by the main program as parameters for GETCHAR. The character read is passed back to the main program in the DATAIN position on the stack. The addresses OUTSTATUS and DATAOUT and the character to be displayed are pushed onto the processor stack in that order by the main program as parameters for PUTCHAR. A stack structure like that shown in Figure 3.46 is used.

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

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

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

Main Program

```
READLINE:  PUSH   OFFSET INSTATUS    Push address parameters
           PUSH   OFFSET DATAIN       onto the stack.
           CALL   GETCHAR
           POP    EAX                Pop the doubleword
                                      containing the character
                                      read into EAX.
           MOV    [ECX],AL           Store character in
                                      low-order byte of EAX
                                      into the memory.
           INC    ECX                Increment the memory pointer.
           ADD    ESP,4              Remove parameter INSTATUS
                                      from top of the stack.
           PUSH   OFFSET OUTSTATUS   Push address parameters
           PUSH   OFFSET DATAOUT      onto the stack.
           PUSH   EAX                Push doubleword containing
                                      the character to be displayed
                                      onto the stack.
           CALL   PUTCHAR
           ADD    ESP,12             Remove three parameters
                                      from the stack.
           CMP    AL,CR              Check for end-of-line
                                      character.
           JNE    READLINE           Go back for more.
```

Subroutine GETCHAR

```
GETCHAR:  PUSH  EAX                   Save registers to be
          PUSH  EBX                    used in the subroutine.
          PUSH  EDX
          MOV   EBX,[ESP + 20]        Load INSTATUS into EBX.
          MOV   EDX,[ESP + 16]        Load DATAIN into EDX.
READ:     BT    DWORD PTR [EBX],3     Wait for character.
          JNC   READ
          MOV   AL,[EDX]              Read character into AL.
          MOV   [ESP + 16],EAX        Overwrite DATAIN in the
                                       stack with the doubleword
                                       containing the character read.
          POP   EDX                   Restore registers.
          POP   EBX
          POP   EAX
          RET
```

Subroutine PUTCHAR

```
PUTCHAR:  PUSH  EAX                   Save registers to be
          PUSH  ESI                    used in the subroutine.
          PUSH  EDI
          MOV   ESI,[ESP + 24]        Load OUTSTATUS.
          MOV   EDI,[ESP + 20]        Load DATAOUT.
          MOV   EAX,[ESP + 16]        Load doubleword containing
                                       character to be displayed
                                       into register EAX.
DISPLAY:  BT    DWORD PTR [ESI],3     Wait for the display.
          JNC   DISPLAY
          MOV   [EDI],AL              Display character.
          POP   EDI                   Restore registers.
          POP   ESI
          POP   EAX
          RET
```

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

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

READ:   BT    INSTATUS,3
        JNC   READ                          Wait for new character
        MOV   AL,DATAIN                     Get new character
        AND   AL,0FH                        Convert to equivalent
                                              binary value
        RET
```

3.66. (*a*) The subroutines convert 3 decimal digits to a binary value.

```
GETCHARS:  PUSH  ECX                          Save registers.
           PUSH  EBX
           PUSH  EAX
           MOV   ECX,3                        Use ECX as character
                                                counter.
           MOV   EBX,[ESP + 20]               Load character buffer
                                                address into EBX.
READ:      BT    INSTATUS,3
           JNC   READ
           MOV   BYTE PTR [EBX],DATAIN        Get and store character.
           INC   EBX                          Increment buffer pointer.
           LOOP  READ                         Repeat until all
                                                characters received.
           MOV   EAX,[ESP + 16]               Pointer to result.
           CALL  CONVERT
           POP   EAX                          Restore registers.
           POP   EBX
           POP   ECX
           RET


CONVERT:   PUSH  ECX                          Save registers.
           PUSH  EDX
           DEC   EBX                          Load low-order digit
           MOV   DL,[EBX]                       numerical value
           AND   DL,0FH                         into EDX.
           DEC   EBX                          Load and add
           MOV   CL,[EBX]                       tens digit value
           AND   CL,0FH                         into EDX.
           ADD   EDX,[TENS + ECX * 4]
           DEC   EBX                          Load and add
           MOV   CL,[EBX]                       hundreds digit value
           AND   CL,0FH                         into EDX.
           ADD   EDX,[HUNDREDS + ECX * 4]
           MOV   [EAX],EDX                    Store result.
           POP   EDX                          Restore registers.
           POP   ECX
           RET
```

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

...
Return address to GETCHARS
[EAX]
[EBX]
[ECX]
Return address to Main
Result address
Buffer address
ORIGINAL TOS
...

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

```
        SHL   EBX,2                Set Stride in EBX.
        SUB   EDI,ESI              Set EDI to y − x.
        SHL   ESI,2                Set EDX to
        ADD   EDX,ESI                address A(0,x).
LOOP:   MOV   ESI,[EDX]            Add A(i,x) to A(i,y).
        ADD   [EDX + EDI * 4],ESI
        ADD   EDX,EBX              Move to next row.
        DEC   EAX                  Repeat loop until all
        JG    LOOP                   entries have been added.
        RET                        Return.
```

3.68. Program trace:

TIME	EDI	ECX	DL	LIST	LIST +1	LIST +2	LIST +3	LIST +4
After 1st	3	−1	120	106	13	67	45	120
After 2nd	2	−1	106	67	13	45	106	120
After 3rd	1	−1	67	45	13	67	106	120
After 4th	0	−1	45	13	45	67	106	120

3.69. Assume that the calling program passes the address LIST − 4 to the subroutine in register EAX.

Subroutine SORT

```
SORT:   PUSH   EDI                     Save registers.
        PUSH   ECX
        PUSH   EDX
        MOV    EDI,[EAX]               Initialize outer loop index
        DEC    EDI                       register EDI to j = n − 1.
        ADD    EAX,4                   Set EAX to contain LIST.
OUTER:  MOV    ECX,EDI                 Initialize inner loop index
        DEC    ECX                       register to k = j − 1.
        MOV    EDX,[EAX + EDI * 4]     Load LIST(j) into EDX.
INNER:  CMP    [EAX + ECX * 4],EDX     Compare LIST(k) to LIST(j).
        JLE    NEXT                    If LIST(k) ≤ LIST(j),
                                         go to next k index entry;
        XCHG   [EAX + ECX * 4],EDX     Otherwise, interchange LIST(k)
        MOV    [EAX + EDI * 4],EDX       and LIST(j), leaving
                                         (new) LIST(j) in EDX.
NEXT:   DEC    ECX                     Decrement inner loop index k.
        JGE    INNER                   Repeat or terminate inner loop.
        DEC    EDI                     Decrement outer loop index j.
        JG     OUTER                   Repeat or terminate outer loop.
        POP    EDX                     Restore registers.
        POP    ECX
        POP    EDI
        RET
```

3.70. Use register ESI to keep track of the index position of the largest element in the inner loop, and use register EDX (DL) to record its value. Register EBX (BL) is used to hold sublist values to be compared to the current largest value.

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

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

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

```
        MOV   EAX,0
        MOV   EBX,0
        MOV   ECX,0
LOOP:   ADD   EAX,[ESI + 8]     Accumulate scores for test 1.
        ADD   EBX,[ESI + 12]    Accumulate scores for test 2.
        ADD   ECX,[ESI + 16]    Accumulate scores for test 3.
        MOV   ESI,[ESI + 4]     Get link.
        CMP   ESI,0             Check if done.
        JNE   LOOP
        MOV   SUM1,EAX          Store sums.
        MOV   SUM2,EBX
        MOV   SUM3,ECX
```

3.72. If the ID of the new record matches the ID of the Head record of the current list, the new record will be inserted as the new Head. If the ID of the new record matches the ID of a later record in the current list, the new record will be inserted immediately after that record, including the case where the matching record is the Tail record. In this latter case, the new record becomes the new Tail record.

Modify Figure 3.51 as follows:

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

```
INSERTION:  MOV   EDX, 0               Anticipate successful
                                         insertion of the new
                                         record.
            MOV   RNEWID,[RNEWREC]     (Existing instruction.)
```

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

```
             JNE   CONTINUE1   Three new instructions.
             MOV   EDX,RHEAD
             RET
CONTINUE1:   JG    SEARCH      (Existing instruction.)
```

- After the fourth CMP instruction, insert the following three instructions:

```
             JNE   CONTINUE2   Three new instructions.
             MOV   EDX,RNEXT
             RET
CONTINUE2:   JL    INSERT      (Existing instruction.)
```

3.73. If the list is empty, the result is unpredictable because the first instruction will compare the ID of the new record to the contents of memory location zero. If the list is not empty, the following happens. If the contents of RIDNUM are less than the ID number of the Head record, the Head record will be deleted. Otherwise, the routine loops until register RCURRENT points to the Tail record. Then RNEXT gets loaded with zero by the instruction at LOOPSTART, and the result is unpredictable.

Replace Figure 3.52 with the following code:

```
DELETION:    CMP   RHEAD, 0               If the list is empty,
             JNE   CHECKHEAD                return with RIDNUM
             RET                            unchanged.
CHECKHEAD:   CMP   RIDNUM,[RHEAD]         Check if Head record
             JNE   CONTINUE1                is to be deleted and
             MOV   RHEAD,[RHEAD + 4]        perform deletion if it
             MOV   RIDNUM,0                 is, returning with zero
             RET                            in RIDNUM.
CONTINUE1:   MOV   RCURRENT,RHEAD         Otherwise, continue
                                            searching.
LOOPSTART:   MOV   RNEXT,[RCURRENT + 4]
             CMP   RNEXT,0                If all records checked,
             JNE   CHECKNEXT                return with IDNUM
             RET                            unchanged.
CHECKNEXT:   CMP   RIDNUM,[RNEXT]         Check if next record is
             JNE   CONTINUE2                to be deleted and
             MOV   RTEMP,[RNEXT + 4]        perform deletion if
             MOV   [RCURRENT + 4],RTEMP     it is, returning with
             MOV   RIDNUM,0                 zero in RIDNUM.
             RET
CONTINUE2:   MOV   RCURRENT,RNEXT         Otherwise, continue
             JMP   LOOPSTART                the search.
```