Appendix B
The Altera Nios II Processor

B.1. Program that computes SUM = 580 + 6840 + 80000:

```
            movia   r2, NUMBERS          /* Load address of numbers.     */
            ldw     r3, (r2)             /* Load 580.                    */
            ldw     r4, 4(r2)            /* Load 68400.                  */
            add     r3, r3, r4           /* Generate 580 + 80000.        */
            ldw     r4, 8(r2)            /* Load 80000.                  */
            add     r3, r3, r4           /* Generate the final sum.      */
            stw     r3, 12(r2)           /* Store the sum.               */
            next instruction

            .org    0x500
NUMBERS:    .word   580, 68400, 80000    /* Numbers to be added.         */
SUM:        .skip   4                    /* Space for the sum.           */
```

B.2. Program that computes ANSWER = A × B + C × D:

```
            movia   r2, A           /* Get the address of A.        */
            ldw     r3, (r2)        /* Load the operand A.          */
            movia   r2, B           /* Get the address of B.        */
            ldw     r4, (r2)        /* Load the operand B.          */
            mul     r5, r3, r4      /* Generate A x B.              */
            movia   r2, C           /* Get the address of C.        */
            ldw     r3, (r2)        /* Load the operand C.          */
            movia   r2, D           /* Get the address of D.        */
            ldw     r4, (r2)        /* Load the operand D.          */
            mul     r6, r3, r4      /* Generate C x D.              */
            add     r7, r5, r6      /* Compute the final answer.    */
            movia   r2, ANSWER      /* Get the address and          */
            stw     r7, (r2)        /*   store the answer.          */
            next instruction

            .org    0x500
A:          .word   100             /* Test data.                   */
B:          .word   50
C:          .word   20
D:          .word   400
ANSWER:     .skip   4               /* Space for the answer.        */
```

B.3. The following program determines the number of negative integers.

```
            movia  r2, N           /* Get the address N.                */
            ldw    r2, (r2)        /* Load the size of the list.        */
            mov    r3, r0          /* Initialize the counter to 0.      */
            movia  r4, NUMBERS     /* Load address of the first number. */
LOOP:       ldw    r5, (r4)        /* Get the next number.              */
            bge    r5, r0, NEXT    /* Test if number is negative.       */
            addi   r3, r3, 1       /* Increment the count.              */
NEXT:       addi   r4, r4, 4       /* Increment the pointer to list.    */
            subi   r2, r2, 1       /* Decrement the list counter.       */
            bgt    r2, r0, LOOP    /* Loop back if not finished.        */
            movia  r6, NEGNUM      /* Get the address NEGNUM.           */
            stw    r3, (r6)        /* Store the result.                 */
            next instruction

            .org   0x500
NEGNUM:     .skip  4               /* Space for the result.             */
N:          .word  6               /* Size of the list.                 */
NUMBERS:    .word  23, −5, −128    /* Test data.                        */
            .word  44, −23, −9
```

B.4. The program can be arranged as:

```
        .org    0x100
        movia   r2, LIST
        mov     r3, r0
        mov     r4, r0
        mov     r5, r0
        movia   r6, N
        ldw     r6, (r6)
LOOP:   ldw     r7, 4(r2)
        add     r3, r3, r7
        ldw     r7, 8(r2)
        add     r4, r4, r7
        ldw     r7, 12(r2)
        add     r5, r5, r7
        addi    r2, r2, 16
        subi    r6, r6, 1
        bgt     r6, r0, LOOP
        movia   r7, SUM1
        stw     r3, (r7)
        movia   r7, SUM2
        stw     r4, (r7)
        movia   r7, SUM3
        stw     r5, (r7)
        next instruction

        .org    0x500
SUM1:   .skip   4                   /* Space for SUM1.    */
SUM2:   .skip   4                   /* Space for SUM2.    */
SUM3:   .skip   4                   /* Space for SUM3.    */
N:      .word   3                   /* Size of the list.  */
LIST:   .word   1234, 62, 85, 75    /* Example records.   */
        .word   1235, 90, 82, 88
        .word   1236, 72, 65, 80
```

B.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.

```
        movia  r2, J            /* Compute and place                        */
        ldw    r2, (r2)         /*   Stride = 4(j + 1)                      */
        addi   r2, r2, 1        /*   into register r2.                      */
        slli   r2, r2, 2
        movia  r3, LIST         /* Initialize register r3 to the location   */
        addi   r3, r3, 4        /*   of the test 1 score for student 1.     */
        movia  r4, SUM          /* Initialize register r4 to the location   */
                                /*   of the sum for test 1.                 */
        movia  r5, J            /* Initialize outer loop counter            */
        ldw    r5, (r5)         /*   r5 to j.                               */
OUTER:  movia  r6, N            /* Initialize inner loop counter            */
        ldw    r6, (r6)         /*   r6 to n.                               */
        mov    r7, r0
        mov    r8, r0           /* Clear the sum register r8.               */
        add    r9, r3, r7       /* Use r9 as an index register.             */
INNER:  ldw    r10, (r9)        /* Accumulate the sum                       */
        add    r8, r8, r10      /*   of test scores.                        */
        add    r9, r9, r2       /* Increment index register by Stride value. */
        subi   r6, r6, 1        /* Check if all student scores on current   */
        bgt    r6, r0, INNER    /*   test have been accumulated.            */
        stw    r8, (r4)         /* Store sum of current test scores and     */
        addi   r4, r4, 4        /*   increment sum location pointer.        */
        addi   r3, r3, 4        /* Increment r3 to point to the next        */
                                /*   test score for student 1.              */
        subi   r5, r5, 1        /* Check if the sums for all tests have     */
        bgt    r5, r0, OUTER    /*   been computed.                         */
        next instruction
```

B.6. A Nios II program for Example 2.5 is:

```
          movia   r2, LIST          /* Get the address LIST.          */
          movia   r3, N             /* Get the address N.             */
          ldw     r3, (r3)          /* Initialize outer loop pointer  */
          add     r3, r2, r3        /*   to LIST + n.                 */
OUTER:    subi    r3, r3, 1         /* Decrement the pointer.         */
          ble     r3, r2, DONE      /* Check if last entry.           */
          ldb     r5, (r3)          /* Starting max value in sublist. */
          subi    r4, r3, 1         /* Initialize inner loop pointer. */
INNER:    ldb     r6, (r4)          /* Check if the next entry        */
          bge     r5, r6, NEXT      /*   is lower.                    */
          stb     r6, (r3)          /* If yes, then swap              */
          stb     r5, (r4)          /*   the entries and              */
          mov     r5, r6            /*   update the max value.        */
NEXT:     subi    r4, r4, 1         /* Adjust the inner loop pointer. */
          bge     r4, r2, INNER
          br      OUTER
          next instruction

          .org    0x500
N:        .word   10                /* Size of the list.              */
LIST:     .string ”zZbB53kK24”      /* Test data.                     */
```

B.7. The Nios II implementation is:

```
                .equ    TIM_STATUS, 0x4020
Exception Handler
                .org    0x020
ILOC:           subi    sp, sp, 12              /* Save registers.                      */
                stw     ra, 8(sp)
                stw     et, 4(sp)
                stw     r2, (sp)
                rdctl   et, ipending            /* Check contents of ipending.          */
                andi    r2, et, 8               /* Check if request from timer.         */
                beq     r2, r0, NEXT
                movia   r2, TIM_STATUS
                ldb     r2, (r2)                /* Clear TIRQ and ZERO bits.            */
                call    DISPLAY                 /* Call the DISPLAY routine.            */
NEXT:           ...                             /* Check for other interrupts.          */

                ldw     r2, (sp)                /* Restore registers.                   */
                ldw     et, 4(SP)
                ldw     ra, 8(sp)
                addi    sp, sp, 12
                eret                            /* Return from exception.               */

Main Program
START:          ...                             /* Set up parameters for interrupts.    */
                orhi    r2, r0, 0x3B9A          /* Prepare the initial                  */
                ori     r2, r2, 0xCA00          /*   count value.                       */
                movia   r3, TIM_STATUS
                stw     r2, 8(r3)               /* Set the initial count value.         */
                movi    r2, 7                   /* Set the timer to free run            */
                stb     r2, 4(r3)               /*   and enable interrupts.             */
                rdctl   r2, ienable
                ori     r2, r2, 8               /* Enable timer interrupts in           */
                wrctl   ienable, r2             /*   the processor control register.    */
                rdctl   r2, status
                ori     r2, r2, 1
                wrctl   status, r2              /* Set PIE bit in status register.      */
COMPUTE:        next instruction
```

B.8. A possible program is:

```
        .equ    DIGIT, 0x800               /* Location of ASCII-encoded digit.    */
        .equ    SEVEN, 0x4030              /* Address of 7-segment display.       */
        movia   r6, DIGIT                  /* Get the address DIGIT. */
        ldb     r2, (r6)                   /* Load the ASCII-encoded digit.       */
        andi    r3, r2, 0xF0               /* Extract high-order bits of ASCII.   */
        andi    r2, r2, 0x0F               /* Extract the decimal number.         */
        movi    r4, 0x30                   /* Check if high-order bits of         */
        beq     r3, r4, HIGH3              /*   ASCII code are 0011.              */
        movi    r2, 0x0F                   /* Not a digit, display a blank.       */
HIGH3:  ldb     r5, TABLE(r2)              /* Get the 7-segment pattern.          */
        movia   r6, SEVEN
        stb     r5, (r6)                   /* Display the digit.                  */

        .org    0x1000
TABLE:  .byte   0x7E,0x30,0x6D,0x79        /* Table that contains                 */
        .byte   0x33,0x5B,0x5F,0x70        /*   the necessary                     */
        .byte   0x7F,0x7B,0x00,0x00        /*   7-segment patterns.               */
        .byte   0x00,0x00,0x00,0x00
```

B.9. The program for Problem B.8 can be augmented as follows:

```
         .equ    DIGIT, 0x800          /* Location of ASCII-encoded digit.   */
         .equ    SEVEN, 0x4030         /* Address of 7-segment display.      */
         movia   r6, DIGIT             /* Get the address DIGIT. */
         ldb     r2, (r6)              /* Load the ASCII-encoded digit.      */
         andi    r3, r2, 0xF0          /* Extract high-order bits of ASCII.  */
         andi    r2, r2, 0x0F          /* Extract the decimal number.        */
         movi    r4, 0x30              /* Check if high-order bits of        */
         beq     r3, r4, HIGH3         /*   ASCII code are 0011.             */
         movi    r2, 0x0F              /* Not a digit, display a blank.      */
HIGH3:   movia   r7, TABLE             /* Compute the address of the         */
         add     r7, r7, r2            /*   required table entry.            */
         ldb     r5, (r7)              /* Get the 7-segment pattern.         */
         movia   r6, SEVEN
         stb     r5, (r6)              /* Display the digit.                 */

         .org    0x10100
TABLE:   .byte   0x7E,0x30,0x6D,0x79   /* Table that contains                */
         .byte   0x33,0x5B,0x5F,0x70   /*   the necessary                    */
         .byte   0x7F,0x7B,0x00,0x00   /*   7-segment patterns.              */
         .byte   0x00,0x00,0x00,0x00
```

B.10. Assuming the display interface registers in Figure 3.3, the following program can be used:

```
          movia  r2, LOC                  /* Get the address LOC.            */
          movia  r3, DISP_DATA            /* Get the address of display.     */
          movi   r4, 10                   /* Initialize the byte counter.    */
LOOP:     ldbu   r5, (r2)                 /* Load a byte.                    */
          andi   r6, r5, 0xF0             /* Select the high-order 4 bits.   */
          srli   r6, r6, 4                /* Shift right by 4 bit positions. */
          ldb    r6, TABLE(r6)            /* Get the character for display.  */
          call   DISPLAY
          andi   r6, r5, 0x0F             /* Select the low-order 4 bits.    */
          ldb    r6, TABLE(r6)            /* Get the character for display.  */
          call   DISPLAY
          movi   r6, 0x20                 /* ASCII code for SPACE.           */
          call   DISPLAY
          addi   r2, r2, 1                /* Increment the pointer.          */
          subi   r4, r4, 1                /* Decrement the byte counter.     */
          bgt    r4, r0, LOOP             /* Branch back if not finished.    */
          next instruction

DISPLAY:  ldbio  r7, 4(r3)
          andi   r7, r7, 4                /* Check the DOUT flag.            */
          beq    r7, r0, DISPLAY
          stb    r6, (r3)                 /* Send the character to display.  */
          ret

          .org   0x1000
TABLE:    .byte  0x30,0x31,0x32,0x33      /* Table that contains             */
          .byte  0x34,0x35,0x36,0x37      /*   the necessary                 */
          .byte  0x38,0x39,0x0041,0x42    /*   ASCII characters.             */
          .byte  0x43,0x44,0x45,0x46
```

B.11. Assuming the display interface registers in Figure 3.3, a possible program is:

```
          movia  r2, BINARY      /* Get the address BINARY.         */
          ldhu   r2, (r2)        /* Load the 16-bit pattern.        */
          movia  r3, DISP_DATA   /* Get the address of display.     */
          movi   r4, 16          /* Initialize the bit counter.     */
          ori    r5, r0, 0x8000  /* Set bit 15 to 1.                */
LOOP:     and    r6, r2, r5      /* Test a bit.                     */
          cmpgt  r7, r6, r0      /* Check if 0 or 1, and then       */
          ori    r7, r7, 0x30    /* form the ASCII character.       */
          call   DISPLAY
          srli   r5, r5, 1       /* Shift to check the next bit.    */
          subi   r4, r4, 1       /* Decrement the bit counter.      */
          bgt    r4, r0, LOOP    /* Branch back if not finished.    */
          next instruction

DISPLAY:  ldbio  r8, 4(r3)
          andi   r8, r8, 4       /* Check the DOUT flag.            */
          beq    r8, r0, DISPLAY
          stb    r7, (r3)        /* Send the character to display.  */
          ret
```

B.12. Setting the timer count to indicate one-second periods, and using polling to detect when the timer reaches the end of a period, the following program can be used:

```
        movia   r2, SEVEN
        movia   r3, TIMER
        orhi    r4, r0, 0x5F5           /* Set the count period in the timer    */
        ori     r4, r4, 0xE100          /*   circuit to the value 0x5F5E100,    */
        stwio   r4, 8(r3)               /*   to provide the desired delay.      */
        movi    r4, 6                   /* Start the timer in the               */
        stbio   r4, 4(r3)               /*   continuous mode.                   */
        mov     r5, r0                  /* Clear the digit counter.             */
LOOP:   ldbio   r6, (r3)                /* Wait for the timer to reach the      */
        andi    r6, r6, 2               /*   end of the one-second period.      */
        beq     r6, r0, LOOP
        ldb     r7, TABLE(r5)           /* Look up the 7-segment pattern        */
        stbio   r7, (r2)                /*   and display it.                    */
        addi    r5, r5, 1               /* Increment the digit counter,         */
        movi    r4, 10                  /*   and check if >9.                   */
        blt     r5, r4, LOOP
        mov     r5, r0                  /* Clear the digit counter.             */
        br      LOOP
        next instruction

        .org    0x1000
TABLE:  .byte   0x7E,0x30,0x6D,0x79     /* Table that contains                  */
        .byte   0x33,0x5B,0x5F,0x70     /*   the necessary                      */
        .byte   0x7F,0x7B,0x00,0x00     /*   7-segment patterns.                */
        .byte   0x00,0x00,0x00,0x00
```

B.13. Assume that the two 7-segment displays are concatenated into one device that is accessed by loading the segment patterns into a 16-bit data register, called SEVEN, in the device interface. Then, setting the timer count to indicate one-second periods, and using polling to detect when the timer reaches the end of a period, the following program can be used:

```
        movia   r2, SEVEN
        movia   r3, TIMER
        orhi    r4, r0, 0x5F5            /* Set the count period in the timer     */
        ori     r4, r4, 0xE100           /*   circuit to the value 0x5F5E100,     */
        stwio   r4, 8(r3)                /*   to provide the desired delay.       */
        movi    r4, 6                    /* Start the timer in the                */
        stbio   r4, 4(r3)                /*   continuous mode.                    */
        mov     r5, r0                   /* Clear low and                         */
        mov     r6, r0                   /*   high digit counters.                */
LOOP:   ldbio   r7, (r3)                 /* Wait for the timer to reach the       */
        andi    r7, r7, 2                /*   end of the one-second period.       */
        beq     r7, r0, LOOP
        ldb     r8, TABLE(r5)            /* Look up the 7-segment                 */
        ldb     r9, TABLE(r6)            /*   patterns of digits.                 */
        slli    r9, r9, 8                /* Concatenate the patterns              */
        or      r8, r8, r9               /*   for the two digits, and send        */
        sthio   r8, (r2)                 /*   them to 7-segment displays.         */
        addi    r5, r5, 1                /* Increment the low-digit counter,      */
        movi    r4, 10                   /*   and check if >9.                    */
        blt     r5, r4, LOOP
        mov     r5, r0                   /* Clear the low-digit counter.          */
        addi    r6, r6, 1                /* Increment the high-digit counter,     */
        blt     r6, r4, LOOP             /*   and check if >9.                    */
        mov     r6, r0                   /* Clear the high-digit counter.         */
        br      LOOP
        next instruction

        .org    0x1000
TABLE:  .byte   0x7E,0x30,0x6D,0x79      /* Table that contains                   */
        .byte   0x33,0x5B,0x5F,0x70      /*   the necessary                       */
        .byte   0x7F,0x7B,0x00,0x00      /*   7-segment patterns.                 */
        .byte   0x00,0x00,0x00,0x00
```

B.14. Assume that the four 7-segment displays are concatenated into one device that is accessed by loading the segment patterns into a 32-bit data register, called SEVEN, in the device interface. Then, setting the timer count to indicate one-second periods, and using polling to detect when the timer reaches the end of a period, the following program can be used:

```
            movia   r2, SEVEN
            movia   r3, TIMER
            orhi    r4, r0, 0x5F5           /* Set the count period in the timer  */
            ori     r4, r4, 0xE100          /*   circuit to the value 0x5F5E100,  */
            stwio   r4, 8(r3)               /*   to provide the desired delay.    */
            movi    r4, 6                   /* Start the timer in the             */
            stbio   r4, 4(r3)               /*   continuous mode.                 */
            mov     r5, r0                  /* Set the minute counter to 0.       */
LOOP1:      mov     r6, r0                  /* Clear the seconds counter.         */
LOOP2:      ldbio   r7, (r3)                /* Wait for the timer to reach the    */
            andi    r7, r7, 2               /*   end of one-second period.        */
            beq     r7, r0, LOOP2
            addi    r6, r6, 1               /* Wait for 60 seconds before         */
            movi    r4, 60                  /*   updating the display.            */
            ble     r6, r4, LOOP2
            call    DISPLAY
            addi    r5, r5, 1               /* Increment the minute counter.      */
            movi    r4, 1440                /* When the time 24:00 is reached,    */
            blt     r5, r4, LOOP1           /*   have to reset the minute         */
            mov     r5, r0                  /*   counter to 0.                    */
            br      LOOP1

DISPLAY:    mov     r11, r5
            movi    r12, 600                /* To determine the first digit of    */
            divu    r13, r11, r12           /*   hours, divide by 600.            */
            ldb     r15, TABLE(r13)         /* Get the 7-segment pattern.         */
            slli    r15, r15, 8             /* Make space for next digit.         */
            mul     r14, r13, r12           /* Compute the remainder of the       */
            sub     r11, r11, r14           /*   division operation.              */
            movi    r12, 60                 /* Divide the remainder by 60 to      */
            divu    r13, r11, r12           /*   get the second digit of hours.   */
            ldb     r16, TABLE(r13)         /* Get the 7-segment pattern,         */
            or      r15, r15, r16           /*   concatenate it to the first      */
            slli    r15, r15, 8             /*   digit, and shift.                */
            mul     r14, r13, r12           /* Determine the minutes that have    */
            sub     r11, r11, r14           /*   to be displayed.                 */
            movi    r12, 10                 /* To determine the first digit of    */
            divu    r13, r11, r12           /*   minutes, divide by 10.           */
            ldb     r16, TABLE(r13)         /* Get the 7-segment pattern,         */
            or      r15, r15, r16           /*   concatenate it to the first      */
            slli    r15, r15, 8             /*   two digits, and shift.           */
            mul     r14, r13, r12           /* Compute the remainder, which       */
            sub     r11, r11, r14           /*   is the last digit.               */
            ldb     r16, TABLE(r11)         /* Concatenate the last digit to      */
            or      r15, r15, r16           /*   the preceding 3 digits.          */
            stw     r15, (r2)               /* Display the obtained pattern.      */
            ret

            .org    0x1000
TABLE:      .byte   0x7E,0x30,0x6D,0x79     /* Table that contains                */
            .byte   0x33,0x5B,0x5F,0x70     /*   the necessary                    */
            .byte   0x7F,0x7B               /*   7-segment patterns.              */
```

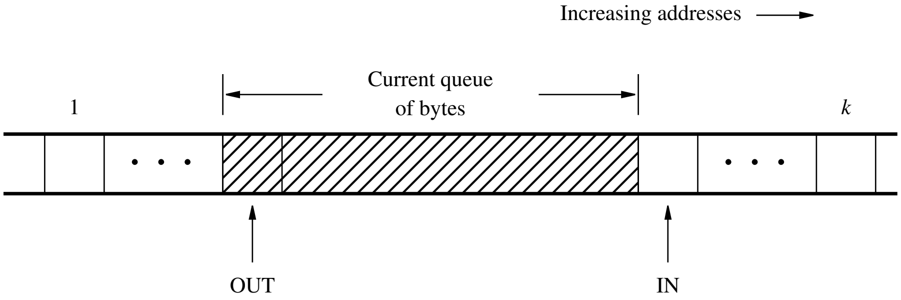
B.15. The contents of register r2 can be safely pushed on the second stack, or poped from it, by calling the following subroutines:

```
SPUSH:   subi    sp, sp, 4              /* Save register r3 on      */
         stw     r3, (sp)               /*   the processor stack.   */
         movia   r3, TOP
         bleu    r5, r3, FULLERROR
         subi    r5, r5, 4
         stw     r2, (r5)
         ldw     r3, (sp)               /* Restore register r3.     */
         addi    sp, sp, 4
         ret

SPOP:    subi    sp, sp, 4              /* Save register r3 on      */
         stw     r3, (sp)               /*   the processor stack.   */
         movia   r3, BOTTOM
         bgeu    r5, r3, EMPTYERROR
         ldw     r2, (r5)
         addi    r5, r5, 4
         ldw     r3, (sp)               /* Restore register r3.     */
         addi    sp, sp, 4
         ret
```

B.16. (*a*) Wraparound must be used. That is, the next item must be entered at the beginning of the memory region, assuming that location is empty.

(*b*) A current queue of bytes is shown in the memory region from byte location 1 to byte location *k* in the following diagram.



The IN pointer points to the location where the next byte will be appended to the queue. If the queue is not full with *k* bytes, this location is empty, as shown in the diagram.

The OUT pointer points to the location containing the next byte to be removed from the queue. If the queue is not empty, this location contains a valid byte, as shown in the diagram.

Initially, the queue is empty and both IN and OUT point to location 1.

(*c*) Initially, as stated in Part *b*, when the queue is empty, both the IN and OUT pointers point to location 1. When the queue has been filled with *k* bytes and none of them have been removed, the OUT pointer still points to location 1. But the IN pointer must also be pointing to location 1, because (following the wraparound rule) it must point to the location where the next byte will be appended. Thus, in both cases, both pointers point to location 1; but in one case the queue is empty, and in the other case it is full.

(*d*) One way to resolve the problem in Part (*c*) is to maintain at least one empty location at all times. That is, an item cannot be appended to the queue if ([IN] + 1) Modulo *k* = [OUT]. If this is done, the queue is empty only when [IN] = [OUT].

(*e*) Append operation:

- LOC ← [IN]
- IN ← ([IN] + 1) Modulo *k*
- If [IN] = [OUT], queue is full. Restore contents of IN to contents of LOC and indicate failed append operation, that is, indicate that the queue was full. Otherwise, store new item at LOC.

Remove operation:

- If [IN] = [OUT], the queue is empty. Indicate failed remove operation, that is, indicate that the queue was empty. Otherwise, read the item pointed to by OUT and perform OUT ← ([OUT] + 1) Modulo *k*.

B.17. Use the following register assignment:

r2 – Item to be appended to or removed from queue
r3 – IN pointer
r4 – OUT pointer
r5 – Address of beginning of queue area in memory
r6 – Address of end of queue area in memory
r7 – Temporary storage for [IN] during append operation

Assume that the queue is initially empty, with [r3] = [r4] = [r5]. The following APPEND and REMOVE routines implement the procedures required in part (*e*) of Problem B.16.

APPEND routine:

```
                mov     r7, r3
                addi    r3, r3, 1          /* Increment IN pointer             */
                bgeu    r6, r3, CHECK      /*    modulo k.                     */
                mov     r3, r5
CHECK:          beq     r3, r4, FULL       /* Check if queue is full.          */
                stb     r2, (r7)           /* If queue not full, append item.  */
                br      CONTINUE
FULL:           mov     r3, r7             /* Restore IN pointer and send      */
                call    QUEUEFULL          /*    message that queue is full.   */
CONTINUE:       . . .
```

REMOVE routine:

```
REMOVE:         beq     r3, r4, EMPTY          /* Check if queue is empty.    */
                ldb     r2, (r4)               /* Remove byte and             */
                addi    r4, r4, 1              /*    increment r4 modulo k.   */
                bgeu    r6, r4, CONTINUE
                mov     r4, r5
                br      CONTINUE
EMPTY:          call    QUEUEEMPTY
CONTINUE:       . . .
```

B.18. The values of OUT signals can be computed using the expression

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

A possible program is:

```
        movia   r2, N           /* Get the number of entries, n, that  */
        ldw     r2, (r2)        /*   that have to be generated.        */
        movia   r3, IN          /* Pointer to the IN list.             */
        movia   r4, OUT         /* Pointer to the OUT list.            */
LOOP:   ldw     r5, (r3)        /* Get the value IN(k) and             */
        srai    r5, r5, 3       /*   divide it by 8.                   */
        ldw     r6, 4(r3)       /* Get the value IN(k+1) and           */
        srai    r6, r6, 2       /*   divide it by 4.                   */
        add     r5, r5, r6
        ldw     r6, 8(r3)       /* Get the value IN(k+2) and           */
        srai    r6, r6, 1       /*   divide it by 2.                   */
        add     r5, r5, r6      /* Compute the sum and                 */
        stw     r5, (r4)        /*   store it in OUT list.             */
        addi    r3, r3, 4       /* Increment the pointers              */
        addi    r4, r4, 4       /*   to IN and OUT lists.              */
        subi    r2, r2, 1       /* Continue until all values in        */
        bgt     r2, r0, LOOP    /*   OUT list have been generated.     */
        next instruction
```

B.19. A sequence of bytes can be copied using the program:

```
            movia   r2, N           /* Load the length parameter        */
            ldw     r2, (r2)        /*   into r2.                       */
            movia   r3, FROM        /* Pointer to from list.            */
            movia   r4, TO          /* Pointer to to list.              */
            call    MEMCPY
            next instruction

MEMCPY:     subi    sp, sp, 12      /* Save registers.                  */
            stw     r5, 8(sp)
            stw     r6, 4(sp)
            stw     r7, (sp)
            add     r5, r3, r2      /* Compute address of the last      */
            subi    r5, r5, 1       /*   entry in the from list.        */
            bgeu    r4, r5, UP      /* Scan upwards if to list          */
            bleu    r4, r3, UP      /*   begins inside from list.       */
            add     r6, r4, r2      /* Compute the pointer for          */
            subi    r6, r6, 1       /*   scanning downwards.            */
DOWN:       ldb     r7, (r5)        /* Transfer a byte and              */
            stb     r7, (r6)
            subi    r5, r5, 1       /*   adjust the pointers downwards. */
            subi    r6, r6, 1
            bge     r5, r3, DOWN
            br      DONE
UP:         ldb     r7, (r3)        /* Transfer a byte and              */
            stb     r7, (r4)
            addi    r3, r3, 1       /*   adjust the pointers upwards.   */
            addi    r4, r4, 1
            bleu    r3, r5, UP
DONE:       ldw     r7, (sp)        /* Restore registers.               */
            ldw     r6, 4(sp)
            ldw     r5, 8(sp)
            addi    sp, sp, 12
            ret
```

B.20. The comparison task can be performed as follows:

```
           movia   r2, N          /* Load the length parameter   */
           ldw     r2, (r2)       /*   into r2.                  */
           movia   r3, FIRST      /* Pointer to first list.      */
           movia   r4, SECOND     /* Pointer to second list.     */
           call    MEMCMP
           next instruction

MEMCMP:    subi    sp, sp, 12     /* Save registers.             */
           stw     r5, 8(sp)
           stw     r6, 4(sp)
           stw     r7, (sp)
           mov     r5, r0         /* Clear the counter.          */
LOOP:      ldb     r6, (r3)       /* Load the bytes that have    */
           ldb     r7, (r4)       /*   to be compared.           */
           beq     r6, r7, NEXT
           addi    r5, r5, 1      /* Increment the counter.      */
NEXT:      addi    r3, r3, 1      /* Increment the pointers      */
           addi    r4, r4, 1      /*   to the lists.             */
           subi    r2, r2, 1      /* Branch back if the end of   */
           bgtu    r2, r0, LOOP   /*   lists is not reached.     */
           mov     r2, r5         /* Return the result via r2.   */
           ldw     r7, (sp)       /* Restore registers.          */
           ldw     r6, 4(sp)
           ldw     r5, 8(sp)
           addi    sp, sp, 12
           ret
```

B.21. The subroutine may be implemented as follows:

```
          movia   r2, STRING      /* Pointer to the string.              */
          call    EXCLAIM
          next instruction

EXCLAIM:  subi    sp, sp, 12      /* Save registers.                     */
          stw     r3, 8(sp)
          stw     r4, 4(sp)
          stw     r5, (sp)
          movi    r3, 0x2E        /* ASCII code for period.              */
          movi    r4, 0x21        /* ASCII code for exclamation mark.    */
LOOP:     ldb     r5, (r2)
          beq     r5, r0, DONE    /* Check if NUL.                       */
          bne     r5, r3, NEXT    /* If period, then replace             */
          stb     r4, (r2)        /*   with exclamation mark.            */
NEXT:     addi    r2, r2, 1
          br      LOOP
DONE:     ldw     r5, (sp)        /* Restore registers.                  */
          ldw     r4, 4(sp)
          ldw     r3, 8(sp)
          addi    sp, sp, 12
          ret
```

B.22. ASCII codes for lower-case letters are in the hexadecimal range 61 to 7A. Whenever a character in this range is found, it can be converted into upper case by clearing bit 5 to zero. A possible program is:

```
            movia   r2, STRING       /* Pointer to the string.                */
            call    ALLCAPS
            next instruction

ALLCAPS:    subi    sp, sp, 12       /* Save registers.                       */
            stw     r3, 8(sp)
            stw     r4, 4(sp)
            stw     r5, (sp)
            movi    r3, 0x61         /* ASCII code for a.                     */
            movi    r4, 0x7a         /* ASCII code for z.                     */
LOOP:       ldb     r5, (r2)
            beq     r5, r0, DONE     /* Check if NUL.                         */
            bltu    r5, r3, NEXT     /* Check if in the range                 */
            bgtu    r5, r4, NEXT     /*   a to z.                             */
            andi    r5, r5, 0xDF     /* Create ASCII for the capital letter.  */
            stb     r5, (r2)         /* Store the capital letter.             */
NEXT:       addi    r2, r2, 1        /* Move to the next character.           */
            br      LOOP
DONE:       ldw     r5, (sp)         /* Restore registers.                    */
            ldw     r4, 4(sp)
            ldw     r3, 8(sp)
            addi    sp, sp, 12
            ret
```

B.23. Words can be counted by detecting the SPACE character. Assuming that words are separated by single SPACE characters, a possible program is:

```
          movia  r2, STRING     /* Pointer to the string.         */
          call   WORDS
          next instruction

WORDS:    subi   sp, sp, 12     /* Save registers.                */
          stw    r3, 8(sp)
          stw    r4, 4(sp)
          stw    r5, (sp)
          movi   r3, 0x20       /* ASCII code for SPACE.          */
          mov    r4, r0         /* Clear the word counter.        */
LOOP:     ldb    r5, (r2)
          beq    r5, r0, DONE   /* Check if NUL.                  */
          bne    r5, r3, NEXT   /* Check if SPACE.                */
          addi   r4, r4, 1.     /* Increment the word count.      */
NEXT:     addi   r2, r2, 1      /* Move to the next character.    */
          br     LOOP
DONE:     mov    r2, r4         /* Pass the result in r2.         */
          ldw    r5, (sp)       /* Restore registers.             */
          ldw    r4, 4(sp)
          ldw    r3, 8(sp)
          addi   sp, sp, 12
          ret
```

B.24. Assume that the calling program passes the parameters via registers, as follows:

r2 contains the length of the list

r3 contains the starting address of the list

r4 contains the new value to be inserted into the list

Then, the desired subroutine may be implemented as follows:

```
INSERT:     subi   sp, sp, 20          /* Save registers.                   */
            stw    r2, 16(sp)
            stw    r3, 12(sp)
            stw    r4, 8(sp)
            stw    r5, 4(sp)
            stw    r6, (sp)
            slli   r2, r2, 2           /* Multiply by 4.                    */
            add    r5, r3, r2          /* End of the list.                  */
LOOP:       ldw    r6, (r3)            /* Check entries in the list         */
            ble    r4, r6, TRANSFER    /*   until insertion point is reached. */
            addi   r3, r3, 4
            blt    r3, r5, LOOP
            br     DONE
TRANSFER:   ldw    r6, (r3)            /* Insert the new entry and          */
            stw    r4, (r3)            /*   move the rest of the entries    */
            mov    r4, r6              /*   upwards in the list.            */
            addi   r3, r3, 4           /* Increment the list pointer.       */
            bltu   r3, r5, TRANSFER
DONE:       stw    r4, (r3)            /* Store the last entry.             */
            ldw    r6, (sp)            /* Restore registers.                */
            ldw    r5, 4(sp)
            ldw    r4, 8(sp)
            ldw    r3, 12(sp)
            ldw    r2, 16(sp)
            addi   sp, sp, 20
            ret
```

B.25. Assume that the calling program passes the parameters via registers, as follows:

r10 contains the starting address of the unsorted list

r11 contains the length of the unsorted list

r12 contains the starting address of the new list

Then, using the INSERT subroutine derived in Problem B.24, the desired subroutine may be implemented as follows:

```
INSERTSORT:  subi   sp, sp, 20           /* Save registers.                      */
             stw    ra, 16(sp)
             stw    r2, 12(sp)
             stw    r3, 8(sp)
             stw    r4, 4(sp)
             stw    r10, (sp)
             ldw    r4, (r10)            /* Transfer one number from old list    */
             stw    r4, (r12)            /*   to new list.                       */
             mov    r3, r12
             movi   r2, 1
SCAN:        addi   r10, r10, 4          /* Increment pointer to old list.       */
             ldw    r4, (r10)            /* Next number to be inserted.          */
             call   INSERT
             addi   r2, r2, 1            /* Increment the length of new list.    */
             blt    r2, r11, SCAN
             ldw    r10, (sp)            /* Restore registers.                   */
             ldw    r4, 4(sp)
             ldw    r3, 8(sp)
             ldw    r2, 12(sp)
             ldw    ra, 16(sp)
             addi   sp, sp, 20
             ret

INSERT:      subi   sp, sp, 20           /* Save registers.                      */
             stw    r2, 16(sp)
             stw    r3, 12(sp)
             stw    r4, 8(sp)
             stw    r5, 4(sp)
             stw    r6, (sp)
             slli   r2, r2, 2            /* Multiply by 4.                       */
             add    r5, r3, r2           /* End of the list.                     */
LOOP:        ldw    r6, (r3)             /* Check entries in the list            */
             ble    r4, r6, TRANSFER     /*   until insertion point is reached.  */
             addi   r3, r3, 4
             blt    r3, r5, LOOP
             br     DONE
TRANSFER:    ldw    r6, (r3)             /* Insert the new entry and             */
             stw    r4, (r3)             /*   move the rest of the entries       */
             mov    r4, r6               /*   upwards in the list.               */
             addi   r3, r3, 4            /* Increment the list pointer.          */
             bltu   r3, r5, TRANSFER
DONE:        stw    r4, (r3)             /* Store the last entry.                */
             ldw    r6, (sp)             /* Restore registers.                   */
             ldw    r5, 4(sp)
             ldw    r4, 8(sp)
             ldw    r3, 12(sp)
             ldw    r2, 16(sp)
             addi   sp, sp, 20
             ret
```

B.26. The program can be based on the scheme presented in Figure B.12, assuming the device-interface given in Figure 3.3.

We will store the promt "Type your name" as a string of ASCII-encoded characters, by using an assembler directive. Similarly, we will store "Your name reversed" as the output message that precedes the display of the reversed name. Each string is terminated by a Carriage Return character.

A possible program is:

```
          .equ    KBD_DATA, 0x4000    /*  Specify addresses for keyboard        */
          .equ    DISP_DATA, 0x4010   /*    and display data registers.         */
          movia   r2, LOC             /*  Location where line will be stored.   */
          movia   r3, KBD_DATA        /*  r3 points to keyboard data register.  */
          movia   r4, DISP_DATA       /*  r4 points to display data register.   */
          addi    r5, r0, 0x0D        /*  Load ASCII code for Carriage Return.  */

/* Display the promt. */
          movia   r8, PROMPT
PLOOP:    ldbio   r6, 4(r4)           /*  Read display status register.         */
          andi    r6, r6, 4           /*  Check the DOUT flag.                  */
          beq     r6, r0, PLOOP
          ldb     r7, (r8)            /* Send a character of the prompt         */
          stbio   r7, (r4)            /*   to the display.                      */
          addi    r8, r8, 1
          bne     r7, r5, PLOOP

/* Read the name. */
READ:     ldbio   r6, 4(r3)           /*  Read keyboard status register.        */
          andi    r6, r6, 2           /*  Check the KIN flag.                   */
          beq     r6, r0, READ
          ldbio   r7, (r3)            /*  Read character from keyboard.         */
          stb     r7, (r2)            /*  Write character into main memory      */
          addi    r2, r2, 1           /*    and increment the pointer.          */
ECHO:     ldbio   r6, 4(r4)           /*  Read display status register.         */
          andi    r6, r6, 4           /*  Check the DOUT flag.                  */
          beq     r6, r0, ECHO
          stbio   r7, (r4)            /*  Send the character to display.        */
          bne     r5, r7, READ        /*  Loop back if character is not CR.     */

/* Display output message. */
          movia   r8, MESSAGE
MLOOP:    ldbio   r6, 4(r4)           /*  Read display status register.         */
          andi    r6, r6, 4           /*  Check the DOUT flag.                  */
          beq     r6, r0, MLOOP
          ldb     r7, (r8)            /* Send a character of the message        */
          stbio   r7, (r4)            /*   to the display.                      */
          addi    r8, r8, 1
          bne     r7, r5, MLOOP
```

. . .continued on the next page

```
/* Display the reversed name. */
              subi    r8, r2, 2        /* Set the pointer for backward scan.     */
              movia   r2, LOC
NLOOP:        ldbio   r6, 4(r4)        /*  Read display status register.         */
              andi    r6, r6, 4        /*  Check the DOUT flag.                  */
              beq     r6, r0, NLOOP
              ldb     r7, (r8)
              stbio   r7, (r4)         /*  Send a character to display.          */
              subi    r8, r8, 1
              bge     r8, r2, NLOOP
              next instruction

              .org    0x1000
PROMPT:       .byte  0x54, 0x79, 0x70, 0x65, 0x20, 0x79, 0x6F, 0x75
              .byte  0x72, 0x20, 0x6E, 0x61, 0x6D, 0x65, 0x0D
MESSAGE:      .byte  0x59, 0x6F, 0x75, 0x72, 0x20, 0x6E, 0x61, 0x6D, 0x65, 0x20
              .byte  0x72, 0x65, 0x76, 0x65, 0x72, 0x73, 0x65, 0x64, 0x0D
```

B.27. The program can be based on the scheme presented in Figure B.12, assuming the device-interface given in Figure 3.3.

We will store the promt “Type the word” as a string of ASCII-encoded characters, by using an assembler directive. Similarly, we will store “It is a palindrome” and “It is not a palindrome” as the output messages indicating the result of the test. Each string is terminated by a Carriage Return character.

A possible program is:

```
          .equ    KBD_DATA, 0x4000    /*  Specify addresses for keyboard          */
          .equ    DISP_DATA, 0x4010   /*     and display data registers.          */
          movia   r2, LOC             /*  Location where line will be stored.     */
          movia   r3, KBD_DATA        /*  r3 points to keyboard data register.    */
          movia   r4, DISP_DATA       /*  r4 points to display data register.     */
          addi    r5, r0, 0x0D        /*  Load ASCII code for Carriage Return.    */

          movia   r8, PROMPT
          call    DISPLAY             /* Display the prompt.                      */

/* Read the word. */
READ:     ldbio   r6, 4(r3)           /*  Read keyboard status register.          */
          andi    r6, r6, 2           /*  Check the KIN flag.                     */
          beq     r6, r0, READ
          ldbio   r7, (r3)            /*  Read character from keyboard.           */
          stb     r7, (r2)            /*  Write character into main memory        */
          addi    r2, r2, 1           /*    and increment the pointer.            */
ECHO:     ldbio   r6, 4(r4)           /*  Read display status register.           */
          andi    r6, r6, 4           /*  Check the DOUT flag.                    */
          beq     r6, r0, ECHO
          stbio   r7, (r4)            /*  Send the character to display.          */
          bne     r5, r7, READ        /*  Loop back if character is not CR.       */
```

. . .continued on the next page

```
/* Determine if palindrome. */
          subi    r8, r2, 2           /* Set the pointer for backward scan.    */
          movia   r2, LOC
          mov     r9, r2
DLOOP:    ldb     r6, (r2)            /* Scan the word in                      */
          ldb     r7, (r8)            /*   opposite directions and check       */
          bne     r6, r7, NOTP        /*   if there is a match.                */
          addi    r2, r2, 1
          subi    r8, r8, 1
          ble     r8, r9, DLOOP

/* Display the result. */
          movia   r8, YESPAL          /* Display that it is                    */
          call    DISPLAY             /*   a palindrome.                       */
          br      DONE
NOTP:     movia   r8, NOPAL           /* Display that it is                    */
          call    DISPLAY             /*   not a palindrome.                   */
DONE:     next instruction

DISPLAY:  ldbio   r6, 4(r4)           /*  Read display status register.        */
          andi    r6, r6, 4           /*  Check the DOUT flag.                 */
          beq     r6, r0, DISPLAY
          ldb     r7, (r8)            /* Keep sending characters               */
          stbio   r7, (r4)            /*   to the display until                */
          addi    r8, r8, 1           /*   the Carriage Return character       */
          bne     r7, r5, DISPLAY     /*   is reached.                         */
          ret

          .org    0x1000
PROMPT:   .byte  0x54, 0x79, 0x70, 0x65, 0x20, 0x74, 0x68, 0x65
          .byte  0x20, 0x77, 0x6F, 0x72, 0x64, 0x0D
YESPAL:   .byte  0x49, 0x74, 0x20, 0x69, 0x73, 0x20, 0x61, 0x20, 0x70, 0x61
          .byte  0x6C, 0x69, 0x6E, 0x64, 0x72, 0x6F, 0x6D, 0x65, 0x0D
NOPAL:    .byte  0x49, 0x74, 0x20, 0x69, 0x73, 0x20, 0x6E, 0x6F, 0x78
          .byte  0x20, 0x61, 0x20, 0x70, 0x61, 0x6C, 0x69, 0x6E, 0x64
          .byte  0x72, 0x6F, 0x6D, 0x65, 0x0D
```

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

```
          movia  r2, STRING     /* Load address of string.                                  */
          call   LENGTH         /* Compute length of string.                                */
          movi   r3, 78         /* Check if length is greater than 78.                      */
          ble    r2, r3, CONT1  /* If not, continue to subsequent instructions,             */
          movi   r2, 78         /*    otherwise, truncate to 78.                            */
CONT1:    mov    r6, r2         /* Save length in r6.                                       */
          srai   r7, r2, 1      /* Use r7 to calculate and hold 39−length/2.                */
          subi   r7, r7, 39     /* After division by 2, subtract 39, and then change the sign */
          sub    r7, r0, r7     /*    to obtain number of leading spaces.                   */
          mov    r2, r6         /* Prepare arguments for call to subroutine                 */
          mov    r3, r7         /*    to display upper line of                              */
          call   DISPA          /*    bounding box with carriage return.                    */
          mov    r3, r7         /* Initialize counter with number of leading spaces.        */
          movi   r2, 0x20       /* Load space character into D0.                            */
LOOP1:    call   WRITECHAR      /* Repeat this loop to display spaces                       */
          subi   r3, r3, 1      /*    until count has reached zero.                         */
          bgt    r3, r0, LOOP1
DISP2:    movi   r2, 0x7C       /* Load vertical bar character into D0.                     */
          call   WRITECHAR      /* Display the character.                                   */
          movia  r4, STRING     /* Initialize pointer to string.                            */
          mov    r3, r6         /* Initialize the counter for the string length.            */
LOOP2:    ldb    r2, 0(r4)      /* Get a character.                                         */
          addi   r4, r4, 1      /* Increment the pointer.                                   */
          call   WRITECHAR      /* Display the character.                                   */
          subi   r3, r3, 1      /* Decrement the counter                                    */
          bgt    r3, r0, LOOP2  /*    and repeat until all characters displayed.            */
          movi   r2, 0x7C       /* Load the vertical bar character into D0.                 */
          call   WRITECHAR      /* Display the character.                                   */
          movi   r2, 0xD        /* Display a carriage return.                               */
          call   WRITECHAR
          mov    r2, r6         /* Prepare arguments for call to subroutine                 */
          mov    r3, r7         /*    to display lower line of                              */
          call   DISPA          /*    bounding box with carriage return.                    */
          next instruction
```

The subroutines called from the program above are shown below, and they are written in a modular manner to save/restore registers. The WRITECHAR subroutine is based on the example of Figure B.12.

```
LENGTH:      subi    sp, sp, 8            /* Save registers.                                */
             stw     r3, 4(sp)
             stw     r4, 0(sp)
             movi    r4, 0                /* Initialize count of characters.                */
LEN_LOOP:    ldb     r3, 0(r2)            /* Loop until zero is detected.                   */
             beq     r3, r0, LEN_DONE
             addi    r4, r4, 1            /* Increment count.                               */
             addi    r2, r2, 1            /* Increment pointer.                             */
             br      LEN_LOOP
LEN_DONE:    mov     r2, r4               /* Copy count to register for return value.       */
             ldw     r3, 4(sp)            /* Restore registers.                             */
             ldw     r4, 0(sp)
             addi    sp, sp, 8
             ret
```

```
WRITECHAR:   subi    sp, sp, 8            /* Save registers.                                */
             stw     r4, 4(sp)
             stw     r3, 0(sp)
             movia   r4, DISP_STATUS      /* EQU directives are assumed for I/O addresses.  */
WCLOOP:      ldbio   r3, 4(r4)
             andi    r3, r3, 4
             beq     r3, r0, WCLOOP
             stbio   r2, 0(r4)
             ldw     r4, 4(sp)            /* Restore registers.                             */
             ldw     r3, 0(sp)
             addi    sp, sp, 8
             ret
```

```
DISPA:     subi   sp, sp, 16          /* Save registers.                         */
           stw    ra, 12(sp)
           stw    r2, 8(sp)
           stw    r3, 4(sp)
           stw    r5, 0(sp)
           mov    r5, r2              /* Save length.                            */
           movi   r2, 0x20            /* Load space character into D0.           */
SPLOOP:    call   WRITECHAR           /* Repeat this loop to display spaces      */
           subi   r3, r3, 1           /*    until count has reached zero.        */
           bgt    r3, r0, SPLOOP
           movi   r2, 0x2B            /* Display ‘+’ character.                  */
           call   WRITECHAR
           movi   r2, 0x2D            /* Load ‘-’ character into D0.             */
DSHLOOP:   call   WRITECHAR           /* Repeat this loop to display ‘-’         */
           subi   r5, r5, 1           /*    until the other count has reached zero. */
           bgt    r5, r0, DSHLOOP
           movi   r2, 0x2B            /* Display ‘+’ character.                  */
           call   WRITECHAR
           movi   r2, 0xD             /* Display carriage return.                */
           call   WRITECHAR
           ldw    ra, 12(sp)          /* Restore registers.                      */
           stw    r2, 8(sp)
           stw    r3, 4(sp)
           ldw    r5, 0(sp)
           addi   sp, sp, 16
           ret
```

B.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.

```
            movia  r4, TEXT          /* Register r4 points to start of text.                      */
            movi   r5, 80            /* Register r5 reflects space left on the current line.      */
RESET:      movi   r3, 0             /* Clear count of characters in current word.                */
            mov    r2, r4            /* Save the starting point of current word.                  */
SCAN:       ldb    r6, 0(r4)         /* Read the next character.                                  */
            addi   r4, r4, 1         /* Advance the pointer.                                      */
            movi   r7, 0x20          /* Check for a control character or space,                   */
            ble    r6, r7, HAVEWORD  /*    and process a complete word appropriately.             */
            addi   r3, r3, 1         /* Otherwise, increment count of characters,                 */
            br     SCAN              /*    and repeat inner loop for current word.                */
HAVEWORD:   mov    r8, r5            /* For complete word, use space left on current line         */
            sub    r8, r8, r3        /*    and count of characters in current word                */
            bge    r8, r0, DISP      /*    to determine if word will fit.                         */
            call   NEWLINE           /* Otherwise, move to a new line,                            */
            movi   r5, 80            /*    and reinitialize space left for new line.              */
DISP:       call   DISPLAY           /* Display word using r2 pointer and r3 count.               */
            sub    r5, r5, r3        /* Reduce space left on line using count.                    */
            beq    r8, r0, SKIP      /* If previous calculation indicated no space on line,       */
                                     /*    skip printing a space after current word.              */
            movi   r2, 0x20          /* Display a space (it is safe to use r2 here).              */
            call   WRITECHAR
            subi   r5, r5, 1         /* Reduce space on current line by one character.            */
SKIP:       beq    r6, r0, DONE      /* Finally, check if last character was NUL,                 */
                                     /*    and end program.                                       */
            br     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 B.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:   subi   sp, sp, 8          /* Save registers.                                 */
           stw    ra, 4(sp)
           stw    r2, 0(sp)
           movi   r2, 0xD            /* Send carriage return to move to new line.       */
           call   WRITECHAR
           ldw    ra, 4(sp)          /* Restore registers.                              */
           ldw    r2, 0(sp)
           addi   sp, sp, 8
           ret

DISPLAY:   subi   sp, sp, 20
           stw    ra, 16(sp)         /* Save subroutine linkage register.               */
           stw    r5, 12(sp)         /* Save register to use for pointer.               */
           stw    r2, 8(sp)          /* Save original word start for modularity.        */
           stw    r4, 4(sp)          /* Save register to use for count.                 */
           stw    r3, 0(sp)          /* Save original character count for modularity.   */
           mov    r4, r3             /* Prepare counter for loop.                       */
           mov    r5, r2             /* Prepare pointer for loop.                       */
DLOOP:     ldb    r2, 0(r5)          /* Read next character.                            */
           addi   r5, r5, 1          /* Increment pointer.                              */
           call   WRITECHAR          /* Display the character.                          */
           subi   r3, r3, 1          /* Decrement the count.                            */
           bgt    r3, r0, DLOOP      /* Repeat if not finished with current word.       */
           ldw    ra, 16(sp)         /* Restore registers.                              */
           ldw    r5, 12(sp)
           ldw    r2, 8(sp)
           ldw    r4, 4(sp)
           ldw    r3, 0(sp)
           addi   sp, sp, 20
           ret
```