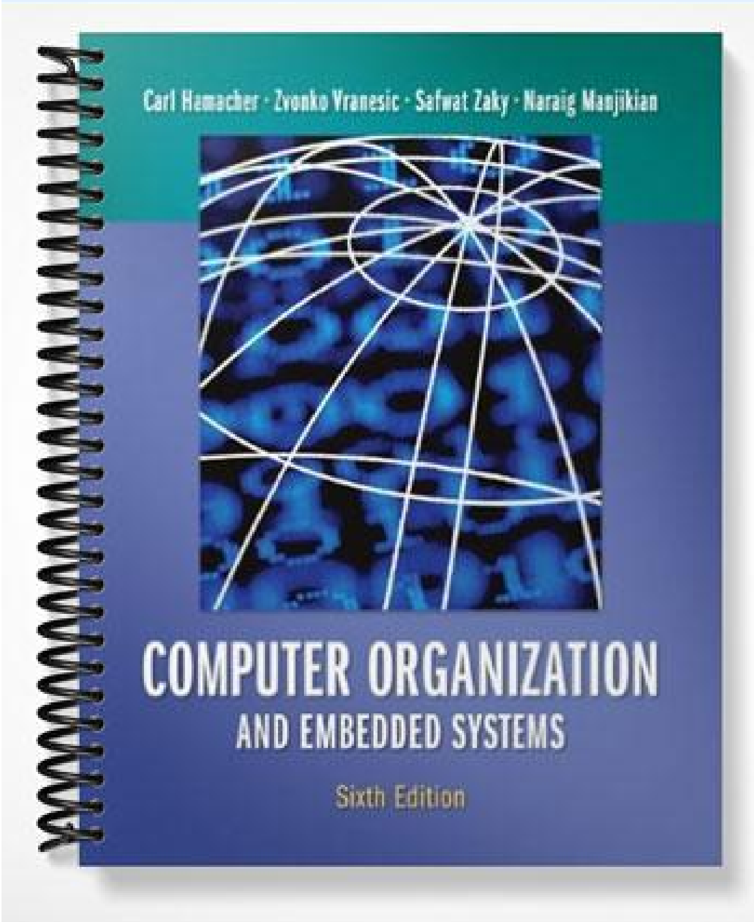
SOLUTIONS MANUAL



Chapter 2
Instruction Set Architecture

2.1. No; any binary pattern can be interpreted as a number or as an instruction.

2.2. Byte contents in hex, starting at location 1000, will be 43, 6F, 6D, 70, 75, 74, 65, 72. The two words at 1000 and 1004 will be 436F6D70 and 75746572.

2.3. Byte contents in hex, starting at location 1000, will be 43, 6F, 6D, 70, 75, 74, 65, 72. The two words at 1000 and 1004 will be 706D6F43 and 72657475.

2.4. (a) 2012, (b) 5000, (c) 5028, (d) 2000, (e) 1996.

2.5. A RISC-style program that computes SUM = 580 + 6840 + 80000:

```
            Move      R2, #NUMBERS         Get the address of numbers.
            Load      R3, (R2)             Load 580.
            Load      R4, 4(R2)            Load 68400.
            Add       R3, R3, R4           Generate 580 + 80000.
            Load      R4, 8(R2)            Load 80000.
            Add       R3, R3, R4           Generate the final sum.
            Store     R3, 12(R2)           Store the sum.
            next instruction

            ORIGIN    0x500
NUMBERS:    DATAWORD  580, 68400, 80000    Numbers to be added.
SUM:        RESERVE   4                    Space for the sum.
```

2.6. A CISC-style program that computes SUM = 580 + 6840 + 80000:

```
            Move      R2, #NUMBERS         Get the address of numbers.
            Move      R3, (R2)+            Load 580.
            Add       R3, (R2)+            Generate 580 + 80000.
            Add       R3, (R2)             Generate the final sum.
            Move      SUM, R3              Store the sum.
            next instruction

            ORIGIN    0x500
NUMBERS:    DATAWORD  580, 68400, 80000    Numbers to be added.
SUM:        RESERVE   4                    Space for the sum.
```

2.7. A RISC-style program that computes ANSWER = A × B + C × D:

```
          Move              R2, #A          Get the address of A.
          Load              R3, (R2)        Load the operand A.
          Move              R2, #B          Get the address of B.
          Load              R4, (R2)        Load the operand B.
          Multiply          R5, R3, R4      Generate A × B.
          Move              R2, #C          Get the address of C.
          Load              R3, (R2)        Load the operand C.
          Move              R2, #D          Get the address of D.
          Load              R4, (R2)        Load the operand D.
          Multiply          R6, R3, R4      Generate C × D.
          Add               R7, R5, R6      Compute the final answer.
          Move              R2, #ANSWER     Get the address and
          Store             R7, (R2)          store the answer.
          next instruction

          ORIGIN            0x500
A:        DATAWORD          100             Test data.
B:        DATAWORD          50
C:        DATAWORD          20
D:        DATAWORD          400
ANSWER:   RESERVE           4               Space for the answer.
```

2.8. A CISC-style program that computes ANSWER = A × B + C × D:

```
          Move              R2, A           Load the operand A.
          Multiply          R2, B           Generate A × B.
          Move              R3, C           Load the operand C.
          Multiply          R3, D           Generate C × D.
          Add               R3, R2          Compute the final answer.
          Move              ANSWER, R3      Store the answer.
          next instruction

          ORIGIN            0x500
A:        DATAWORD          100             Test data.
B:        DATAWORD          50
C:        DATAWORD          20
D:        DATAWORD          400
ANSWER:   RESERVE           4               Space for the answer.
```

2.9. An alternative program is given below. The size of the list in bytes is computed by shifting the value n to the left by two bit positions, which multiplies the value by 4. This is then added to the starting address of the list to generate the address that follows the last entry in the list.

The loop in this program has only four instructions. Note that we could use a similar arrangement to process the list in the direction of increasing addresses.

```
        Load                 R2, N          Load the size of the list.
        LShiftL              R2, R2, #2     Multiply by 4.
        Clear                R3             Initialize sum to 0.
        Move                 R4, #NUM1      Get address of the first number.
        Add                  R2, R2, R4     Address past the last entry.
LOOP:   Subtract             R2, R2, #4     Decrement the pointer to the list.
        Load                 R5, (R2)       Get the next number.
        Add                  R3, R3, R5     Add this number to sum.
        Branch_if_[R4]<[R2]  LOOP           Branch back if not finished.
        Store                R3, SUM        Store the final sum.
```

2.10. 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 2.10. The parameter Stride = $4(j + 1)$ is the distance in bytes between scores on a particular test for adjacent students in the list.

```
        Move                 R2, #J         Compute and place
        Load                 R2, (R2)         Stride = 4(j + 1)
        Add                  R2, R2, #1       into register R2.
        LShiftL              R2, R2, #2
        Move                 R3, #LIST      Initialize register R3 to the location
        Add                  R3, R3, #4       of the test 1 score for student 1.
        Move                 R4, #SUM       Initialize register R4 to the location
                                              of the sum for test 1.
        Move                 R5, #J         Initialize outer loop counter
        Load                 R5, (R5)         R5 to j.
OUTER:  Move                 R6, #N         Initialize inner loop counter
        Load                 R6, (R6)         R6 to n.
        Move                 R7, R0
        Move                 R8, R0         Clear the sum register R8.
        Add                  R9, R3, R7     Use R9 as an index register.
INNER:  Load                 R10, (R9)      Accumulate the sum
        Add                  R8, R8, R10      of test scores.
        Add                  R9, R9, R2     Increment index register by Stride value.
        Subtract             R6, R6, #1     Check if all student scores on current
        Branch_if_[R6]>[R0]  INNER            test have been accumulated.
        Store                R8, (R4)       Store sum of current test scores and
        Add                  R4, R4, #4       increment sum location pointer.
        Add                  R3, R3, #4     Increment R3 to point to the next
                                              test score for student 1.
        Subtract             R5, R5, #1     Check if the sums for all tests have
        Branch_if_[R5]>[R0]  OUTER            been computed.
        next instruction
```

2.11. The following program determines the number of negative integers.

```
            Move                 R2, #N             Get the address N.
            Load                 R2, (R2)           Load the size of the list.
            Move                 R3, R0             Initialize the counter to 0.
            Move                 R4, #NUMBERS       Load address of the first number.
LOOP:       Load                 R5, (R4)           Get the next number.
            Branch_if_[R5]≥[R0]  NEXT               Test if number is negative.
            Add                  R3, R3, #1         Increment the count.
NEXT:       Add                  R4, R4, #4         Increment the pointer to list.
            Subtract             R2, R2, #1         Decrement the list counter.
            Branch_if_[R2]>[R0]  LOOP               Loop back if not finished.
            Move                 R6, #NEGNUM        Get the address NEGNUM.
            Store                R3, (R6)           Store the result.
            next instruction

            ORIGIN               0x500
NEGNUM:     RESERVE              4                  Space for the result.
N:          DATAWORD             6                  Size of the list.
NUMBERS:    DATAWORD             23, −5, −128       Test data.
            DATAWORD             44, −23, −9
```

2.12. The assembler directives ORIGIN and DATAWORD cause the object program memory image constructed by the assembler to indicate that 300 is to be placed at memory word location 1000 at the time the program is loaded into memory prior to execution.

The Move and Store instructions place 300 into memory word location 1000 when these instructions are executed as part of a program.

2.13. An assembly-language program in the style of Figure 2.13 is:

```
        ORIGIN      100
        MOV         R2, #LIST       Get the address LIST.
        CLR         R3
        CLR         R4
        CLR         R5
        LD          R6, N           Load the value n.
LOOP:   LD          R7, 4(R2)       Add the mark for next student's
        ADD         R3, R3, R7        Test 1 to the partial sum.
        LD          R7, 8(R2)       Add the mark for that student's
        ADD         R4, R4, R7        Test 2 to the partial sum.
        LD          R7, 12(R2)      Add the mark for that student's
        ADD         R5, R5, R7        Test 3 to the partial sum.
        ADD         R2, R2, #16     Increment the pointer.
        SUB         R6, R6, #1      Decrement the counter.
        BGT         R6, R0, LOOP    Branch back if not finished.
        ST          R3, SUM1        Store the total for Test 1.
        ST          R4, SUM2        Store the total for Test 2.
        ST          R5, SUM3        Store the total for Test 3.
        next instruction

        ORIGIN      300
SUM1:   RESERVE     4
SUM2:   RESERVE     4
SUM3:   RESERVE     4
N:      DATAWORD    50
LIST:   RESERVE     800
        END
```

2.14. A CISC-style program corresponding to Figure 2.33 is:

```
        Move         R2, #STRING     R2 points to the start of the string.
        Clear        R3              R3 is a counter that is cleared to 0.
        Move         R4, #0x0D       ASCII code for Carriage Return.
LOOP:   CompareByte  R4, (R2)+       Check the next character.
        Branch=0     DONE            Finished if character is CR.
        Add          R3, R3, #1      Increment the counter.
        Branch       LOOP            Not finished, loop back.
DONE:   Move         LENGTH, R3      Store the count in location LENGTH.
```

2.15. A CISC-style program corresponding to Figure 2.34 is:

```
LIST      EQU       1000          Starting address of the list.

          ORIGIN    400
          Move      R2, #LIST     R2 points to the start of the list.
          Move      R3, 4(R2)     R3 is a counter, initialize it with n.
          Move      R4, R2
          Add       R4, #8        R4 points to the first number.
          Move      R5, (R4)      R5 holds the smallest number found so far.
LOOP:     Subtract  R3, #1        Decrement the counter.
          Branch=0  DONE          Finished if R3 is equal to 0.
          Compare   R5, (R4)+
          Branch≤0  LOOP          Check if smaller number found.
          Move      R5, −4(R4)    Update the smallest number found.
          Branch    LOOP
DONE:     Move      (R2), R5      Store the smallest number into SMALL.

          ORIGIN    1000
SMALL:    RESERVE   4             Space for the smallest number found.
N:        DATAWORD  7             Number of entries in the list.
ENTRIES:  DATAWORD  4,5,3,6,1,8,2 Entries in the list.
          END
```

2.16. A CISC-style program corresponding to Figure 2.35 is:

```
          Move      R2, N         Initialize counter R2 with n.
          Move      R3, #DECIMAL  R3 points to the ASCII digits.
          Clear     R4            R4 will hold the binary number.
LOOP:     MoveByte  R5, (R3)+     Get the next ASCII digit.
          And       R5, #0x0F     Form the BCD digit.
          Add       R4, R5        Add to the intermediate result.
          Subtract  R2, #1        Decrement the counter.
          Branch=0  DONE
          Multiply  R4, #10       Multiply by 10.
          Branch    LOOP          Loop back if not done.
DONE:     Move      BINARY, R4    Store result in location BINARY.
```

2.17. Assume that the subroutine can change the contents of any register used to pass parameters.

```
SUB:      LShiftL   R5, #2        Use R5 to contain distance in bytes
                                    between successive elements in a column.
          Subtract  R3, R2        Form (y − x).
          LShiftL   R3, #2        Form 4(y − x).
          LShiftL   R2, #2        Set R6 to
          Add       R6, R2          address A(0,x).
LOOP:     Move      R2, (R6)      Add corresponding
          Add       (R6, R3), R2    column elements.
          Add       R6, R5        Move to next row.
          Decrement R4            Repeat until all
          Branch>0  LOOP            elements are added.
          Return                  Return to calling program.
```

2.18. A RISC-style program for Example 2.5 is:

	Move	R2, #LIST	Get the address LIST.
	Move	R3, #N	Get the address N.
	Load	R3, (R3)	Initialize outer loop pointer
	Add	R3, R2, R3	to LIST + n.
OUTER:	Subtract	R3, R3, #1	Decrement the pointer.
	Branch_if_[R3]$\leq$[R2]	DONE	Check if last entry.
	LoadByte	R5, (R3)	Starting max value in sublist.
	Subtract	R4, R3, #1	Initialize inner loop pointer.
INNER:	LoadByte	R6, (R4)	Check if the next entry
	Branch_if_[R5]$\geq$[R6]	NEXT	is lower.
	StoreByte	R6, (R3)	If yes, then swap
	StoreByte	R5, (R4)	the entries and
	Move	R5, R6	update the max value.
NEXT:	Subtract	R4, R4, #1	Adjust the inner loop pointer.
	Branch_if_[R4]$\geq$[R2]	INNER	
	Branch	OUTER	

2.19. The tasks can be performed as follows:

(a)

```
Move   R2, (R5)+
Add    R2, (R5)+
Move   −(R5), R2
```

(b)

```
Move   R3, 16(R5)
```

(c)

```
Add    R5, #40
```

2.20. (a) The stack will contain the first 4 entries shown in Figure 2.19. However, the stack pointer will point to address 976, because it has already been adjusted to this value by the first Subtract instruction in the subroutine.

(b) The stack pointer will have the value 976. The stack contents will be the same as shown in Figure 2.19, except that NUM1 will have been replaced by the sum.

(c) The stack pointer will have value 992. There will be 2 entries in the stack - n and the sum.

2.21. (*a*) Neither nesting nor recursion are supported.

(*b*) Nesting is supported, because different Call instructions will save the return address at different memory locations. Recursion is not supported.

(*c*) Both nesting and recursion are supported.

2.22. The contents of register R2 can be safely pushed on the second stack, or poped from it, by calling the following RISC-style subroutines:

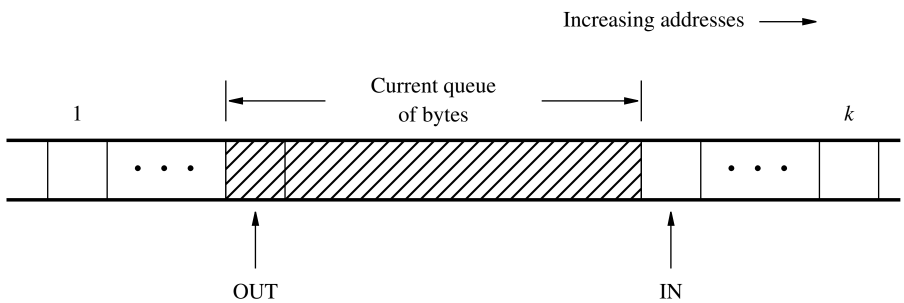
SPUSH:	Subtract	SP, SP, #4	Save register R3 on
	Store	R3, (SP)	the processor stack.
	Move	R3, #TOP	
	Branch_if_[R5]≤[R3]	FULLERROR	
	Subtract	R5, R5, #4	
	Store	R2, (R5)	
	Load	R3, (SP)	Restore register R3.
	Add	SP, SP, #4	
	Return		
SPOP:	Subtract	SP, SP, #4	Save register R3 on
	Store	R3, (SP)	the processor stack.
	Move	R3, #BOTTOM	
	Branch_if_[R5]≥[R3]	EMPTYERROR	
	Load	R2, (R5)	
	Add	R5, R5, #4	
	Load	R3, (SP)	Restore register R3.
	Add	SP, SP, #4	
	Return		

2.23. The contents of register R2 can be safely pushed on the second stack, or poped from it, as follows:

SPUSH:	Compare	R5, #TOP	If R5 has a value equal to
	Branch≤0	FULLERROR	or less than TOP, then stack is full.
	Move	−(R5), R2	Otherwise, push the new entry.
SPOP:	Compare	R5, #BOTTOM	If R5 has a value equal to or
	Branch≥0	EMPTYERROR	greater than BOTTOM, then stack is empty.
	Move	R2, (R5)+	Otherwise, pop the entry from stack.

2.24. (*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 $\leftarrow$ [IN]
- IN $\leftarrow$ ([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 $\leftarrow$ ([OUT] + 1) Modulo k.

2.25. 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 2.24.

APPEND routine:

```
            Move                 R7, R3
            Add                  R3, R3, #1      Increment IN pointer
            Branch_if_[R6]≥[R3]  CHECK             modulo k.
            Move                 R3, R5
CHECK:      Branch_if_[R3]=[R4]  FULL            Check if queue is full.
            Store                R2, (R7)        If queue not full, append item.
            Branch               CONTINUE
FULL:       Move                 R3, R7          Restore IN pointer and send
            Call                 QUEUEFULL         message that queue is full.
CONTINUE:   ...
```

REMOVE routine:

```
REMOVE:     Branch_if_[R3]=[R4]  EMPTY           Check if queue is empty.
            Load                 R2, (R4)        Remove byte and
            Add                  R4, R4, #1        increment r4 modulo k.
            Branch_if_[R6]≥[R4]  CONTINUE
            Move                 R4, R5
            Branch               CONTINUE
EMPTY:      Call                 QUEUEEMPTY
CONTINUE:   ...
```

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

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

A possible program is:

```
         Move                 R2, #N        Get the number of entries, n, that
         Load                 R2, (R2)        that have to be generated.
         Move                 R3, #IN       Pointer to the IN list.
         Move                 R4, #OUT      Pointer to the OUT list.
LOOP:    Load                 R5, (R3)      Get the value IN(k) and
         AShiftR              R5, R5, #3      divide it by 8.
         Load                 R6, 4(R3)     Get the value IN(k+1) and
         AShiftR              R6, R6, #2      divide it by 4.
         Add                  R5, R5, R6
         Load                 R6, 8(R3)     Get the value IN(k+2) and
         AShiftR              R6, R6, #1      divide it by 2.
         Add                  R5, R5, R6    Compute the sum and
         Store                R5, (R4)        store it in OUT list.
         Add                  R3, R3, #4    Increment the pointers
         Add                  R4, R4, #4      to IN and OUT lists.
         Subtract             R2, R2, #1    Continue until all values in
         Branch_if_[R2]>[R0]  LOOP            OUT list have been generated.
         next instruction
```

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

```
            Move                R2, #N        Load the length parameter
            Load                R2, (R2)        into R2.
            Move                R3, #FROM     Pointer to from list.
            Move                R4, #TO       Pointer to to  list.
            Call                MEMCPY
            next instruction

MEMCPY:     Subtract            SP, SP, #12   Save registers.
            Store               R5, 8(SP)
            Store               R6, 4(SP)
            Store               R7, (SP)
            Add                 R5, R3, R2    Compute address of the last
            Subtract            R5, R5, #1      entry in the from list.
            Branch_if_[R4]≥[R5] UP            Scan upwards if to list
            Branch_if_[R4]≤[R3] UP              begins inside from list.
            Add                 R6, R4, R2    Compute the pointer for
            Subtract            R6, R6, #1      scanning downwards.
DOWN:       LoadByte            R7, (R5)      Transfer a byte and
            StoreByte           R7, (R6)
            Subtract            R5, R5, #1      adjust the pointers downwards.
            Subtract            R6, R6, #1
            Branch_if_[R5]≥[R3] DOWN
            Branch              DONE
UP:         LoadByte            R7, (R3)      Transfer a byte and
            StoreByte           R7, (R4)
            Add                 R3, R3, #1      adjust the pointers upwards.
            Add                 R4, R4, #1
            Branch_if_[R3]≤[R5] UP
DONE:       Load                R7, (SP)      Restore registers.
            Load                R6, 4(SP)
            Load                R5, 8(SP)
            Add                 SP, SP, #12
            Return
```

2.28. The comparison task can be performed as follows:

```
              Move                 R2, #N          Load the length parameter
              Load                 R2, (R2)          into R2.
              Move                 R3, #FIRST      Pointer to first list.
              Move                 R4, #SECOND     Pointer to second list.
              Call                 MEMCMP
              next instruction

MEMCMP:       Subtract             SP, SP, #12     Save registers.
              Store                R5, 8(SP)
              Store                R6, 4(SP)
              Store                R7, (SP)
              Move                 R5, R0          Clear the counter.
LOOP:         LoadByte             R6, (R3)        Load the bytes that have
              LoadByte             R7, (R4)          to be compared.
              Branch_if_[R6]=[R7]  NEXT
              Add                  R5, R5, #1      Increment the counter.
NEXT:         Add                  R3, R3, #1      Increment the pointers
              Add                  R4, R4, #1        to the lists.
              Subtract             R2, R2, #1      Branch back if the end of
              Branch_if_[R2]>[R0]  LOOP              lists is not reached.
              Move                 R2, R5          Return the result via R2.
              Load                 R7, (SP)        Restore registers.
              Load                 R6, 4(SP)
              Load                 R5, 8(SP)
              Add                  SP, SP, #12
              Return
```

2.29. The subroutine may be implemented as follows:

```
          Move                  R2, #STRING     Pointer to the string.
          Call                  EXCLAIM
          next instruction

EXCLAIM:  Subtract              SP, SP, #12     Save registers.
          Store                 R3, 8(SP)
          Store                 R4, 4(SP)
          Store                 R5, (SP)
          Move                  R3, #0x2E       ASCII code for period.
          Move                  R4, #0x21       ASCII code for exclamation mark.
LOOP:     LoadByte              R5, (R2)
          Branch_if_[R5]=[R0]   R5, R0, DONE    Check if NUL.
          Branch_if_[R5]≠[R3]   NEXT            If period, then replace
          StoreByte             R4, (R2)          with exclamation mark.
NEXT:     Add                   R2, R2, #1
          Branch                LOOP
DONE:     Load                  R5, (SP)        Restore registers.
          Load                  R4, 4(SP)
          Load                  R3, 8(SP)
          Add                   SP, SP, #12
          Return
```

2.30. 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:

```
            Move                 R2, #STRING     Pointer to the string.
            Call                 ALLCAPS
            next instruction

ALLCAPS:    Subtract             SP, SP, #12     Save registers.
            Store                R3, 8(SP)
            Store                R4, 4(SP)
            Store                R5, (SP)
            Move                 R3, #0x61       ASCII code for a.
            Move                 R4, #0x7a       ASCII code for z.
LOOP:       LoadByte             R5, (R2)
            Branch_if_[R5]=[R0]  DONE            Check if NUL.
            Branch_if_[R5]<[R3]  NEXT            Check if in the range
            Branch_if_[R5]>[R4]  NEXT              a to z.
            And                  R5, R5, #0xDF   Create ASCII for the capital letter.
            StoreByte            R5, (R2)        Store the capital letter.
NEXT:       Add                  R2, R2, #1      Move to the next character.
            Branch               LOOP
DONE:       Load                 R5, (SP)        Restore registers.
            Load                 R4, 4(SP)
            Load                 R3, 8(SP)
            Add                  SP, SP, #12
            Return
```

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

```
         Move                  R2, #STRING    Pointer to the string.
         Call                  WORDS
         next instruction

WORDS:   Subtract              SP, SP, #12    Save registers.
         Store                 R3, 8(SP)
         Store                 R4, 4(SP)
         Store                 R5, (SP)
         Move                  R3, #0x20      ASCII code for SPACE.
         Move                  R4, R0         Clear the word counter.
LOOP:    LoadByte              R5, (R2)
         Branch_if_[R5]=[R0]   DONE           Check if NUL.
         Branch_if_[R5]≠[R3]   NEXT           Check if SPACE.
         Add                   R4, R4, #1.    Increment the word count.
NEXT:    Add                   R2, R2, #1     Move to the next character.
         Branch                LOOP
DONE:    Move                  R2, R4         Pass the result in R2.
         Load                  R5, (SP)       Restore registers.
         Load                  R4, 4(SP)
         Load                  R3, 8(SP)
         Add                   SP, SP, #12
         Return
```

2.32. 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:    Subtract              SP, SP, #20    Save registers.
           Store                 R2, 16(SP)
           Store                 R3, 12(SP)
           Store                 R4, 8(SP)
           Store                 R5, 4(SP)
           Store                 R6, (SP)
           LShiftL               R2, R2, #2     Multiply by 4.
           Add                   R5, R3, R2     End of the list.
LOOP:      Load                  R6, (R3)       Check entries in the list
           Branch_if_[R4]≤[R6]   TRANSFER         until insertion point is reached.
           Add                   R3, R3, #4
           Branch_if_[R3]<[R5]   LOOP
           Branch                DONE
TRANSFER:  Load                  R6, (R3)       Insert the new entry and
           Store                 R4, (R3)         move the rest of the entries
           Move                  R4, R6           upwards in the list.
           Add                   R3, R3, #4     Increment the list pointer.
           Branch_if_[R3]<[R5]   TRANSFER
DONE:      Store                 R4, (R3)       Store the last entry.
           Load                  R6, (SP)       Restore registers.
           Load                  R5, 4(SP)
           Load                  R4, 8(SP)
           Load                  R3, 12(SP)
           Load                  R2, 16(SP)
           Add                   SP, SP, #20
           Return
```

2.33. 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 2.32, the desired subroutine may be implemented as follows:

```
INSERTSORT:  Subtract             SP, SP, #20        Save registers.
             Store                LINK_reg, 16(SP)
             Store                R2, 12(SP)
             Store                R3, 8(SP)
             Store                R4, 4(SP)
             Store                R10, (SP)
             Load                 R4, (R10)          Transfer one number from old list
             Store                R4, (R12)            to new list.
             Move                 R3, R12
             Move                 R2, #1
SCAN:        Add                  R10, R10, #4       Increment pointer to old list.
             Load                 R4, (R10)          Next number to be inserted.
             Call                 INSERT
             Add                  R2, R2, #1         Increment the length of new list.
             Branch_if_[R2]<[R11] SCAN
             Load                 R10, (SP)          Restore registers.
             Load                 R4, 4(SP)
             Load                 R3, 8(SP)
             Load                 R2, 12(SP)
             Load                 LINK_reg, 16(SP)
             Add                  SP, SP, #20
             Return

INSERT:      Subtract             SP, SP, #20        Save registers.
             Store                R2, 16(SP)
             Store                R3, 12(SP)
             Store                R4, 8(SP)
             Store                R5, 4(SP)
             Store                R6, (SP)
             LShiftL              R2, R2, #2         Multiply by 4.
             Add                  R5, R3, R2         End of the list.
LOOP:        Load                 R6, (R3)           Check entries in the list
             Branch_if_[R4]≤[R6]  TRANSFER             until insertion point is reached.
             Add                  R3, R3, #4
             Branch_if_[R3]<[R5]  LOOP
             Branch               DONE
TRANSFER:    Load                 R6, (R3)           Insert the new entry and
             Store                R4, (R3)             move the rest of the entries
             Move                 R4, R6               upwards in the list.
             Add                  R3, R3, #4         Increment the list pointer.
             Branch_if_[R3]<[R5]  TRANSFER
DONE:        Store                R4, (R3)           Store the last entry.
             Load                 R6, (SP)           Restore registers.
             Load                 R5, 4(SP)
             Load                 R4, 8(SP)
             Load                 R3, 12(SP)
             Load                 R2, 16(SP)
             Add                  SP, SP, #20
             Return
```