PROBLEMS (Cap. 4 - Istruzioni macchina)

2.1 Represent the decimal values 5, −2, 14, −10, 26, −19, 51, and −43, as signed, 7-bit numbers in the following binary formats:

(*a*) Sign-and-magnitude
(*b*) 1's-complement
(*c*) 2's-complement

(See Appendix E for decimal-to-binary integer conversion.)

2.2 (*a*) Convert the following pairs of decimal numbers to 5-bit, signed, 2's-complement, binary numbers and add them. State whether or not overflow occurs in each case.

(*a*) 5 and 10
(*b*) 7 and 13
(*c*) −14 and 11
(*d*) −5 and 7
(*e*) −3 and −8
(*f*) −10 and −13

(*b*) Repeat Part *a* for the subtract operation, where the second number of each pair is to be subtracted from the first number. State whether or not overflow occurs in each case.

2.3 Given a binary pattern in some memory location, is it possible to tell whether this pattern represents a machine instruction or a number?

2.4 A memory byte location contains the pattern 00101100. What does this pattern represent when interpreted as a binary number? What does it represent as an ASCII code?

2.5 Consider a computer that has a byte-addressable memory organized in 32-bit words according to the big-endian scheme. A program reads ASCII characters entered at a keyboard and stores them in successive byte locations, starting at location 1000. Show the contents of the two memory words at locations 1000 and 1004 after the name "Johnson" has been entered.

2.6 Repeat Problem 2.5 for the little-endian scheme.

2.7 A program reads ASCII characters representing the digits of a decimal number as they are entered at a keyboard and stores the characters in successive memory bytes. Examine the ASCII code in Appendix E and indicate what operation is needed to convert each character into an equivalent binary number.

2.8 Write a program that can evaluate the expression

$$A \times B + C \times D$$

in a single-accumulator processor. Assume that the processor has Load, Store, Multiply, and Add instructions, and that all values fit in the accumulator.

2.12 "Having a large number of processor registers makes it possible to reduce the number of memory accesses needed to perform complex tasks." Devise a simple computational task to show the validity of this statement for a processor that has four registers compared to another that has only two registers.

2.13 Registers R1 and R2 of a computer contain the decimal values 1200 and 4600. What is the effective address of the memory operand in each of the following instructions?

(*a*)	Load	20(R1),R5
(*b*)	Move	#3000,R5
(*c*)	Store	R5,30(R1,R2)
(*d*)	Add	−(R2),R5
(*e*)	Subtract	(R1)+,R5

2.14 Assume that the list of student test scores shown in Figure 2.14 is stored in the memory as a linked list as shown in Figure 2.36. Write an assembly language program that accomplishes the same thing as the program in Figure 2.15. The head record is stored at memory location 1000.

2.15 Consider an array of numbers $A(i,j)$, where $i = 0$ through $n - 1$ is the row index, and $j = 0$ through $m - 1$ is the column index. The array is stored in the memory of a computer one row after another, with elements of each row occupying m successive word locations. Assume that the memory is byte-addressable and that the word length is 32 bits. Write a subroutine for adding column x to column y, element by element, leaving the sum elements in column y. The indices x and y are passed to the subroutine in registers R1 and R2. The parameters n and m are passed to the subroutine in registers R3 and R4, and the address of element A(0,0) is passed in register R0. Any of the addressing modes in Table 2.1 can be used. At most, one operand of an instruction can be in the memory.

2.17 Register R5 is used in a program to point to the top of a stack. Write a sequence of instructions using the Index, Autoincrement, and Autodecrement addressing modes to perform each of the following tasks:

(*a*) Pop the top two items off the stack, add them, and then push the result onto the stack.

(*b*) Copy the fifth item from the top into register R3.

(*c*) Remove the top ten items from the stack.

2.18 A FIFO queue of bytes is to be implemented in the memory, occupying a fixed region of k bytes. You need two pointers, an IN pointer and an OUT pointer. The IN pointer keeps track of the location where the next byte is to be appended to the queue, and the OUT pointer keeps track of the location containing the next byte to be removed from the queue.

(*a*) As data items are added to the queue, they are added at successively higher addresses until the end of the memory region is reached. What happens next, when a new item is to be added to the queue?

(*b*) Choose a suitable definition for the IN and OUT pointers, indicating what they point to in the data structure. Use a simple diagram to illustrate your answer.

(*c*) Show that if the state of the queue is described only by the two pointers, the situations when the queue is completely full and completely empty are indistinguishable.

(*d*) What condition would you add to solve the problem in part *c*?

(*e*) Propose a procedure for manipulating the two pointers IN and OUT to append and remove items from the queue.

2.19 Consider the queue structure described in Problem 2.18. Write APPEND and REMOVE routines that transfer data between a processor register and the queue. Be careful to inspect and update the state of the queue and the pointers each time an operation is attempted and performed.

Chapter 2
Machine Instructions and Programs

2.1. The three binary representations are given as:

Decimal values	Sign-and-magnitude representation	1's-complement representation	2's-complement representation
5	0000101	0000101	0000101
−2	1000010	1111101	1111110
14	0001110	0001110	0001110
−10	1001010	1110101	1110110
26	0011010	0011010	0011010
−19	1010011	1101100	1101101
51	0110011	0110011	0110011
−43	1101011	1010100	1010101

2.2. (a)

(a)	(b)	(c)
00101	00111	10010
+ 01010	+ 01101	+ 01011
01111	10100	11101
no overflow	overflow	no overflow

(d)	(e)	(f)
11011	11101	10110
+ 00111	+ 11000	+ 10011
00010	10101	01001
no overflow	no overflow	overflow

(b) To subtract the second number, form its 2's-complement and add it to the first number.

(a)	(b)	(c)
00101	00111	10010
+ 10110	+ 10011	+ 10101
11011	11010	00111
no overflow	no overflow	overflow

(d)	(e)	(f)
11011	11101	10110
+ 11001	+ 01000	+ 01101
10100	00101	00011
no overflow	no overflow	no overflow

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

2.4. The number 44 and the ASCII punctuation character “comma”.

2.5. Byte contents in hex, starting at location 1000, will be 4A, 6F, 68, 6E, 73, 6F, 6E. The two words at 1000 and 1004 will be 4A6F686E and 736F6EXX. Byte 1007 (shown as XX) is unchanged. (See Section 2.6.3 for hex notation.)

2.6. Byte contents in hex, starting at location 1000, will be 4A, 6F, 68, 6E, 73, 6F, 6E. The two words at 1000 and 1004 will be 6E686F4A and XX6E6F73. Byte 1007 (shown as XX) is unchanged. (See section 2.6.3 for hex notation.)

2.7. Clear the high-order 4 bits of each byte to 0000.

2.8. A program for the expression is:

```
Load      A
Multiply  B
Store     RESULT
Load      C
Multiply  D
Add       RESULT
Store     RESULT
```

2.12. The dot product program in Figure 2.33 uses five registers. Instead of using R0 to accumulate the sum, the sum can be accumulated directly into DOTPROD. This means that the last Move instruction in the program can be removed, but DOTPROD is read and written on each pass through the loop, significantly increasing memory accesses. The four registers R1, R2, R3, and R4, are still needed to make this program efficient, and they are all used in the loop. Suppose that R1 and R2 are retained as pointers to the A and B vectors. Counter register R3 and temporary storage register R4 could be replaced by memory locations in a 2-register machine; but the number of memory accesses would increase significantly.

2.13. 1220, part of the instruction, 5830, 4599, 1200.

2.14. Linked list version of the student test scores program:

```
        Move      #1000,R0
        Clear     R1
        Clear     R2
        Clear     R3
LOOP    Add       8(R0),R1
        Add       12(R0),R2
        Add       16(R0),R3
        Move      4(R0),R0
        Branch>0  LOOP
        Move      R1,SUM1
        Move      R2,SUM2
        Move      R3,SUM3
```

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

Subroutine

```
        Move      R5,−(SP)    Save R5 on stack.
        Multiply  #4,R4       Use R4 to contain distance in
                              bytes (Stride) between successive
                              elements in a column.
        Multiply  #4,R1       Byte distances from A(0,0)
        Multiply  #4,R2       to A(0,x) and A(0,y)
                              placed in R1 and R2.
LOOP    Move      (R0,R1),R5  Add corresponding
        Add       R5,(R0,R2)  column elements.
        Add       R4,R1       Increment column element
        Add       R4,R2       pointers by Stride value.
        Decrement R3          Repeat until all
        Branch>0  LOOP        elements are added.
        Move      (SP)+,R5    Restore R5.
        Return                Return to calling program.
```

2.17. (a)

```
Move   (R5)+,R0
Add    (R5)+,R0
Move   R0,−(R5)
```

(b)

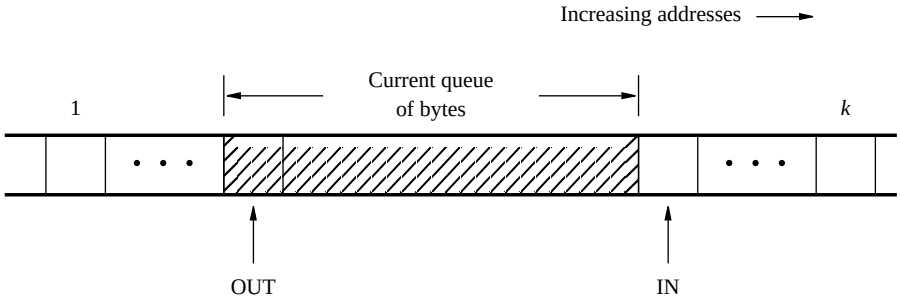
```
Move   16(R5),R3
```

(c)

```
Add    #40,R5
```

2.18. (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.

2.19. Use the following register assignment:

R0 − Item to be appended to or removed from queue
R1 − IN pointer
R2 − OUT pointer
R3 − Address of beginning of queue area in memory
R4 − Address of end of queue area in memory
R5 − Temporary storage for [IN] during append operation

Assume that the queue is initially empty, with [R1] = [R2] = [R3].

The following APPEND and REMOVE routines implement the procedures required in Part (e) of Problem 2.18.

APPEND routine:

```
            Move        R1,R5
            Increment   R1          Increment IN pointer
            Compare     R1,R4         Modulo k.
            Branch≥0    CHECK
            Move        R3,R1
CHECK       Compare     R1,R2       Check if queue is full.
            Branch=0    FULL
            MoveByte    R0,(R5)     If queue not full, append item.
            Branch      CONTINUE
FULL        Move        R5,R1       Restore IN pointer and send
            Call        QUEUEFULL     message that queue is full.
CONTINUE    ...
```

REMOVE routine:

```
            Compare     R1,R2       Check if queue is empty.
            Branch=0    EMPTY       If empty, send message.
            MoveByte    (R2)+,R0    Otherwise, remove byte and
            Compare     R2,R4         increment R2 Modulo k.
            Branch≥0    CONTINUE
            Move        R3,R2
            Branch      CONTINUE
EMPTY       Call        QUEUEEMPTY
CONTINUE    ...
```