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
```

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

The Base with index addressing mode (R1,R2) is used to access the scores on a particular test. Register R1 points to the test score for student 1, and R2 is incremented by Stride in the inner loop to access scores on the same test by successive students in the list.

```
        Move        J,R4          Compute and place Stride = 4(j + 1)
        Increment   R4              into register R4.
        Multiply    #4,R4
        Move        #LIST,R1      Initialize base register R1 to the
        Add         #4,R1           location of the test 1 score for student 1.
        Move        #SUM,R3       Initialize register R3 to the location
                                    of the sum for test 1.
        Move        J,R10         Initialize outer loop counter R10 to j.
OUTER   Move        N,R11         Initialize inner loop counter R11 to n.
        Clear       R2            Clear index register R2 to zero.
        Clear       R0            Clear sum register R0 to zero.
INNER   Add         (R1,R2),R0    Accumulate the sum of test scores in R0.
        Add         R4,R2         Increment index register R2 by Stride value.
        Decrement   R11           Check if all student scores on current
        Branch>0    INNER           test have been accumulated.
        Move        R0,(R3)       Store sum of current test scores and
        Add         #4,R3           increment sum location pointer.
        Add         #4,R1         Increment base register to next test
                                    score for student 1.
        Decrement   R10           Check if the sums for all tests have
        Branch>0    OUTER           been computed.
```

2.10. (a)

			Memory accesses
	Move	#AVEC,R1	1
	Move	#BVEC,R2	1
	Load	N,R3	2
	Clear	R0	1
LOOP	Load	(R1)+,R4	2
	Load	(R2)+,R5	2
	Multiply	R4,R5	1
	Add	R5,R0	1
	Decrement	R3	1
	Branch>0	LOOP	1
	Store	R0,DOTPROD	2

(b) $k_1 = 1 + 1 + 2 + 1 + 2 = 7$; and $k_2 = 2 + 2 + 1 + 1 + 1 + 1 = 8$

2.11. (a) The original program in Figure 2.33 is efficient on this task.

(b) $k_1 = 7$; and $k_2 = 7$

This is only better than the program in Problem 2.10(a) by a small amount.

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.
```

2.16. 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 instruction places 300 into memory word location 1000 when the instruction is executed as part of a program.

2.17. (*a*)

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

(*b*)

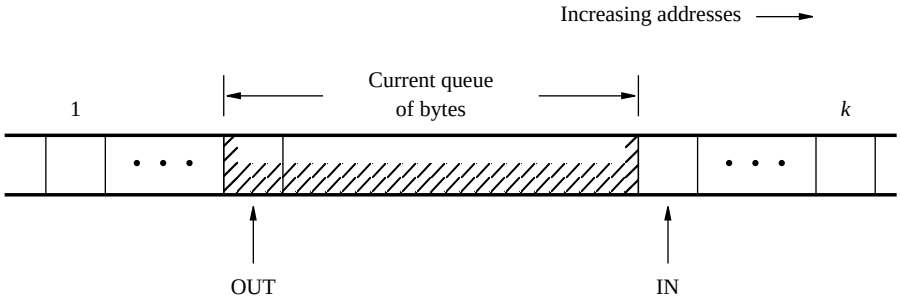
```
Move   16(R5),R3
```

(*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    ...
```

2.20. (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.

2.21. To allow nesting, the first action performed by the subroutine is to save the contents of the link register on a stack. The Return instruction pops this value into the program counter. This supports recursion, that is, when the subroutine calls itself.

2.22. Assume that register SP is used as the stack pointer and that the stack grows toward lower addresses. Also assume that the memory is byte-addressable and that all stack entries are 4-byte words. Initially, the stack is empty. Therefore, SP contains the address [LOWERLIMIT] + 4. The routines CALLSUB and RETRN must check for the stack full and stack empty cases as shown in Parts (b) and (a) of Figure 2.23, respectively.

```
CALLSUB   Compare     UPPERLIMIT,SP
          Branch≤0    FULLERROR
          Move        RL,−(SP)
          Branch      (R1)

RETRN     Compare     LOWERLIMIT,SP
          Branch>0    EMPTYERROR
          Move        (SP)+,PC
```

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

Modify Figure 2.37 as follows:

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

```
INSERTION   Move      #0, ERROR    Anticipate successful
                                     insertion of the new record.
            Compare   #0, RHEAD    (Existing instruction.)
```

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

```
            Branch≠0    CONTINUE1       Three new instructions.
            Move        RHEAD, ERROR
            Return
CONTINUE1   Branch>0    SEARCH          (Existing instruction.)
```

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

```
            Branch≠0    CONTINUE2       Three new instructions.
            Move        RNEXT, ERROR
            Return
CONTINUE2   Branch<0    INSERT          (Existing instruction.)
```

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

Replace Figure 2.38 with the following code:

```
DELETION    Compare     #0, RHEAD           If the list is empty,
            Branch≠0    CHECKHEAD             return with RIDNUM unchanged.
            Return
CHECKHEAD   Compare     (RHEAD), RIDNUM     Check if Head record
            Branch≠0    CONTINUE1             is to be deleted and
            Move        4(RHEAD), RHEAD       perform deletion if it
            Move        #0, RIDNUM            is, returning with zero
            Return                            in RIDNUM.
CONTINUE1   Move        RHEAD, RCURRENT     Otherwise, continue searching.
LOOP        Move        4(CURRENT), RNEXT
            Compare     #0, RNEXT           If all records checked,
            Branch≠0    CHECKNEXT             return with IDNUM unchanged.
            Return
CHECKNEXT   Compare     (RNEXT), RIDNUM     Check if next record is
            Branch≠0    CONTINUE2             to be deleted and perform
            Move        4(RNEXT), RTEMP       deletion if it is,
            Move        RTEMP, 4(RCURRENT)    returning with zero
            Move        #0, RIDNUM            in RIDNUM.
            Return
CONTINUE2   Move        RNEXT, RCURRENT     Otherwise, continue
            Branch      LOOP                  the search.
```