PROGRAMMING AND CODING THE IBM 709-7090-7094 COMPUTERS

PROGRAMMING AND CODING THE IBM 709-7090-7094 COMPUTERS

PHILIP M. SHERMAN

BELL TELEPHONE LABORATORIES
MURRAY HILL, NEW JERSEY

JOHN WILEY AND SONS, INC.

NEW YORK, LONDON AND SYDNEY

Printed in the United States of America

PREFACE

The purpose of this booklet is to explain the differences that exist between the hypothetical DELTA 63 (in PROGRAMMING AND CODING DIGITAL COMPUTERS) and the IBM 709-7090-7094 digital computers. It is deemed important that the reader "go on" a computer early in his studies. This booklet, used in conjunction with the book, attempts to permit him to do just that.

The book itself is self-contained; it stands by itself and makes no references to this booklet. The booklet, however, is tied intimately to the book. There are many references to the latter, indicated by the specific mention of pages.

The plan for the joint use of book and booklet is as follows. The reader follows the book with reference to the workplan of this booklet (placed at the start of this booklet). The workplan indicates when material here is to supplement, modify, or replace material in the book. The reader then makes appropriate references as noted. Material here follows the plan of the book and is placed in proper sequence. In general, the book material in small type, which is of a specific nature (specific to the DELTA 63), is supplemented, modified or replaced.

The effect of this joint usage is to yield a textbook that is of general structure, illustrated by coding for the IBM 709-7090-7094 computers. Characteristics of these computers, their repertoires of instructions, and examples of their coding appear in this booklet. A number of additional examples are included to reflect the special features and instructions of the IBM 709-7090-7094 computers.

Computer manuals, published by IBM on the three computers, should also be used.

Part I of the book is general in its approach and so needs no modification here. Parts II and III, however, are largely specific and so are well represented in this booklet.

An index to the 7090 instructions appearing in this booklet follows the regular index.

Philip M. Sherman

TABLE OF CONTENTS

WORKPLAN

This workplan indicates where the material in the booklet is to be used to supplement, modify, or replace the corresponding material in the book, PROGRAMMING AND CODING DIGITAL COMPUTERS. (S, M, and R indicate supplement, modify, and replace. These characters appear at each section within the booklet.)

	Pages	Material
	CHAPTER 5	
R	84.4 - 85.5	General structure of 7090
R	85.8 - 86.3	Addition and subtraction: instructions
R	86.7 - 87.2	Example 5.1
R	87.4 - 87.8	Example 5.2
S	At 87.10	Comments on overflow
R	88.4 - 89.1	Mult. and division: instructions
R	89.5 - 91.1	Examples 5.3,5.4,5.5
R	92.1 - 92.6	Example 5.6
R	93.3 - 93.6	Transfer instructions
M	93.6 - 94.4	Comments on transfers
R	94.5 - 97.10	Examples 5.7,5.8,5.9
	CHAPTER 6	
R	102.7 - 103.1	Example 6.1
S	At 104.4	What FAP is
R	106.8 - 107.1	Instruction format
M	107.4 - 109.7	Pseudo-operations
S	At 109.10	Qualifiers
R	110.2 - 110.6	Example 6.2
R	111.5 - 112.1	Example 6.3

Chapter 5
BASIC OPERATIONS

(R)----------(84.4 - 85.5*)----------

GENERAL STRUCTURE OF 709/7090/7094 COMPUTERS

The three computers are very similar, having the same memory capacity and essentially the same special registers and instructions. The 709 is slower than the 7090 by a factor of approximately 5; the 7094 has a few additional features and instructions. The three will be referred to by reference to the 7090.**

The IBM 7090 computer has 32,768 36-bit words, usually addressed octally, 00000 through 77777. Bits in memory words are labeled S, 1, 2, ..., 35. The S-bit holds the sign, so that a signed 35-bit number can be stored in each word; a positive sign is stored as a 0 and a negative sign is stored as a 1.

Magnetic tapes are connected to the computer for input-output purposes. Information may be read from magnetic tape or punched cards and may be written on tape, punched on cards, or printed on paper. Data transmitted between memory and an input-output unit must pass through a data channel.

Each instruction is placed in one word in memory Most instructions have the format shown in Figure 5.1. Bits S and 1 through 11, the operation field, hold the operation code. Bits 21 through 35, the address field, hold the operand address of the instruction. The octal representation of the instruction shown in Figure 5.1 is +050000015056. The operation code is $+0500_8$ and the operand address is 15056_8.

---

*Pages given at heads of sections indicate the pages in the text replaced by the material here. The digit after the decimal point indicates position on the page; thus "84.4" indicates a point about 4/10 down the page.

**Details on the 7090 and 7094 computers are available in these IBM Manuals: "Reference Manual - IBM 7090 Data Processing System" (Form A22-6528-4, 1962) and "Reference Manual - IBM 7094 Data Processing System" (Form A22-6703, 1962).

Pages	Material

CHAPTER 7

R 120.1 - 120.4	Example 7.1
R 120.8 - 122.3	Example 7.2
R 123.7 - 124.9	Example 7.3
R 125.6 - 126.4	Example 7.4
S At 129.2	Information on tape
S At 129.4	Input-output equipment
R 131.7 - 132.7	Example 7.5
R 133.1 - 134.6	Examples 7.6,7.7
R 135.2 - 136.6	Example 7.8

CHAPTER 8

R 141.8 - 142.7	Index Registers
R 143.1 - 143.9	Indexing instructions
M 143.9 - 144.3	Levels of addressing
M 144.4 - 144.10	The variable field
R 145.6 - 146.3	Example 8.1
R 146.4 - 146.8	Example 8.2
R 147.2 - 147.8	Example 8.3, indexing instructions
R 148.4 - 152.7	Examples 8.4,8.5,8.6
R 153.1 - 153.3	Time-space balance example
S At 154.7	Index register pointers
R 155.0 - 156.3	Indexing instructions, Ex. 8.7
R 156.7 - 157.8	Example 8.8
M 159.4 - 161.5	Indirect addressing
R 162.4 - 164.6	Example 8.9

Pages	Material

CHAPTER 9

M 168.2 - 168.6	Compare instruction
R 168.7 - 169.8	Example 9.1
S At 170.1	Test instructions
R 170.5 - 171.7	Examples 9.2,9.3
R 172.7 - 174.3	Examples 9.4,9.5
R 176.4 - 178.7	Example 9.6

CHAPTER 10

R 182.10 - 183.4	Example 10.1
R 185.4 - 185.9	Example 10.2
S At 186.4	Macro-instructions
R 186.5 - 186.7	Macro-instruction example
R 188.1 - 188.5	TSX instruction and linkages
R 189.7 - 190.2	Transfer of information
R 190.7 - 191.6	Example 10.3
R 191.7 - 192.4	Example 10.4
R 192.6 - 192.8	Subroutine call example
R 193.0 - 195.3	Examples 10.5,10.6,10.7
R 196.4 - 196.5	Multiple returns

CHAPTER 11

S At 206.8	Correction cards
R 208.8 - 209.10	BCD codes
R 210.7 - 211.1	BCD pseudo-operation
M 211.2 - 213.2	Input-output subroutines

Pages	Material
	CHAPTER 12
R 235.1 - 235.4	Instruction execution times
R 235.8 - 236.3	Example 12.2
R 240.3 - 241.6	Example 12.3
R 243.1 - 244.10	Shifting and masking
R 245.3 - 246.9	Example 12.4
R 247.2 - 247.9	Example 12.5
	CHAPTER 13
R 254.8 - 255.8	Floating pt. oper'ns and instructions
M 255.8 - 256.4	Floating-point examples
	CHAPTER 14
R 267.4 - 267.7	Compiler coding example
	CHAPTER 15
S At 296.5	Logical instructions
R 297.4 - 298.10	Example 15.1
M 299.3 - 299.8	Packing binary information
R 299.8 - 300.2	Example 15.2
R 301.2 - 301.7	Example 15.3
S At 302.1	ERA, PBT, and the sense indicators
R 307.7 - 311.7	Example 15.4
R 316.1 - 317.3	Word formats, Example 15.5
R 320.1 - 320.4	Example 15.6
R 320.6 - 321.5	Example 15.7
S At 322.0	PAC instruction
R 322.1 - 322.7	Example 15.8

Pages	Material

CHAPTER 16

CHAPTER 17

Pages	Material
R 371.2 - 371.8	Simulation macro-instructions
R 372.4 - 373.2	Example 17.13
R 373.6 - 374.8	Example 17.14
R 375.3 - 376.2	Example 17.15
R 376.5 - 378.4	Example 17.16

CHAPTER 18

R 387.5 - 391.2	Example 18.2
M 391.7 - 394.8	Comments on generality of material
R 395.3 - 395.10	Example 18.4

CHAPTER 19

S At 402.3	Assembler aids
R 402.4 - 403.8	Example 19.1
R 403.9 - 404.2	Example 19.2
R 406.4 - 406.7	Example 19.3
R 409.4 - 409.10	Trap feature (STR)
R 413.1 - 413.9	Examples 19.4, 19.5
R 414.3 - 415.3	Example 19.6

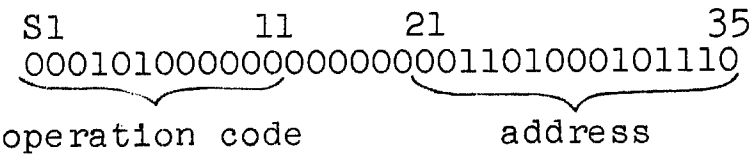


Figure 5.1. IBM 7090 instruction word.

Integers and fixed-point numbers each may occupy 35 bits, filling bits 1-35; the S-bit is used for the sign. Floating-point numbers each also occupy one word; one number is shown in Figure 5.2. Bits 1 through 8 hold the characteristic, and bits 9 through 35 hold the absolute value of the fraction. Characteristics are formed by adding 200$_8$ (128) to the powers of 2 in floating-point form. The binary point is assumed to be immediately to the left of the fraction. The octal form of the number shown in Figure 5.2 is -20622050400, which is the number -18.0790.

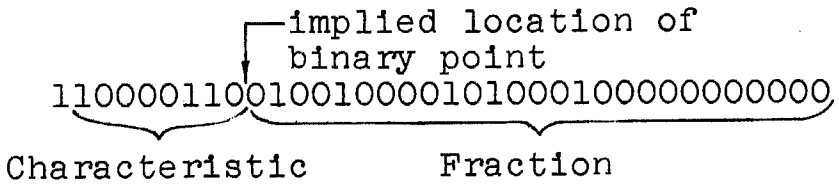


Figure 5.2. IBM 7090 floating-point number.

The accumulator register (AC) contains 38 bits, labeled S, Q, P, and 1 through 35. The Q and P bits are considered to be to the left of bit 1 and provide for overflow in the AC. The multiplier-quotient register (MQ) contains 36 bits, numbered as in a memory word. During the execution of special instructions, including multiplication and division, the MQ is used as the right-hand extension of the AC.

(R)----------(85.8 - 86.3)----------

ADDITION AND SUBTRACTION

Following are several of the arithmetic and data-moving instructions of the 7090. In these descriptions, the following sequence of information is given: The

instruction name in full, a 3-letter mnemonic abbreviation for the instruction, the operation code, the execution time in machine cycles, and the description. A machine cycle is 12, 2.18, and 2 microseconds for the 709, 7090 and 7094 computers, respectively. The Y that is mentioned refers to a memory location and represents an operand address. All registers affected by the instruction are mentioned.

CLEAR AND ADD (CLA Y) (+0500); 2 cycles. The C(Y) replaces the $C(AC)_{S,1-35}$.* Positions P and Q of the AC are set to zero. The C(Y) is unchanged.

STORE (STO Y) (+0601); 2 cycles. The $C(AC)_{S,1-35}$ replaces the C(Y). The C(AC) is unchanged.

ADD (ADD Y) (+0400); 2 cycles. The C(Y) is added algebraically to the C(AC) and the sum is placed in the AC. The C(Y) is unchanged.

SUBTRACT (SUB Y) (+0402); 2 cycles. The C(Y) is subtracted algebraically from the C(AC) and the difference is placed in the AC. The C(Y) is unchanged.

HALT AND TRANSFER (HTR Y) (+0000); 2 cycles. The computer stops upon execution of this instruction. (If the start key on the console is pushed, the computer takes its next instruction from location Y and proceeds from there.)

As the result of an addition or subtraction, if the C(AC) is zero, the sign of the AC is unchanged. Thus if the C(AC) is -60 and the C(Y) is +60, then after the addition of the C(Y) the C(AC) = -0.

(R)----------(86.7 - 87.2)----------

Example 5.1 Find the sum of 56, -45, 23, and -39. These numbers are located in sequence, beginning at location 00300. Place the sum in location 00304.

Since addition is performed in the AC, the first number must be loaded into the AC, and all other numbers must then be added to the first. Finally, the sum must be stored in 00304. The program is written to begin at location 00100 and end at location 00105. Location 00304 is set aside for the sum.

---

*Subscripts on an expression of the form C(X), where X is a word or register, refer to the only bits involved; bits not mentioned are not involved.

Location	Contents	Remarks
00100	+0500 000 00300	Load 56 into the AC
00101	+0400 000 00301	Add -45 to AC, giving 11
00102	+0400 000 00302	Add 23, giving 34
00103	+0400 000 00303	Add -39, giving -5
00104	+0601 000 00304	Store sum in 00304
00105	+0000 000 00000	Halt
00300	+000000000070	56 (Numbers are listed
00301	-000000000055	-45 at left in octal)
00302	+000000000027	23
00303	-000000000047	-39
00304	+000000000000	For sum

(R)----------(87.4 - 87.8)----------

Example 5.2 Find the value of m, where

$$m = a + b - c + d$$

The quantities a, b, c, and d are stored in sequence, starting at location 00675. Place the sum in location 00674.

The structure of this program is similar to the one in Example 5.1, except that one quantity (c) is subtracted from the C(AC). The program is written to start at location 00020. Location 00674 is set aside for the sum.

Location	Contents	Remarks
00020	+0500 000 00675	Load a into AC
00021	+0400 000 00676	Add b, forming a + b
00022	+0402 000 00677	Subtract c, forming a + b - c
00023	+0400 000 00700	Add d: a + b - c + d
00024	+0601 000 00674	Store sum
00025	+0000 000 00000	Halt
00674	+000000000000	For sum
00675	+xxxxxxxxxxxx	a
00676	+xxxxxxxxxxxx	b
00677	+xxxxxxxxxxxx	c
00700	+xxxxxxxxxxxx	d

The contents of the four words containing a, b, c, and d are shown as x's with plus signs. The x's stand for any digits, and the signs may be negative.

(S)----------(At 87.10*)----------

If two numbers are added, there may be an overflow bit (a carry) to the left of bit 1 in the accumulator, into bit P. Carries from bit P are placed in bit Q, and carries from bit Q are lost. When a "1" bit is so placed in bit P, overflow occurs and the overflow indicator is turned on. An instruction, TRANSFER ON OVERFLOW, may be used to test the status of this indicator.

(R)----------(88.4 - 89.1)----------

MULTIPLICATION AND DIVISION

MULTIPLY (MPY Y) (+0200); 2-14 cycles. The C(Y) is multiplied algebraically by the C(MQ), and the product is placed in the AC and the MQ. The less significant half of the product is placed in the MQ, and the more significant half is placed in the AC. Positions P and Q of the AC are set to zero. The sign of the product is placed in the signs of both registers.

To illustrate multiplication, let us assume for simplicity that the AC, the MQ, and location Y have 4 bits and a sign each. Let

$$C(MQ) = -1011_2$$

$$C(Y) = +0111_2$$

The product of these numbers is -01001101; it appears in the AC and the MQ as follows:

AC: -0100 MQ: -1101

Note that, if the product is small enough (4 bits here, or 35 bits in the actual MQ) all significant bits are located in the MQ.

DIVIDE OR HALT (DVH Y) (+0220); 3-14 cycles. The $C(AC)_{Q,P,1-35}$ and the $C(MQ)_{1-35}$ are treated as a 72-bit dividend and the C(Y) is treated as a 35-bit divisor. The sign of the AC is the sign of the dividend. If the

---

*This means "at the bottom of 87."

magnitude of the C(Y) is greater than the magnitude of the C(AC), division takes place. The 35-bit quotient is placed in the MQ and the remainder is placed in the AC. The C(Y) is unchanged. If the magnitude of the C(Y) is not greater than the magnitude of the C(AC), division does not occur and the computer stops with the divide-check indicator on; the C(AC) and the C(MQ) remain unchanged.

A similar instruction, DIVIDE OR PROCEED, is available. If division does not occur because the magnitude of the C(Y) is too small, division does not occur and the computer continues in sequence with the next instruction.

To illustrate division, assume again that registers and words have 4 bits and a sign each. Let the MQ contain the number 14 (16_8) and the C(AC) = 0. Let the C(Y) = 4. The quotient is 3 and the remainder is 2. The answer appears as follows:

AC: +0010 MQ: +0011

Instructions to load and store the MQ are required to perform these operations.

LOAD MQ (LDQ Y) (+0560); 2 cycles. The C(Y) replaces the C(MQ). The C(Y) is unchanged.

STORE MQ (STQ Y) (-0600); 2 cycles. The C(MQ) replaces the C(Y). The C(MQ) is unchanged.

It is sometimes necessary to move the C(AC) to the MQ, or vice versa, and to clear the AC except for its sign. The following instructions are so used.

EXCHANGE AC AND MQ (XCA) (+0131); 1 cycle. The $C(AC)_{S,1-35}$ and the C(MQ) are exchanged. Positions P and Q of the AC are set to zero.

This instruction requires no operand; the address field is left empty (00000) normally.

CLEAR MAGNITUDE (CLM) (+0760,0); 2 cycles. The $C(AC)_{Q,P,1-35}$ are cleared and the $C(AC)_S$ is unchanged.

(R)----------(89.5 - 91.1)----------

Example 5.3 Determine the value of the expression

$$f = (a+b)(c+d)/ac$$

The quantities a, b, c, and d, having the values 1.5, -3.5, 12.1 and 14, respectively, are stored in sequence, starting at location 01000. Place the value of f in location 00777. Scale all numbers upward by a factor of 10.

The numbers in this problem are small enough so that the MQ alone suffices for all calculations; the AC is not needed. It is necessary to store an intermediate result, (a+b), temporarily. This is stored in the location set aside for f, location 00777.

Before a division occurs, it is necessary to clear the AC unless it is known for certain that it contains zero. Here, we are assuming that all products are less than 35 bits, so that the AC is zero after each multiplication. After the first division in this program, a remainder might be left in the AC, so that the register is cleared, except for sign. The sign must be kept in the AC because that is taken as the sign of the dividend. Note the use of the XCA instruction.

(In this listing, the six rightmost octal digits of the AC and the MQ are shown with each instruction; the contents after execution are shown. Unknown quantities are shown by x's.)

Location	Contents	C(AC)	C(MQ)	Remarks
00100	+0500 000 01000	+..000017	+..xxxxxx	Load a
00101	+0400 000 01001	-..000024	+..xxxxxx	Add b
00102	+0601 000 00777	-..000024	+..xxxxxx	Store temp.
00103	+0500 000 01002	+..000171	+..xxxxxx	Load c
00104	+0400 000 01003	+..000405	+..xxxxxx	Add d
00105	+0131 000 00000	+..xxxxxx	+..000405	AC to MQ
00106	+0200 000 00777	-..000000	-..012144	(a+b)(c+d)
00107	+0220 000 01000	-..000000	-..000534	Divide by a
00100	+0760 000 00000	-..000000	-..000534	Clear AC, keeping sign
00111	+0220 000 01002	-..000152	-..000002	Divide by c
00112	-0600 000 00777	-..000152	-..000002	Store f
00113	+0000 000 00000	-..000152	-..000002	Halt
00777	+000000000000	For result (f)		
01000	+000000000017	a (Numbers are scaled up by 10)		
01001	-000000000043	b		
01002	+000000000171	c		
01003	+000000000214	d		

After multiplication, the C(MQ) = 121448 (5220). Division by 17_8 (15) gives 5348 (348) with no remainder. Division of this by 1718 (121) gives 2 with a remainder of 1528 (106). The value of f stored is 2; a more accurate value is 2.9, but the 9-digit is lost unless precautions are taken. Scaling all four original quantities as indicated does not improve accuracy. To avoid the loss of accuracy in division, it is necessary to scale the dividend up more than the divisor. This problem is characteristic of fixed-point division in any computer and provides a good argument for floating-point arithmetic.

In loading the MQ for division using LDQ (which is not done here), the sign of that register must be placed in the AC. This may be accomplished as follows:

```
+0560 000 xxxxx   Load MQ
+0500 000 xxxxx   Load AC with same number
+0760 000 00000   Clear AC, keeping sign
```

Example 5.4 Evaluate p^4; p = -13 and is stored in location 00160. Place the answer in the MQ.

Location	Contents	C(MQ)	Remarks
00200	+0560 000 00160	-000000000015	Load p into MQ
00201	+0200 000 00160	+000000000251	Multiply: p^2
00202	+0200 000 00160	-000000004225	Multiply: p^3
00203	+0200 000 00160	+000000067621	Multiply: p^4
00204	+0000 000 00000	+000000067621	Halt
00160	-000000000015	p	

Example 5.5 Evaluate the polynomial

$$F = 8x^5 + 4x^3 - x^2$$

x is stored in location 01000; F is to be left in the MQ.

If the program is written in the manner of earlier programs - evaluating each term separately and storing it temporarily - 16 instructions are required. If we note, however, that terms have common factors, some coding and program execution time can be saved. For example, all three terms have the factor x^2. The function can be regrouped as follows:

$$F = x^2(x(4+8x^2) - 1)$$

The program can be written by starting within the inner parentheses and performing all operations in sequence, ending the program outside the brackets.

Location	Contents	Remarks
00100	0560 000 01000	Load x to MQ
00101	0200 000 01000	Multiply by x: x^2
00102	0200 000 00201	Multiply by 8: $8x^2$
00103	0131 000 00000	$8x^2$ to AC
00104	0400 000 00200	Add 4: $4 + 8x^2$
00105	0131 000 00000	Sum to MQ
00106	0200 000 01000	Multiply by x
00107	0131 000 00000	Product to AC
00110	0402 000 00202	Subtract 1
00111	0131 000 00000	Difference to MQ
00112	0400 000 01000	Multiply by x
00113	0400 000 01000	Multiply by x
00114	0000 000 00000	Halt
00200	+000000000004	4
00201	+000000000010	8
00202	+000000000001	1
01000	+xxxxxxxxxxxx	x

(R)----------(92.1 - 92.6)----------

ANALYSIS FOR CODING

Example 5.6 Write a program to evaluate a general fifth-order polynomial, leaving the result in the AC. The coefficients a, b, c, d, e, and f are located in sequence starting at location 01000; x is in location 00700.

The coding for this problem follows directly from the last form for G at the bottom of page 91 in the book. The program starts within the inner parentheses and proceeds outward.

Location	Contents	Remarks
00100	+0560 000 01000	Load a into MQ
00101	+0200 000 00700	Multiply by x: ax
00102	+0131 000 00000	Product to AC
00103	+0400 000 01001	Add b: ax + b
00104	+0131 000 00700	Sum to MQ
00105	+0200 000 00700	Multiply by x: (ax+b)x
00106	+0131 000 00000	Product to AC
00107	+0400 000 01002	(ax+b)x + c
...	.....	
00123	+0400 000 01005	G in AC now
00124	+0000 000 00000	Halt

Several instructions are omitted; the sequence of four instructions (EXCHANGE, MULTIPLY, EXCHANGE, and ADD) is repeated three times after location 00107.

The time for the execution of instructions should be considered in setting up a problem of this type, especially if the sequence is to be repeated many times. The approach taken in Example 5.6 is relatively efficient, since operations are minimized for the general case. Note that multiplication takes 2 to 14 cycles, addition takes 2 cycles, and the exchange instruction takes 1 cycle.

(R)----------(93.3 - 93.6)----------

TRANSFER INSTRUCTIONS

The transfer instructions on the IBM 7090 correspond to the jump instructions of the book and on some other computers.

TRANSFER (TRA Y) (+0020); 1 cycle. The computer takes its next instruction from location Y and proceeds in sequence from there.

TRANSFER ON PLUS (TPL Y) (+0120); 1 cycle. If the sign of the AC is plus, the computer takes its next instruction from location Y and proceeds from there. If it is minus, the computer takes the next instruction in sequence.

TRANSFER ON MINUS (TMI Y); (-0120); 1 cycle. If the sign of the AC is minus, the computer takes its next instruction from location Y and proceeds from there. If it is plus, the computer takes the next instruction in sequence.

TRANSFER ON ZERO (TZE Y) (0100); 2 cycles. If the $C(AC)_{Q,P,1-35}$ is zero, the computer takes its next instruction from Y and proceeds from there. If it is not zero, the computer takes the next instruction in sequence.

TRANSFER ON NO ZERO (TNZ Y) (-0100); 2 cycles. If the $C(AC)_{Q,P,1-35}$ is not zero, the computer takes its next instruction from Y and proceeds from there. If it is zero, the computer takes the next instruction in sequence.

(M)----------(93.6 - 94.4)----------

(The following comments apply to 93.6 - 94.4, which should be read with these in mind.)

The following correspondence of instructions exists:

DELTA 63	IBM 7090
JUMP	TRA
JUMPMI	TMI
JUMPNZ	TNZ
JUMPPL	TPL
LOAD	CLA

The concepts described are of a general nature.

(R)----------(94.5 - 97.10)----------

CODING SOME DECISIONS

Example 5.7 Code the following operation: If $i \leq n$, continue at location 00150; if $i > n$, continue in sequence. The flowchart in Fig. 5.3a in the book pictures this decision.

The two conditions can be rewritten as "if $i - n \leq 0$" and "if $i - n > 0$." Since a test against n is not available directly, this revision is necessary. Conditional jump instructions are used to check for the first condition, which is really two decisions as far as the computer is concerned: "if $i - n < 0$" and "if $i - n = 0$." If neither condition holds, the program continues in sequence. The flowchart in Fig. 5.3b in the book pictures the revised decision.

Location	Contents	Remarks
00170	CLA 00500	Load i, located in 00500
00171	SUB 00501	Form i - n; n is in 00501
00172	TMI 00150	Jump if $(i - n) < 0$
00173	TZE 00150	Jump if $(i - n) = 0$

Control will go to 00174 if $i - n > 0$, as required.

In the next two examples, three-way and four-way decisions must be made. Since all transfer instructions can make only two-way decisions, it is necessary to place transfer instructions in sequence to accomplish these multiple decisions.

Example 5.8 If the C(00500) is (1) negative or zero, (2) positive but less than 20, or (3) 20 or greater, send control, respectively, to (1) 00600, (2) 00700, or (3) 01000. This decision appears in Fig. 5.4a in the book.

Let the C(00500) = x. The conditions are

1. If $x \leq 0$, go to 00600;
2. if $0 < x < 20$, go to 00700;
3. if $20 \leq x$, go to 01000.

The steps in the coding process can be listed as follows:

1. Place x in the AC; jump to 00600 if negative.
2. Jump to 00600 if zero.
3. Having taken care of nonpositive x, form x - 20 because condition 2 now becomes "if x - 20 < 0, go to 00700." Jump as indicated.
4. Having taken care of x < 20, jump to 01000.

A modified flowchart is drawn in Fig. 5.4b in the book.

Location	Contents	Remarks
00100	CLA 00500	Load x
00101	TMI 00600	Transfer if x is negative
00102	TZE 00600	Transfer if x is zero
00103	SUB 00200	Form x - 20
00104	TMI 00700	Transfer if $(x - 20) < 0$
00105	TRA 01000	Transfer if $(x - 20) \geq 0$
00200	+000000000024	20

The instructions at 00600, 00700, and 01000, and subsequent instructions are not listed.

Example 5.9 Either the quantity a (located in 00400) or the quantity b (located in 00402) is to be stored in location 00000, depending on these conditions:

If a is positive and b is zero, store a;
if a is positive and b is nonzero, store b;
if a is negative and b is zero, store b;
if a is negative and b is nonzero, store a.

To simplify the coding, assume that a is not zero. The flowchart for this problem is drawn in Fig. 5.5a in the book. The coding follows directly from the flowchart, which is labeled with addresses to match the program following. As the result of the two tests (on a and on b), a four-way branch occurs. The four paths merge into two paths, however, because there are only two actions to be taken. A modification of part of the flowchart is shown in Fig. 5.5b in the book.

Location	Contents	Remarks
00120	CLA 00400	Load a
00121	TPL 00125	Go to 00125 if a is +
00122	CLA 00402	Load b
00123	TZE 00130	Go to 00130 if b is 0
00124	TRA 00127	Go to 00127 if nonzero
00125	CLA 00402	Load b
00126	TNZ 00130	Go to 00130 if b is nonzero
00127	CLA 00400	Load a again
00130	STØ 00000*	Store AC (a or b) in 00000
00131	HTR	Halt

*Some IBM printing equipment uses the symbol "Ø" for the letter O and the symbol "0" for zero. This printing equipment uses no small letters.

Chapter 6
SYMBOLIC CODING

(R)----------(102.7 - 103.1)----------

A SYMBOLIC PROGRAM

Example 6.1 Evaluate the polynomial

$$F = 8x^5 + 4x^3 - x^2$$

x is stored in location X; F is to be left in the AC.

In this program, written in the symbolic language FAP for the IBM 7090, ØNE, FOUR, and EIGHT are used for the address of the constants 1, 4, and 8. This problem was coded in Example 5.5.

```
Locn.   Oper.     Address
 START  LDQ        X
        MPY        X
        MPY        EIGHT
        XCA
        ADD        FØUR
        XCA
        MPY        X
        XCA
        SUB        ØNE
        XCA
        MPY        X
        MPY        X
        HTR

 ØNE    +000000000001
 FØUR   +000000000004
 EIGHT  +000000000010
 X      +xxxxxxxxxxxx
```

(S)----------(At 104.4)----------

THE ASSEMBLER LANGUAGE

An assembly language very commonly used for the IBM 7090 computer is FAP (FORTRAN Assembly Program). FAP is a modification of SAP (Symbolic Assembly Program), written by United Aircraft for the IBM 704 computer.

(R)----------(106.8 - 107.1)----------

INSTRUCTION FORMAT

The use of the columns on a FAP symbolic card and the fields they comprise are as follows:

Columns	Field	Contents
1 - 6	Location field	Symbol (definition)
8 - 14	Operation field	Symbolic operation
16 - 72	Address field	Address and remarks

Figure 6.1 in the book also applies to a FAP symbolic card.

The location field may be left blank; several instructions in Example 6.1 have no symbols in their location fields. A symbol is defined by being placed in the location field of an instruction. The symbol may be placed anywhere in the field. Column 7 must be blank.

The operation field must begin in column 8. The variable field contains a symbolic address which must begin after at least one blank column following the operation, but no later than column 16. Remarks may be used, provided at least one blank column precedes them.

(M)----------(107.4 - 109.7)----------

(The following comments apply to 107.4 - 109.7, which should be read with these in mind.)

FAP pseudo-operations correspond exactly in their function and use to the pseudo-operations in the book. The following correspondence exists:

HAP	FAP
ØRIGIN	ØRG
END	END
ØCTAL	ØCT
DECML	DEC
BLØCK	BSS

The symbol "BSS" stands for block starting with symbol; a symbol in the location field is normally used to identify the block.

If the operation and variable fields of a symbolic card are left blank, FAP assembles a full word of 0-bits. If the operation field alone is blank, 0-bits will fill bits S, 1, and 2. If the address field alone is blank, 0-bits will fill bits 21-35, the address field of the instruction.

(S)----------(At 109.10)----------

QUALIFIERS

If it is desired to modify integer interpretation for several cards, so that all integers are treated as octal, the SAK pseudo-operation, placed in the operation field, is used. Cards following it, until a second SAK card is encountered, are so treated. Successive SAKs reverse the mode. A decimal qualifier, /D/, is also available. Any integer immediately following this qualifier is treated as decimal.

(R)----------(110.2 - 110.6)----------

THE ASSEMBLY LISTING

Example 6.2 Following is a listing of the program to evaluate the polynomial

$$F = 8x^5 + 4x^3 - x^2$$

coded in Example 5.6.

Object program (octal)		Source program (symbolic)		
Location	Contents	Location	Oper.	Address
00100			ØRG	/Ø/100
00100	+0560 000 00120	START	LDQ	X
00101	+0200 000 00120		MPY	X
00102	+0200 000 00117		MPY	EIGHT
00103	+0131 000 00000		XCA	
00104	+0400 000 00116		ADD	FØUR
00105	+0131 000 00000		XCA	
00106	+0200 000 00120		MPY	X
00107	+0131 000 00000		XCA	
00110	+0402 000 00115		SUB	ØNE
00111	+0131 000 00000		XCA	
00112	+0400 000 00120		MPY	X
00113	+0400 000 00120		MPY	X
00114	+0000 000 00000		HTR	
00115	+000000000001	ØNE	DEC	1
00116	+000000000004	FØUR	DEC	4
00117	+000000000010	EIGHT	DEC	8
00120	+000000000000	X		
	00100		END	START

(R)----------(111.5 - 112.1)----------

STEPS IN PROGRAM ASSEMBLY

Example 6.3 Compare the quantities p and q, stored in P and Q, respectively. If $p < q$, place the number 1 in NUMBER; if $p = q$, place the number 2 in NUMBER; if $p > q$, place the number 3 in NUMBER.

These conditions can be rewritten:

1. If $p - q < 0$, store 1;
2. if $p - q = 0$, store 2;
3. if $p - q > 0$, store 3.

Case 2 must be checked first, because -0 and +0 are treated differently. The flowchart is drawn in Fig. 6.2 in the book. Note that control is sent to one of three places so that the proper number (1, 2, or 3) can be obtained for storage in NUMBER. The three possible store operations are performed at one location, STØRE.

```
Locn.   Oper.    Address

        ØRG      /Ø/200
START   CLA      P
        SUB      Q          Form p - q
        TZE      GET2       Jump if zero
        TMI      GET1       Jump if minus
GET3    CLA      THREE      Here if plus...3 to AC
        TRA      STØRE
GET1    CLA      ØNE        1 to AC
        TRA      STØRE
GET2    CLA      TWØ        2 to AC
STØRE   STØ      NUMBER
        HTR

P
Q
NUMBER
ØNE     DEC      1
TWØ     DEC      2
THREE   DEC      3
        END      START
```

Chapter 7
PROGRAM LOOPS

(R)----------(120.1 - 120.4)----------

WHY USE LOOPS?

Example 7.1 Compute the value of x^{10}. The value of x is small enough so that the number x^{10} does not exceed the capacity of a computer word.

From Example 5.4, we note that a sequence of MPY instructions suffices.

```
Locn.   Oper.    Address
        LDQ      X
        MPY      X
        MPY      X
        MPY      X
        MPY      X
        MPY      X
        MPY      X
        MPY      X
        MPY      X
        MPY      X
        STQ      RESULT
        HTR

X
RESULT
```

(R)----------(120.8 - 122.3)----------

A SIMPLE LOOP

Example 7.2 Compute the value of x^n.

A flowchart appears in Fig. 7.1 in the book. The quantity p is the current value of the product; its initial value is 1. Counting is done with index i; its initial value is also 1. The important step is the multiplication of the accumulated product by x, producing one more power of x:

$$p \times x \rightarrow p$$

To allow for the case n = 0 a test is made; in that event, p is set equal to 1. The symbolic names used to label flowchart boxes correspond to the symbols in the following program. The test for loop termination is accomplished by checking (i - n) against zero.

```
Locn.    Oper.    Address
START    CLA      ØNE        1 to p and i
         STØ      P
         STØ      I
TESTN    CLA      N          Test for zero n
         TZE      DØNE
MLTPY    LDQ      P          p·x to p
         MPY      X
         STQ      P
INCRSE   CLA      I          i + 1 to i
         ADD      ØNE
         STØ      I
TEST     SUB      N          Test for end
         TMI      MLTPY      Back if not done
         TZE      MLTPY
DØNE     HTR

N
X
P
I
ØNE      DEC      1
```

Note that through the use of a program loop it is a simple matter to include n as a variable of the problem.

(R)----------(123.7 - 124.9)----------

A LOOP WITH ADDRESS MODIFICATION

STORE ZERO (STZ Y) (+0600); 2 cycles. The C(Y) is set to zero and its sign is set plus.

Example 7.3 Determine the sum of a given set of n numbers. The numbers are stored in the block beginning at NUMBRS; their sum is to be placed at SUM.

The flowchart is modified to include the operation of address modification; it is redrawn in Fig. 7.2 in the book. The memory box indicates that a_i is stored in location NUMBRS+i-1. Thus, initially, the program sums the C(NUMBRS); to sum a_i, the program sums the C(NUMBRS+i-1). After the last number is summed, the operand address of the ADD instruction is NUMBRS+n-1. The flowchart shows, however that the test for the end of the problem follows the modification of the index, so that the operand address at the time of the test is NUMBRS+n. Thus, the loop must terminate when the ADD instruction has been modified exactly n times.

In the following program, the ADD instruction is modified after each number is summed. A data instruction, or instruction used as a constant, can be set initially to check for the final value of the ADD instruction. When this constant matches the ADD, the loop terminates; a TNZ does the matching. In addition, another instruction (at SETWD) is used to "initialize" the ADD instruction.

It is necessary to set aside a block of words for the n numbers. Here, 1000 words are reserved. A word is also set aside for n.

```
Locn.    Oper.    Address

         STZ      SUM          0 to sum
         CLA      SETWD        Initialize instr.
         STØ      ADDNUM
         ADD      N            Set test word
         STØ      CØMPAR
LØØP     CLA      SUM          Add a number
ADDNUM   ADD      NUMBRS
         STØ      SUM
         CLA      ADDNUM       Modify instr.
         ADD      ØNE
         STØ      ADDNUM
         SUB      CØMPAR
         TNZ      LØØP
DONE     HTR

SUM
SETWD    ADD      NUMBRS
CØMPAR                         (ADD NUMBRS+n)
N
ØNE      DEC      1
NUMBRS   BSS      1000
```

A number of 7090 instructions have negative operation codes; STQ and TNZ are among them. Adding +1 to such instructions has the effect of decreasing the address portion of the instruction. This difficulty may be avoided by the consistent use of the following instructions in place of CLA and STØ for address modification; their use ignores the operation code sign, in effect, and arithmetic is performed as desired.

CLEAR AND ADD LOGICAL (CAL Y) (-0500); 2 cycles. The C(Y) replaces the $C(AC)_{P,1-35}$. Positions S and Q of the AC are set to zero.

STORE LOGICAL WORD (SLW Y) (+0602); 2 cycles. The $C(AC)_{P,1-35}$ replaces the C(Y). The C(AC) is unchanged.

These instructions move bit P of a storage word to bit S of the accumulator and vice versa, so that arithmetic may be performed on the storage word as though it were a 36-bit positive integer. There are other, more significant uses for these two instructions; refer to Chapter 12.

(R)----------(125.6 - 126.4)----------

POLYNOMIAL EVALUATION

Example 7.4 Write a program to evaluate a polynomial of order n, for n as large as 100. The number n, the n + 1 coefficients, and the variable x are all given. These are located, respectively, in N, the block starting at CØEFF, and X. The coefficients are $b_0, b_1, \ldots, b_n$.

$$F = b_0 x^n + b_1 x^{n-1} + \ldots + b_n$$

A flowchart appears in Fig. 7.3 in the book. The program has a structure similar to that of Example 7.3 as regards its address modification and initialization. The significant operation is the calculation of q, the accumulated partial polynomial value. The calculation is

$$qx + b_i \rightarrow q$$

Reference to Example 5.6 (page 92 in the book) indicates why this operation repeated for successive coefficients b_i yields the value of F.

Locn.	Oper.	Address	
	CLA	CØEFF	b_0 to Q
	STØ	Q	
	CAL	SETWD	Initialize
	SLW	MØD	
	ADD	N	
	SLW	CØMPAR	
LØØP	LDQ	Q	$q \cdot x + b_i$ to q
	MPY	X	
	XCA		

(cont'd)

```
Locn.   Oper.    Address

MØD     ADD      CØEFF+1
        STØ      Q
        CAL      MØD          Modify instruction
        ADD      ØNE
        SLW      MØD
        CLA      MØD
        SUB      CØMPAR
        TNZ      LØØP         Back if not done
DØNE    HTR

Q
SETWD   ADD      COEFF+1
CØMPAR                        (ADD CØEFF+1+n)
N
X
ØNE     DEC      1
CØEFF   BSS      101
```

(S)----------(At 129.2)----------

INFORMATION ON TAPE

Magnetic tapes for the 7090 computer are $\frac{1}{2}$" in width and are normally 2400' long. Information is stored in seven channels, in the manner described in the book.

Data are recorded on tape in one of two modes; the difference is usually of little concern to the programmer. In the binary mode, information appears as described in the book. In the BCD mode, some bit configurations are changed and the check bit is such that there are an even number of 1's across the tape width.

Record gaps are $\frac{3}{4}$" long. The ends of files are indicated by special marks and/or end-of-file gaps; the latter are $3\frac{3}{4}$" long.

(S)----------(At 129.4)----------

TAPE READING AND WRITING

(The material in this section and in the remainder of Chapter 7, although hypothetical, is of interest to the 7090 programmer. He will rarely write his own input-output coding; rather he will use a monitor system, described in Chapter 11 in the book. Therefore, in order to appreciate input-output operations, the DELTA 63 instructions should be studied. They are simplified versions of instructions that actually exist on the 7090. The latter instructions are much more complex.)

Magnetic tape may be read from or written on at the rates of 75" per second (709 and 7090) or 112.5" per second (7090). Information passes between core storage and magnetic tape at rates of 15,000 to 62,500 lines of bits per second. Each line of bits represents one character--such as a letter, digit, or punctuation mark--so that the maximum rate is 62,500 characters per second. Recent equipment uses rates up to 90,000 characters per second.

Information is written or read in one direction only. A tape may be backspaced or rewound, however, and be read or written on again. Instructions are available to backspace one record, to backspace one file, to write an end-of-file gap and mark, and to rewind a tape.

Data transmitted between core storage and an input-output device (magnetic tape, card reader, card punch, printer) must pass through a data channel. The operation of a data channel is initiated within a program in the computer, but once started the channel operates independently of the program. Data channels control the quantity and destination of the data transmitted through them.

The computer and a data channel cannot both make a reference to core storage at the same time, so that the execution of a main program instruction may be delayed until the needs of the data channel are satisfied. The delays do not interfere with the main program in any other way. If the instruction being executed does not use core storage when a channel requires a reference to storage, normally no delay occurs.

A maximum of 10 tapes per channel can be used. Each tape unit has an address, as does each channel. The combination of the two addresses specifies a particular tape unit attached to a particular channel.

A card reader reads cards at the rate of 250 cards per minute. Information punched on the cards may be binary, decimal, alphabetic, or in another format. The reading format is controlled by the stored program and a control panel attached to the reader. Any sequence of channel commands calling for the uninterrupted transmission of 24 words causes the reading of one complete card. The words read are stored in consecutive core storage locations, starting with the address specified in the channel instruction. Word counts of other than 24 may also be given and cards are read as required.

One card punch may be attached to any channel. Punched card output may be decimal, alphabetic, binary, or any other desired form. Cards are punched at the rate

of 100 per minute. The punching format is controlled by the stored program and a control panel on the unit. Starting with the location in core storage specified by the channel instruction, 24 words from consecutive locations are punched on a card. Counts other than 24 may also be punched by a single instruction.

One printer may be attached to any channel. Information may be printed in any form within the limitations of the set of characters available. A set of 48 different characters is available. Information is printed at the rate of 150 lines per minute. The format of the information is controlled by the stored program and a control panel on the printer.

(R)----------(131.7 - 132.7)----------

Example 7.5 Write a loading program to load from tape A.

The READTA instruction in the program below is a DELTA 63 instruction.* A reading loop, using the skip feature of this instruction, is established. A flowchart is drawn in Fig. 7.6 in the book. After a record of 24 words is read in, the reading instruction is address modified so that the next record is read into memory at a location 24 addresses later. This process repeats until the file is exhausted.

```
Locn.   Oper.    Address

READIN  READTA   /Ø/1000     Loading point
        TRA      /Ø/1005     Starting point
        CAL      READIN      Modify instruction
        ADD      N24
        SLW      READIN
        TRA      READIN

N24     DEC      24
```

*The DELTA 63 input-output instructions in these examples can be simulated by macro-instructions, described in Section 10.2 and Chapter 17 of the book.

(R.)----------(133.1 - 134.6)----------

LOADING DATA

STORE ADDRESS (STA Y) (+0621); 2 cycles. The contents of the address field of the AC, i.e., bits 21-35, replaces the contents of the address field of location Y. The C(AC) and the other bits in Y are unchanged.

Example 7.6 Write a card-loading program that stops loading on encountering an end-of-program card which contains the octal number 777777777777 in the first word position.

This loading program is similar to the one coded in the last example, which reads information from tape. The only change (aside from the reading instruction) is that a test for the end-of-program card must be made after each card is read. As the first word of each card is read into a memory location, the contents of that location must be checked for 777777777777; if that number is found, control goes to the object program for execution. If the end-of-program card is omitted, card reading would be attempted when no cards are present in the card reader, and the computer would stop. A flowchart appears in Fig. 7.7 in the book.

```
Location    Oper.    Address

READIN      READC    /Ø/1000      Loading point
            HTR                   No 777777777777
TEST        CLA      /Ø/1000      Test first word for 7's
            SUB      SEVENS
            TZE      /Ø/1005      Go to program
            CAL      READIN       Modify instructions
            ADD      N24
            SLW      READIN
            STA      TEST
            TRA      READIN

N24         DEC      24
SEVENS      ØCT      777777777777
```

As an example of a program that reads its data, consider the summation of n numbers; this problem was coded in Example 7.3.

Example 7.7 Determine the sum of a given set of n numbers. The numbers are stored in the block beginning at NUMBRS; their sum is to be placed in SUM.

The numbers are stored on data cards. The number n' is in the first word position of the first data card, and the n numbers are stored on the following cards, punched in binary, 24 to a card. The last card is filled out with zeros. The reading loop is similar to the loop in Example 7.5, where an object program is read in.

```
Location    Oper.    Address
            READC    N,1        Read in n
            TRA      ERRØR
LØØP        READC    NUMBRS     Read in 24 numbers
            TRA      START      Go to summation seq.
            CAL      LØØP
            ADD      N24
            SLW      LØØP
            TRA      LØØP
START       STZ      SUM
            ...      ...
ERRØR       ...      ...
```

The program continues as in Example 7.3.

(R)----------(135.2 - 136.6)----------

READING OUT RESULTS

Example 7.8 A deck of data cards contains n integers, one to a card, in the first word position, in binary form. Write a program that computes the sum of each set of three integers in succession and writes the n/3 sums on tape. The number n, a multiple of 3, appears on the first data card.

(Refer to the book for an analysis and a flowchart. Note that STØRAD is equivalent to STA.)

```
Location    Oper.    Address

STEP1       READC    N,1          Read n
            TRA      ERRØR        Tra if end of file
            CAL      SETWD1       Initialize first loop
            SLW      LØØP1
            ADD      N
            SLW      CMPAR1
LØØP1       READC    NMBRS,1      Read a number
            TRA      ERRØR
            CAL      LØØP1
            ADD      ØNE
            SLW      LØØP1
            CLA      LØØP1
            SUB      CMPAR1
            TNZ      LØØP1
STEP2       CAL      SETWD2       Initialize second loop
            SLW      LØØP2
            ADD      ØNE
            STA      LØØP2+2
            ADD      ØNE
            STA      LØØP2+2
            ADD      N
            STA      CMPAR2
LØØP2       CLA      NMBRS        Add 3 numbers
            ADD      NMBRS+1
            ADD      NMBRS+2
            STØ      SUM
            WRITEB   SUM,1        Write sum on tape B
            CAL      LØØP2        Modify instrs.
            ADD      THREE
            SLW      LØØP2
            ADD      ØNE
            STA      LØØP2+1
            ADD      ØNE
            STA      LØØP2+2
            CLA      LØØP2+2
            SUB      CMPAR2       Test for last sum
            TNZ      LØØP2
            HTR

ØNE         DEC      1
THREE       DEC      3
N
SUM
SETWD1      READC    NMBRS,1
SETWD2      CLA      NMBRS
CMPAR1                            (READC NMBRS+n,1)
CMPAR2      ADD      **           (NMBRS+2+n)
NMBRS       BSS      3000
ERRØR       ...      ...
```

Chapter 8
INDEX REGISTERS

(R)----------(141.8 - 142.7)----------

THE INDEX REGISTERS

The 709 and 7090 computers each have three index registers, designated 1, 2, and 4. (The 7094 computer has seven index registers.) Associated with most instructions is a tag which specifies one of these registers. Bits 18-20 in the instruction word comprise the tag field, pictured in Fig. 8.1 below. This 3-bit field may contain the integers 0, 1, 2, and 4.* A tag of 0 indicates that

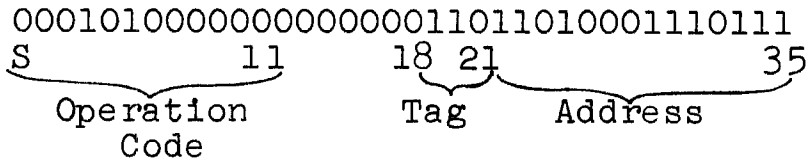


Fig. 8.1 Format of 7090 instruction.

no index register is specified, whereas a nonzero integer designates a particular one. The symbol XR is used for "index register," and XR1, XR2, and XR4 refer to the specified registers. Each index register contains 15 bits.

The seven index registers of the 7094 are designated 1, 2, ..., 7, and the tag field is the same as the other computers. Integers 0, 1, ..., 7 are used as described above.

A tag is indicated in FAP by placing its numerical designator in the variable field, after a comma following the address, without an intervening blank space. The following instructions, shown both in symbolic and assembled form, indicate the use of XR1 and XR4, respectively:

Machine word	Symbolic instruction
+0500 00 1 04500	CLA LIST,1
+0400 00 4 04512	ADD LIST+10,4

*The integers 3, 5, and 6 may be used in the multiple-tag mode. Refer to the IBM 7090 Manual.

The extra blank space, just to the left of the tag, is included in the assembled machine word for clarity.

In an instruction with no tag (zero in the tag field), the address of the word that is processed is simply the operand address. In an instruction with a tag, however, the address of the processed word is the operand address decreased by the contents of the specified index register. This address modification is automatic and temporary; the instruction does not change, but the effect is as though it were changed during the execution of the instruction. As an example, let the C(XR1) = 100; the instruction

```
ADD   WØRD,1
```

will cause the C(WØRD-100) to be added to the accumulator.

(R)----------(143.1 - 143.9)----------

INSTRUCTIONS

LOAD INDEX FROM ADDRESS (LXA Y,T) (+0534); 2 cycles. The $C(Y)_{21-35}$ replaces the contents of the specified index register. The C(Y) is unchanged.

ADDRESS TO INDEX TRUE (AXT Y,T) (+0774); 1 cycle. Positions 21-35 (Y) of this instruction replace the contents of the specified index register.

The following examples illustrate these instructions; (1) if the C(NUMBER) = 500, the instruction

```
LXA   NUMBER,1
```

places the number 500 in XR1; (2) the instruction

```
AXT   1000,1
```

places the number 1000 in XR1.

Several indexing instructions are similar to two-address instructions; they contain two operands. These instructions have a fourth field, the decrement field, which has 15 bits; it is pictured in Fig. 8.2. In these instructions, the operation code occupies only three bits,

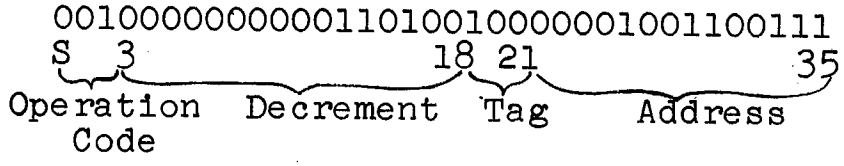


Fig. 8.2 Format of certain indexing instructions

S, 1, and 2; the decrement field occupies bits 3-17. The 15-bit, unsigned number in the decrement field is the decrement.

TRANSFER WITH INDEX INCREMENTED (TXI Y,T,D) (+1); 2 cycles. The decrement of this instruction is added to the contents of the specified index register, and the sum is placed in the index register. The computer takes its next instruction from location Y.

This instruction does two distinct things, independently of each other: it modifies an index register by a specified amount and it transfers control unconditionally.

The form of the instruction in FAP format is the following:

```
TXI   NEXT,2,26
```

This instruction increases the C(XR2) by 26 and transfers control to NEXT; if the address of NEXT is 01147, then this instruction is pictured in Fig. 8.2. Note that the decrement is placed in the variable field, directly after the tag with an intervening comma. The three fields in the variable field, the address, tag, and decrement, appear in that order, but appear in the reverse order within the instruction.

Frequently, it is necessary only to modify the index register without transfering control elsewhere; the following form is then used:

```
TXI   *+1,2,26
```

The decrement may be written as a negative number:

```
TXI   NEXT,1,-10
```

Here, the C(XR1) is decreased by 10. Since decrements are unsigned numbers, the 2's complement is placed in the instruction word; the 2's complement of 12_8 (10) is 77766_8, so that the assembled word for this last instruction is

```
+1 77766 1 01147
```

If the C(XR1) = 24_8 (20) prior to execution of this instruction, it is $00024_8 + 77766_8 = 00012_8 \pmod{100000_8}$ afterward; $12_8 = 10$.

TRANSFER ON INDEX LOW OR EQUAL (TXL Y,T,D) (-3); 2 cycles. If the contents of the specified index register is less than or equal to the decrement of this instruction,

the computer takes its next instruction from location Y. If the contents of the index register is greater than the decrement, the computer takes the next instruction in sequence.

TRANSFER ON INDEX HIGH (TXH Y,T,D) (+3); 2 cycles. If the contents of the specified index register is greater than the decrement of this instruction, the computer takes its next instruction from location Y. If the contents of the index register is less than or equal to the decrement, the computer takes the next instruction in sequence.

These last two instructions are conditional transfer instructions; a condition in an index register is tested. An example of the former instruction is the following:

```
TXL   LØØP,4,3
```

Control goes to LØØP if the C(XR4) is less than or equal to 3.

(M)----------(143.9 - 144.3)----------

(The following comments apply to 143.9 - 144.3.)
The following correspondence of instructions exists:

DELTA 63	IBM 7090
SETXRI	AXT
SETXR	LXA
INCRXM	TXI with an address of "*+1"
XJUMP	TXL (approximately)

(M)----------(144.4 - 144.10)----------

(The following comments apply to 144.4 - 144.10.)

THE VARIABLE FIELD

The material applies equally well to FAP, except for the examples of instructions. Those instructions, however, are merely illustrative.

(R)----------(145.6 - 146.3)----------

Example 8.1 Compute the value of x^n. This problem was previously coded in Example 7.2. The result is placed in P.

In this program, XR1 is used to represent the index i of Example 7.2. XR1 is set to 1 initially and increased by 1 each loop cycle, after the multiplication occurs. Then a test of the C(XR1) is made by a TXL instruction; when the index i exceeds n, after just being increased, the computation must cease and control passes to DØNE instead of being returned to MLTPY for further multiplication.

A test for the case n = 0 is included by a test for 0. Prior to this, 1 is stored in P to allow for this possibility. If $n \neq 0$, P is later set to x^n.

Initially, it is assumed that n is a known number; it is coded in the decrement of the TXL instruction.

```
Locn.    Oper.    Var. Field

START    AXT      1,1              Set i = 1
         CLA      ØNE              Set P = 1
         STØ      P
         CLA      N                Out if n = 0
         TZE      DØNE
         LDQ      ØNE
MLTPY    MPY      X                p·x to p
         TXI      *+1,1,1          i+1 to i
         TXL      MLTPY,1,n        Test for i = n
         STØ      P
DØNE     HTR

N                 n
X
P
ØNE      DEC      1
```

Note that in this program, n is an integer, placed initially in the decrement of the TXL instruction. Normally, n would be supplied as data and would be stored in the decrement during the running of the program.

An alternate approach is to have the XR value run backwards, from n down to 1, decreasing by 1 each loop cycle. This is more common in FAP, because of the fact that effective addresses are formed by subtraction of index register contents. The following program uses this approach. The index register is initially set to n; it is tested for equality with 1 at the end of the loop (actually it is tested to see if it exceeds 0). As long as i exceeds 0, control returns to MLTPY. The initial test for zero n is done with a TXL instruction.

```
Locn.   Oper.    Var. Field

START   LXA      N,1            Set i = n
        CLA      ØNE            Set P = 1
        STØ      P
        TXL      DØNE,1,0       Out if n = 0
        LDQ      ØNE
MLTPY   MPY      X              p·x to p
        TXI      *+1,1,-1       i-1 to i
        TXH      MLTPY,1,0      Test for i = 1
        STØ      P
DØNE    HTR

N                n
X
P
ØNE     DEC      1
```

(R)----------(146.4 - 146.8)----------

Example 8.2 Determine the sum of a given set of n numbers. The numbers are stored in the block beginning at NUMBRS; their sum is to be placed in SUM.

This problem was previously coded in Example 7.3.

The ADD instruction must refer to all n numbers in sequence, so that it is tagged; XR1 is used. The effective address initially must be NUMBRS; it must then be NUMBRS+1, etc. Since index registers decrement direct addresses, it is most convenient to provide a direct address that includes the size of the block to be processed. If the index register is set to that size, the initial effective address is that of the first word in the block. If the final index register value is 1, the final effective address is that of the last word in the block. This approach is feasible if the quantities to be processed are stored in ascending memory locations.

Initially, we assume that n is known to be 100. The operand address of the ADD instruction is NUMBRS+100, while the index register is initially 100; the final value of the index register is 1, so that the final effective address is NUMBRS+99, the address of the 100th and final number.

```
Locn.   Oper.    Var. Field
        STZ      SUM
        AXT      100,1
        CLA      SUM
LØØP    ADD      NUMBRS+100,1     Add a number
        TXI      *+1,1,-1
        TXH      LØØP,1,0         Test for end
        STØ      SUM
DØNE    HTR

SUM
NUMBRS  BSS      100
```

If n is a variable of the problem, as it usually is, it is necessary to set the address of the ADD instruction during the run of the program. For this purpose, the data instruction at XNUMBR is used, and the STA instruction sets the address. (Alternately, the decrement of an indexing instruction can be set, as in the first program in Example 8.1.) However, it is still necessary to know the upper bound on n, so that a block can be set aside. Assume that bound is 100; the resulting program follows.

```
Locn.   Oper.    Var. Field
        CLA      XNUMBR           Set ADD instr.
        ADD      N
        STA      LØØP
        STZ      SUM
        LXA      N,1
        CLA      SUM
LØØP    ADD      **,1             (NUMBRS+n)
        TXI      *+1,1,-1
        TXH      LØØP,1,0
        STØ      SUM
DØNE    HTR

SUM
N
XNUMBR           NUMBRS
NUMBRS  BSS      100
```

The "**" in the variable field of the second ADD instruction indicates that an address is to be supplied when the program is run.

(R)----------(147.2 - 147.8)----------

Example 8.3 Write a program to evaluate a polynomial of order n, for n as large as 100. The number n, the coefficients, and the variable x are located in N, the block starting at CØEFF, and X, respectively.

This program was previously coded in Example 7.4.

```
Locn.   Oper.   Var. Field

        CLA     XCØEFF          Set ADD instr.
        ADD     N
        STA     LØØP+2
        LXA     N,1
        LDQ     CØEFF           b0 to MQ
LØØP    MPY     X               C(AC)·x + b1 to AC
        XCA
        ADD     **,1
        XCA
        TXI     *+1,1,-1
        TXH     LØØP,1,0
        STQ     Q
        HTR

Q
N
X
XCØEFF          CØEFF+1
CØEFF   BSS     101
```

Three more indexing instructions are described. The first two permit the contents of an index register to be saved in storage. The third combines the operations performed by the pair of TXI and TXH instructions in the last four programs into one instruction.

STORE INDEX IN ADDRESS (SXA Y,T) (+0634); 2 cycles. The contents of the specified index register replaces the $C(Y)_{21-35}$. The $C(Y)_{S,1-20}$ is unchanged.

STORE INDEX IN DECREMENT (SXD Y,T) (-0634); 2 cycles. The contents of the specified index register replaces the $C(Y)_{3-17}$. The $C(Y)_{S,1,2,18-35}$ is unchanged.

TRANSFER ON INDEX (TIX Y,T,D) (+2); 2 cycles. If the contents of the specified index register is greater than the decrement of this instruction, the number in the index register is decreased by the decrement and the computer takes its next instruction from location Y. If the contents of the index register is less than or equal to the decrement, the index register is unchanged and the computer takes the next instruction in sequence.

The action of this instruction is pictured in Fig. 8.3. Location Y in this instruction is usually the return point in a loop and hence precedes the test at the TIX instruction.

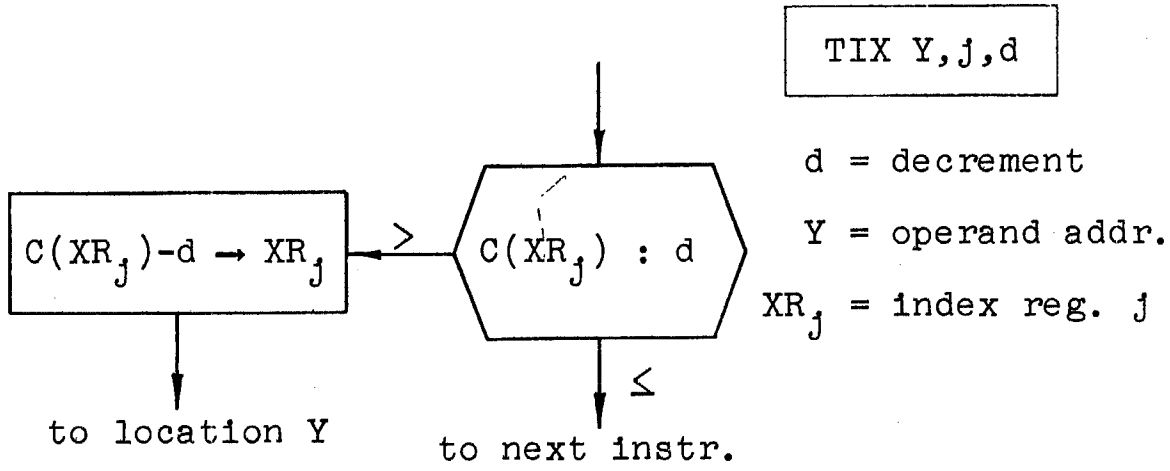


Fig. 8.3 The TIX instruction.

(R)----------(148.4 - 152.7)----------

Example 8.4 Given a set of n numbers in LIST, count the number of negative numbers present; n is at most 1000. Place the count in CØUNT.

This problem was analyzed in Section 2.2. A flowchart appears in Fig. 8.2 in the book. Index i counts loop cycles and index j counts the number of negative integers. XR1 and XR2 are used for these, respectively. At the end of the program, the C(XR2), the desired count, is placed in CØUNT.

The TIX instruction is used for loop control and the STA instruction is used to set the address of the CLA instruction that places a number in the AC for testing.

```
Locn.    Oper.    Var. Field
I        SET      1              Set index symbols
J        SET      2
         CLA      XLIST          Set address of instr.
         ADD      N
         STA      TEST
         LXA      N,I            Set index i
         AXT      0,J            Zero index j
TEST     CLA      **,I           (LIST+n)
         TZE      TIXPT          No count if 0
         TPL      TIXPT          No count if +
INCRSE   TXI      *+1,J,1        j+1 to j (if number is -)
TIXPT    TIX      TEST,I,1
         SCA      CØUNT,J        Store count
         HTR

N
CØUNT
XLIST             LIST
LIST     BSS      1000
```

Example 8.5 Given 80 numbers, find the sum of the tenth powers of the numbers. The numbers are stored in TABLEZ; the sum is to go in SUMZ.

This problem was analyzed in Section 2.4 (page 36); a flowchart appears in Fig. 8.3 in the book.

Since the flowchart shows two nested loops, two index registers are required for the two indices i and j. The current value of the accumulator during the multiplication process is called p. The inner loop of this problem, which computes the tenth power of a number, is similar to the loop in Example 8.1. The outer loop, which sums the powers, is similar to the loop in Example 8.2. Summation cannot accumulate in the AC, since that register is also used during multiplication. Therefore the computed partial sums are stored in SUMZ. Symbolic indices are not used, although they would serve well in this program. Because the number of inner and outer loop cycles are known, addresses can be coded into the program.

```
Locn.   Oper.   Var. Field
        STZ     SUMZ            Clear sum
        AXT     80,1
LØØPI   AXT     10,2
        LDQ     ØNE
LØØPJ   MPY     TABLEZ+80,1     p·number to p(AC)
TESTJ   TIX     LØØPJ,2,1       j+1 to j and test
        XCA
SUMI    ADD     SUMZ            Sum+p to Sum
        STØ     SUMZ
TESTI   TIX     LØØPI,1,1
        HTR

ØNE     DEC     1
SUMZ
TABLEZ  BSS     80
```

Example 8.6 Given a list of 1000 integers, sort them into negative and positive integers, and compute the sums of the two lists.

The numbers a_i start at location LIST. Let the negative integers be placed in the block starting at NEGLST and the positive integers be placed in the block starting at PØSLST. Let N_j be the jth location in the NEGLST block; let P_k be the kth location in the PØSLST block. Place the sums in NEGSUM and PØSSUM; let S_n and S_p be the negative and positive sums. A flowchart appears in Fig. 8.4 in the book.

Three index registers are to be used. XR1 is the loop control XR (index i); XR2 and XR4 will be used as pointers to designate where in NEGLST and PØSLST the integers are to be stored (indices j and k). Since index registers modify by decrementing rather than by incrementing, the index register contents must be decreased each time entries are made into the tables. Thus, to store a positive number this sequence is used:

```
Oper.   Var. Field
STØ     PØSLST+1000,K
TXI     MØD,K,-1
```

The first instruction stores the number, and the second modifies XR4 (k) so that the next time a positive number is stored, it is placed in the next word in the list. The TXI instruction transfers control to MØD, where the TIX instruction controls the outer loop. In this manner, XR4 indicates at any time the next available location for

storage of an integer. The same is true for XR2. These index registers point to the locations in memory; hence they are called pointers when so used.

```
Locn.    Oper.    Var. Field
I        SET      1                 Set index symbols
J        SET      2
K        SET      4
         STZ      NEGSUM
         STZ      PØSSUM
         AXT      1000,I            Set main counter
         AXT      1000,J            Set j,k counters
         AXT      1000,K
FETCH    CLA      LIST+1000,I       Fetch a number
         TPL      PLUS              Test sign
MINUS    STØ      NEGLST+1000,J     Store in list
         ADD      NEGSUM            Sn + no. to Sn
         STØ      NEGSUM
         TXI      MØD,J,-1          Modify j
PLUS     STØ      PØSLST+1000,K     Store in list
         ADD      PØSSUM            Sp + no. to Sp
         STØ      PØSSUM
         TXI      *+1,K,-1
MØD      TIX      FETCH,I,1
         HTR

NEGSUM
PØSSUM
LIST     BSS      1000
NEGLST   BSS      1000
PØSLST   BSS      1000
```

The number of entries in NEGLST and PØSLST are available in the index registers after the program is finished, in complemented form. For example, XR2 contains the difference between 1000 and the number of negative numbers.

Two methods of loop control have been used several places in the examples in this chapter. One method uses the TXI-TXH pair of instructions; the other uses the TIX instruction. In the examples here, both approaches accomplish the same functions: (1) index modification by constant amounts and (2) testing for the end of the looping process. In both cases, also, the index runs from an initial value, n, down to a final value, usually 1, in steps of 1: n, n - 1, n - 2, ..., 3, 2, 1. If, however, it is desired that the index run from one limit down to another, where the lower limit is not equal to or less than the decrementing amount (the step), the TIX instruction

cannot be used; the TXI-TXH pair is required. Assume, for example, that the index is required to run as follows: 100, 98, 96, ..., 44, 42, 40. These instructions can be used:

```
        AXT   100,1
         .      .
         .      .
        TXI   *+1,1,-2
        TXH   LØØP,1,39
```

As soon as the index decreases to 38, the loop will stop. The TIX instruction cannot provide this flexibility.

(R)----------(153.1 - 153.3)----------

THE TIME-SPACE BALANCE

This material is discussed in the book. The coding example for the 7090 is the following.

```
Locn.   Oper.   Var. Field
        CLA     ZERØ
        AXT     60,1
LØØP    ADD     NUMBRS+60,1     Add 3 numbers
        ADD     NUMBRS+61,1
        ADD     NUMBRS+62,1
        TXI     *+1,1,-3        Modify index by 3
        TXH     LØØP,1,0
        STØ     SUM
```

(S)----------(At 154.7)----------

NONLOOP INDEX REGISTER USAGE

When index registers are used as counters, their contents generally start at 0 or 1 and are increased regularly, usually by steps of 1. When used as pointers, however, their contents usually decrease so that they point to successive memory words at increasing memory addresses; this is due to the decrementing nature of these registers.

(R)----------(155.0 - 156.3)----------

TABLE-LOOK-AT

PLACE ADDRESS IN INDEX (PAX 0,T) (+0734); 1 cycle. The $C(AC)_{21-35}$ replaces the contents of the specified index register. The C(AC) is unchanged.

No address is involved, although a tag is necessary. To indicate that the integer in the variable field is a tag rather than an address, a comma must precede the tag. This is required by the convention in Section 8.1 in the book. An example of one instruction is the following:

```
PAX   0,2
```

Another useful instruction is the following.

PLACE INDEX IN ADDRESS (PXA 0,T) (+0754); 1 cycle. The contents of the specified index register replaces the $C(AC)_{21-35}$ and the remainder of the AC is cleared. (If the tag is 0, the AC is cleared completely.)

This instruction is approximately the opposite of PAX, except that the rest of the AC (bits S,Q,P,1-20) is cleared. Thus the instruction

```
PXA   0,0
```

may be used to clear the AC.

Example 8.7 Given 2000 positive integers, all less than 100 in value, determine a histogram as follows: compute the distribution of integers in ten equal intervals: 0 - 9, 10 - 19, ..., and 90 - 99. The integers are located in the block starting at LIST.

The interval to which each integer belongs can most readily be found by dividing it by 10, discarding the remainder. If the quotient is q, the integer lies in the (q+1)th interval.

The value q is then used to set an index register and thereby to select one of 10 counters, which is then incremented by 1. These counters count the number of integers in the 10 intervals. A flowchart is drawn in Fig. 8.5 in the book. The 10 counts are $n_1, n_2, \ldots, n_{10}$.

```
Locn.    Oper.    Var. Field
         AXT      10,2              Clear block
         STZ      CTABLE+10,2
         TIX      *-1,2,1
         AXT      2000,1
NEWØNE   LDQ      LIST+2000,1       Fetch an integer
         PXA      0,0               Clear AC
         DVP      TEN
         XCA                        Int./10 to AC
         PAX      0,2                  ... to XR2
         CLA      CTABLE+9,2        Fetch proper counter
         ADD      ØNE               n_q + 1 to n_q
         STØ      CTABLE+9,2
         TIX      NEWØNE,1,1
         HTR
```

(Cont'd.)

```
Locn.   Oper.   Var. Field

ØNE     DEC     1
TEN     DEC     10
CTABLE  BSS     10              The 10 counters
LIST    BSS     2000
```

(R)----------(156.7 - 157.8)----------

PUSH-DOWN LISTS

Example 8.8 The block of 1000 words at PDLIST contains a set of k items (numbers), the first of which is located in PDLIST, the head of the list. The items are stored in successive memory words; the number (1000-k) is stored in XR4.

The following actions occur:

(1) An item is added to the bottom of the list, and the C(XR4) is decreased by 1 to reflect the addition.

(2) An item is removed from the top of the list, the list is moved up in its entirety one position, and the C(XR4) is increased by 1 to reflect the removal.

The sequence of these actions is unknown; they may occur in any sequence, e.g., (1), (2), (2), (2), (1), ... This situation may be likened to a purchase order processing scheme, where orders are handled in the order received; new orders go at the bottom while the top order is processed first. This approach is sometimes called "first-in-first-out" sequencing.

The number k may be zero, but we assume it is never negative; that is, no more items are removed than are added, if we start with an empty list.

Two subprograms (or routines) are required; one to add an item and one to remove an item. These must be coded independently, since that is how they are used. Let the item to be removed be stored in ØLD.

The add-item routine (ADDITM) consists of these steps:

1. The C(NEW) must be placed at the bottom of the list; the address of the first free location is given as PDLIST+1000,4 since the C(XR4) is -k, the number of items in the list.
2. The C(XR4) must be decreased by 1.

The flowchart in Fig. 8.6a in the book shows the ADDITM routine. The i^{th} word in PDLIST is L_i.

The remove-item routine (REMITM) consists of these steps:

1. The C(PDLIST), the first item, must be stored in ØLD.
2. The k-1 remaining items must be moved up one word each.
3. The C(XR4) must be increased by 1.

To accomplish the movement of k-1 items, a loop is established. The TXI-TXH pair of instructions is used because the index runs from 1000 down to 1002 - k; to effect this, the decrement in the TXH instruction must be 1 less, 1001 - k. After XR4 is increased by 1 so that it contains this number, its contents are placed in the TXH decrement by the SXD instruction.

```
Locn.    Oper.    Var. Field
ADDITM   CLA      NEW              Put item at bottom
         STØ      PDLIST+1000,4      of list
         TXI      *+1,4,-1

REMITM   CLA      PDLIST           Put first item in ØLD
         STØ      ØLD
         TXI      *+1,4,1          XR4:  1001 - k
         SXD      TEST,4
         AXT      1000,1           Move k - 1 items
MØVE     CLA      PDLIST+1001,1      up 1 position
         STØ      PDLIST+1000,1
         TXI      *+1,1,-1
TEST     TXH      MØVE,1,**        (1001 - k)
```

(M)----------(159.4 - 161.5)----------

(The following comment applies to 159.4 - 161.5.)
This correspondence exists:

IBM 7090	DELTA 63
CLA	LØAD

(R)----------(162.4 - 164.6)----------

USING INDIRECT ADDRESSING

The following pseudo-operation is also used to set aside blocks of storage in memory. Under certain conditions, it is more useful than BSS.

BES (Block ending with symbol). A block of words of the size indicated in the variable field of this pseudo-operation is set aside for later use at the point in the program at which this card occurs. The location associated with the symbol appearing in the location field is the first address after the block. Thus, in the following

```
LIST    BES    200
```

if the last location used before the block was 00300, then 00301 through 00610, 3108 (200) locations long, are set aside for the block, and location 00611 is associated with LIST.

Example 8.9 Given five blocks of numbers and a sorted list of the starting addresses of the blocks, write a program to process the blocks in the indicated sequence. The program is to be written so that each number to be processed is loaded into the accumulator and processed in some manner. (This processing is not of interest here and is therefore not coded.)

Much detail and analysis of this problem is given in the book. Note, however, that the addresses in the SØRTED block are the BES type. The coding in FAP follows.

```
Locn.   Oper.    Var. Field

        AXT      5,2              Set i index
START   LXA      BLSIZE,1         Set j index
        CLA*     SØRTED+5,2

        (Processing routine here)

        TIX      START+1,1,1
        CAL      START
        ADD      ØNE
        SLW      START
        TIX      START,2,1
        HTR
```

Chapter 9
SEQUENCING IN MEMORY

(M)----------(168.2 - 168.6)----------

(The following comment applies to 168.2 - 168.6.)
The 7090 compare instruction is abbreviated as CAS, and bits S and 1-35 of the accumulator are involved.

(R)----------(168.7 - 169.8)----------

Example 9.1 Given a set of n numbers (a_1,a_2,...), determine the largest number. The numbers, of which there are no more than 1000, are stored in the block starting at SET; thelargest number is to be stored in BIG.

A flowchart appears in Fig. 9.1 in the book. Initially, the first number is placed in the AC. Then, the C(AC) is successively compared with a_2, a_3, ..., until a larger number is found. When a larger number is found, it is placed in the AC and the process repeats, the C(AC) being compared to all numbers next in sequence. After all numbers are tested, the AC contains the largest number in SET; it is stored in BIG. The flowchart indicates that the index is initially set to 1, which causes a_1 to be compared to itself. Although unnecessary, this is done for uniformity with loops in other programs.

Locn.	Oper.	Var. Field	
	CLA	XSET	Set instructioss
	ADD	N	
	STA	FETCH	
	STA	FETCH+3	
	CLA	SET	a_1 to AC
	LXA	N,1	
FETCH	CAS	**,1	(SET+n)
	TRA	NEXT	C(AC) greater
	TRA	NEXT	C(AC) equal
	CLA	**,1	C(AC) less; a_1 to AC
NEXT	TIX	FETCH,1,1	
	STØ	BIG	
	HTR		

(Cont'd.)

```
Locn.   Oper.    Var. Field

BIG
N
XSET             SET
SET     BSS      1000
```

(S)----------(At 170.1)----------

The 7090 computer has a number of test instructions that are of the skip variety; each tests a condition within the computer and causes the computer to continue in sequence or skip one instruction, depending on the outcome of the test. Two such instructions are the following.

STORAGE ZERO TEST (ZET Y) (+0520); 2 cycles. If the $C(Y)_{1-35}$ is zero, the computer skips the next instruction and proceeds from there. If it is not zero, the computer takes the next instruction in sequence. The C(Y) is unchanged.

STORE NOT ZERO TEST (NZT Y) (-0520); 2 cycles. If the $C(Y)_{1-35}$ is not zero, the computer skips the next instruction and proceeds from there. If it is zero, the computer takes the next instruction in sequence. The C(Y) is unchanged.

(R)----------(170.5 - 171.7)----------

FIXED BRANCHING

CLEAR AND SUBTRACT (CLS Y) (+0502); 2 cycles. The negative of the C(Y) replaces the $C(AC)_{S,1-35}$. Positions P and Q of the AC are set to zero. The C(Y) is unchanged.

Example 9.2 Code a branch point that sends control alternately to operations P and Q.

There are a number of ways to code an alternating branch point. Using instructions already studied, we shall base the decision on the sign of a number in memory. The sign is reversed every time control is sent to the branch point, after which the sign is tested with a TPL instruction. Since the sign is alternately plus and minus, the branching occurs. A flowchart appears in Fig. 9.2 in the book.

```
Locn.   Oper.    Var. Field
        CLS      SIGNWD          Change sign of SIGNWD
        STØ      SIGNWD
        TPL      ØPERP           Tra to oper. P
        TRA      ØPERQ           Tra to oper. Q
        ...      .....
SIGNWD  DEC      1
```

If the initial sign of SIGNWD is plus (as here), control will first go to Q, since the sign is changed before the test. The C(SIGNWD) will be alternately +1 and -1.

Example 9.3 Code a branch point that sends control to one of four operations in cyclic sequence: P, Q, R, S, P, Q, ...

A tagged TRA instruction is used to produce a cycling transfer of control (with XR1). The effective address of this instruction is BRANCH, BRANCH+1, ..., BRANCH+3, in sequence, sending control, respectively, to ØPERP, ØPERQ, ..., ØPERS. Control then goes to operations P, Q, R, and S. To produce these effective addresses, XR1 is set to 3, 2, 1, 4, 3, 2, 1, ... Decreasing the C(XR1) by 1 is no problem, but following 1 with 4 requires special treatment. Initially, early in the program, the C(XR1) is set to 4 and is then decreased by 1 just prior to each transfer at the branch point. A TXH instruction tests for the case when the (XR1) falls to 0, at which time it is reset to 4 before the transfer.

A flowchart appears in Fig. 9.3 in the book.

```
Locn.   Oper.   Var. Field
        AXT     4,1             (Set early in program)
        ...     ...
        TXI     *+1,1,-1
        TXH     JUMPS,1,0       Test index; tra if 1
        AXT     4,1               or greater; reset if 0
JUMPS   TRA     BRANCH+4,1
BRANCH  TRA     ØPERP           C(XR1) = 4
        TRA     ØPERQ                    3
        TRA     ØPERR                    2
        TRA     ØPERS                    1
```

The program can be made slightly simpler with the use of indirect addressing, flagging the first TRA instruction (at JUMPS) and removing the operations from the next four instructions. The extension of this technique to any number of paths is straightforward.

(R)----------(172.7 - 174.3)----------

VARIABLE BRANCHING

Example 9.4 Code a branch point that sends control to one of five locations (X1, X2, ..., X5) if the C(DIGIT) = 1, 2, ..., 5, respectively.

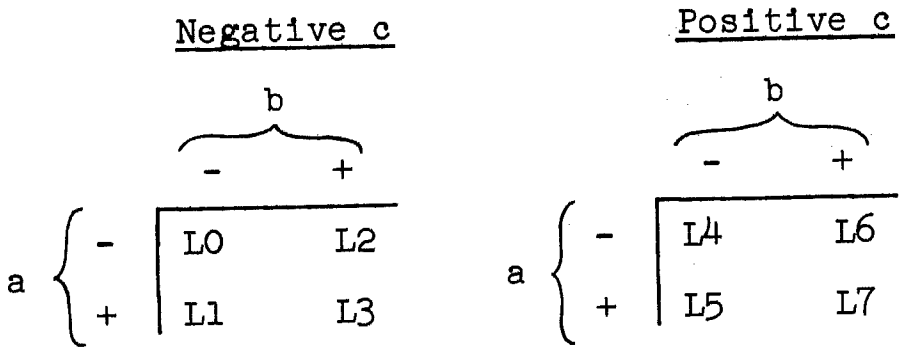
This problem is similar to the four-way branching problem of Example 9.3, except that the branching depends on the data. (Presumably, the C(DIGIT) is determined

during the program execution and thus depends on data.) All that is necessary is that the C(DIGIT) be loaded into an index register and that a jump be effected with a tagged TRA instruction.

```
Locn.    Oper.    Var. Field
         LXA      DIGIT,1
         TRA*     JTABLE+5,1
JTABLE             X5
                   X4
                   X3
                   X2
                   X1
```

If the C(DIGIT) = j, then j is placed in XR1 nd the effective address of the TRA instruction is JTABLE+5-j, so that a transfer to Xj occurs, as required. A check on the C(DIGIT) might be necessary to avoid an erroneous transfer.

Example 9.5 Code a branch point that sends control to one of eight locations (L0, L1, ..., L7), depending upon the signs of three variables, a, b, and c, in the following manner:



Two distinct approaches are possible here. In one, the signs of the three variables are tested in sequence; an eight-way branch results. A flowchart is given in Fig. 9.4a in the book. In the other, a digit j is built up for an eventual jump to location Lj. Weights can be assigned to the algebraic signs of the variables:

a: "-": 0; "+": 1
b: "-": 0; "+": 2
c: "-": 0; "+": 4

As the signs are checked, the appropriate weights are summed to form the proper value of j. Finally, j is used to modify a transfer address as in the last example.

```
Locn.    Oper.     Var. Field

         ...       ...
         AXT       0,1              0 to j
         CLA       A                Test a
         TMI       *+2
         TXI       *+1,1,+1         j+1 to j
         CLA       B                Test b
         TMI       *+2
         TXI       *+1,1,+2         j+2 to j
         CLA       C                Test c
         TMI       *+2
         TXI       *+1,1,+4         j+4 to j
         TRA*      BRANCH+7,1
BRANCH             L7
                   L6
                   L5
                   L4
                   L3
                   L2
                   L1
                   L0
```

(R)----------(176.4 - 178.7)----------

Example 9.6 Five coding sequences are available for use in a computer problem: A, B, C, D, and E. They are to be used in sequence three ways, depending upon which of three conditions is met by data being processed:

Case 1: use A, B, and D.
Case 2: use A, C, D, and E.
Case 3: use B, C, and D.

Two approaches are analyzed and flowcharted in the book. The first method uses a pair of instructions to effect the switching as follows:

```
         Oper.     Var. Field

         CLA       INDA             Skip A if zero
         TZE       SKIPA
```

The second method uses the following types of instructions before each box:

```
TRA     *+3,1
TRA     *+2             Here if C(XR1) = 2
TRA     SKIPA           Here if C(XR1) = 1
```

Chapter 10
SUBROUTINES

(R)----------(182.10 - 183.4)----------

Example 10.1 Write a program to evaluate e:

$$e = a^2 + b^2 + c^2 + d^2$$

The squares, as they are computed, must be stored temporarily. A three-word block (TEMP) is set aside for this purpose. As in earlier programs, it is assumed all products are small enough to remain in the MQ with fixed-point multiplication.

```
Locn.   Oper.   Var. Field
SUMSQ   LDQ     A           Square a
        MPY     A
        STQ     TEMP
        LDQ     B           Square b
        MPY     B
        STQ     TEMP+1
        LDQ     C           Square c
        MPY     C
        STQ     TEMP+2
        LDQ     D           Square d
        MPY     D
        XCA
        ADD     TEMP        Sum the squares
        ADD     TEMP+1
        ADD     TEMP+2
        STØ     E
        ...
TEMP    BSS     3
```

(R)----------(185.4 - 185.9)----------

Example 10.2 Write three open subroutines, each of which computes the sum of a set of numbers; the sets are 50, 100, and 250 in size.

The sets begin at locations LIST1, LIST2, and LIST3; place their sums at SUM1, SUM2, and SUM3, respectively. Assume that the latter three locations are cleared.

```
Oper.    Var. Field

AXT      50,1
PXA      0,0
ADD      LIST1+50,1
TIX      *-1,1,1
STØ      SUM1

AXT      100,1
PXA      0,0
ADD      LIST2+100,1
TIX      *-1,1,1
STØ      SUM2

AXT      250,1
PXA      0,0
ADD      LIST3+250,1
TIX      *-1,1,1
STØ      SUM3
```

(S)----------(At 186.4)----------

MACRO-INSTRUCTIONS

In BE-FAP, the coding sequence defining the basic structure of an open subroutine is delimited by the pseudo-operations MACRØ and END.* There is no ambiguity between this END and the last card of a FAP program because each MACRØ pseudo-operation is matched by the assembler to an END pseudo-operation.

(R)----------(186.5 - 186.7)----------

The routine of Example 10.2 would be written as follows as a macro-definition:

```
Locn.    Oper.    Var. Field

SUMBLK   MACRØ    A,B,C
         AXT      B,1
         PXA      0,0          Clear AC
         ADD      A+B,1
         TIX      *-1,1,1
         STØ      C
         END
```

*BE-FAP is the assembler at Bell Telephone Laboratories for use with the 7090 and 7094 computers; it is widely used, with variations, at other installations of that computer.

(R)----------(188.1 - 188.5)----------

TRANSFER OF CONTROL

A special instruction is available for the purpose of transfering control to a closed subroutine.

TRANSFER AND SET INDEX (TSX Y) (+0074); 2 cycles. The 2's complement of the computer's instruction counter contents is placed in the specified index register. The computer takes its next instruction from location Y.

By storing in an index register the location of the transfer instruction, i.e., the location from where control came, a means is provided to return control to the main program. For example, assume control is to return to the instruction following the TSX instruction, i.e., to READY+1 in the following:

```
Locn.   Oper.    Var. Field
READY   TSX      SUBRTE,4
```

The following instruction, located within the subroutine, effects the return:

```
        TRA      1,4
```

Let L be the location of the TSX instruction, i.e., the address READY. Since the C(XR4) = -L, the effective address of the transfer instruction is 1-(-L), or L+1, as required.

The use of the TSX instruction or other instructions to establish a means for the return of control is termed a linkage. By convention, XR4 is almost always used for this purpose. An example of a linkage that does not use index registers is the following. In the main program, these instructions are used:

```
Locn.   Oper.    Var. Field
        CLA      *          Place address of this
        TRA      SUBRTE       instr. in AC
RETURN  ...      ...        Return here
```

The subroutine coding is started and terminated by these instructions:

```
SUBRTE  ADD     TWØ             Add 2 to AC to produce
        STA     GØBACK            return location
        ...     ...
        ...     ...
GØBACK  TRA     **
```

The location address in the accumulator must be increased by two to effect a return to RETURN.

(R)----------(189.7 - 190.2)----------

TRANSFER OF INFORMATION

As an example, if a closed subroutine SUMBLK is written to sum a set of numbers, as coded in Example 10.2, one calling sequence might be:

```
Locn.   Oper.   Var. Field

MØVE    TSX     SUMBLK,4
                LIST1
                50
                SUM1
```

As seen here, the information stored in the calling sequence may be of several types: (a) the address for a result or the address of one data word may be given (e.g., SUM1); (b) the starting address of a block of words may be given (e.g., LIST1); (c) the size of a block of words may be given (e.g., 50).

The subroutine has the job of obtaining the information for its use from the calling sequence. The effective address of the first word following the TSX instruction is given by "1,4," so that the following instructions are equivalent:

```
        CLA     1,4
        CLA     MØVE+1
```

These instructions load the address LIST1 into the accumulator. Similarly, the other arguments in the calling sequence can be addressed with "2,4" and "3,4".

The alternate linkage described, which uses no index registers, can also be used with parameters placed in the words immediately following the transfer to the subroutine.

Within the subroutine, the addition of 2, 3, or 4 to the address in the accumulator at the start provides the addresses of the parameters in the main program. Addition of 5 provides the return address.

(R)----------(190.7 - 191.6)----------

Example 10.3 Write the SUMBLK macro-instruction of Section 10.2 as a closed subroutine.

Since all parameters are given in the calling sequence, they must be moved to the body of the subroutine. Three instructions of the form

```
CLA     M,4
```

where M is 1, 2, and 3, load the contents of the address fields of the calling sequence into the AC. STA instructions store these addresses in the proper places in the routine. The heart of the subroutine, which computes the sum of the numbers in the block, is the same in form as the macro-instruction. Because XR1 is used by the subroutine, its contents must be saved. Note that the address at SUB1 is set by adding two parameters from the calling sequence.

```
Locn.   Oper.   Var. Field

SUMBLK  SXA     SAVEX1,1        Save XR1
        CLA     1,4             Fetch list address
        ADD     2,4             Add list size
        STA     SUB1
        CLA     2,4             Fetch list size
        STA     SUB2
        CLA     3,4             Fetch sum address
        STA     SUB3
SUB2    AXT     **,1            (size)
        PXA     0,0             Clear AC
SUB1    ADD     **,1            (list addr + size)
        TIX     *-1,1,1
SUB3    STØ     **              (sum address)
        LXA     SAVEX1,1        Restore XR1
        TRA     4,4

SAVEX1
```

This subroutine is generalized so that any calling sequence in the proper form can call upon it. The following calling sequence will result in the summation of the 100 words at LIST2:

```
        TSX     SUMBLK,4
                LIST2
                100
                SUM2
```

(R)----------(191.7 - 192.4)----------

Example 10.4 Write SUMBLK as a closed subroutine. Since the size of the block, rather than the address of a location containing the size, is given in the calling sequence, indirect addressing is not used to obtain that parameter. The tag on the ADD instruction (see Example 10.3) must be placed in the calling sequence.

The calling sequence:

```
Locn.   Oper.   Var. Field
        TSX     SUMBLK,4
                LIST2+100,1
                100
                SUM2
```

The subroutine (note the alternate method of restoring XR1):

```
SUMBLK  SXA     SAVEX1,1
        CLA     2,4             Fetch list size
        STA     SUB2
SUB2    AXT     **,1            (size)
        PXA     0,0
        ADD*    1,4
        TIX     *-1,1,1
        STØ*    3,4
SAVEX1  AXT     **,1            Restore XR1
        TRA     4,4
```

The flexibility offered by the combined use of indirect addressing and tags on both the direct and indirect addresses is illustrated.

(R)----------(192.6 - 192.8)----------

```
   Locn.   Oper.    Var. Field

   SUMBLK  MACRØ    A,B,C
           TSX      SUMBLK,4
                    A
                    B
                    C
           END
```

The macro-call:

```
           SUMBLK   LIST1,50,SUM1
```

(R)----------(193.0 - 195.3)----------

Example 10.5 Write a closed subroutine for the evaluation of e:

$$e = a^2 + b^2 + c^2 + d^2$$

The calling sequence:

```
   Locn.   Oper.    Var. Field

           TSX      SUMSQ,4
                    A
                    B
                    C
                    D
                    E
```

The subroutine:

```
   SUMSQ   LDQ*     1,4
           MPY*     1,4
           STQ      TEMP
           LDQ*     2,4
           MPY*     2,4
           STQ      TEMP+1
           LDQ*     3,4
           MPY*     3,4
           STQ      TEMP+2
           LDQ*     4,4
           MPY*     4,4
           XCA
```

(Cont'd.)

```
Locn.   Oper.    Var. Field

        ADD      TEMP
        ADD      TEMP+1
        ADD      TEMP+2
        STØ*     5,4
        TRA      6,4

TEMP    BSS      3
```

In this example, the use of indirect addressing does not slow down the subroutine. In Example 10.4 flagging is effective within a loop that cycled n times. In this subroutine no loop is present.

Example 10.6 Write a program to evaluate

$$F = \sqrt{x} + \sqrt{x^2 - y^2} + (x^2 + y^2 + z^2 + u^2)$$

Two subroutines are assumed available for this purpose: SQRØØT, which computes the square root of the C(AC) and leaves the result in the AC, and SUMSQ, as in Example 10.5.

```
Locn.   Oper.    Var. Field

        CLA      X                 x to AC
        TSX      SQRØØT,4
        STØ      TEMP              Store sq. root of x
        LDQ      Y
        MPY      Y
        STQ      TEMP+1            Store y^2
        LDQ      X
        MPY      X
        SCA
        SUB      TEMP+1            Form x^2 - y^2
        TSX      SQRØØT,4
        STØ      TEMP+1
        TSX      SUMSQ,4
                 X
                 Y
                 Z
                 U
                 TEMP+2            (For result)
```

(Cont'd.)

```
Locn.   Oper.    Var. Field

        CLA      TEMP
        ADD      TEMP+1
        ADD      TEMP+2
        STØ      F
        HTR

TEMP    BSS      3
X
Y
Z
U
```

The symbols SQRØØT and SUMSQ must be defined.

In Section 8.4, it was pointed out that indirect addressing may be used with tags on both the direct and indirect addresses. This technique is illustrated by an example.

Example 10.7 The block at LISTA contains 100 numbers whose cube roots are to be computed; the results are to be placed in the block at LISTB. Write a routine to perform the operations, making use of a CBRØØT subroutine.

It is assumed that CBRØØT has two addresses in its calling sequence, the address of the argument (to be cubed) and the address for the result. A loop is established containing the calling sequence. The addresses in the calling sequence are tagged. Within the subroutine, a flagged reference places one of the arguments in LISTA in the AC; because of the tag, all arguments are fetched in sequence.

```
Locn.   Oper.    Var. Field

        AXT      100,1
BEGIN   TSX      CBRØØT,4
                 LISTA+100,1
                 LISTB+100,1
        TIX      BEGIN,1,1
        HTR

LISTA   BSS      100
LISTB   BSS      100
```

Within the subroutine, the argument is placed in the AC and the result is subsequently stored in LISTB by the instructions

```
CLA*    1,4
...
...
STØ     2,4
```

(R)----------(196.4 - 196.5)----------

Oper.	Var. Field	
TZE	5,4	Zero return
TPL	6,4	Positive return
TMI	7,4	Negative return

Chapter 11
INPUT-OUTPUT OPERATIONS

(S)----------(At 206.8)----------

Following are some examples of corrections cards used with 7090 monitors:

	Locn.	Oper.	Var. Field
octal correction card:	237	ØCT	050000211145
decimal correction card:	4420	DEC	22,33,88

As the result of the first card, the octal word 050000211145, which corresponds to the instruction

CLA /Ø/11145,2

is placed at location 00237, overwriting whatever was there previously.

As the result of the second card, the integers 22, 33, and 88 would be placed at locations 04420, 04421, and 04422, respectively. The convention on the use of several fields of a FAP DEC card applies to these correction cards.

(R)----------(208.8 - 209.10)----------

ALPHANUMERIC INFORMATION

The 7090 BCD character codes are given in the accompanying table. Six bits are used for each character. The coding is done by the card reader to put information from cards on tape or in memory. The codes listed apply to characters in memory; in some cases, the codes on magnetic tape differ. The code is generally termed binary-coded-decimal or BCD. For compactness, the codes are generally expressed as 2-digit octal numbers, as in the table. The term Hollerith is used synonomously with BCD.

BCD CHARACTER CODES

Character	BCD code	Character	BCD code	Character	BCD code
0	00	D	24	Q	50
1	01	E	25	R	51
2	02	F	26	$	53
3	03	G	27	*	54
4	04	H	30	(blank)	60
5	05	I	31	/	61
6	06	.	33	S	62
7	07	)	34	T	63
8	10	-	40	U	64
9	11	J	41	V	65
=	13	K	42	W	66
"	14	L	43	X	67
+	20	M	44	Y	70
A	21	N	45	Z	71
B	22	Ø	46	,	73
C	23	P	47	(	74

(R)----------(210.7 - 211.1)----------

PSEUDO-OPERATIONS

The FAP assembler has two pseudo-operations for the generation of alphanumeric information within a symbolic program.

(1) BCI (binary-coded information). The first character in the variable field of this pseudo-operation is a decimal digit n, from 1 to 9. The second character is a comma. The following string of 6n characters, if n is in the range from 1 to 9, is stored by the assembler in the n successive computer words at the point in the program at which this card occurs. If n = 10, the comma must appear in column 12, thus providing 60 columns (columns 13 through 72) for characters; 10 BCD words are then generated. Thus, to store 6 words of BCD information, one writes

Locn.	Oper.	Var. Field
STRING	BCI	6,TØDAY THE DATE IS ØCTØBER 24, 1961.

The symbol STRING is assigned to the first word containing this information, which is stored in STRING through STRING+5. If the count n is insufficient to account for the entire string, only the first 6n characters are stored in n words. If the count is too large, blanks fill up remaining space to a total of n words.

(2) BCD. This pseudo-operation is used in the same manner as BCI, except that the comma is omitted. If 10 BCD words are to be generated, the character 0 must appear in column 12, followed in column 13 by the start of the string.

(M)----------(211.2 - 213.2)----------

(The material in 211.2 - 213.2 consists of a typical approach to monitor input-output subroutine usage. Actual usage varies a great deal among 7090 computers.)

Chapter 12
PROGRAM PLANNING

(R)----------(235.1 - 235.4)----------

MINIMIZING RUNNING TIME

The time for each instruction to be executed, in number of cycles, is indicated in the individual instruction descriptions. The 709 cycle time is 12 microseconds, the 7090 cycle time is 2.18 microseconds, and the 7094 cycle time is 2 microseconds.

There are a few general rules that indicate the cycle times of 7090 instructions. (1) Instructions involving only the arithmetic unit registers (AC, MQ, XRs) require 1 cycle; examples are XCA, PAX, and PXA. In addition, TRA, TMI, TPL, and AXT require 1 cycle. (2) Most instructions require 2 cycles. (2) The CAS instruction and some instructions requiring testing of each bit in the AC and/or memory word require 3 to 4 cycles. (4) All floating-point instructions and fixed-point multiplication and division require approximately 2 to 15 cycles, depending on several factors.*

(R)----------(235.8 - 236.3)----------

MINIMIZING MEMORY SPACE

Example 12.2 Assume that, at location PUT in a routine, the contents of the accumulator is to be stored in BLØCK if the sequence is being used for processing a list of 1000 numbers or more, and is to be stored in LIST otherwise.

The decision on which version to use is made just before its use. If the C(NUMBER) is nonzero, the version containing the following instruction is used:

Locn.	Oper.	Var. Field
	STØ	LIST

*There are some reductions in these figures for the 7094 instructions.

The sequence to set the STØ instruction is the following:

```
         CLA     NUMBER          Test NUMBER...
         TZE     ZERØ            transfer if zero
         CLA     WØRD1
         STA     PUT
         TRA     PAST
ZERØ     CLA     WORD2
         STA     PUT
PAST     ...     ...             The routine
         ...     ...
PUT      STØ     **
         ...     ...
WØRD1            LIST
WORD2            BLØCK
```

(R)----------(240.3 - 241.6)----------

DYNAMIC ALLOCATION

Example 12.3 Assume that the quantities p, q, and r are read initially into memory from data cards, into locations P, Q, and R. Write a program that assigns blocks of words to blocks A, B, C, and D.

(Refer to the book for a description of this problem and an analysis.)

The coding to accomplish the test on the total size of blocks required is the following:

```
Locn.    Oper.    Var. Field

         CLA      P              Form 2p+2q+3r
         ADD      P
         ADD      Q
         ADD      Q
         ADD      R
         ADD      R
         ADD      R
         CAS      TWELVE
         TRA      TØØBIG
         TRA      *+1
         ...      ....
```

(Refer again to the book for further comments.)

```
ASSIGN  CLA     BEGINA
        ADD     ASIZE
        STØ     BEGINB
        ADD     BSIZE
        STØ     BEGINC
        ADD     CSIZE
        STØ     BEGIND
        ...     ....
BEGINA          A
BEGINB          **
BEGINC          **
BEGIND          **
A       BSS     24000
```

(Refer again to the book.)

(R)----------(243.1 - 244.10)----------

SHIFTING

ACCUMULATOR RIGHT SHIFT (ARS Y) (+0771); 2-4 cycles. The $C(AC)_{Q,P,1-35}$ are shifted right Y bit positions. Bits shifted past position 35 are lost. Vacated positions are filled with zeros.

ACCUMULATOR LEFT SHIFT (ALS Y) (+0767); 2-4 cycles. The $C(AC)_{Q,P,1-35}$ are shifted left Y bit positions. Bits shifted past position Q are lost. Vacated positions are filled with zeros.

LOGICAL RIGHT SHIFT (LGR Y) (-0765); 2-7 cycles. The $C(AC)_{Q,P,1-35}$ and the C(MQ), considered as a single register, are shifted right Y bit positions. Bits leaving position 35 of the AC enter the sign of the MQ. The sign of the AC is unchanged. Bits shifted past position 35 of the MQ are lost. Vacated positions are filled with zeros.

LOGICAL LEFT SHIFT (LGL Y) (-0763); 2-7 cycles. The $C(AC)_{Q,P,1-35}$ and the C(MQ), considered as a single register, are shifted left Y bit positions. Bits leaving the sign of the MQ enter bit 35 of the AC. The sign of the AC is unchanged. Bits shifted past the Q position are lost. Vacated positions are filled with zeros.

ROTATE MQ LEFT (RQL Y) (-0773); 2-4 cycles. The C(MQ) are shifted left Y bit positions in an "end-around" fashion. Bits shifted out of the sign reappear in position 35. No bits are lost.

In addition to these shift instructions, there are two others, LONG RIGHT SHIFT and LONG LEFT SHIFT, that are similar to LOGICAL RIGHT and LOGICAL LEFT. In the LONG

shifts, only bits 1-35 of the AC and MQ shift. The sign of the register that bits are shifted into is made to agree with the sign of the other register, while the latter is unchanged.*

Examples of shifting operations follow.
If the C(AC) are (in binary)

```
 QP
-11011000000110111000110101011111000110
```

then execution of the instruction

```
ARS   10
```

changes the C(AC) to

```
 QP
-00000000001101100000011011100011010101
```

If the C(AC) and the C(MQ) are respectively (in binary)

```
 QP
+00000000110101101100000010100100000000

                    -00000100011010010111111111001001001
```

then execution of the instruction

```
LGL   24
```

changes the contents of these registers to

```
 QP
+10100100000001000001000110100101111111

                    -11001001001000000000000000000000000
```

If the C(MQ) are (in octal)

```
+025044210776
```

---

*A long left shift (LLS) instruction with an address of 0 has the effect of moving only the MQ sign to the AC sign. This means may be used in integer division for putting the dividend sign in the AC.

then execution of the instruction

RQL 27

changes the C(MQ) to

-376025044210

Note that the initial "-3" is actually 7.

MASKING

Masking is accomplished by the use of the following logical instructions.

"AND" TO ACCUMULATOR (ANA Y) (-0320); 3 cycles. Corresponding bits in the $AC_{P,1-35}$ and location Y are compared; where both contain a 1 in any position, the bit in the AC is set to 1; where either or both are 0, the bit in the AC is set to 0. The C(Y) is unchanged. The S and Q positions in the AC are set to zero.

"AND" TO STORAGE (ANS Y) (+0320); 4 cycles. Corresponding bits in the $AC_{P,1-35}$ and location Y are compared; where both contain a 1 in any position, the bit in location Y is set to 1; where either or both are 0, the bit in location Y is set to 0. The C(AC) is unchanged.

As an example of the use of the ANA instruction, we assume the following:

	octal	binary
$C(AC)_{P,1-35}$	= 033200577740	= 000011011010000000101111111111100000
C(Y)	= 007777777400	= 000000111111111111111111111100000000

Then, execution of this instruction

ANA Y

changes the $C(AC)_{P,1-35}$ to the following:

003200577440 = 000000011010000000101111111100000000

In this manner, any selected portion of the C(AC) may be retained while other portions are masked out. The ANS instruction operates similarly on a word in memory, using the AC as a mask. The 36 bits in positions P and 1-35 of the accumulator comprise what is frequently called the logical accumulator; the subscript "L" will be used to refer to it. Positions S and 1-35 comprise the arithmetic accumulator, since those bits are involved in arithmetic.

(R)----------(245.3 - 246.9)----------

PACKING AND UNPACKING

Example 12.4 Four positive 9-bit integers are stored in successive words starting at WØRDS. Write a routine to pack them into a single 36-bit word, PACKED. The C(WØRDS) is to be placed in the leftmost 9 bits of PACKED, the C(WØRDS+1) is to be placed in the next 9 bits, and so on.

The AC will be used to accumulate the four numbers; they will be packed there from the right. To accomplish this, each number will be placed in the leftmost portion of the MQ and the AC-MQ double register will then be shifted left, to place the number in the AC. If this is done four times, all four numbers will be packed in the AC; the SLW instruction is used to store the result. A flowchart appears in Fig. 12-1 below; this flowchart is a computer flowchart for the 7090. An analysis of the packing procedure follows the program.

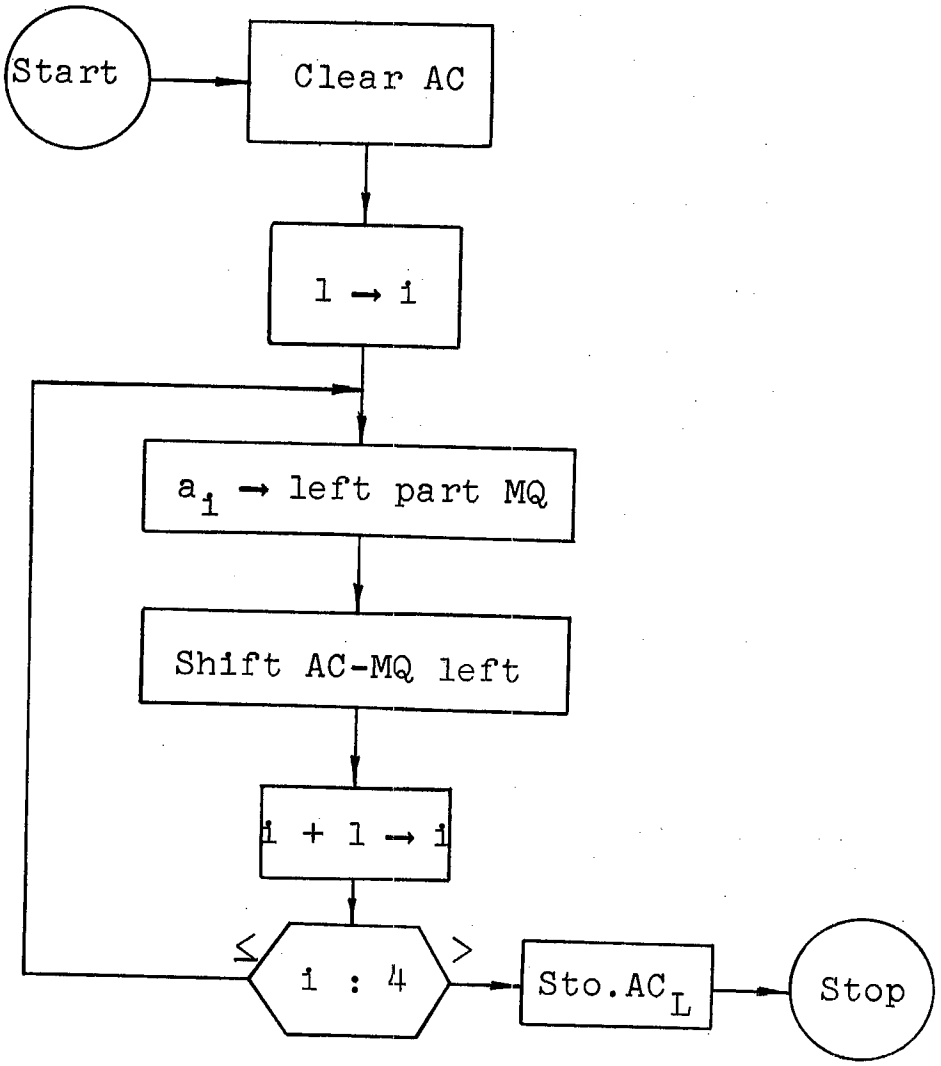


Fig. 12-1. Flowchart for packing routine.

```
Locn.    Oper.    Var. Field
         PXA      0,0              Clear AC
         AXT      4,1
GETWRD   LDQ      WØRDS+4,1        One number to MQ
         RQL      27               Shift to left in MQ
         LGL      9                Shift in AC
         TIX      GETWRD,1,1
         SLW      PACKED
         HTR

PACKED
WØRDS    BSS      4
```

The $C(AC)_L$ and the C(MQ) throughout the execution of this program are listed below; the four loop cycles are shown. The action of the TIX instruction is omitted; effective addresses are listed. The four octal numbers being packed are assumed to be 510, 327, 222, and 106.

Instruction	$C(AC)_L$	C(MQ)
LDQ WØRDS	000000000000	000000000510
RQL 27	000000000000	510000000000
LGL 9	000000000510	000000000000
LDQ WØRDS+1	000000000510	000000000327
RQL 27	000000000510	327000000000
LGL 9	000000510327	000000000000
LDQ WØRDS+2	000000510327	000000000222
RQL 27	000000510327	222000000000
LGL 9	000510327222	000000000000
LDQ WØRDS+3	000510327222	000000000106
RQL 27	000510327222	106000000000
LGL 9	510327222106	000000000000

At the end of this sequence, the numbers are packed in the AC_L.

(R)----------(247.2 - 247.9)----------

Example 12.5 Given a 72-character alphanumeric string, stored in a block from STRING to STRING+11 (6 characters to a word), write a routine to unpack the string, placing each character in the rightmost 6 bits of a word in the block UNPAKD, in the order of appearance in the string.

Each word in the STRING block must be unpacked, its contents being placed in 6 words in UNPAKD. This requires a loop of the form in Example 12.4, except that now unpacking is required. This can be accomplished by loading a word from STRING into the MQ and shifting the AC-MQ pair left 6 bits, putting one character at the right of the AC. After that is stored in UNPAKD and the AC is cleared, another rotation puts the next character into the AC. Surrounding this loop is an outer loop that fetches a new word from STRING each cycle. After that word is unpacked, the next word from STRING is placed in the MQ. A flowchart appears in the book; that flowchart applies to the 7090 if "MQ" is substituted for "MR".

In this program, XR1 and XR2 are used for the outer and inner loop indices i and j, respectively. XR4 is used as a pointer (k), indicating the next available location in UNPAKD for an unpacked character. After each character is stored, the AC must be cleared for the next one. (The indexing discrepancy, described in Section 8.2 of the book, should be recalled.)

```
Locn.    Oper.    Var. Field
K        SET      4
I        SET      1
J        SET      2
         AXT      72,K
         AXT      12,I
LØØP1    LDQ      STRING+12,I
         AXT      6,J
LØØP2    PXA      0,0
         LGL      6
         SLW      UNPAKD+72,K
         TXI      *+1,K,-1          Modify pointer
         TIX      LØØP2,J,1
         TIX      LØØP1,I,1
         HTR

STRING   BSS      12
UNPAKD   BSS      72
```

Chapter 13
NUMERICAL PROBLEMS

(R)----------(254.8 - 255.8)----------

FLOATING-POINT OPERATIONS

Floating-point numbers were described in Chapter 5. In this form, the fraction parts are stored in bits 9-35 and the characteristics are stored in bits 1-8. Characteristics are formed by adding 200$_8$ to the powers of 2 in floating point form. Examples appear in Chapter 4 of the book. If the number is written so that a 1 bit appears in position 9 (at the left of the fraction) the number is normalized.

To incorporate floating-point numbers into a FAP program, the DEC pseudo-operation is used with a decimal point in each number field. The following instruction leads to assembly of the integer 75:

Oper.	Var. Field
DEC	75

This appears as +000000000113 (octal) in the program. The following instruction leads to assembly of 75 as a normalized floating-point number:

```
DEC     75.
```

This appears as 207454000000 (octal) in the program. Alternately, the exponential form described in Section 11.2 of the book may be used:

```
DEC     .75E+2
```

Regardless of the form given, provided a decimal point is present, the number is assembled in normalized form.

In the following floating-point instructions, the operands are treated as floating-point numbers (not necessarily normalized). The results are normalized, except at noted.

FLOATING ADD (FAD Y) (+0300); 6-15 cycles. The C(Y) is added algebraically to the C(AC), and the sum is placed in the AC and the MQ. The less significant half of the sum is placed in the MQ and the more significant half is placed in the AC; the characteristic of the MQ is 33_8 less than the characteristic of the AC. The sign of the sum is placed in both registers. The C(Y) is unchanged.

FLOATING SUBTRACT (FSB Y) (+0302); 6-15 cycles. The C(Y) is subtracted algebraically from the C(AC), and the difference is placed as the sum is in FLOATING ADD. The C(Y) is unchanged

FLOATING MULTIPLY (FMP Y) (+0260); 2-13 cycles. The C(Y) is multiplied algebraically by the C(MQ) and the product is placed in the AC and the MQ; the product is normalized if the original factors were normalized and not necessarily otherwise. The less significant half of the product is placed in the MQ and the more significant half is placed in the AC; the characteristic of the MQ is set as in FLOATING ADD. The sign of the product is placed in both registers. The C(Y) is unchanged.

FLOATING DIVIDE OR HALT (FDH Y) (+0240); 3-13 cycles. The C(AC) is divided algebraically by the C(Y), and the quotient is placed in the MQ. The remainder is placed in the AC. The quotient is normalized if the original operands were normalized and not necessarily otherwise. If the magnitude of the AC fraction is greater than or equal to twice that of the fraction in Y, or if the magnitude of the fraction in Y is zero, division does not occur, the divide-check light is turned on, and the computer stops.

In addition to these floating-point instructions, there are others as follows: instructions to add or subtract the magnitude of a number to or from the C(AC), instructions to add, subtract, and multiply without placing the result in normalized form, and a divide instruction that lets the computer continue in sequence if division cannot occur.

(M)----------(255.8 - 256.4)----------

(The following comments apply to 255.8 - 256.4.)

In floating-point operations, the following correspondence of registers exists:

DELTA 63	IBM 7090
MR	AC
AC	MQ

Thus, the result of the summation is

C(AC) = +212435710424 C(MQ) = +157222000000

Chapter 14
ALGEBRAIC LANGUAGES

(R)----------(267.4 - 267.7)----------

THE COMPILATION PROCESS

A FAP sequence that evaluates r is the following:

```
Oper.    Var. Field
 LDQ      A
 MPY      C              OK if product
 MPY      FØUR             remains in MQ
 STQ      TEMP
 LDQ      B
 MPY      B
 XCA
 SUB      TEMP
 TSX      SQRØØT,4       Sq. root of C(AC);
 SUB      B                result in AC
 XCA
 DVP      A
 DVP      TWØ
 STQ      R
```

Chapter 15
NONNUMERICAL PROBLEMS

(S)----------(At 296.5)----------

NONNUMERICAL CONCEPTS

The following logical instructions are available to perform the operations described in the book.

"OR" TO ACCUMULATOR (ORA Y) (-0501); 2 cycles. Corresponding bits in the logical AC and location Y are compared; where either or both contain a 1 in any position, the bit in the AC is set to 1; where both are 0, the bit in the AC is set to 0. The C(Y) and the S and Q positions in the AC are unchanged.

"OR" TO STORAGE (ORS Y) (-0602); 2 cycles. Corresponding bits in the logical AC and location Y are compared; where either or both contain a 1 in any position the bit in location Y is set to 1; where both are 0, the bit in location Y is set to 0. The C(AC) is unchanged.

COMPLEMENT MAGNITUDE (COM) (+0760,6); 2 cycles.* All 1's are replaced by 0 and all 0's are replaced by 1 in the $C(AC)_{Q,P,1-35}$. The sign of the AC is unchanged.

LOGICAL COMPARE ACCUMULATOR WITH STORAGE (LAS Y) (-0340); 3 cycles. The $C(AC)_{Q,P,1-35}$ and the C(Y) are compared, both considered as unsigned numbers If the C(AC) is greater than the C(Y), the computer executes the next instruction in sequence If the C(AC) equals the C(Y), the computer skips the next instruction and proceeds from there. If the C(AC) is less than the C(Y), the computer skips the next two instructions and proceeds from there.

---

*The operation code of this instruction consists of +0760 in bits S and 1-11 and, an addition, 00006 in bits 21-35; both numbers must be present. This format is true of several 7090 instructions; all of these have either -0760 or +0760 in the leftmost 12 bits. None of these instructions makes any references to memory.

The CAL, SLW, ANA, and ANS instructions introduced in Chapters 7 and 12 and the four instructions above are termed logical instructions. Each refers to the logical accumulator (bits P and 1-35), treating those 36 bits and the 36 bits of a memory word as unsigned numbers.

(R)----------(297.4 - 298.10)----------

ANALYSIS OF SYMBOLIC EXPRESSIONS

Example 15.1 Write a routine that analyzes a symbolic expression, placing each symbol and each operator in sequence in a list, one to a word. Only expressions involving symbols, "+", and "-" need be considered.

The problem is flowcharted in Fig. 15.1 here. After the string is unpacked and stored in LIST, one character at a time is brought into the AC (at the right). The character is examined; if it is an operator ("+" or "-"), then a symbol has just ended; that symbol is to be stored in NEWLST. At that point, the operator can also be stored in NEWLST. If the character examined is a letter, it is shifted within the AC so that it can be stored in SYMBØL, where the symbol, letter by letter, is reconstructed. With each additional letter within the symbol, the amount of the shift decreases (6 bits at a time) so that each letter can be properly placed. This varying shift is accomplished by a tagged ALS instruction; XR4 is used for this purpose and is modified by 6 after each shift. The symbol letters are combined by use of the ØRS instruction, which "or"'s each letter to SYMBØL. When a blank is encountered, the process stops, after the last symbol is stored.

Each character is examined to see if it is (a) a letter, (b) an operator, or (c) a blank. The last two possibilities are checked by testing for (c) BCD 20 ("-") or 40 ("+") and (c) BCD 60 (blank). Any other possibility is treated as a letter. In the program below, it is assumed that unpacking into LIST has been completed, characters stored at the right. A pointer (XR2) is used to indicate the next available space in NEWLST. LIST is assumed to be 100 words long.

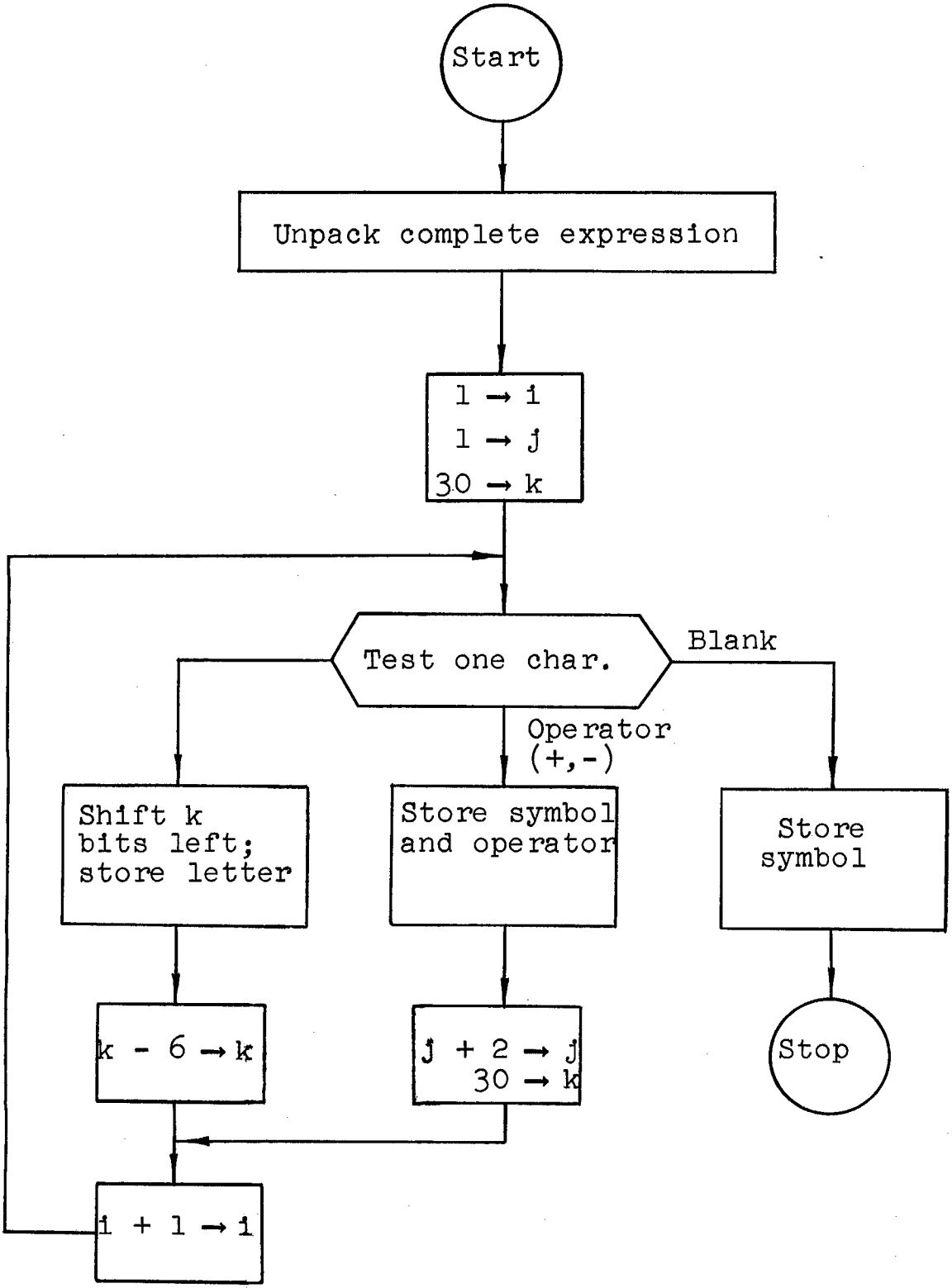


Fig. 15.1 Flowchart of symbolic expression analysis.

```
Locn.   Oper.   Var. Field
        AXT     100,1           Set i
        AXT     100,2           Set j
LØØP    AXT     0,4             Set k
        STZ     SYMBØL          Clear symbol
NEXT    CAL     LIST+100,1      Fetch 1 character
        LAS     BLANKT          Test for blank
        TRA     *+2               no
        TRA     BLANK             yes
        LAS     PLUS            Test for +
        TRA     *+2               no
        TRA     ØPER              yes
        LAS     MINUS           Test for -
        TRA     *+2               no
        TRA     ØPER              yes
        ALS     30,4            Here if letter
        ØRS     SYMBØL
        TXI     *+1,4,6         k-6 to k
        TXI     NEXT,1,-1       i+1 to i; go back
ØPER    SLW     NEWLST+101,2    Store operator
        CAL     SYMBØL
        SLW     NEWLST+100,2    Store symbol
        TXI     *+1,2,-2        j-2 to j
        TXI     LØØP,1,-1       i+1 to i; go back
BLANK   CAL     SYMBØL
        SLW     NEWLST+100,2    Store symbol
        HTR

BLANKT  BCI     1,00000
PLUS    BCI     1,00000+
MINUS   BCI     1,00000-
SYMBØL
LIST    BSS     100
NEWLST  BSS     100
```

At the end of this program, the symbols in NEWLST are left-adjusted, that is, they are stored at the extreme left of the words in NEWLST. If it is desired that they be stored right-adjusted, then the complete symbol must be shifted right before it is stored in NEWLST. Because XR4 begins at 0 and is decreased by 6 for each letter in a symbol, the amount of the final shift needed for right-adjusting is 36 - C(XR4) at the time of storing. For example, if a symbol has 4 characters, the C(XR4) will be 24 at the time the symbol is to be stored; a right shift of 12 bit positions will right-adjust the symbol. Therefore, the insertion of the following instruction just after the two CAL instructions that place the C(SYMBØL) into the AC accomplishes the right-adjusting:

```
        ARS     36,4
```

(M)----------(299.3 - 299.8)----------

(The following comments apply to 299.3 - 299.8.)

PACKING BINARY INFORMATION

To effect the packing, each word is placed in the MQ and the following two instructions are executed as a pair six times (after the AC is cleared):

```
RQL   5
LGL   1
```

If this is done with the 36-bit word given, the right part of the AC will look as follows (in binary):

...0000000000110100

(R)----------(299.8 - 300.2)----------

Example 15.2 The program below takes 6 words of BCD 0's and 1's (in BITS to BITS+5) and packs them into one word, BITPAK.

```
Locn.    Oper.    Var. Field
         AXT      6,2
NEWWRD   LDQ      BITS+6,2        Next word to MQ
         AXT      6,1
NWCHAR   RQL      5               Next character to AC
         LGL      1
         TIX      NWCHAR,1,1
         TIX      NEWWRD,2,1
         SLW      BITPAK
         HTR
```

(R)----------(301.2 - 301.7)----------

CODING ALPHANUMERIC INFORMATION

Example 15.3 The following program codes the C(TERM) and puts the code in CØDTRM. If the C(TERM) is not found in the list, control will pass to ERRØR.

```
Locn.   Oper.    Var. Field
        AXT      160,1            Table has 160 entries
        CAL      TERM
AGAIN   LAS      TABLE+150,1      Compare with i^th entry
        TRA      *+2
        TRA      FØUND            Found term
        TIX      AGAIN,1,1
        TRA      ERRØR            Tra if not in table
FØUND   CAL      CØDTBL+160,1     Fetch i^th code
        SLW      CØDTRM
        ...      .....
TABLE   BCI      1,TRIANG
        BCI      1,SQUARE
        BCI      1,RECTAN
        BCI      1,PARALL
        ...      .....
CØDTBL  ØCT      20
        ØCT      21
        ØCT      23
        ØCT      30
        ...      .....
```

Here, the code for "TRIANGLE" is 20, the code for "SQUARE" is 21, etc.

(S)----------(At 302.1)----------

NONNUMERICAL CONCEPTS

The following logical instruction is useful in nonnumerical problems

EXCLUSIVE "OR" TO ACCUMULATOR (ERA Y) (+0322); 3 cycles. Each bit in Y is matched with the corresponding bit in the logical accumulator. Where corresponding bits match, a 0 replaces the bit in the accumulator; where corresponding bits do not match, a 1 replaces the bit in the accumulator. The C(Y) is unchanged.

For example, if

the $C(AC)_L$ = 011100001010000111111100001100001000

the (Y) = 000000011111000011101010110001111000

then execution of the ERA instruction places this result in the AC:

011100010101000100010110111101110000

This instruction may thus be used to identify the matching bits in two words.

The following instruction tests the status of the P-bit.

P-BIT TEST (PBT) (-0760,1); 2 cycles. If the $C(AC)_P$ is 1, the computer skips the next instruction and proceeds from there. If the $C(AC)_P$ is 0, the computer continues in sequence.

The 7090 computer has a special 36-bit register, the sense indicator (SI) register. The following instructions treat the register as switches which may be logically treated and tested individually or in groups.

LOAD INDICATORS (LDI Y) (+0441); 2 cycles. The C(Y) replaces the C(SI). The C(Y) is unchanged.

STORE INDICATORS (STI Y) (+0604); 2 cycles. The C(SI) replaces the C(Y). The C(SI) is unchanged.

PLACE ACCUMULATOR IN INDICATORS (PAI) (+0044); 1 cycle. The $C(AC)_L$ replaces the C(SI). The C(AC) is unchanged.

ON TEST FOR INDICATORS (ONT Y) (+0446); 4 cycles. For each bit in the C(Y) that is 1, the corresponding bit in the SI is examined. If all the examined positions in the SI contain a 1, the computer skips the next instruction and proceeds from there. If any of the examined positions does not contain 1, the computer takes the next instruction. The C(Y) and the C(SI) are unchanged.

Another way of stating the operation of ONT is to say that, considering the C(Y) and C(SI) as ordered 36-bit sets, the computer skips the next instruction if and only if the C(SI) covers the C(Y), i.e., if the SI has 1's wherever Y has 1's. Thus, if

the C(SI) = 000001110100000001111000101000000000

then a skip occurs in cases (a) and (b), but not (c):

case (a): C(Y) = 000001110000000000000000101000000000

case (b): C(Y) = 000000010000000001111000000000000000

case (c): C(Y) = 110001110000000001000000001000000000

These and the other SI instructions have many applications, of which the following is but a simple example:

Generalize the PBT instruction to permit a test for 1 in any bit of the logical accumulator; let BITWRD contain a 1 in the desired test position and 0 elsewhere. The desired sequence:

Oper.	Var. Field	
PAI		C(AC) to SI
ØNT	BITWRD	Test word

By using other bit patterns in BITWRD, any set of 1's can be tested for.

Another SI instruction of interest is the following.

INVERT INDICATORS FROM ACCUMULATOR (IIA) (+0041); 1 cycle. Each bit of the logical accumulator is matched with the corresponding bit of the SI. Where a bit in the AC is 1, the contents of that position in the SI is unchanged. The C(AC) is unchanged.

The effect of the IIA instruction is to place the exclusive or of the C(AC) and the C(SI) into the SI, just as the ERA instruction places the exclusive or of the C(Y) and C(AC) into the AC.

(R)----------(307.7 - 311.7)----------

NIM

Example 15.4 Write a program to make a Nim move; if the position presented is even, make a "random" move by removing one coin from the first nonzero group; if the position presented is odd, make a move to create an even position.

A large range of positions will be accepted; up to 35 bits may be present in each count (i.e., 235 - 1 is the maximum count), and up to 1000 groups of coins may be present. While these limits are extraordinarily high, coding this case is no more difficult than coding a 5-group game with count limits of 20.

The counts are present in the block starting at CØUNTS; the number of groups (n) is located in GRPNUM. The counts are stored internally in binary form (since the 7090 is a binary computer), which makes a good deal of the coding simple. CØLUMN is used to indicate the columns requiring change; each bit position corresponds to one column, and 1 indicates a change. LEFCØL is used to indicate the leftmost column requiring change.

The program is coded in several stages. In the first stage, steps 1 and 2 are coded. To determine which columns have an odd number of 1's, the ERA instruction is very useful.

Consider one column, say position 35, in each word containing a count. Imagine that the first word is combined with the cleared AC by the ERA instruction. If the word contains a 1 in bit 35, that position in the AC becomes 1, otherwise it remains 0. In fact, as a series of words is successively combined with the AC (which contains the result of all previous combinations), that bit will remain 0 until the first 1 (in bit 35) occurs in a word. Then, as more words are combined with the AC, that bit remains 1 until another 1 occurs in a word. In summary, the number

in bit 35 in the AC indicates, at all times, whether an odd number (1) or an even number (0) of 1's has occurred in that position. The same reasoning applies to all bit positions.

```
Locn.   Oper.    Var. Field
        CLA      XCØUNT          Set address
        ADD      GRPNUM
        STA      GET             (Other addresses also
        LXA      GRPNUM,1          set here)
        PXA      0,0             Clear AC
GET     ERA      **,1            (CØUNTS+n)
        TIX      GET,1,1
        SLW      CØLUMN
        ...
XCØUNT           CØUNTS
```

At the end of the first stage, the columns requiring change are indicated by 1's in CØLUMN. The second stage consists of the coding for step 3 and contains two parts -- 3a: determination of the leftmost column to be changed, and 3b: determination of the first row with a 1 in that column. To code 3a, the C(CØLUMN) are placed in the AC and shifted left one bit position at a time until the P-bit becomes 1. XR1 is used for loop control and when the P-bit is 1, XR1 "points" to the leftmost column. For example, if the leftmost 1 was in bit 20, that bit moves to position P after 20 left shifts and the C(XR1) then decreases to 15 from its original 35. The proper word from a table of single bits, BITS, can then be selected by XR1 and stored in LEFCØL. Part 3a is coded below; 3b is coded later.

Provision is made here for the even-position possibility (Fig. 15.4 in the book); if the loop is cycled 35 times and position P is never 1, then no 1's are present in CØLUMN and no columns require change.

```
Locn.   Oper.    Var. Field
        AXT      35,1
        CAL      CØLUMN
SHIFT   ALS      1              Shift left 1 bit position
        PBT                     Test P-bit
        TRA      *+2            Go on if 0
        TRA      LEFTC          Tra when 1
        TIX      SHIFT,1,1
        TRA      EVEN           Tra if no 1's
LEFTC   CAL      BITS+35,1      Fetch proper bit
        SLW      LEFCØL
        ...      ...
BITS    ØCT      200000000000   1 in bit 1
        ØCT      100000000000   1 in bit 2
        ØCT      040000000000   1 in bit 3
        ...      ...
```

To code 3b, each row count is checked to see if it has a 1 in the bit identified in LEFCØL. This is done by loading each count into the AC, masking with the C(LEFCØL) and transfering out on a nonzero AC. If the AC is nonzero, a 1 must be present in the count just tested in the proper position. Now, XR1 is used to point to the count to be modified.

```
Locn.   Oper.    Var. Field
        LXA      GRPNUM,1
GETBIT  CAL      **,1           (Must be set as in first
        ANA      LEFCØL           stage) (COUNTS+n)
        TNZ      FØUND          Tra when count with bit
        TIX      GETBIT,1,1       found
FØUND   ...      ...            XR1 still points to count
                                  to be modified.
```

In the third stage of the program, step 4 is coded. Here, the number of coins to be removed from the selected row must be determined. Actually, what is required is the new count; the actual number of coins removed need not be computed. The sense indicators are useful here; the IIA instruction performs precisely the required operation: inverting a selected set of bits within a word.

```
Locn.   Oper.    Var. Field

FØUND   LDI      **,1          (Must be set...(CØUNTS+n))
        CAL      CØLUMN        Cols to be changed to AC
        IIA
        STI      **,1          (CØUNTS+n)
        TRA      PRINT
```

PRINT is a routine that prints the counts after the program has calculated the move. If the computer is to play against a man, it must inform him of its move. Somehow, then, the man must inform the computer of his move. This can be done by supplying new data cards each time or by entering the move into the console keys. In any event, the modified count is placed in the CØUNTS block and control goes to the start of the program.

One other routine is needed; in the event that the program finds an even position, it is to make a "random" move: remove one coin from the first nonzero group.

```
Locn.   Oper.    Var. Field

EVEN    LXA      GRPNUM,1
        CLA      **,1          (CØUNTS+n)
        TNZ      EVEN1
        TIX      EVEN+1,1,1
        TRA      ALLDØN        Tra if all are zero
EVEN1   SUB      ØNE
        STØ      **,1          (CØUNTS+n)
        TRA      PRINT
        ...
ØNE     DEC      1
```

ALLDØN is a routine that prints an appropriate comment.

(R)----------(316.1 - 317.3)----------

OTHER TYPES OF LISTS

In order to place three subfields into a word, the address, tag, and decrement fields of an instruction can be used, provided the desired subfields have 15, 3, and 15 bits, respectively. If the operation field is left blank, zeros are assembled into the prefix, i.e., bits S, 1, and 2; alternately the pseudo-operation PZE (plus zero) may be used for the same purpose. (There are other, similar pseudo-operations that cause the 7 other possible prefixes to be assembled; these are PØN (plus one), PTW (plus two), PTH (plus three), MØN (minus one), etc.)

Thus the following correspondences exist within FAP:

Machine word	Symbolic instruction
0 02215 2 11002	PZE LIST,2,WØRD
0 02215 0 11002	LIST,,WORD
-2 11002 1 00277	MTW WØRD,1,NAME

Example 15.5 Refer to the book for a description of the list structure used. Tag fields are used here to hold codes.

Location	Contents	Location	Oper.	Var. Field
01000	0 01004 3 01001	L0	PZE	L1,3,L4
01001	0 01007 1 01002	L1	PZE	L2,1,L7
01002	0 01003 2 01010	L2	PZE	L8,2,L3
01003	0 01012 4 01011	L3	PZE	L9,4,L10
01004	0 01006 1 01005	L4	PZE	L5,1,L6
01005	0 01014 3 01013	L5	PZE	L11,3,L12
01006	0 01016 4 01015	L6	PZE	L13,4,L14
01007	0 00000 7 00115	L7	PZE	ABC,7
01010	0 00000 7 00400	L8	PZE	NAME,7
01011	0 00000 7 42000	L9	PZE	LIST,7
01012	0 00000 7 00221	L10	PZE	WØRD,7
01013	0 00000 7 03300	L11	PZE	X1,7
01014	0 00000 7 03301	L12	PZE	X2,7
01015	0 00000 7 03305	L13	PZE	X3,7
01016	0 00000 7 03310	L14	PZE	X4,7

(R)----------(320.1 - 320.4)----------

INTEGER CONVERSION

Example 15.6 Let a block of integers n_i ($1 \leq n_i \leq N_o$ for $i = 1,2,\ldots,k$) be located starting at BLØCK. It is desired to classify them by assigning a class number c_i to each. To perform this classification, the integer n_i (the argument) is placed in XR2 and the number c_i, located in the $(m_i-1)^{th}$ word from the end of a block at TABLE, is fetched. This number is stored in the n_i^{th} position of a parallel table, CLASS. N_o is assumed to be 100.

```
Locn.   Oper.   Var. Field
        CLA     XBLØCK          Set addresses
        ADD     K
        STA     ARG
        CLA     XCLASS
        ADD     K
        STA     PUT
        LXA     K,1
ARG     CLA     **,1            (BLØCK+k)
        PAX     0,2             Put argument in XR2
        CLA     TABLE+100,2
PUT     STØ     **,1            (CLASS+k)
        TIX     ARG,1,1
        ...
XBLOCK          BLØCK
XCLASS          CLASS
K
```

In this problem, the argument domain is 100 (N_o) in size and a table (TABLE) of that size contains the numbers c_i. These numbers cover the image range. A limitation to this method is the maximum size of N_o permitted, since a table of that size is required.

(R)----------(320.6 - 321.5)----------

PATTERN DETECTION

Example 15.7 Count all appearances of bits "1011" and "0100" in the block of 50 words starting at PATTRN, and place these counts in CNTR1 and CNTR2, respectively. Consider each word to consist of nine 4-bit bites.

The arguments are numbers from 0 to 15 inclusive; these are the decimal numbers represented by the 16 different bites. A transfer of control is made to one address in a 16-word table, depending on the argument. In all but two of these locations, a transfer is made immediately back to the main subroutine. In the other two, transfers are made to counting routines, after the execution of which control is returned to the main routine.

```
Locn.   Oper.   Var. Field
        AXT     50,1
LØØP1   LDQ     PATTRN+50,1     Put a word in MQ
        AXT     9,2
LØØP2   PXA     0,0             Clear AC
        LGL     4               Put 1 bite in AC
        PAX     0,4
        TRA     TABLE+16,4      (TABLE is defined below)
ØUT     TIX     LØØP2,2,1
        TIX     LØØP1,1,1
```

14 of the 16 instructions in the block at TABLE are the following:

```
        TRA     ØUT
```

These correspond to the 14 different bites that are not processed. The other two instructions are

```
        TRA     CØUNT1
        TRA     CØUNT2
```

which occupy the sixth and eleventh words from the end of the block, corresponding to the two patterns to be processed. The routine at CØUNT1 is

```
CØUNT1  CLA     CNTR1           Add 1 to CNTR1
        ADD     ØNE
        STØ     CNTR1
        TRA     ØUT
```

The routine at CØUNT2 is similar.

(S)----------(At 322.0)----------

PLACE COMPLEMENT OF ADDRESS IN INDEX (PAC) (+0737); The 2's complement of the $C(AC)_{21-35}$ replaces the contents of the specified index register. The C(AC) is unchanged.

(R)----------(322.1 - 322.7)----------

Example 15.8 Refer to the book for a description and analysis of this problem. In this program, k, the number of integers, is taken to be 500. The maximum value of the numbers is taken to be 10000. The complement of each integer is placed in XR2 so that the indexing on the STØ instruction places n_i in LIST+n_1.

```
Locn.   Oper.   Var. Field
        AXT     500,1
LØØP1   CLA     BLØCK+500,1     Fetch an integer
        PAC     0,2             Place in XR2, complemented
        STØ     LIST,2          Store in LIST
        TIX     LØØP1,1,1
```

This loop constitutes the integer-ordering process. Note, for example, that the number 45 is stored in LIST+45, 873 is stored in LIST+873, etc. The zero-removal routine is the following:

```
        AXT     10000,1
        AXT     10000,2
LØØP2   CLA     LIST+10000,1    Fetch a number
        TZE     *+3
        STØ     LIST+10000,2    Move if nonzero
        TXI     *+1,2,-1
        TIX     LØØP2,1,1
```

The two XR's act as pointers here; XR1 also serves for loop control. XR1 points to the number being tested; XR2 points to the next available location for its final position. The presence of a zero (indicating a missing value) leaves XR2 unchanged, so that each nonzero number is stored in a successive position in the list. A count of the number of entries in LIST is given at the end by 10000 - C(XR2).

Chapter 16
DATA PROCESSING

(S)----------(At 331.10)----------

CONVERSION

The 7090 computer has three special instructions useful in the conversion of information in one form to another form. These convert instructions treat each 36-bit word as a set of six 6-bit bites, operating upon each bite in sequence. One of the instructions follows.

CONVERT BY ADDITION FROM THE MQ (CAQ Y) (-0114); 2-8 cycles. The MQ is considered to consist of six 6-bit quantities; these may be designated as follows: L1: Bits S and 1-5; L2: bits 6-11; ...; L6: bits 30-35. An effective address Y + L1 is formed and the C(Y+L1) is added to the $AC_{Q,P,1-35}$ (the sign is unchanged). Then the C(MQ) is rotated six positions to the left. As a next step, a new effective address Y' + L2 is formed, where $Y' = C(Y+L1)_{21-35}$, and the C(Y'+L2) is added to the AC. Then the MQ is rotated six more positions to the left. This process occurs n times, n being the $C(INSTR)_{10-17}$, where INSTR contains the CAQ instruction.

Examples of the use of the CAQ and the other two 7090 convert instructions appear in the IBM manuals, under "Programming Examples." Other uses are given below in Examples 16.2 and 16.3A.

Within FAP, the count is designated as the third subfield in the variable field.

(R)----------(332.2 - 333.10)----------

Example 16.1 Write a routine to convert alphanumeric octal information to binary form.

This conversion process is a simple one and does not benefit particularly from the use of CAQ. Refer to the book for an analysis. Here, the number to be converted is placed initially in the MQ. The converted word remains in the AC.

```
Locn.   Oper.   Var. Field
        AXT     6,1
        PXA     0,0             Zero AC
LØØP    RQL     3
        LGL     3
        TIX     *-2,1,1
```

Example 16.2 Write a routine to convert alphanumeric decimal information to binary form. A number in DIGITS is to be converted; the result is to be stored in SUM.

The CAQ instruction can be used to great advantage, since it permits a sum of numbers to be accumulated rapidly as the result of a series of table references. Consider a BCD word containing an integer: 002258. The number can be calculated by summing 8×10^0, 5×10^1, 2×10^2, and 2×10^3. For this purpose, six tables of 10 words each are required. The address fields of the words in these tables each contain the head of the following table, so that reference is made in succession to the six tables. The decrements of the first table contain the 10 digits from 0 to 9. The decrements of the second table contain the numbers 0, 10, 20, ..., 90. The decrements of the sixth table contain the numbers 0, 100000, 200000, ..., 900000. The addresses of the sixth-table words are irrelevant. (Special care is required that the sum of the six addresses from the table does not overflow into the left part of the AC, where the sum is accumulated. This overflow can never extend past the decrement for a count of 6.)

```
Locn.   Oper.   Var. Field
        LDQ     DIGITS          Load number in MQ
        PXA     0,0
        CAQ     CTABLE,,6
        ARS     18              Place sum in address field
        SLW     SUM
```

The table begins as follows, where 00500 is the address of CTABLE:

Location	Contents			
00500	0 00000 0 00512	CTABLE	PZE	CTABLE+10,,0
00501	0 00001 0 00512		PZE	CTABLE+10,,1
00502	0 00002 0 00512		PZE	CTABLE+10,,2
00503	0 00003 0 00512		PZE	CTABLE+10,,3
00504	0 00004 0 00512		PZE	CTABLE+10,,4
00505	0 00005 0 00512		PZE	CTABLE+10,,5
00506	0 00006 0 00512		PZE	CTABLE+10,,6
00507	0 00007 0 00512		PZE	CTABLE+10,,7
00510	0 00010 0 00512		PZE	CTABLE+10,,8
00511	0 00011 0 00512		PZE	CTABLE+10,,9
00512	0 00000 0 00524		PZE	CTABLE+20,,0
00513	0 00012 0 00524		PZE	CTABLE+20,,10
00514	0 00024 0 00524		PZE	CTABLE+20,,20
...	.....		...	.....

Example 16.3 Write a routine to convert binary numbers to alphanumeric decimal form.

This is the reverse of the process in Example 16.2, but it is not possible to program the problem in the opposite manner. Instead, a simple approach is obtained by using the repeated division process described in Section 4.2. Arithmetic must be done in binary; in a binary machine this is automatic. The remainders are saved in binary, which happens to be their BCD form, since all remainders are digits. The number to be converted is located in NUMBER. A 35-bit number may contain as many as 11 decimal digits, when converted.

```
Locn.   Oper.   Var. Field
        AXT     11,1
        LDQ     NUMBER
MØRE    PXA     0,0
        DVP     TEN
        STØ     LIST+11,1       Store remainder
        TIX     MØRE,1,1
```

After execution of this routine, the 11 digits are stored in sequence in the block at LIST, with the low-order digit in LIST. A packing routine can place them in two words; they will then be in BCD form.

(S)----------(At 334.2)----------

CONVERSION

The following instruction is useful in the conversion of 6-bit bites to other 6-bit bites.

CONVERT BY REPLACEMENT FROM THE MQ (CRQ Y) (-0154); 2-8 cycles. This instruction is similar to CAQ in its execution, except that the leftmost 6 bits (S and 1-5) of the referenced words replace the 6-bit quantities in the MQ, L1, L2, ..., L6. As before, the MQ is rotated six positions left after each such substitution. The count n applies as in CAQ.

Example 16.3A* Write a program that scans the 100 words starting at LIST and makes the following BCD substitutions:

replace all even digits by X;
replace all vowels by Y;
leave other codes unchanged.

The program is the following:

```
Locn.   Oper.   Var. Field
        AXT     100,2
LØØPS   LDQ     LIST+100,2      Word to MQ
        CRQ     TABLE,,6        Convert word
        STQ     LIST+100,2      Store word
        TIX     LØØPS,2,1
        HTR
```

In addition, a table is required which contains the new BCD codes in bits 1-6 and the address of the table (i.e., TABLE) in all address fields.

To create such a table, the FAP pseudo-operation VFD is useful. As an example of its use, consider

```
Locn.   Oper.   Var. Field
PLACE   VFD     H6/X,15/,15/NAME
```

*This example has no counterpart in the book.

Here, the items in the variable field mean the following: generate a 36-bit word at PLACE with its first 6 bits the Hollerith X, its next 15 bits ignored (zeros), and its next 15 bits the equivalence of the symbol NAME.* The table thus begins as follows:

```
TABLE    VFD     H6/X,15/,15/TABLE     0
         VFD     H6/1,15/,15/TABLE     1
         VFD     H6/X,15/,15/TABLE     2
         VFD     H6/3,15/,15/TABLE     3
         VFD     H6/X,15/,15/TABLE     4
         VFD     H6/5,15/,15/TABLE     5
         VFD     H6/X,15/,15/TABLE     6
         VFD     H6/7,15/,15/TABLE     7
         VFD     H6/X,15/,15/TABLE     8
         VFD     H6/9,15/,15/TABLE     9
         BSS     1                     nil
         VFD     H6/=,15/,15/TABLE     =
         VFD     H6/",15/,15/TABLE     "
         BSS     3                     3 nils
         VFD     H6/+,15/,15/TABLE     +
         VFD     H6/Y,15/,15/TABLE     A
         VFD     H6/B,15/,15/TABLE     B
         VFD     H6/C,15/,15/TABLE     C
         ...          .....            ..
```

(R)----------(338.7 - 340.2)----------

SORTING (ORDERING)

TRANSFER ON LOW MQ (TLQ Y) (+0040); 2 cycles. If the C(MQ) is algebraically less than the C(AC), the computer takes its next instruction from location Y. If the C(MQ) is algebraically greater than or equal to the C(AC), the computer continues in sequence.

Example 16.4 Write a routine to perform an interchange sort of 100 numbers, located starting at NUMBRS.

Let ℓ_i be the i^{th} location in the list. During the first pass, the first comparison is made between the $C(\ell_1)$ and the $C(\ell_2)$, and the last comparison is made between the $C(\ell_{99})$ and the $C(\ell_{100})$. The i^{th} comparison is made between the $C(\ell_i)$ and the $C(\ell_{i+1})$; i runs from 1 to 99. In the program, the index, as given by the C(XR1), runs initially from 99 to 1, then on succeeding passes runs from 99 to limits that increase by 1 each pass. This is

---

*Refer to a FAP manual for more details on VFD.

accomplished by modifying the test instruction (TXH) after each pass through the numbers. The whole process terminates when XR2 is reduced to 1, after the 99th pass. The AC and MQ are used for comparison and exchange. The following routine places the smallest number first.

```
Locn.    Oper.    Var. Field
         CLA      ZERO              Initialize test
         STD      TEST
         AXT      99,2
NEWPAS   AXT      99,1              Start new pass
NEWNUM   LDQ      NUMBRS+99,1       ℓi to MQ
         CLA      NUMBRS+100,1      ℓi+1 to AC
         TLQ      NØEX              ℓi+1 greater; no exch.
         STQ      NUMBRS+100,1
         STØ      NUMBRS+99,1
         TXI      *+1,1,-1
TEST     TXH      NEWNUM,1,**       Test for last number
         CAL      TEST              Modify test; add 1 to
         ADD      DECR1               decrement
         STD      TEST
         TIX      NEWPAS,2,1
         ...
DECR1    ØCT      1000000           1 in decrement
ZERØ
NUMBRS   BSS      100
```

(M)----------(340.2 - 340.5)----------

The book discusses, in 340.2 - 340.5, the efficiency of this sorting procedure. The following instruction can be inserted in the program to test for an interchange in a given pass:

```
         STZ      INDIC
```

(R)----------(341.1 - 341.10)----------

MERGING

Example 16.5 Write a routine to merge the lists in the blocks at LIST1 and LIST2 of sizes m and n, respectively, forming a single ordered list, NEWLST. The original lists have no more than 1000 numbers each.

The flowchart in Fig. 16.3 in the book shows the process. The entries in LIST1 are a_i; the entries in LIST2 are b_j; the locations in NEWLST are ℓ_k. XR1 and XR2 act as pointers to the two original lists; XR4 acts as a pointer to NEWLST.

In order to insure that trouble does not occur when either list is exhausted, an extra number is assumed present at the end of each list. This number exceeds any number in the list and is not counted (in the counts m and n). When m + n numbers are merged and the process stops, these numbers are ignored. At the end of the merging process, a dummy number (in LARGE) must be attached to the end of NEWLST.

```
Locn.   Oper.   Var. Field

        CLA     X2001           Form 2001-m-n;
        SUB     M                 set into test instr.
        SUB     N
        ALS     18
        STD     SET
        AXT     2001,4
        AXT     1001,1
        AXT     1001,2
NEXT    CLA     LIST1+1001,1    Compare a_i and b_j
        LDQ     LIST2+1001,2
        TLQ     SMALL2
SMALL1  STØ     NEWLST+2001,4   Store a_i (smaller)
        TXI     MØD,1,-1
SMALL2  STQ     NEWLST+2001,4   Store b_j (smaller)
        TXI     MØD,2,-1
MØD     TXI     *+1,4,-1
SET     TXH     NEXT,4,**       (2001-m-n)
        CLA     LARGE           Store large number
        STØ     NEWLST+2001,4     (dummy) at end
        HTR

M
N
LARGE   ØCT     377777777777
X2001   DEC     2001
LIST1   BSS     1001
LIST2   BSS     1001
NEWLST  BSS     2001
```

Chapter 17
MACRO-INSTRUCTIONS

(S)----------(At 348.10)----------

MACRO-INSTRUCTION PSEUDO-OPERATIONS

The pseudo-operations described here are all available in BE-FAP, Bell Telephone Laboratories' version of FAP. They are not all available in other 7090 assemblers, while other features, not here described, may be present in other systems.

(R)----------(349.0 - 349.9)----------

Example 17.1 Define and use a macro-instruction that sums three numbers and stores the result.
The macro-definition:

```
Locn.   Oper.    Var. Field

SUM3    MACRO    A,B,C,S
        CLA      A
        ADD      B
        ADD      C
        STØ      S
        END
```

This macro-instruction sums the C(A), C(B), and C(C), and stores the sum in S.
Consider this macro-call:

```
        SUM3     WORD,DIGIT,NUMBER,RESULT
```

This call expands into the following sequence:

```
        CLA      WØRD
        ADD      DIGIT
        ADD      NUMBER
        STØ      RESULT
```

There is no restriction on the nature of the alphanumeric strings that may be substituted for dummy arguments. The following indicates some possibilities.

Call the SUM3 macro-instruction using more complex call arguments.

(1) The following call

```
Oper.    Var. Field

SUM3     WORD+1,(DIGIT+1,3),NUMBER,RESULT
```

expands into the following:

```
CLA      WØRD+1
ADD      DIGIT+1,3
ADD      NUMBER
STØ      RESULT
```

(2) The following call

```
SUM3     WØRD+NAME-23,,NUMBER+3,/Ø/44000
```

expands into the following:

```
CLA      WØRD+NAME-23
ADD
ADD      NUMBER+3
STØ      /Ø/44000
```

(R)----------(350.1 - 350.3)----------

Location	Contents	Location	Oper.	Var. Field
00100	+0774 00 1 00012		AXT	10,1
00101	+0500 00 0 00223	LØØP	CLA	X
00102	+0601 00 0 00224		STØ	Y
00103			SUM3	NAME,W,X,Z
00107	+0120 00 0 00117		TPL	NEXT
00110	+1 00001 2 00111		TXI	*+1,2,1

(S)----------(At 350.4)----------

The FAP pseudo-operation used to cause printing of macro-instruction expansions is PMC (print macro-instruction). Repeated use of PMC "turns on" and then "turns off" this printing.

(R)----------(351.4 - 351.10)----------

PSEUDO-OPERATIONS

Example 17.2 Compute the value of p:

$$p = (a+b+c)(d+e+f+g+h)/(a+d+h)$$

We can use the SUM3 macro-instruction:

Oper.	Var. Field
SUM3	A,B,C,TEMP
SUM3	D,E,F,TEMP+1
SUM3	TEMP+1,G,H,TEMP+1
SUM3	A,D,H,TEMP+2
LDQ	TEMP
MPY	TEMP+1
DVP	TEMP+2
STQ	P

In this program, to sum five numbers, the SUM3 macro-instruction must be used twice in succession. A temporary location TEMP+1 is required for the storage of the first sum. The second and third calls should be examined; they expand to the following:

CLA	D
ADD	E
ADD	F
STØ	TEMP+1
CLA	TEMP+1
ADD	G
ADD	H
STØ	TEMP+1

(R)----------(352.4 - 352.9)----------

CONDITIONAL ASSEMBLY

The following pseudo-operation in the FAP assembler is used to effect conditional assembly:

Oper.	Var. Field
IFF	P,A,B

The P subfield represents a symbolic expression, while the A and B subfields represent alphanumeric strings. The instruction that is next after the IFF pseudo-operation is assembled if and only if the quantities p and q have the same values, where

$$p = \begin{cases} 0 \text{ if the equivalence of P is zero} \\ 1 \text{ if the equivalence of P is nonzero} \end{cases}$$

$$q = \begin{cases} 0 \text{ if A and B are nonidentical strings} \\ 1 \text{ if A and B are identical strings} \end{cases}$$

If at least one symbol in P is undefined prior to the occurrence of the IFF pseudo-operation, then p = 0.

As examples, the following instructions will cause the suppression of the next instructions:

```
IFF   1,R,S
IFF   0,XXX,XXX
IFF   LØC+20,AA+1,1+AA
```

provided the equivalence of LØC is not -20. Note that AA+1 and 1+AA are not identical strings (although they have the same equivalence). The following instructions will cause the assembly of the next instructions:

```
IFF   1,SS,SS
IFF   0,X,Y
IFF   NAME-NAME,A+B+C,A+C+B
```

Arguments to be substituted in the three subfields of IFF may be substituted within a macro-definition as may any argument.

(R)----------(353.0 - 354.7)----------

This pseudo-operation may be used to suppress the instructions discussed in Example 17.2 For example, if we write

```
IFF   0,AC,A
```

then assembly if the next instruction will occur if and only if A is not AC.

Example 17.3 Revise the SUM3 macro-instruction to suppress AC instructions as indicated. Recode the evaluation of p, given in Example 17.2.

The macro-definition is

```
Locn.   Oper.    Var. Field

SUM3    MACRØ    A,B,C,S
        IFF      0,AC,A
        CLA      A
        ADD      B
        ADD      C
        IFF      0,AC,S
        STØ      S
        END
```

Consider the following call and how it will be treated by the assembler.

```
        SUM3     D,E,F,AC
```

Consider the first IFF. Since the IFF A and B subfields are different (A = D and B = AC upon substitution), q = 0, while p = 0 also. Therefore, assembly of the next instruction occurs. Next consider the second IFF. Here, the IFF A and B subfields are the same (they are both "AC" upon substitution) so that q = 1 while p = 0. Hence, the next instruction is not assembled. Therefore this call expands as follows:

```
        CLA      D
        ADD      E
        ADD      F
```

By similar reasoning, the call

```
        SUM3     AC,G,H,TEMP+1
```

expands into

```
        ADD      G
        ADD      H
        STØ      TEMP+1
```

The revised coding begins as follows (see Example 17.2):

```
        SUM3     A,B,C,TEMP
        SUM3     D,E,F,AC
        SUM3     AC,G,H,TEMP+1
        SUM3     A,D,H,TEMP+2
        ...      .....
```

The second and third calls expand into the following:

```
CLA     D
ADD     E
ADD     F
ADD     G
ADD     H
STØ     TEMP+1
```

The redundant coding of Example 17.2 is not assembled.

In this use of the IFF pseudo-operation, the symbol "AC" is used because it is suggestive of the accumulator. Of course, any desired symbol may be used.

(M)----------(355.4 - 355.9)----------

This material has no exact counterpart in FAP, since the IFF pseudo-operation is more structured differently from each of the IF-type HAP pseudo-operations. However, there are parallels; the following usages are equivalent in the two languages:

HAP		FAP	
IFSAME	A,B	IFF	1,A,B
IFDIFF	A,B	IFF	0,A,B
IFZERØ	Q	IFF	Q,1
IFNØNZ	Q	IFF	Q

(The reader should now read the material of 355.4 - 355.9 and then read the following.)

The first two cases of IFF above, illustrated in earlier examples, permit the variability of the second and third subfields. The last two cases illustrate its use with a variable first subfield, which can then be tested for zeroness. Thus, with subfield A = 1 and subfield B blank, q = 0 and assembly of the next instruction occurs if and only if Q is zero. Conversely, in the last case, assembly occurs if and only if Q is nonzero. The book's examples of the following form apply in FAP:

```
IFF   Q/5
IFF   Q-Q/2*2,1
```

(R)----------(356.0 - 356.4)----------

REPETITION IN CODING

Example 17.4 Write a macro-instruction to determine the sum of the cubes of three quantities and to store the result.

The macro-definition is

```
Locn.   Oper.    Var. Field

SUMCUB  MACRØ    A,B,C,S
        STZ      TEMP
        LDQ      A
        MPY      A
        MPY      A
        XCA
        ADD      TEMP
        STØ      TEMP
        LDQ      B
        MPY      B
        MPY      B
        XCA
        ADD      TEMP
        STØ      TEMP
        LDQ      C
        MPY      C
        MPY      C
        XCA
        ADD      TEMP
        STØ      S
        END
```

(R)----------(356.6 - 357.8)----------

Coding delimited by the IRP (indefinite repeat) pseudo-operation at the start and end is repeated once for each call argument supplied for a dummy argument A, the coding being assembled with the call arguments given for A.

At the start:

```
Oper.    Var. Field

IRP      A
```

At the end:

```
IRP
```

The several call arguments for A are placed within parentheses, separated by commas.

Example 17.4, continued Recode the SUMCUB macro-instruction, using IRP.

The new macro-definition:

```
Locn.    Oper.     Var. Field

SUMCUB   MACRØ     X,S
         STZ       TEMP
         IRP       X
         LDQ       X
         MPY       X
         MPY       X
         XCA
         ADD       TEMP
         STØ       TEMP
         IRP
         STØ       S
         END
```

The call

```
         SUMCUB    (X1,X2,X3),RESULT
```

expands into

```
         STZ       TEMP
         LDQ       X1
         MPY       X1
         MPY       X1
         XCA
         ADD       TEMP
         STØ       TEMP
         LDQ       X2
         MPY       X2
         ...       ...
```

The sequence is the same as given earlier, except for an extra STØ instruction near the end of the sequence.

The second macro-definition is shorter than the first and serves for any number of repetitions. Thus the same definition can be called upon by the following instructions:

```
         SUMCUB    (WØRD1,WØRD2),ANSWER
         SUMCUB    (Y1,Z1,X1,A1,B2,C3),SUM
```

In the first two cases two parameters are involved; in the second, six parameters are involved. Through the use of the IRP pseudo-operation a macro-instruction of variable length may be defined. The length of the expanded coding depends on the manner in which the macro-instruction is called.

(R)----------(357.9 - 358.3)----------

CREATED SYMBOLS

Example 17.5 Write a macro-instruction that adds the larger of two numbers to a third, leaving the sum in the accumulator. If the numbers are equal, a jump to EQUAL should occur.

The macro-definition is

```
Locn.   Oper.    Var. Field

LARSUM  MACRØ    A,B,C
        CLA      A
        CAS      B
        TRA      DD
        TRA      EQUAL
        CLA      B
DD      ADD      C
        END
```

(R)----------(358.4 - 358.10)----------

Any dummy arguments that appear in a macro-definition that are not called for within a macro-call, provided they are omitted at the end of the call, are replaced by the assembler by created symbols. The following symbols are created: ..001, ..002, ..003, etc.; they are created in this sequence as needed.

Example 17.5, continued Rewrite the macro-instruction LARSUM, permitting a created symbol.

The macro-definition, which has one extra dummy argument, is

```
Locn.   Oper.    Var. Field

LARSUM  MACRØ    A,B,C,DD
        CLA      A
        CAS      B
        TRA      DD
        TRA      EQUAL
        CLA      B
DD      ADD      C
        END
```

The call

```
        LARSUM  NUMBR1,NUMBR2,DIGIT
```

which means "add the larger of the C(NUMBR1) and the C(NUMBR2) to the C(DIGIT), and leave the sum in the AC," expands into

```
          CLA     NUMBR1
          CAS     NUMBR2
          TRA     ..001
          TRA     EQUAL
          CAL     NUMBR2
..001     ADD     DIGIT
```

(M)----------(359.0 - 359.3)----------

This material applies here, with just one change. Dummy arguments to be replaced by created symbols do not appear on CREATE cards, as described in the book.

(R)----------(359.3 - 359.7)----------

Example 17.6 Write a macro-instruction that places the larger of two numbers in the accumulator. If the numbers are equal, a jump to EQUAL should occur.

The macro-definition is

```
Locn.   Oper.    Var. Field

LARGER  MACRØ    A,B,DD
        CLA      A
        CAS      B
        TRA      DD
        TRA      EQUAL
        CLA      B
DD      BSS      0
        END
```

The call

```
        LARGER  WØRD5,WØRD7
```

expands into

```
          CLA     WØRD5
          CAS     WØRD7
          TRA     ..001
          TRA     EQUAL
          CLA     WØRD7
..001     BSS     0
```

(S)----------(At 359.10)----------

A special pseudo-operation is available to determine whether or not a given symbol within a macro-definition has been replaced by a created symbol.

Oper.	Var. Field
IFF	P,/CRS/X

P has the same significance as previously, in the IFF pseudo-operation. The quantity q has the value 0 if X is not a created symbol and the value 1 if it is. Thus,

```
IFF     1,/CRS/Q
```

causes assembly of the next instruction if and only if Q is created.

(R)----------(360.3 - 361.3)----------

REMOTE ASSEMBLY

Coding delimited by the card

```
RMT
```

(for remote) at the start and at the end is assembled as normally but is not assigned to memory locations until the end of the program.

Example 17.7 Write a macro-instruction to evaluate and store the function

$$f(x) = 5a + bc$$

At constant, 5, and one word of temporary storage are required. In the following macro-instruction, these two words are remotely assembled.

```
Locn.   Oper.     Var. Field

FUNCTN  MACRØ     A,B,C,R,FIVE,TEMP
        LDQ       A
        MPY       FIVE
        STQ       TEMP
        LDQ       B
        MPY       C
        XCA
        ADD       TEMP
        STØ       R
        RMT
FIVE    DEC       5
TEMP
        RMT
        END
```

The call

```
        FUNCTN  XX,YYY,ZZZZ,ANS
```

expands into

```
        LDQ       XX
        MPY       ..001
        STØ       ..002
        LDQ       YYY
        MPY       ZZZZ
        XCA
        ADD       ..002
        STØ       ANS
```

while the two words

```
..001   DEC       5
..002
```

are assembled at the end of the program. Created symbols are used to refer to these two words to avoid multiple definitions as before.

(This method actually wastes space, however, since two words are assembled per FUNCTN macro-call, whereas only two are needed in all. However, it points out that macro-instructions can be completely self-contained, which is useful.)

NESTED MACRO-INSTRUCTIONS

Example 17.8 Write a macro-instruction that computes one of these two functions:

$f = a + b$, if a and b are both positive;
$g = a - b$, otherwise.

Two macro-instructions for addition and subtraction are defined first:

```
Locn.    Oper.     Var. Field

ADDMAC   MACRØ     A,B,C
         CLA       A
         ADD       B
         STØ       C
         END

SUBMAC   MACRØ     A,B,C
         CLA       A
         SUB       B
         STØ       C
         END
```

The main macro-definition is

```
CØMPUT   MACRØ     K,L,S,SUB,ØUT
         CLA       K
         TMI       SUB
         CLA       L
         TMI       SUB
         ADDMAC    K,L,S
         TRA       ØUT
SUB      SUBMAC    K,L,S
ØUT      BSS       0
         END
```

The call

```
         CØMPUT  THIS,THAT,RESULT
```

expands into the following coding, where L(a) = THIS, L(b) = THAT, and L(f) or L(g) is RESULT.

```
         CLA     THIS
         TMI     ..001
         CLA     THAT
         TMI     ..001
         CLA     THIS    }
         ADD     THAT    } Expansion of ADDMAC
         STØ     RESULT  }
         TRA     ..002
..001    CLA     THIS    }
         SUB     THAT    } Expansion of SUBMAC
         STØ     RESULT  }
..002    BSS     0
```

(M)----------(363.7 - 364.5)----------

(This material has no counterpart in FAP, since there is no pseudo-operation in the latter equivalent to TØ. However, the pseudo-operation GØ (with a blank variable field) may be used. It works like the book's TØ with a variable address that refers to the end of the macro-instruction. In other words, GØ causes assembly of the rest of the macro-instruction to be suppressed.)

(R)----------(365.3 - 366.4)----------

Example 17.10 Write a macro-instructions to perform two of the five block operations.

1. Clearing By this is meant placing zeros in every word in the block. The call is to have the form

```
Locn.   Oper.    Var. Field

        CLRBLK   BLØCK,N
```

This means "Clear the block starting at BLØCK of size C(N)"; the number of words in the block is located in N.

The macro-definition is

```
CLRBLK  MACRØ    A,B
        AXT      B,4
        STZ      A+B,4
        TIX      *-1,4,1
        END
```

XR4 is used in all the block macro-instructions and is therefore to be used outside with caution.

2. Moving The call is to have the form

```
MOVBLK          BLØCK1,N1,BLØCK2,N2
```

This means "Move the block at BLØCK1 of size C(N1) so that it immediately follows the block at BLØCK2 of size C(N2).

The macro-definition is

```
MØVBLK  MACRØ           B1,N1,B2,N2,Z1,Z2,Y1,Y2
        CLA             Z1
        ADD             N1
        STA             Y1
        CLA             Z2
        ADD             N1
        ADD             N2
        STA             Y2
        AXT             N1,4
Y1      CLA             **,4    (B1+N1)
Y2      STØ             **,4    (B2+N1+N2)
        TIX             *-2,4,1
        RMT
Z1                      B1
Z2                      B2
        RMT
        END
```

The first 7 instructions are used to set the addresses of locations Y1 and Y2. The first such address (of Y1) must be set to "BLØCK1+n_1" and the second (of Y2) must be set to "BLØCK2+n_1+n_2" if the first word in BLOCK1 is to go after the last word in BLØCK2.

(Refer to the book for the continuation of the example.)

(R)----------(366.8 - 367.9)----------

Example 17.11 Write a program that (1) reads three blocks of data (with no more than 2500 numbers in each) from tape F, placing them in blocks starting at LIST, TABLE, and DIGITS, (2) combines these into one larger block at DIGITS, (3) places in a new block at NUMBRS all positive numbers from this block that are less than 1000, and (4) then prints out the list of such numbers on tape G. The number of words in each of the three records on tape appears in a three-word record at the start of the tape.

After the three blocks are read in, they are combined at DIGITS. In order to add the third block (TABLE) the size of the combined blocks at DIGITS and LIST must be computed. The total size of the combined blocks is needed for the loop within which all positive numbers less than 1000 are stored in NUMBRS. As each number is stored in that

block, the C(XR2) is decreased by 1; XR2 is used as a pointer to the next available location in the block at NUMBRS. Finally, the number of words in NUMBRS is placed in SZ4 and used in the PRTBLK macro-instruction. In the program, note that the three-word record will be read into SZ1 through SZ1+2, i.e., into SZ1, SZ2, and SZ3.

```
Locn.   Oper.    Var. Field
        READBK   SZ1,THREE,F     Read 3-word record
        READBK   LIST,SZ1,F
        READBK   TABLE,SZ2,F
        READBK   DIGITS,SZ3,F
        MØVBLK   LIST,SZ1,DIGITS,SZ3
        CLA      SZ1             Form size of 2 blocks
        ADD      SZ3
        STØ      SUMSZ
        ADD      SZ2             Form size of 3 blocks
        STØ      TØTLSZ
        MØVBLK   TABLE,SZ2,DIGITS,SUMSZ
        AXT      7500,2
        CLA      XDIGIT          Set address
        ADD      TØTLSZ
        STA      MØRE
        LXA      TØTLSZ,1
MØRE    CLA      **,1            (DIGITS+totlsz)
        TMI      NØLIST          Place pos. numbers
        CAS      THØUS             smaller than 1000
        TRA      NØLIST            in NUMBRS
        TRA      NØLIST
        STØ      NUMBRS+7500,2
        TXI      *+1,2,-1
NØLIST  TIX      MØRE,1,1
        SXA      TEMP,2          Determine count
        CLA      X7500             of numbers stored
        SUB      TEMP
        STØ      SZ4
        PRTBLK   NUMBRS,SZ4,G
        HTR
```

(Cont'd.)

```
Locn.   Oper.     Var. Field

SZ1
SZ2
SZ3
SZ
SUMSZ
TØTLSZ
THØUS   DEC       1000
THREE   DEC       3
XDIGIT
TEMP
LIST    BSS       2500
TABLE   BSS       2500
DIGITS  BSS       7500
NUMBRS  BSS       7500
```

(R)----------(368.4 - 370.7)----------

Example 17.12 Write macro-instructions to simulate 100 index registers, using the CLA instruction as a specific illustration.

Normal assembler instruction names, e.g., CLA, ADD, and SUB, are to be used by the programmer in the usual manner, but with tags as high as 100. Since an instruction such as

```
CLA   LIST,68
```

would be misinterpreted by the assembler under normal circumstances, the symbol CLA must refer to a macro-instruction. This macro-instruction must produce the proper coding for the simulation of XR68.

In order that the assembler operation codes be interpreted as macro-instructions, new names must be assigned to the machine instructions. This may be done by a series of pseudo-operations as follows:

```
Locn.   Oper.     Var. Field

CLA.    ØPSYN     CLA
ADD.    ØPSYN     ADD
SUB.    ØPSYN     SUB
        ...
```

Now the original names (in the variable field above) may be used as macro-instruction names.

Since the 7094 has 7 index registers, 93 will be simulated. XR8 through XR100 will be simulated by 93 words in memory in a block, starting at SIMXR. Thus, XRJ, where J = 8, 9, ..., 100, will be simulated by location SIMXR+J-8.

Consider the simulation of the CLA instruction. One of three coding sequences must be assembled, conditional on the tag: (1) an untagged CLA instruction is required if no tag is present; (2) a normal, tagged CLA instruction is required if a tag from 1 to 7 is given; and (3) a coding sequence as follows is required if a tag from 8 to 100 is given: the instructions assembled must, when they are executed later, modify the CLA operand address by the contents of the simulated XR. These conditions are depicted in Fig. 17.1 in the book.

Conditional-assembly techniques are required. Two decisions must be made. The test for a tag can be made with the IFF test for a created symbol; if the tag is omitted, a created symbol can be produced. The test for the "size" of the tag can be made with another form of the IFF pseudo-operation. Because IFF affects only the following instruction, an inner macro-call is required, since several instructions must be assembled conditionally.

The instructions assembled in the event that a tag from 8 through 100 is given provide for saving and restoring XR1, which is used as the actual index register, for executing the CLA instruction, and for loading XR1 with the C(SIMXR+J-8).

The macro-definitions are

```
Locn.   Oper.    Var. Field

CLA     MACRØ    A,T
        IFF      1,/CRS/T
        CLA.     A                Assemble if no tag
        IFF      0,/CRS/T
        CLAM     A,T              Assemble if tag
        END

CLAM    MACRØ    A,T
        IFF      T/8,1
        CLA.     A,T              Assemble if tag is
        IFF      T/8                1 through 7
        CLAN     A,T              Assemble otherwise
        END

CLAN    MACRØ    A,T
        SXA.     SAVX1,1          Save XR1
        LXA.     SIMXR+T-8,1
        CLA.     A,1
        LXA.     SAVX1,1
        END
```

The use of a normal machine operation code as a macro-instruction name reassigns that name to the latter function and deletes its use as a machine operation for that assembly. Thus CLA now refers only to the macro-instruction.

Note that operation codes ending in "." are used when a machine instruction is to be assembled. The following three macro-calls lead to the accompanying expnasions:

		Oper.	Var. Field
1.	Call:	CLA	SWITCH
	Expansion:	CLA.	SWITCH
2.	Call:	CLA	LIST,5
	Expansion:	CLA.	LIST,5
3.	Call:	CLA	NAME,32
	Expansion:	SXA.	SAVX1,1
		LXA.	SIMXR+32-8,1
		CLA.	NAME,1
		LXA.	SAVX1,1

Other instructions are similarly simulated, but a different coding structure is needed for such instructions as LXA, TXI, and TSX. To simulate LXA, e.g., we need only place a number in the proper SIMXR word (if the tag is 8 or greater); it is not necessary to use a real index register in the process. Similarly, to simulate TXI, we need only increase the proper SIMXR word contents.

(R)----------(371.2 - 371.8)----------

```
Locn.   Oper.    Var. Field

HALT    MACRØ
        HTR
        END

ADD.    ØPSYN    ADD
ADD     MACRØ    A,B,C
        CLA      A
        ADD.     B
        STØ      C
        END
```

```
Locn.    Oper.     Var. Field

SUBT     MACRØ     A,B,C
         CLA       A
         SUB       B
         STØ       C
         END

MULT     MACRØ     A,B,C
         LDQ       A
         MPY       B
         STQ       C
         END

DIV      MACRØ     A,B,C
         LDQ       A
         PXA       0,0
         LLS       0
         DVP       B
         STQ       C
         END

MØVE     MACRØ     A,B,C
         AXT       B-A+1,4
         CLA       B+1,4
         STØ       C+B-A+1,4
         TIX       *-2,4,1
         END

JUMP     MACRØ     A
         TRA       A
         END

JUMPPM   MACRØ     A,B,C
         CLA       A
         TPL       B
         TMI       C
         END
```

(R)----------(372.4 - 373.2)----------

Example 17.13 The first technique might use a PRINTL macro-instruction which provided a printout (probably with a standard format) of an indefinite number of specified words (a list), as in

```
PRINTL  (NUMBR,WØRD,WØRD+3,XYZ)
```

The structure of PRINTL depends on the form of the call for the monitor input-output subroutine. It can be inserted at any desired points in the program.

The second technique can be exemplified by a procedure that automatically supplies a printout of the contents of the referenced location in any STØ instruction. To accomplish this, the operation STØ must be defined as a macro-instruction. Another printing macro-instruction, PRINT, is used.

```
Locn.   Oper.    Var. Field

STØ.    ØPSYN    STØ
STØ     MACRØ    A,T
        IFF      1,/CRS/T
        STØ.     A
        IFF      0,/CRS/T
        STØ.     A,T
        PRINT    A,T
        END
```

To allow for the presence or absence of a tag, the IFF pseudo-operations are used. Here, we shall assume that a tag may also be present on the PRINT call.

(R)----------(373.6 - 374.8)----------

Example 17.14 Write the TURNØN and TURNØF macro-instructions described.

To accomplish this, STØ must be used for both the normal machine instruction (when no printing is desired) and for the macro-instruction (when printing is desired). The TURNØN and TURNØF macro-instructions have the function of the switching the significance (and interpretation) of the word STØ back and forth between these two, the machine instruction and the macro-instruction. In this way the printing feature is "turned on and off."

The TURNØN and TURNØF macro-instructions have no arguments; their calls appear to be pseudo-operations.

```
Locn.   Oper.    Var. Field

STØ.    ØPSYN    STØ
STØ..   MACRØ    A,T
        IFF      1,/CRS/T
        STØ.     A
        IFF      0,/CRS/T
        STØ.     A,T
        PRINT    A,T
        END

TURNØN  MACRØ
STØ     ØPSYN    STØ..
        END

TURNØF  MACRØ
STØ     ØPSYN    STØ.
        END
```

A short program, using these features, follows:

```
Locn.   Oper.    Var. Field

        TURNØN
        AXT      200,1
QQQ     CLA      LIST+200,1
        ADD      SIX
        STØ      LIST+200,1
        TIX      QQQ,1,1
        TURNØF
        CLA      NUMBER
        STØ      TABLE
        TZE      ØUT
```

This sequence expands into the following coding (macro-calls are not given):

```
        AXT      200,1
QQQ     CLA      LIST+200,1
        ADD      SIX
        STØ.     LIST+200,1
        PRINT    LIST+200,1
        TIX      QQQ,1,1
        CLA      NUMBER
        STØ.     TABLE
        TZE      ØUT
```

(R)----------(375.3 - 376.2)----------

Example 17.15 Write coding so that printing occurs (1) at every third STØ instruction, and (2) at every third STØ execution.

1. The STØ macro-instruction of Example 17.13 is modified to provide for conditional assembly of the output macro-instruction (PRINT). The symbol Q is used as a counter, increased by 1 each time the macro-instruction is called. The IFF variable field is similar to one given earlier, in Section 17.1, under "Conditional Assembly." Assembly of PRINT occurs every time Q is a multiple of 3.

```
Locn.   Oper.    Var. Field

Q       SET      0
STØ.    ØPSYN    STØ

STØ     MACRØ    A,T
        IFF      1,/CRS/T
        STØ.     A
        IFF      0,/CRS/T
        STØ.     A,T
Q       SET      Q+1
        IFF      Q-Q/3*3,1
        PRINT    A,T
        END
```

2. Now counting must be done when the program is executed. To achieve this, a sequence of coding must be included that calculates the function Q-Q/3*3 where the contents of the counter is Q (when the program is executed).

```
        CLA      CNTR          Q+1 to Q
        ADD      ØNE
        STØ      CNTR
        XCA
        PXA      0,0
        DVP      THREE
        MPY      THREE         Q/3*3
        TNZ      NØPRINT       Test for Q-Q/3*3 = 0
        PRINT    A,T
NØPRNT  ...      ...
```

It is easier to compute the remainder directly, but this approach is taken as a parallel to method 1.

(R)----------(376.5 - 378.4)----------

Example 17.16 Write a macro-instruction that will cause a single word to be assembled, containing n! The call is to be as follows:

FACTL N

where N represents an integer to be supplied.

Two nested macro-instructions are used. In the inner macro-instruction (FACTLX) the actual recursion occurs. The macro-instruction repeatedly calls itself, each time computing one more factor in n!, as follows: $1 \cdot 2 \cdot 3 \cdots n$. At the same time, a "counter" Q is used for loop control; the counter runs from 1 to n. An IFF is used to control the recursion; when the counter contains the value n, the process stops.

The outer macro-instruction is used to initialize both the counter Q (at 0) and a partial product F (at 1). If n is given as 0, assembly of the FATCLX macro-call is suppressed and F (which is n!) is set equal to 1. After FACTLX computes n! (for $n \neq 0$), the word containing n! is assembled. A flowchart of the process appears in Fig. 17.2 in the book.

```
Locn.   Oper.    Var. Field

FACTL   MACRØ    N,Z
Q       SET      0
F       SET      1
        IFF      N
        FACTLX   N
Z       DEC      F
        END

FACTLX  MACRØ    N
Q       SET      Q+1
F       SET      F*Q
        IFF      Q-N
        FACTLX   N
        END
```

As an example of how this works, consider the call

```
FACTL   4
```

Following is a list of most of the pseudo-operations as they are generated in the assembly process during the recursive calling of FACTLX. The SET and IFF pseudo-operations are listed in the order of their generation:

```
Expansion of FACTL:                    Q       SET   0
                                       F       SET   1
                                               IFF   4

Expansion of FACTLX (1st time):        Q       SET   1
                                       F       SET   1
                                               IFF   -3

Expansion of FACTLX (2nd time):        Q       SET   2
                                       F       SET   2
                                               IFF   -2

Expansion of FACTLX (3rd time):        Q       SET   3
                                       F       SET   6
                                               IFF   -1

Expansion of FACTLX (4th time):        Q       SET   4
                                       F       SET   24
                                               IFF   0

Expansion of FACTL:                    ..001   DEC   24
```

Chapter 18
INTERPRETERS AND SIMULATION

(R)----------(387.5 - 391.2)----------

AN INTERPRETIVE PROGRAM

The following instruction is useful in Example 18.2.
LOAD COMPLEMENT OF ADDRESS IN INDEX (LAC Y) (+0535); 2 cycles. The 2's complement of the $C(Y)_{21-35}$ replaces the contents of the specified index register. The C(Y) is unchanged.

Example 18.2 Refer to the book for the introduction to an analysis and flowchart of this program. That material applies here with one modification. The complement of the C(10000) is placed in XR2, rather than the C(10000), as noted at 398.3. This is necessary because 7090 index registers work by decrementing. Notice that 100008 must be added to addresses in the program before they are used to set addresses or SIMAR, as in the MØVE, JUMPSR, and JPPMSR routines.

Locn.	Oper.	Var. Field	
	LAC	/Ø/10000,2	Place compl. of C(10000)
	CLA	/Ø/10001	in XR2
	STØ	SIMAR	Place starting addr in SIMAR
NEXT	CAL*	SIMAR	Obtain A-address by
	ANA	AMASK	masking out rest of
	ARS	18	instruction; place in
	STA	ADDRA	ADDRA
	CAL*	SIMAR	Obtain B-address...
	ANA	BMASK	
	ARS	9	
	STA	ADDRB	
	CAL*	SIMAR	Obtain C-address...
	ANA	CMASK	
	STA	ADDRC	

(Cont'd.)

```
Locn.     Oper.     Var. Field

          CAL*      SIMAR          Obtain operation code
          ANA       ØPMASK
          ARS       27
          PAX       0,1            Place op. code in XR1
          TRA*      SUBRT+7,1      Transfer on op. code
SUBRT                JPPMSR         7
                     JUMPSR         6
                     MØVESR         5
                     DIVSR          4
                     MULTSR         3
                     SUBTSR         2
                     ADDSR          1
                     HALTSR         0
RETURN    CLA       SIMAR          Return point after non-
          ADD       ØNE              tra. execution; modify
          STØ       SIMAR            SIMAR by 1
          TRA       NEXT

HALTSR    HTR
ADDSR     CLA*      ADDRA
          ADD*      ADDRB
          STØ*      ADDRC
          TRA       RETURN
SUBTSR    CLA*      ADDRA
          SUB*      ADDRB
          STØ*      ADDRC
          TRA       RETURN
MULTSR    LDQ*      ADDRA
          MPY*      ADDRB
          STQ*      ADDRC
          TRA       RETURN
DIVSR     LDQ*      ADDRA
          PXA       0,0
          LLS       0
          DVP*      ADDRB
          TRA       RETURN
MØVESR    CLA       ADDRB          Form size of block:
          SUB       ADDRA            B-A+1
          ADD       ØNE
          PAX       0,4            Place size in XR4
          ADD       ADDRA
          ADD       RELØCN         Add 10000 for relocation
          STA       MV1
          SUB       ADDRA
          ADD       ADDRC
          STA       MV2
```

(Cont'd.)

```
Locn.    Oper.    Var. Field
MV1      CLA      **,4          (B+1, which is A+size)
MV2      STØ      **,4          (B-A+C+1, which is
         TIX      *-2,4,1         C+size)
         TRA      RETURN
JUMPSR   CLA      ADDRA         Fetch new address for
         ADD      RELØCN          SIMAR
         STA      SIMAR
         TRA      NEXT
JPPMSR   CLA*     ADDRA         Test sign of C(ADDRA)
         TPL      TR1
         CLA      ADDRC         Set SIMAR to C-address
         ADD      RELØCN
         STA      SIMAR
         TRA      NEXT
TR1      CLA      ADDRB         Set SIMAR to B-address
         ADD      RELØCN
         STA      SIMAR
         TRA      NEXT

SIMAR                           Simulated AR
AMASK    ØCT      000777000000  Masks
BMASK    ØCT      000000777000
CMASK    ØCT      000000000777
ØPMASK   ØCT      007000000000
ADDRA             **,2          Registers for the 3
ADDRB             **,2            addresses
ADDRC             **,2
ØNE      DEC      1
RELØCN   ØCT      10000
```

(M)----------(391.7 - 394.8)----------

(The material in 391.7 - 394.8, though it applies to a program written for the DELTA 63, applies to a study of interpreters on the 7090 as well. The concepts are general, and a study of the material can be understood almost entirely even if details on coding are not.)

(R)----------(395.3 - 395.10)----------

Example 18.4 Write routines for a self-interpreter for the 7090 to simulate the instructions TXI, TSX, and TIX.

In the self-interpreter, the address, tag, and decrement portions of the instruction being executed interpretively are placed in ADDR, TAG, and DECR, respectively (in the address fields). Then control goes to the

individual routines. In addition, the SETTAG routine is executed at that time. Its purpose is to set into a memory location the tag of the instruction being executed; the address field of that word contains the address SIMXR-1. SIMXR heads a block of seven words; SIMXR+j-1 represents the simulated XRj (allowing for seven index registers). An indirect address reference to that word thus references the proper simulated XR. The 15 rightmost bits of the simulated XR represent the contents of that XR.

The routine for setting the index-register address:

```
Locn.    Oper.    Var. Field
SETTAG   CLA      BSIMXR
         ADD      TAG
         STØ      REFXR
         ...      .....
REFXR             **              (SIMXR+j-1)
BSIMXR            SIMXR-1
```

In the following routines, SIMAR, RETURN, and NEXT represent the same instructions as in Example 18.2; control passes to RETURN if a transfer is not executed; control passes to NEXT if a transfer is executed. The STA instruction is used to store a new value in the simulated XRs; by so doing, any arithmetic performed on the XR is effectively done modulo 100000_8 as required.

```
         Oper.    Var. Field
TXI:     CLA*     REFXR          Modify XR
         ADD      DECR
         STA*     REFXR
         CLA      ADDR           "Transfer"
         STØ      SIMAR
         TRA      NEXT

TSX:     CLA      SIMAR          Form 2's complement
         CAS      DECR             with decrement
         TRA      *+3            C(XR) gr. than decr.
         TRA      RETURN         C(XR) equal to decr.;
         TRA      RETURN           "go on"
         SUB      DECR           Modify C(XR) by decrement
         STA*     REFXR
         CLA      ADDR           "Transfer"
         STØ      SIMAR
         TRA      NEXT
```

The last three instructions in these three routines are identical and could be combined.

Chapter 19
PROGRAM DEBUGGING AND TESTING

(S)----------(At 402.3)----------

ASSEMBLER AIDS

In addition to the errors listed in the book, the FAP assembler also flags these errors:

6. Illegal indirect addressing.
7. Improper tag and decrement.
8. Errors in other pseudo-operations.
9. Relocation errors.

(R)----------(402.4 - 403.8)----------

Example 19.1 The following letters are used by the FAP assembler to flag errors:

U - undefined symbol
M - multiply-defined symbol
Ø - illegal operation code
G - error in data-generating card, such as ØCT or DEC
A - improper address or omitted address where required
T - improper tag or omitted tag where required
D - improper decrement or omitted decrement where required
I - illegal indirect addressing
P - illegal use of pseudo-operation
R - relocation error.

Other flags are given, under appropriate conditions.

The program below, taken from Example 8.7, is recoded with several errors that are flaggable by the assembler. The octal listing is given with error flags. Note that portions of octal words are omitted where errors are present.

	Location	Contents	Location	Oper.	Var. Field
	00200			ØRG	/Ø/200
	00201	+0560 00 1 04147	NEWØNE	LDQ	LIST+2000,1
T	00202	+0774 00 0 03720		AXT	2000
	00203	+0754 00 0 00000		PXA	0,0
U	00204	+0221 00 0		DVP	TEN
Ø	00205	00000 0 00000		XAC	
	00206	+0734 00 2 00000		PAX	0,2
	00207	+0500 00 2 00227		CLA	CTABLE+10,2
M	00210	+0400 00 0 00214		ADD	ØNE
	00211	+0501 00 2 00227		STØ	CTABLE+10,2
UD	00212	+2 00000 00201		TIX	NEWØNE,I
	00213	+0000 00 0 00000		HTR	
M	00214	+000000000001	ØNE	DEC	1
	00215		CTABLE	BSS	10
	00227		LIST	BSS	2000
M	04147	+0 00000 0 00001	ØNE	PZE	1

The errors made were as follows: (1) omission of the tag in the AXT instruction; (2) failure to define the symbol "TEN"; (3) mispunching of XCA as "XAC"; (4) multiply defining the symbol "ØNE"; (5) mispunching of the tag "1" as "I" in TIX; this appears as an undefined symbol ("I"); (6) omission of the decrement in the TIX instruction. Note that two flags appear on one line; two errors were made in one symbolic instruction.

In the event that part or all of a word cannot be assembled because of an error, the FAP assembler sometimes leaves blanks or, in the case of a multiply-defined symbol, assembles the earlier address. The resulting object deck will have zeros punched where blanks appear in the listing. Thus, at location 00204, the following word is assembled in the deck:

022100000000

If certain fatal errors occur in a program, an object deck is normally not produced and the program is not run at that time; such errors include those with flags U, M, and Ø. Other errors, called nonfatal, do not inhibit deck punching or immediate execution; these include those with flags A, T, and D. These latter "errors" may be intentional and the assembler permits a run. Optionally, a deck may be punched regardless, if appropriate indications is given. FAP provides, with the listing, a list of undefined and multiply-defined symbols.

The errors made in this example, like all coding errors can be corrected by the use of correction cards, described in Section 11.1, or by reassembly.

The following octal correction cards will correct the errors made in the program. They would be applied to the object deck.

Locn.	Oper.	Var. Field
00202	ØCT	077400103720
00204	ØCT	022100020000
00205	ØCT	013100000000
00212	ØCT	200001100201
20000	ØCT	000000000012

Most of these cards are easily understandable. Note that a word containing 10 (128) has been established at location 20000, a location outside the program.

(R)----------(403.9 - 404.2)----------

Example 19.2 The symbol reference table for the program coded in Example 8.7 is as follows:

Locn.	Symbol	References
00214	ØNE	00210,00214
04147	TEN	00204,04147
00227	LIST	00201,00227
00215	CTABLE	00207,00211,00215
00201	NEWØNE	00201,00212

(R)----------(406.4 - 406.7)----------

HELP AT THE CONSOLE

Example 19.3 Refer to the book for a description and analysis of this problem. There are, of course, only three index registers, in the 7090; this will not effect the description given. The instructions under consideration are the following:

```
TRA   02123,2
TRA   02234,4
```

(R)----------(409.4 - 409.10)----------

THE USE OF DUMPS

In the 7090 a special trapping feature is provided. If bits S, 1, and 2 of an instruction being executed contain the bits 1, 0, and 1, respectively, the instruction is interpreted as STR:

STORE LOCATION AND TRAP (STR); 2 cycles. The location of this instruction, plus one, replaces positions 21-35 of location 00000. The computer then takes its next instruction from location 00002. The contents of positions 3-35 of this instruction are not interpreted.

Through the use of STR, the monitor system is able to gain control. As the program deck is loaded, the SNAP cards are also loaded. As these are encountered by the monitor, the information on them is stored within the snapshot routine. A snapshot table is established with a list of addresses where snapshots are requested and with other appropriate information. The monitor places the STR prefix in bits S, 1, and 2, after saving the instructions' original prefixes.

During the running of the program, when control passes to an instruction containing an STR prefix, a trap occurs and control passes to 00002. At that location, a transfer instruction sends control to the snapshot routine. There a search is made of the snapshot table, and if a dump was requested at the address in location 00000 (from where control just came), the dump is given.* Then the instruction with the STR prefix must be executed. This is done remotely, within the monitor, where the proper prefix is combined with the other 33 bits of the instruction so that execution can occur. Finally, control returns to the program being run so that it may continue at the point from which control left it.

To assist the programmer when his program unintentionally sends control outside the block of executable instructions, the monitor, just prior to loading a program, places STR prefixes throughout memory (except for the monitor area). Programs loaded into memory of course write over some of these STR's, but the areas not so covered retain them; areas set aside by BSS pseudo-operations retain their STR's. If, then, control passes to a location outside the program (or possibly to within a block of data), a trap occurs and the monitor, recognizing the fact that no snapshot was requested at that location, stops the program, indicating where control went erroneously.

(R)----------(413.1 - 413.9)----------

THE USE OF MACRO-INSTRUCTIONS

Example 19.4 Write a short program containing a loop in which a debugging macro-call is placed

---

*Actually, 00000 contains one more than that address.

The program:

```
Locn.   Oper.     Var. Field

        AXT       100,1
START   CLA       SUM
        ADD       NUMBRS+100,1
        STØ       SUM
        PRTBLK    Ø,SUM,SUM,UNTIL,5
        TIX       START,1,1
        HTR
```

The resultant printout, which merely shows the contents of SUM, might appear as follows:

```
000000000010
000000000034
000000000055
000000000067
000000000102
```

The next example shows a more complex printout, the dump of two different blocks.

Example 19.5 Consider Example 8.6, which sorts a list of 1000 numbers into two blocks, PØSLST and NEGLST. Assume that the following two cards are inserted in the program immediately preceding the instruction at MØD:

```
PRTBLK  D,PØSLST,POSLST+3,UNTIL,4
PRTBLK  D,NEGLST,NEGLST+2,UNTIL,3
```

The request here is for a decimal output, assumed to be given with three or four words to a line. The resultant output might be as follows:

```
+23498        0        0        0
     0        0        0
+23498        0        0        0
  -232        0        0
+23498        0        0        0
  -232   -86001        0
+23498      +77        0        0
```

Note that four dumps of PØSLST (four numbers on a line) were given, while three dumps of NEGLST (three numbers on a line) were given.

(R)----------(414.3 - 415.3)----------

Example 19.6 Write a macro-instruction that will provide the following information, all in octal, when control passes to it:

1. The C(AC) and the C(MQ):
2. the contents of the three index registers;
3. the contents of any three specified words in memory;
4. the contents of a block of any size in memory.

A typical call is:

```
DUMP    WØRD,XXX,SUM,LIST,LIST+10
```

which means "Dump the AC, the MQ, the three XRs, locations WØRD, XX, and SUM, and the block from LIST through LIST+10."

In the macro-instruction that follows, the index registers, the AC, and the MQ are first saved and subsequently restored. The PRINTL macro-instruction of Example 17.13 is used to print a list. A new macro-instruction, PRINTB, is used to print a block.

```
Locn.   Oper.    Var. Field
DUMP    MACRØ    A,B,C,L,M
        SXA      Q,1              Save XRs
        SXA      Q+1,2
        SXA      Q+2,4
        SLW      Q+3              These 3 instructions save
        ARS      36                 all 38 bits of the AC
        STØ      Q+4
        STQ      Q+5              Save MQ
        PRINTL   (Q,Q1,Q2)        Print XRs
        PRINTL   (A,B,C)          Print 3 words
        PRINTB   L,M              Print block
        CLA      Q+4              These 3 instructions
        ALS      36                 restore the complete AC
        ØRA      Q+3
        LDQ      Q+5
        LXA      Q,1
        LXA      Q,1,2
        LXA      Q+2,4
        END
        ...
Q       BSS      6
N3777   ØCT      377777777777
```

INDEX

INDEX TO INSTRUCTIONS

STI	Store indicators	129
STO	Store	3
STQ	Store MQ	6
STZ	Store zero	19
SUB	Subtract	3
SXA	Store index in address	35
SXD	Store index in decrement	35
TIX	Transfer on index	35
TLQ	Transfer on low MQ	94
TMI	Transfer on minus	10
TNZ	Transfer on nonzero	10
TPL	Transfer on plus	10
TRA	Transfer	10
TSX	Transfer and set index	53
TXH	Transfer on index high	31
TXI	Transfer with index incremented	30
TXL	Transfer on index low or equal	30
XCA	Exchange AC and MQ	6
ZET	Zero test	46