


1.3. MIX

IN MANY PLACES throughout this book we will have occasion to refer to a computer's internal machine language. The machine we use is a mythical computer called "MIX." MIX is very much like nearly every computer of the 1960s and 1970s, except that it is, perhaps, nicer. The language of MIX has been designed to be powerful enough to allow brief programs to be written for most algorithms, yet simple enough so that its operations are easily learned.

The reader is urged to study this section carefully, since MIX language appears in so many parts of this book. There should be no hesitation about learning a machine language; indeed, the author once found it not uncommon to be writing programs in a half dozen different machine languages during the same week! Everyone with more than a casual interest in computers will probably get to know at least one machine language sooner or later. MIX has been specially designed to preserve the simplest aspects of historic computers, so that its characteristics are easy to assimilate.

 However, it must be admitted that MIX is now quite obsolete. Therefore MIX will be replaced in subsequent editions of this book by a new machine called MMIX, the 2009. MMIX will be a so-called reduced instruction set computer (RISC), which will do arithmetic on 64-bit words. It will be even nicer than MIX, and it will be similar to machines that have become dominant during the 1990s.

The task of converting everything in this book from MIX to MMIX will take a long time; volunteers are solicited to help with that conversion process. Meanwhile, the author hopes that people will be content to live for a few more years with the old-fashioned MIX architecture — which is still worth knowing, because it helps to provide a context for subsequent developments.

1.3.1. Description of MIX

MIX is the world's first polyunsaturated computer. Like most machines, it has an identifying number — the 1009. This number was found by taking 16 actual computers very similar to MIX and on which MIX could easily be simulated, then averaging their numbers with equal weight:

$$\lfloor (360 + 650 + 709 + 7070 + U3 + SS80 + 1107 + 1604 + G20 + B220 + S2000 + 920 + 601 + H800 + PDP-4 + II)/16 \rfloor = 1009. \quad (1)$$

The same number may also be obtained in a simpler way by taking Roman numerals.

MIX has a peculiar property in that it is both binary and decimal at the same time. MIX programmers don't actually know whether they are programming a machine with base 2 or base 10 arithmetic. Therefore algorithms written in MIX can be used on either type of machine with little change, and MIX can be simulated easily on either type of machine. Programmers who are accustomed to a binary machine can think of MIX as binary; those accustomed to decimal may regard MIX as decimal. Programmers from another planet might choose to think of MIX as a ternary computer.

Words. The basic unit of MIX data is a *byte*. Each byte contains an *unspecified* amount of information, but it must be capable of holding at least 64 distinct values. That is, we know that any number between 0 and 63, inclusive, can be contained in one byte. Furthermore, each byte contains *at most* 100 distinct values. On a binary computer a byte must therefore be composed of six bits; on a decimal computer we have two digits per byte.*

Programs expressed in MIX's language should be written so that no more than sixty-four values are ever assumed for a byte. If we wish to treat the number 80, we should always leave two adjacent bytes for expressing it, even though one byte is sufficient on a decimal computer. *An algorithm in MIX should work properly regardless of how big a byte is.* Although it is quite possible to write programs that depend on the byte size, such actions are anathema to the spirit of this book; the only legitimate programs are those that would give correct results with all byte sizes. It is usually not hard to abide by this ground rule, and we will thereby find that programming a decimal computer isn't so different from programming a binary one after all.

Two adjacent bytes can express the numbers 0 through 4,095.

Three adjacent bytes can express the numbers 0 through 262,143.

Four adjacent bytes can express the numbers 0 through 16,777,215.

Five adjacent bytes can express the numbers 0 through 1,073,741,823.

A computer word consists of five bytes and a sign. The sign portion has only two possible values, + and −.

Registers. There are nine registers in MIX (see Fig. 13):

The A-register (Accumulator) consists of five bytes and a sign.

The X-register (Extension), likewise, comprises five bytes and a sign.

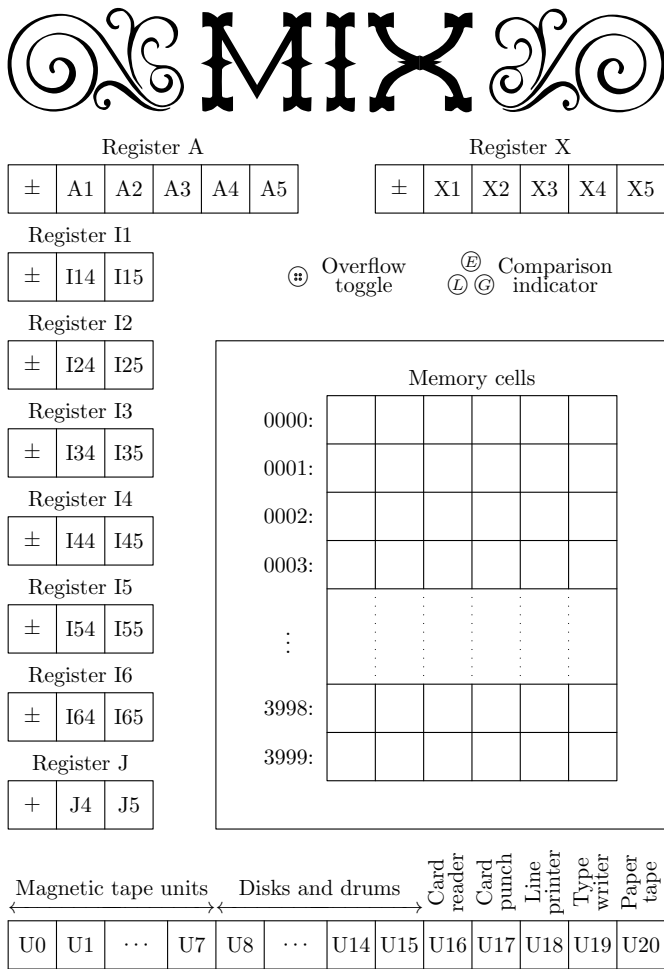
The I-registers (Index registers) I1, I2, I3, I4, I5, and I6 each hold two bytes together with a sign.

The J-register (Jump address) holds two bytes; it behaves as if its sign is always +.

We shall use a small letter “r”, prefixed to the name, to identify a MIX register. Thus, “rA” means “register A.”

The A-register has many uses, especially for arithmetic and for operating on data. The X-register is an extension on the “right-hand side” of rA, and it is used in connection with rA to hold ten bytes of a product or dividend, or it can be used to hold information shifted to the right out of rA. The index registers rI1, rI2, rI3, rI4, rI5, and rI6 are used primarily for counting and for referencing variable memory addresses. The J-register always holds the address of the instruction following the most recent “jump” operation, and it is primarily used in connection with subroutines.

* Since 1975 or so, the word “byte” has come to mean a sequence of precisely eight binary digits, capable of representing the numbers 0 to 255. Real-world bytes are therefore larger than the bytes of the hypothetical MIX machine; indeed, MIX's old-style bytes are just barely bigger than nybbles. When we speak of bytes in connection with MIX we shall confine ourselves to the former sense of the word, harking back to the days when bytes were not yet standardized.



Most of the instructions allow a programmer to use only part of a word if desired. In such cases a nonstandard “field specification” can be given. The allowable fields are those that are adjacent in a computer word, and they are represented by (L:R), where L is the number of the left-hand part and R is the number of the right-hand part of the field. Examples of field specifications are:

- (0:0), the sign only.
- (0:2), the sign and the first two bytes.
- (0:5), the whole word; this is the most common field specification.
- (1:5), the whole word except for the sign.
- (4:4), the fourth byte only.
- (4:5), the two least significant bytes.

The use of field specifications varies slightly from instruction to instruction, and it will be explained in detail for each instruction where it applies. Each field specification (L:R) is actually represented inside the machine by the single number $8L + R$; notice that this number fits easily in one byte.

Instruction format. Computer words used for instructions have the following form:

0	1	2	3	4	5
\pm	A	A	I	F	C

(3)

The rightmost byte, C, is the *operation code* telling what operation is to be performed. For example, $C = 8$ specifies the operation LDA, “load the A-register.”

The F-byte holds a *modification* of the operation code. It is usually a field specification $(L:R) = 8L + R$; for example, if $C = 8$ and $F = 11$, the operation is “load the A-register with the (1:3) field.” Sometimes F is used for other purposes; on input-output instructions, for example, F is the number of the relevant input or output unit.

The left-hand portion of the instruction, $\pm AA$, is the *address*. (Notice that the sign is part of the address.) The I-field, which comes next to the address, is the *index specification*, which may be used to modify the effective address. If $I = 0$, the address $\pm AA$ is used without change; otherwise I should contain a number i between 1 and 6, and the contents of index register Ii are added algebraically to $\pm AA$ before the instruction is carried out; the result is used as the address. This indexing process takes place on *every* instruction. We will use the letter M to indicate the address after any specified indexing has occurred. (If the addition of the index register to the address $\pm AA$ yields a result that does not fit in two bytes, the value of M is undefined.)

In most instructions, M will refer to a memory cell. The terms “memory cell” and “memory location” are used almost interchangeably in this book. We assume that there are 4000 memory cells, numbered from 0 to 3999; hence every memory location can be addressed with two bytes. For every instruction in which M refers to a memory cell we must have $0 \leq M \leq 3999$, and in this case we will write $\text{CONTENTS}(M)$ to denote the value stored in memory location M.

On certain instructions, the “address” M has another significance, and it may even be negative. Thus, one instruction adds M to an index register, and such an operation takes account of the sign of M.

Notation. To discuss instructions in a readable manner, we will use the notation

OP

ADDRESS, I (F)

(4)

to denote an instruction like (3). Here OP is a symbolic name given to the operation code (the C-part) of the instruction; ADDRESS is the ±AA portion; I and F represent the I- and F-fields, respectively.

If I is zero, the ‘,I’ is omitted. If F is the *normal* F-specification for this particular operator, the ‘(F)’ need not be written. The normal F-specification for almost all operators is (0:5), representing a whole word. If a different F is normal, it will be mentioned explicitly when we discuss a particular operator.

For example, the instruction to load a number into the accumulator is called LDA and it is operation code number 8. We have

Conventional representation	Actual numeric instruction					
LDA 2000,2(0:3)	<table><tr><td>+</td><td>2000</td><td>2</td><td>3</td><td>8</td></tr></table>	+	2000	2	3	8
+	2000	2	3	8		
LDA 2000,2(1:3)	<table><tr><td>+</td><td>2000</td><td>2</td><td>11</td><td>8</td></tr></table>	+	2000	2	11	8
+	2000	2	11	8		
LDA 2000(1:3)	<table><tr><td>+</td><td>2000</td><td>0</td><td>11</td><td>8</td></tr></table>	+	2000	0	11	8
+	2000	0	11	8		
LDA 2000	<table><tr><td>+</td><td>2000</td><td>0</td><td>5</td><td>8</td></tr></table>	+	2000	0	5	8
+	2000	0	5	8		
LDA -2000,4	<table><tr><td>-</td><td>2000</td><td>4</td><td>5</td><td>8</td></tr></table>	-	2000	4	5	8
-	2000	4	5	8		

(5)

The instruction ‘LDA 2000,2(0:3)’ may be read “Load A with the contents of location 2000 indexed by 2, the zero-three field.”

To represent the numerical contents of a MIX word, we will always use a box notation like that above. Notice that in the word

+	2000	2	3	8
---	------	---	---	---

the number +2000 is shown filling two adjacent bytes and sign; the actual contents of byte (1:1) and of byte (2:2) will vary from one MIX computer to another, since byte size is variable. As a further example of this notation for MIX words, the diagram

-	10000	3000
---	-------	------

represents a word with two fields, a three-byte-plus-sign field containing −10000 and a two-byte field containing 3000. When a word is split into more than one field, it is said to be “packed.”

Rules for each instruction. The remarks following (3) above have defined the quantities M, F, and C for every word used as an instruction. We will now define the actions corresponding to each instruction.

Loading operators.

• **LDA** (load A). C = 8; F = field.

The specified field of **CONTENTS**(M) replaces the previous contents of register A.

On all operations where a partial field is used as an input, the sign is used if it is a part of the field, otherwise the sign + is understood. The field is shifted over to the right-hand part of the register as it is loaded.

Examples: If F is the normal field specification (0:5), everything in location M is copied into rA. If F is (1:5), the absolute value of **CONTENTS**(M) is loaded with a plus sign. If M contains an *instruction* word and if F is (0:2), the “±AA” field is loaded as

±	0	0	0	A	A
---	---	---	---	---	---

.

Suppose location 2000 contains the word

-	80	3	5	4
---	----	---	---	---

; (6)

then we get the following results from loading various partial fields:

Instruction		Contents of rA afterwards				
LDA	2000	-	80	3	5	4
LDA	2000(1:5)	+	80	3	5	4
LDA	2000(3:5)	+	0	0	3	5
LDA	2000(0:3)	-	0	0	80	3
LDA	2000(4:4)	+	0	0	0	5
LDA	2000(0:0)	-	0	0	0	0
LDA	2000(1:1)	+	0	0	0	?

(The last example has a partially unknown effect, since byte size is variable.)

• **LDX** (load X). C = 15; F = field.

This is the same as **LDA**, except that rX is loaded instead of rA.

• **LDi** (load i). C = 8 + i; F = field.

This is the same as **LDA**, except that r*i* is loaded instead of rA. An index register contains only two bytes (not five) and a sign; bytes 1, 2, 3 are always assumed to be zero. The **LDi** instruction is undefined if it would result in setting bytes 1, 2, or 3 to anything but zero.

In the description of all instructions, “i” stands for an integer, 1 ≤ i ≤ 6. Thus, **LDi** stands for six different instructions: **LD1**, **LD2**, ..., **LD6**.

• **LDAN** (load A negative). C = 16; F = field.

• **LDXN** (load X negative). C = 23; F = field.

• **LDiN** (load i negative). C = 16 + i; F = field.

These eight instructions are the same as **LDA**, **LDX**, **LDi**, respectively, except that the *opposite* sign is loaded.

Storing operators.

• **STA** (store A). C = 24; F = field.
A portion of the contents of rA replaces the field of CONTENTS(M) specified by F. The other parts of CONTENTS(M) are unchanged.

On a *store* operation the field F has the opposite significance from the *load* operation: The number of bytes in the field is taken from the right-hand portion of the register and shifted *left* if necessary to be inserted in the proper field of CONTENTS(M). The sign is not altered unless it is part of the field. The contents of the register are not affected.

Examples: Suppose that location 2000 contains

-	1	2	3	4	5
---	---	---	---	---	---

and register A contains

+	6	7	8	9	0
---	---	---	---	---	---

 .

Then:

Instruction		Contents of location 2000 afterwards					
STA	2000	+	6	7	8	9	0
STA	2000(1:5)	-	6	7	8	9	0
STA	2000(5:5)	-	1	2	3	4	0
STA	2000(2:2)	-	1	0	3	4	5
STA	2000(2:3)	-	1	9	0	4	5
STA	2000(0:1)	+	0	2	3	4	5

• **STX** (store X). C = 31; F = field.
Same as **STA**, except that rX is stored rather than rA.

• **STi** (store *i*). C = 24 + *i*; F = field.
Same as **STA**, except that rLi is stored rather than rA. Bytes 1, 2, 3 of an index register are zero; thus if rI1 contains

±	<i>m</i>	<i>n</i>
---	----------	----------

 ,

it behaves as though it were

±	0	0	0	<i>m</i>	<i>n</i>
---	---	---	---	----------	----------

 .

• **STJ** (store J). C = 32; F = field.
Same as **STi**, except that rJ is stored and its sign is always +.

With **STJ** the normal field specification for F is (0:2), not (0:5). This is natural, since **STJ** is almost always done into the address field of an instruction.

• **STZ** (store zero). C = 33; F = field.
Same as **STA**, except that plus zero is stored. In other words, the specified field of CONTENTS(M) is cleared to zero.

Arithmetic operators. On the add, subtract, multiply, and divide operations, a field specification is allowed. A field specification of “(0:6)” can be used to indicate a “floating point” operation (see Section 4.2), but few of the programs we will write for MIX will use this feature, since we will primarily be concerned with algorithms on integers.

The standard field specification is, as usual, (0:5). Other fields are treated as in LDA. We will use the letter V to indicate the specified field of CONTENTS(M); thus, V is the value that would have been loaded into register A if the operation code were LDA.

- **ADD.** C = 1; F = field.

V is added to rA. If the magnitude of the result is too large for register A, the overflow toggle is set on, and the remainder of the addition appearing in rA is as though a “1” had been carried into another register to the left of rA. (Otherwise the setting of the overflow toggle is unchanged.) If the result is zero, the sign of rA is unchanged.

Example: The sequence of instructions below computes the sum of the five bytes of register A.

```

STA 2000
LDA 2000(5:5)
ADD 2000(4:4)
ADD 2000(3:3)
ADD 2000(2:2)
ADD 2000(1:1)

```

This is sometimes called “sideways addition.”

Overflow will occur in some MIX computers when it would not occur in others, because of the variable definition of byte size. We have not said that overflow will occur definitely if the value is greater than 1073741823; overflow occurs when the magnitude of the result is greater than the contents of five bytes, depending on the byte size. One can still write programs that work properly and that give the same final answers, regardless of the byte size.

- **SUB** (subtract). C = 2; F = field.

V is subtracted from rA. (Equivalent to ADD but with $-V$ in place of V.)

- **MUL** (multiply). C = 3; F = field.

The 10-byte product, V times rA, replaces registers A and X. The signs of rA and rX are both set to the algebraic sign of the product (namely, + if the signs of V and rA were the same, – if they were different).

- **DIV** (divide). C = 4; F = field.

The value of rA and rX, treated as a 10-byte number rAX with the sign of rA, is divided by the value V. If $V = 0$ or if the quotient is more than five bytes in magnitude (this is equivalent to the condition that $|rA| \geq |V|$), registers A and X are filled with undefined information and the overflow toggle is set on. Otherwise the quotient $\pm \lfloor |rAX/V| \rfloor$ is placed in rA and the remainder $\pm(|rAX| \bmod |V|)$ is placed in rX. The sign of rA afterwards is the algebraic sign of the quotient

(namely, + if the signs of V and rA were the same, − if they were different). The sign of rX afterwards is the previous sign of rA.

Examples of arithmetic instructions: In most cases, arithmetic is done only with MIX words that are single five-byte numbers, not packed with several fields. It is, however, possible to operate arithmetically on packed MIX words, if some caution is used. The following examples should be studied carefully. (As before, ? designates an unknown value.)

ADD	1000	+	1234	1	150	rA before
		+	100	5	50	Cell 1000
		+	1334	6	200	rA after

SUB	1000	−	1234	0	0	9	rA before
		−	2000	150	0		Cell 1000
		+	766	149	?		rA after

MUL	1000	+	1	1	1	1	1	rA before
		+	1	1	1	1	1	Cell 1000
		+	0	1	2	3	4	rA after
		+	5	4	3	2	1	rX after

MUL	1000(1:1)	−				112	rA before	
		?	2	?	?	?	?	Cell 1000
		−				0	rA after	
		−				224	rX after	

MUL	1000	−	50	0	112	4	rA before	
		−	2	0	0	0	0	Cell 1000
		+	100	0	224			rA after
		+	8	0	0	0	0	rX after

DIV	1000	+				0	rA before
		?				17	rX before
		+				3	Cell 1000
		+				5	rA after
		+				2	rX after

	-					0	rA before
	+	1	2	3	5	0 3 1	rX before
	-	0	0	0	2	0	Cell 1000
DIV 1000	+	0	6	1	7	? ?	rA after
	-	0	0	0	? 1		rX after

(These examples have been prepared with the philosophy that it is better to give a complete, baffling description than an incomplete, straightforward one.)

Address transfer operators. In the following operations, the (possibly indexed) “address” M is used as a signed number, not as the address of a cell in memory.

- **ENTA** (enter A). C = 48; F = 2.

The quantity M is loaded into rA. The action is equivalent to ‘LDA’ from a memory word containing the signed value of M. If M = 0, the sign of the instruction is loaded.

Examples: ‘ENTA 0’ sets rA to zeros, with a + sign. ‘ENTA 0,1’ sets rA to the current contents of index register 1, except that −0 is changed to +0. ‘ENTA −0,1’ is similar, except that +0 is changed to −0.

- **ENTX** (enter X). C = 55; F = 2.
- **ENTi** (enter *i*). C = 48 + *i*; F = 2.

Analogous to **ENTA**, loading the appropriate register.

- **ENNA** (enter negative A). C = 48; F = 3.
- **ENNX** (enter negative X). C = 55; F = 3.
- **ENNi** (enter negative *i*). C = 48 + *i*; F = 3.

Same as **ENTA**, **ENTX**, and **ENTi**, except that the opposite sign is loaded.

Example: ‘ENN3 0,3’ replaces rI3 by its negative, although −0 remains −0.

- **INCA** (increase A). C = 48; F = 0.

The quantity M is added to rA; the action is equivalent to ‘ADD’ from a memory word containing the value of M. Overflow is possible and it is treated just as in **ADD**.

Example: ‘INCA 1’ increases the value of rA by one.

- **INCX** (increase X). C = 55; F = 0.

The quantity M is added to rX. If overflow occurs, the action is equivalent to **ADD**, except that rX is used instead of rA. Register A is never affected by this instruction.

- **INCi** (increase *i*). C = 48 + *i*; F = 0.

Add M to rIi. Overflow must not occur; if M + rIi doesn’t fit in two bytes, the result of this instruction is undefined.

- DECA (decrease A). C = 48; F = 1.
- DECX (decrease X). C = 55; F = 1.
- DECI (decrease *i*). C = 48 + *i*; F = 1.

These eight instructions are the same as INCA, INCX, and INCI, respectively, except that M is subtracted from the register rather than added.

Notice that the operation code C is the same for ENTA, ENNA, INCA, and DECA; the F-field is used to distinguish the various operations from each other.

Comparison operators. MIX's comparison operators all compare the value contained in a register with a value contained in memory. The comparison indicator is then set to LESS, EQUAL, or GREATER according to whether the value of the register is less than, equal to, or greater than the value of the memory cell. A minus zero is *equal to* a plus zero.

- CMPA (compare A). C = 56; F = field.

The specified field of rA is compared with the *same* field of CONTENTS(M). If F does not include the sign position, the fields are both considered nonnegative; otherwise the sign is taken into account in the comparison. (An equal comparison always occurs when F is (0:0), since minus zero equals plus zero.)

- CMPX (compare X). C = 63; F = field.

This is analogous to CMPA.

- CMPi (compare *i*). C = 56 + *i*; F = field.

Analogous to CMPA. Bytes 1, 2, and 3 of the index register are treated as zero in the comparison. (Thus if F = (1:2), the result cannot be GREATER.)

Jump operators. Instructions are ordinarily executed in sequential order; in other words, the command that is performed after the command in location P is usually the one found in location P + 1. But several “jump” instructions allow this sequence to be interrupted. When a typical jump takes place, the J-register is set to the address of the next instruction (that is, to the address of the instruction that would have been next if we hadn't jumped). A “store J” instruction then can be used by the programmer, if desired, to set the address field of another command that will later be used to return to the original place in the program. The J-register is changed whenever a jump actually occurs in a program, except when the jump operator is JSJ, and it is never changed by non-jumps.

- JMP (jump). C = 39; F = 0.

Unconditional jump: The next instruction is taken from location M.

- JSJ (jump, save J). C = 39; F = 1.

Same as JMP except that the contents of rJ are unchanged.

- JOV (jump on overflow). C = 39; F = 2.

If the overflow toggle is on, it is turned off and a JMP occurs; otherwise nothing happens.

- JNOV (jump on no overflow). C = 39; F = 3.

If the overflow toggle is off, a JMP occurs; otherwise it is turned off.

- JL, JE, JG, JGE, JNE, JLE (jump on less, equal, greater, greater-or-equal, unequal, less-or-equal). C = 39; F = 4, 5, 6, 7, 8, 9, respectively.
Jump if the comparison indicator is set to the condition indicated. For example, JNE will jump if the comparison indicator is LESS or GREATER. The comparison indicator is not changed by these instructions.
- JAN, JAZ, JAP, JANN, JANZ, JANP (jump A negative, zero, positive, nonnegative, nonzero, nonpositive). C = 40; F = 0, 1, 2, 3, 4, 5, respectively.
If the contents of rA satisfy the stated condition, a JMP occurs, otherwise nothing happens. “Positive” means *greater* than zero (not zero); “nonpositive” means the opposite, namely zero or negative.
- JXN, JXZ, JXP, JXNN, JXNZ, JXNP (jump X negative, zero, positive, nonnegative, nonzero, nonpositive). C = 47; F = 0, 1, 2, 3, 4, 5, respectively.
- JiN, JiZ, JiP, JiNN, JiNZ, JiNP (jump *i* negative, zero, positive, nonnegative, nonzero, nonpositive). C = 40 + *i*; F = 0, 1, 2, 3, 4, 5, respectively. These 42 instructions are analogous to the corresponding operations for rA.

Miscellaneous operators.

- SLA, SRA, SLAX, SRAX, SLC, SRC (shift left A, shift right A, shift left AX, shift right AX, shift left AX circularly, shift right AX circularly). C = 6; F = 0, 1, 2, 3, 4, 5, respectively.
These six are the “shift” commands, in which M specifies a number of MIX bytes to be shifted left or right; M must be nonnegative. SLA and SRA do not affect rX; the other shifts affect both registers A and X as though they were a single 10-byte register. With SLA, SRA, SLAX, and SRAX, zeros are shifted into the register at one side, and bytes disappear at the other side. The instructions SLC and SRC call for a “circulating” shift, in which the bytes that leave one end enter in at the other end. Both rA and rX participate in a circulating shift. The signs of registers A and X are not affected in any way by any of the shift commands.

Examples:		Register A						Register X					
	Initial contents	+	1	2	3	4	5	-	6	7	8	9	10
	SRAX 1	+	0	1	2	3	4	-	5	6	7	8	9
	SLA 2	+	2	3	4	0	0	-	5	6	7	8	9
	SRC 4	+	6	7	8	9	2	-	3	4	0	0	5
	SRA 2	+	0	0	6	7	8	-	3	4	0	0	5
	SLC 501	+	0	6	7	8	3	-	4	0	0	5	0

- MOVE. C = 7; F = number, normally 1.
The number of words specified by F is moved, starting from location M to the location specified by the contents of index register 1. The transfer occurs one word at a time, and rI1 is increased by the value of F at the end of the operation. If F = 0, nothing happens.
Care must be taken when there’s overlap between the locations involved; for example, suppose that F = 3 and M = 1000. Then if rI1 = 999, we transfer

CONTENTS(1000) to CONTENTS(999), CONTENTS(1001) to CONTENTS(1000), and CONTENTS(1002) to CONTENTS(1001); nothing unusual occurred here. But if r11 were 1001 instead, we would move CONTENTS(1000) to CONTENTS(1001), then CONTENTS(1001) to CONTENTS(1002), then CONTENTS(1002) to CONTENTS(1003), so we would have moved the *same* word CONTENTS(1000) into three places.

- NOP (no operation). C = 0.

No operation occurs, and this instruction is bypassed. F and M are ignored.

- HLT (halt). C = 5; F = 2.

The machine stops. When the computer operator restarts it, the net effect is equivalent to NOP.

Input-output operators. MIX has a fair amount of input-output equipment (all of which is optional at extra cost). Each device is given a number as follows:

Unit number	Peripheral device	Block size
t	Tape unit number t ($0 \leq t \leq 7$)	100 words
d	Disk or drum unit number d ($8 \leq d \leq 15$)	100 words
16	Card reader	16 words
17	Card punch	16 words
18	Line printer	24 words
19	Typewriter terminal	14 words
20	Paper tape	14 words

Not every MIX installation will have all of this equipment available; we will occasionally make appropriate assumptions about the presence of certain devices. Some devices may not be used both for input and for output. The number of words mentioned in the table above is a fixed block size associated with each unit.

Input or output with magnetic tape, disk, or drum units reads or writes full words (five bytes and a sign). Input or output with units 16 through 20, however, is always done in a *character code* where each byte represents one alphameric character. Thus, five characters per MIX word are transmitted. The character code is given at the top of Table 1, which appears at the close of this section and on the end papers of this book. The code 00 corresponds to ‘ \perp ’, which denotes a *blank space*. Codes 01–29 are for the letters A through Z with a few Greek letters thrown in; codes 30–39 represent the digits 0, 1, . . . , 9; and further codes 40, 41, . . . represent punctuation marks and other special characters. (MIX’s character set harks back to the days before computers could cope with lowercase letters.) We cannot use character code to read in or write out all possible values that a byte may have, since certain combinations are undefined. Moreover, some input-output devices may be unable to handle all the symbols in the character set; for example, the symbols ‘ \circ ’ and ‘ \sim ’ that appear amid the letters will perhaps not be acceptable to the card reader. When character-code input is being done, the signs of all words are set to +; on output, signs are ignored. If a typewriter is used for input, the “carriage return” that is typed at the end of each line causes the remainder of that line to be filled with blanks.

The disk and drum units are external memory devices each containing 100-word blocks. On every IN, OUT, or IOC instruction as defined below, the particular 100-word block referred to by the instruction is specified by the current contents of rX, which should not exceed the capacity of the disk or drum involved.

- IN (input). C = 36; F = unit.

This instruction initiates the transfer of information from the input unit specified into consecutive locations starting with M. The number of locations transferred is the block size for this unit (see the table above). The machine will wait at this point if a preceding operation for the same unit is not yet complete. The transfer of information that starts with this instruction will not be complete until an unknown future time, depending on the speed of the input device, so a program must not refer to the information in memory until then. It is improper to attempt to read any block from magnetic tape that follows the latest block written on that tape.

- OUT (output). C = 37; F = unit.

This instruction starts the transfer of information from memory locations starting at M to the output unit specified. The machine waits until the unit is ready, if it is not initially ready. The transfer will not be complete until an unknown future time, depending on the speed of the output device, so a program must not alter the information in memory until then.

- IOC (input-output control). C = 35; F = unit.

The machine waits, if necessary, until the specified unit is not busy. Then a control operation is performed, depending on the particular device being used. The following examples are used in various parts of this book:

Magnetic tape: If M = 0, the tape is rewound. If M < 0 the tape is skipped backward -M blocks, or to the beginning of the tape, whichever comes first. If M > 0, the tape is skipped forward; it is improper to skip forward over any blocks following the one last written on that tape.

For example, the sequence 'OUT 1000(3); IOC -1(3); IN 2000(3)' writes out one hundred words onto tape 3, then reads it back in again. Unless the tape reliability is questioned, the last two instructions of that sequence are only a slow way to move words 1000-1099 to locations 2000-2099. The sequence 'OUT 1000(3); IOC +1(3)' is improper.

Disk or drum: M should be zero. The effect is to position the device according to rX so that the next IN or OUT operation on this unit will take less time if it uses the same rX setting.

Line printer: M should be zero. 'IOC 0(18)' skips the printer to the top of the following page.

Paper tape: M should be zero. 'IOC 0(20)' rewinds the tape.

- JRED (jump ready). C = 38; F = unit.

A jump occurs if the specified unit is ready, that is, finished with the preceding operation initiated by IN, OUT, or IOC.

- JBUS (jump busy). C = 34; F = unit.

Analogous to JRED, but the jump occurs when the specified unit is *not* ready.

Example: In location 1000, the instruction ‘JBUS 1000(16)’ will be executed repeatedly until unit 16 is ready.

The simple operations above complete MIX’s repertoire of input-output instructions. There is no “tape check” indicator, etc., to cover exceptional conditions on the peripheral devices. Any such condition (e.g., paper jam, unit turned off, out of tape, etc.) causes the unit to remain busy, a bell rings, and the skilled computer operator fixes things manually using ordinary maintenance procedures. Some more complicated peripheral units, which are more expensive and more representative of contemporary equipment than the fixed-block-size tapes, drums, and disks described here, are discussed in Sections 5.4.6 and 5.4.9.

Conversion Operators.

- NUM (convert to numeric). C = 5; F = 0.

This operation is used to change the character code into numeric code. M is ignored. Registers A and X are assumed to contain a 10-byte number in character code; the NUM instruction sets the magnitude of rA equal to the numerical value of this number (treated as a decimal number). The value of rX and the sign of rA are unchanged. Bytes 00, 10, 20, 30, 40, ... convert to the digit zero; bytes 01, 11, 21, ... convert to the digit one; etc. Overflow is possible, and in this case the remainder modulo b^5 is retained, where b is the byte size.

- CHAR (convert to characters). C = 5; F = 1.

This operation is used to change numeric code into character code suitable for output to punched cards or tape or the line printer. The value in rA is converted into a 10-byte decimal number that is put into registers A and X in character code. The signs of rA and rX are unchanged. M is ignored.

Examples:

	Register A						Register X					
Initial contents	-	00	00	31	32	39	+	37	57	47	30	30
NUM 0	-				12977700		+	37	57	47	30	30
INCA 1	-				12977699		+	37	57	47	30	30
CHAR 0	-	30	30	31	32	39	+	37	37	36	39	39

Timing. To give quantitative information about the efficiency of MIX programs, each of MIX’s operations is assigned an *execution time* typical of vintage-1970 computers.

ADD, SUB, all LOAD operations, all STORE operations (including STZ), all shift commands, and all comparison operations take *two units* of time. MOVE requires one unit plus two for each word moved. MUL, NUM, CHAR each require 10 units and DIV requires 12. The execution time for floating point operations is specified in Section 4.2.1. All remaining operations take one unit of time, plus the time the computer may be idle on the IN, OUT, IOC, or HLT instructions.

Notice in particular that ENTA takes one unit of time, while LDA takes two units. The timing rules are easily remembered because of the fact that, except

for shifts, conversions, MUL, and DIV, the number of time units equals the number of references to memory (including the reference to the instruction itself).

MIX’s basic unit of time is a relative measure that we will denote simply by u . It may be regarded as, say, 10 microseconds (for a relatively inexpensive computer) or as 10 nanoseconds (for a relatively high-priced machine).

Example: The sequence LDA 1000; INCA 1; STA 1000 takes exactly $5u$.

*And now I see with eye serene
The very pulse of the machine.*

— WILLIAM WORDSWORTH,
She Was a Phantom of Delight (1804)

Summary. We have now discussed all the features of MIX, except for its “GO button,” which is discussed in exercise 26. Although MIX has nearly 150 different operations, they fit into a few simple patterns so that they can easily be remembered. Table 1 summarizes the operations for each C-setting. The name of each operator is followed in parentheses by its default F-field.

The following exercises give a quick review of the material in this section. They are mostly quite simple, and the reader should try to do nearly all of them.

EXERCISES

- 1. [00] If MIX were a ternary (base 3) computer, how many “trits” would there be per byte?
- 2. [02] If a value to be represented within MIX may get as large as 99999999, how many adjacent bytes should be used to contain this quantity?
- 3. [02] Give the partial field specifications, (L:R), for the (a) address field, (b) index field, (c) field field, and (d) operation code field of a MIX instruction.
- 4. [00] The last example in (5) is ‘LDA -2000,4’. How can this be legitimate, in view of the fact that memory addresses should not be negative?
- 5. [10] What symbolic notation, analogous to (4), corresponds to (6) if (6) is regarded as a MIX instruction?

► 6. [10] Assume that location 3000 contains

+	5	1	200	15
---	---	---	-----	----

What is the result of the following instructions? (State if any of them are undefined or only partially defined.) (a) LDAN 3000; (b) LD2N 3000(3:4); (c) LDX 3000(1:3); (d) LD6 3000; (e) LDXN 3000(0:0).

7. [M15] Give a precise definition of the results of the DIV instruction for all cases in which overflow does not occur, using the algebraic operations $X \bmod Y$ and $\lfloor X/Y \rfloor$.

8. [15] The last example of the DIV instruction that appears on page 133 has “rX before” equal to

+	1235	0	3	1
---	------	---	---	---

. If this were

-	1234	0	3	1
---	------	---	---	---

 instead, but other parts of that example were unchanged, what would registers A and X contain after the DIV instruction?