PDP-1 COMPUTER
ELECTRICAL ENGINEERING DEPARTMENT
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
CAMBRIDGE, MASSACHUSETTS
02139

PDP-35

INSTRUCTION MANUAL

PART 1 -- BASIC INSTRUCTIONS

9 September 1972

SEE FOLLOWING MEMO

FOR REFERENCES TO "ONES" MODE

Introduction

The PDP-1 is a binary, word-oriented digital computer. It has the ability to perform arithmetic upon numbers represented in either of two formats known as one's complement and two's complement respectively. Two's complement is a relatively recent addition to the PDP-1 but it is the preferred mode of operation. Therefore, this manual will describe in detail only the two's complement mode of operation. However, since many existing programs are written to run in one's complement mode, a brief but complete description of one's complement mode may be found in Appendix I.

Although two's complement mode is preferred, one's complement mode is still the default mode of operation. To assemble and debug a program to run in two's mode, the following three steps must be taken:

1. Place the symbol "twos" at the beginning of the program that you wish to assemble. This tells the assembler that your program is to run in two's mode.

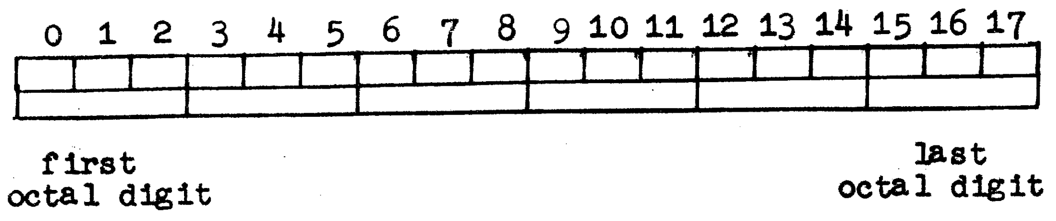
2. Make "e2m" the first instruction of your program. This will set the computer to run in two's complement mode.

3. Type "⊃twos" to ID. This tells ID that numbers are represented in two's complement form. This need only be done once during each console session.

1. Basic Instructions

1.1 Word Formats

Each PDP-1 word is 18 bits long. The bits are numbered, in decimal, 0 to 17 from left to right. In this manual, all numbers are octal, unless otherwise specified. The numbering of bit positions in words is always decimal.



1.1.1 Number Formats

The entire word may be regarded as a signed 18-bit number. Bit 0 is the sign bit. It is on (i.e., a "1") if the number is negative. Positive numbers are represented in the ordinary binary notation. The range of positive numbers that can be represented is 0 to 377777 (131071 decimal). The negative of a number is formed by complementing all bits of the number and then adding 1. Hence -1 is represented as 777777. 400000 (-131072 decimal) is the most negative number that can be represented. Note that the negative of 0 is 777777+1 which is 0.

Examples

Number (base 10)	Two's Complement
0	0
5	5
-5	777773
131071	377777
-131071	400001
-131072	400000

By ignoring the sign convention, a program could deal with data words as unsigned numbers ranging between 0 and 262143 (decimal) or 0 and 777777 (octal).

The addition rule for 2 18-bit numbers is as follows. Add the 2 numbers in the normal fashion, propagating carries to the left. If there is a carry out of the last bit (bit 0) it is ignored. The addition is said to overflow if the result does not correctly represent the algebraically correct sum, i.e., the ~~magnitude of the~~ correct result is ~~greater than 377777.~~ not in the range -400000 ~~to~~ 377777.

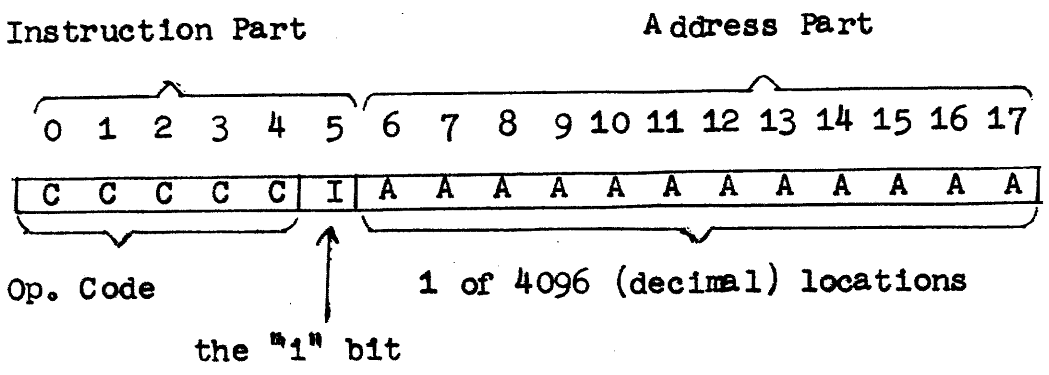
The overflow condition is equivalent to the condition that a carry occurred from bit 1 but not from bit 0 or vice versa.

Example ~~Two's Complement~~

```
      7                      7
+  -100               + 777700    (-100)
  ~~-61~~  -71          777707    ( -71)

 123456                 123456
-111112  + 666666       + 666666    (-111112)
 ~~1~~012344   12344 = 1012344
```

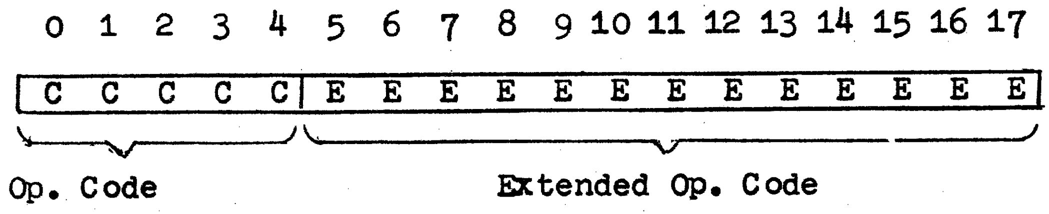
1.1.2 Addressable Instruction Format



The "instruction part" consists of 5 bits (the "op. code") that tell which instruction the computer will do if it executes this word, and the "i-bit" which has to do with address modification.

Except for jmp and jsp, addressable instructions take a minimum of two memory cycles -- one to fetch the actual instruction and another to fetch the operand. jmp and jsp do not take an operand and consequently, require a minimum of one cycle. The amount of time required for the execution of an addressable instruction depends on what addressing is done (see PDP-35, INSTRUCTION MANUAL, Part 2 for complete information). Under the most common conditions, the actual time required for instruction execution is the minimum time given above.

1.1.6 Non-addressable Instruction Format



The low 13 bits are actually an extension of the op. code, further specifying what the computer is to do. Non-addressable instructions never require more than one cycle because they do not reference memory, except for the ivk instruction with PRL on (See PDP-35, INSTRUCTION MANUAL, Part 5).

1.2 Registers

The PDP-1 contains 6 18-bit registers which define the state of the user's process. They are the A (accumulator), I (input/output register), X (index register), G, F (flag register), and W. When the user first logs in, all of these registers contain 0 (all bits off). This defines the initial state of the user's process. Each of these registers has its own properties.

1.2.1 Accumulator

The accumulator is the major register for use in processing data. Most arithmetic and logical instructions operate on data in A and leave results in A.

1.2.2 Input/Output Register

The input/output register was originally used primarily for input/output operations. This is no longer true. I is now a secondary accumulator. Many arithmetic and logical operations can be performed on data contained in I.

1.2.3 Index Register

The index register has two functions. First, it is used in addressing memory (See PDP-35, INSTRUCTION MANUAL, Part 2). Second, it is, like I, a secondary accumulator.

1.2.4 G Register

The G register contains the 15-bit program counter, the overflow bit, the extend mode bit, and the arithmetic mode bit.

The program counter (PC) is bits 3-17 of the G register. The purpose of the program counter is to tell the processor where in memory the instruction that is to be executed next lies. The program counter is incremented after each instruction so that instructions are executed sequentially, according to their locations in memory. When the program counter is incremented, carries out of bit 6 of G are lost. Thus, the instruction executed after the instruction in location 7777 is the instruction in location 0. Certain testing instructions increment the PC an extra time, causing an instruction to be skipped.

Bit 0 of G is the overflow bit (OVF). It is set to 1 by certain arithmetic instructions when overflow occurs, and cleared by the <u>szo</u> instruction. Bit 1 of G is the extend mode bit (EXD). This bit is used in addressing (See PDP-35, INSTRUCTION MANUAL, Part 2). Bit 2 of G is the arithmetic mode bit (TWOS). If this bit is off, the processor is in one's complement mode; if this bit it on, the processor is in two's complement mode.

The format of the G register is shown below.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
OVF	EXD	TWOS	PC3	PC4	PC5	PC6	PC7	PC8	PC9	PC10	PC11	PC12	PC13	PC14	PC15	PC16	PC17

1.2.5 Flag Register

The flag register contains the 6 program flags and several bits which define various states of the processor.

The program flags are 6 1-bit registers that may be quickly and conveniently set and tested by programs (see Operate and Skip class instructions and also PDP-35, INSTRUCTION MANUAL, Part 3, section 3.2.6 concerning the light pen).

The 3 bits AMD, AEF, and AAL determine the mode of addressing which the processor will use on memory referencing instructions (see PDP-35, INSTRUCTION MANUAL, Part 2).

The 2 bits SBH (Sequence Break Hold) and SBM (Sequence Break Mode) determine the state of the sequence break system (see PDP-35, INSTRUCTION MANUAL, Part 3).

The ESI (Execute Single Instruction) bit causes the processor to trap after each instruction is executed (see PDP-35, INSTRUCTION MANUAL, Part 5).

The PRL (Program Reference List) bit affects the way in which the ivk (invoke) instruction works (see PDP-35, INSTRUCTION MANUAL, Part 5). When the PRL bit is 1, references to memory locations 0-77 are illegal.

The format of the F register is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
AMD	AEF	AAL	SBM	SBH	PRL	ESI						PF1	PF2	PF3	PF4	PF5	PF6

1.2.6 W Register

The W register, a software register maintained by the time-sharing supervisor, is used solely for communication with the supervisor. Certain mta and ivk instructions (both are supervisor calls) use the W register (see PDP-35, INSTRUCTION MANUAL, Part 5).

1.3 Instructions to Set the Arithmetic Mode

The PDP-1 processor may operate in either of two arithmetic modes, one's complement and two's complement. The arithmetic mode in which an instruction is executed is determined by the state of the TWOS bit in the G register. When the TWOS bit is off (contains a 0), the processor is in one's complement mode. One's complement mode (TWOS off) is the default mode. The following instructions change the state of TWOS.

Mnemonic	Op.Code	Name	Function
e2m	770060	Enter two's mode	Set TWOS to 1
e1m	770061	Enter one's mode	Set TWOS to 0

The instructions add, adm, sub, mul, div, idx, isp, sft, opr, and opr i (the micro-program instruction) behave differently in one's and two's mode. Address arithmetic is always done in the current arithmetic mode.

1.4 Addressable Instructions

In this section, the symbol "y" when used in the context of "ins y" means the memory location referenced by the instruction "ins". The notation "(y)" means the contents of location y. For information on how addresses are computed, see PDP-35, INSTRUCTION MANUAL, Part 2.

1.4.1 Data Moving Instructions

These instructions serve to move data between memory locations and the A, I, and X registers. These instructions copy data words (or parts of words) from one place to the other and never destroy information at the source.

Mnemonic	Op.Code	Name	Function
lac y	20	load A	Copy (y) into A
lio y	22	load I	Copy (y) into I
lxr y	12	load X	Copy (y) into X
dac y	24	deposit A	Copy A into y
dio y	32	deposit I	Copy I into y
dap y	26	deposit address part of A	Copy the low 12 bits of A into y. The high 6 bits of y are unchanged
dip y	30	deposit instruction part (of A)	Copy the high 6 bits of A into y. The low 12 bits of y are unchanged
dzm y	34	deposit zero in memory	Makes location y contain 0

1.4.2 Logical Instructions

These instructions take one operand in A and the other from a memory location. The result is left in A. Each bit of the result depends only on the corresponding bits of A and the memory word before the operation.

Mnemonic	Op.Code	Name	Each bit of A will be a 1 if and only if the corresponding bits in A (before the instruction) and (y) were --
and y	02	and	both one
ior y	04	inclusive or	not both zero
xor y	06	exclusive or	different

1.4.3 Arithmetic Instructions

The following instructions are used to compute sums and differences. The "left" operand is in A and the "right" operand is taken from memory. The result is left in A and in the case of adm (add to memory), it replaces the contents of the memory location as well.

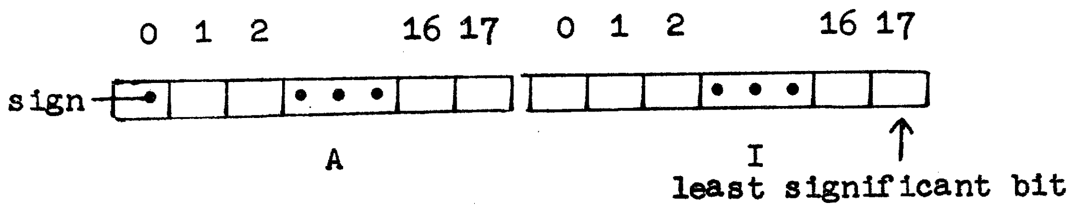
The overflow bit OVF will be set by the add, adm, or sub instructions if the signed result cannot be correctly represented in 18 bits. This is the case if and only if a carry occurred from bit one and no carry occurred from bit 0, or vice versa. See the szo instruction.

Mnemonic	Op.Code	Name	Function
add y	40	add	Sum of A and (y) to A
adm y	36	add to memory	Sum of A and (y) to A and y
sub y	42	subtract	A minus (y) to A

1.4.4 Multiply

2 SPACES

Multiply operates upon two 18 bit numbers to produce a 36 bit product. mul may be viewed as multiplying two 17 bit integers plus signs to produce a 35 bit integer plus sign in the combined A and I registers. MUL NEVER SETS THE OVERFLOW BIT OVF



When two integers are multiplied such that the result can be held in one register, the entire result will be in I in the conventional signed integer format.

Examples --

Before mul		After mul	
A	y	A	I
3	200000	0	600000
-3	200000	777777	200000
3	2	0	6
-3	2	777777	777772
400000	2	777777	0
400000	400000	200000	0

Mnemonic	Op.Code	Name	Function
mul y	54	multiply	A times (y) to A,I

Multiply takes a minimum of two memory cycles plus from 3 to 15 microseconds, depending on the number of one's in A.

1.4.5 Divide

Divide (<u>div</u>) takes a double-length integer in A and I (in the format produced by <u>mul</u>) and divides this by a single length integer in the addressed memory location. The result of a <u>div</u> is a single-length integer quotient in A, and a single-length integer remainder in I. The sign of the remainder will be the same as that of the dividend.

If a quotient overflow occurs, that does is, if the divisor goes into the dividend more times than can be represented in the accumulator (A), the <u>div</u> instruction ~~will~~ not skip. If a <u>div</u> does not skip, the contents of the A and I registers are preserved if the divisor is 0, otherwise they are usually destroyed. <u>div</u> never sets the overflow bit. ORF.

Examples

Before divide			After divide	
A	I	C(Y)	A	I
0	16	2	7	0
0	7	3	2	1
0	11	777773	-1	4
0	25	777774	-5	1
1	400000	200000	6	0
777776	377777	777774	300000	-1 ?
777777	0	400000	2	0
200000	0	400000	400000	0
6	777776	0	no skip, A and I unchanged	
~~100000~~ 77777	222222	100000	no skip, A and I destroyed	

Mnemonic	Op.Code	Name	Function
div y	56	divide	Quotient of (A,I)/(y) to A Remainder to I Skip if the quotient does not overflow

Divide instructions which skip take two cycles plus 20 microseconds. Divide instructions which ~~restore A and I and do not skip take 2 cycles; other non-skipping divides take 2 cycles~~ do not skip take 2 cycles and up to plus 20 ~~microseconds~~.

1.4.6 Counting Instructions

The two instructions idx and isp add one (~~in the current arithmetic mode~~) to the contents of the specified memory location and leave the result both in memory and A. isp skips the next instruction if the result is positive. Neither instruction will set overflow.

Mnemonic	Op.Code	Name	Function
idx y	44	index	(y) + 1 to A and y
isp y	46	index and skip if result positive	(y) + 1 to A and y skip if result ≥ 0

1.4.7 Compare Instructions

The following two instructions are used to compare A with the contents of a memory loction. The comparison is done bit-by-bit, therefore, the contents of A are the same as the contents of memory if and only if every bit of A is the same as the corresponding bit in memory.

Mnemonic	Op.Code	Name and Function
sas y	52	skip if A is the same as (equal to) the contents of y
sad y	50	skip if A is different from (y)

1.4.8 Transfer of Control

All of the following instructions have the effect of changing the program counter (PC) so that the PDP-1 will begin executing instructions in a new sequence.

Mnemonic	Op.Code	Name	Function
jmp y	60	jump	transfer control to location y
jdp y	14	jump and deposit program counter	store G in y, jump to y+1
jsp y	62	jump and save program counter	jump to y and save G in A
jda y	17	jump and deposit accumulator	store A in y, save G in A, jump to y+1 (dac y, jsp y+1)
cal y	16	call	store A in 00100, save G in A, jump to 00101, y ignored (similar to jda 100)

The instructions jdp, jsp, and jda are used chiefly for calling subroutines. The saved G register is the linkage mechanism which allows the subroutine to return to the place from which it was called.

Simple Examples

```
        jdp subr              jsp subr               jda subr
          ...                   ...                    ...

  subr, 0               subr, dap subx         subr, 0
          ...                   ...                  dap subx
          ...                   ...                    ...
        jmp i subr      subx, jmp              subx, jmp
```

These examples show three ways in which a subroutine may be called. In each example the method by which the subroutine returns to the calling program is illustrated. In each example the return is to the location immediately following the subroutine call.

1.4.9 Execute

The xct instruction causes the contents of the specified memory location to be executed as an instruction. xct's may execute other xct's. In all cases the effect is the same as if the xct were replaced by the instruction it addresses. xct takes a minimum of one cycle plus the time to do the addressed instruction.

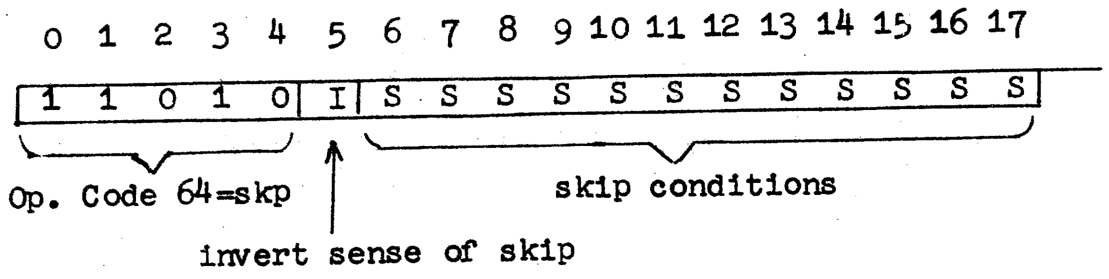
Mnemonic	Op.Code	Name	Function
xct y	10	execute	execute the contents of y as an instruction

1.5 Non-Memory Referencing Instructions

1.5.1 Skip Class

Instructions from the skip class will cause the PDP-1 to jump over one instruction in the normal sequence if the skip condition described by the low 13 bits of the instruction is true.

A skip class instruction has the following format --



Each of the low 12 bits enables a different skip condition.

Mnemonic	Op.Code	Name	Function
skp	640000	skip	never skips
szf n	64000n	skip on zero flag n ($1 \leq n \leq 7$)	skip if program flag n is off. szf 7 skips if all flags are off.
szs n0	6400n0	skip on zero switch n ($1 \leq n \leq 7$)	skip if sense switch n is off. szs 70 skips if all switches are off.
sza	640100	skip on zero A	skip if A(0-17) = 0
spa	640200	skip on positive A	skip if A(0) = 0
sma	640400	skip on minus A	skip if A(0) = 1
szo	641000	skip on zero overflow	skip if the overflow bit (OVF) is off. A szo instruction always clears OVF
spi	642000	skip on positive I	skip if I(0) = 0
sni	644000	skip on nonzero I	skip if I(0-17) $\neq$ 0

If more than one skip condition is enabled, the instruction skips if any (i.e. the logical OR) of the skip conditions is true.

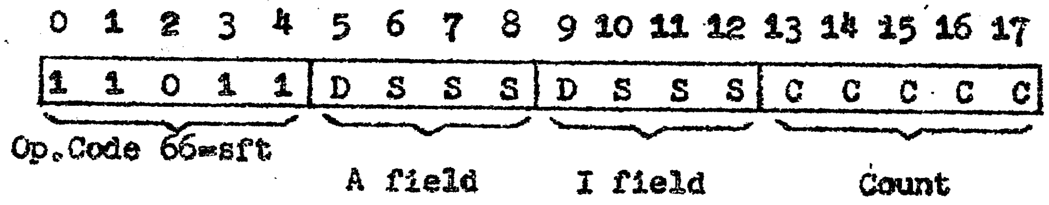
The "i-bit" reverses the sense of the skip, i.e., a skip instruction with the "i-bit" on will skip if and only if the corresponding instruction with the "i-bit" off would not skip. For example, skp 4200 will skip if either I $\neq$ 0 or A(0) = 0. skp i 4200 will not skip if either of these conditions is true. Thus, skp i 4200 will skip only if I = 0 and A(0) = 1.

The following mnemonics define several useful compound skip instructions, i.e, each has several skip conditions enabled.

Mnemonic	Op.Code	Name	Function
szm	640500	skip on zero or minus accumulator	skip if A(0-17) = 0 or if A(0) = 1 (szm = sma$\vee$sza)
spq	650500	skip on positive quantity	skip if A > 0 (spq = sma$\vee$sza i)
clo	651600	clear overflow	this never skips; it is used to clear overflow (clo = spa$\vee$sma$\vee$szo i)

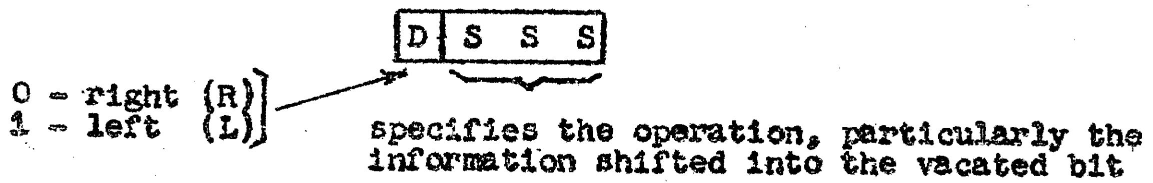
1.5.2 Shift/Rotate Class

The shift instructions perform a variety of functions, including rotates, logical and arithmetic shifts, normalization, and bit counting. The register operated upon may be the A, I, AI (bit 17 of A joined to bit 0 of I), or IA (bit 17 of I joined to bit 0 of A) or even combinations in which one of the registers is reversed. The A and I registers may be operated upon independently, but they must shift the same number of bits.



If the count field is zero, the low 5 bits of the XR are used instead.

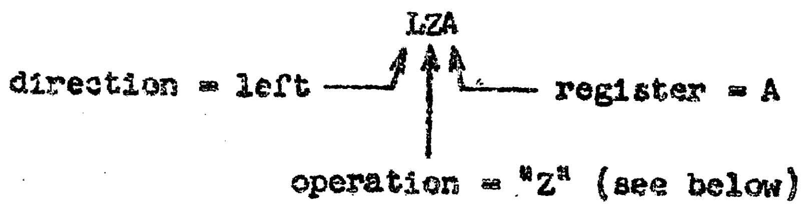
The 4 bits of the A and I fields are interpreted as follows:



D S S S
0 = right (R)
1 = left (L)
specifies the operation, particularly the information shifted into the vacated bit

Because of the great number of possible operations, the mnemonics for this instruction (and the "micro-program" instruction which follows) are "micro-coded". That is, they are assembled in a methodical way from constituent letters. This makes it easier to analyze the instructions in terms of their basic components, rather than trying to describe 256 (or, in the case of the micro-program instruction, 3584) instructions. Shift instructions are written entirely in UPPER CASE characters. (This is to avoid conflicts with other symbols.) Since ID has many upper case commands, shifts must be preceded by a single quote (') when typing them into ID. The assembler has no such problem.

Shift instructions always consist of three upper case letters, specifying a direction, operation, and register.



In a "basic" shift instruction, the register is given as either A or I, and the operation is one of the nine letters given below. The operation and direction are encoded into the "DSSS" field for the indicated register. The field for the other register is set to 0000, making that register do nothing. Instructions directing actions by both A and I may be written by inclusive or'ing basic instructions.

Example -	LZA 5	shift A left 5 bits in a "Z" operation
	LZAVRPI 5	shift I right 5 bits in a "P" operation simultaneously with the above
	LZA	shift A left a number of bits equal to XR(13-17)

The eight possible operations are specified by the following letters ---

letter stands for SSS

F 000

If direction = R, do nothing to this register.
If direction = L, zeroes are shifted into bit 17, bit 0 is unchanged; if bit lost from bit 1 ≠ bit 0, set overflow (arithmetic left shift, doubles the magnitude of the number for each bit shifted)

G 001

If direction = R, see below.
If direction = L, bit 0 of other register is shifted into bit 17, bit 0 is unchanged; if bit lost from bit 1 ≠ bit 0, set overflow (arithmetic shift left combined, if other register is simultaneously doing a left "Z" shift).

Z zero 010

Shift zeros into vacated bit (logical shift).

O one 011

Shift ones into vacated bit.

P propagate 100 if right
101 if left

Vacated bit is shifted into itself, so it propagates itself along the register. If direction = R, this is an arithmetic right shift, halving the magnitude of the number for each position shifted. Non-integral results are rounded toward minus infinity.

R rotate 101 if right
100 if left

The bit at the opposite end of the same register is shifted into the vacated bit. Hence, bits falling off one end are re-introduced at the other, making the register "rotate" as if its ends were tied together in a ring.

V reverse 110 if right
111 if left

The bit at the corresponding end of the other register is shifted into the vacated bit. If the other register is simultaneously shifting in the opposite direction, bits that fall off it will be introduced into this register in the reverse order.

C combine 111 if right
110 if left

The bit at the opposite end of the other register will be shifted into the vacated bit. If the other register is shifting in the same direction, bits that fall off it will be shifted into this register as if it were an extension of the other. E.g. LCAVLZI shifts the AI left together, inserting zeros into I(17). In practice, it is rarely necessary to write such combinations, because there are abbreviations for them. LZC is equivalent to LCAVLZI.

For convenient arithmetic shifts, a ninth letter is defined -

S shift 100 if right
000 if left

Equivalent to F if left or P if right.

RGA is used for normalization and bit counting. The AC does not shift at all, but is initialized to zero and counts. IF the IO operation is F or G, the operation proceeds until either $I(0) \neq I(1)$ or the count runs out. The AC returns the number of places actually shifted. E.g. RGAVLFI 7 (or, equivalently, RGAVLSI 7) shifts the IO left at most 7 places or until it is about to overflow, returning the number of places shifted in the AC. If the IO operation is Z, O, P, R, v= or C, it will shift as usual for the full count. The AC will count the number of times a 1 is shifted out of I(17). E.g. RGAVRZI 5 counts the number of ones in I(13-17).

RGI is reserved for future expansion.

Operations containing RGA may be conveniently specified by giving N (normalize) as if it were the register, and specifying the direction and operation to be performed on the IO. xyN is equivalent to RGAVxyI for all x and y. LSN 7 is equivalent to RGAVLSI 7.

Most of the common combined operations may be written with one symbol by specifying C (combined) or R (reverse combined) in place of A or I for the register. "C" makes the shift operate on AI as a 36 bit register. "R" makes the shift operate on IA.

If the operation is not S or R, a left combined shift causes both registers to shift left, the AC doing a "C" shift and the IO doing whatever operation is specified by the second letter.

LxC = LCAVLxI if $x \neq$ S or R
LSC = LOAVLZI, a combined arithmetic left shift.
LRC = LCAVLCI, a combined rotate.

For example, LOC = LCAVLOI

If the operation is not R, a right combined shift causes both registers to shift right, the IO doing a "C" shift and the AC doing whatever is specified by the second letter.

RxC = RxAVRCI if $x \neq$ R , $x \neq$ S
RRC = RCAVRCI, a combined rotate

For example, ROC = ROAVRCI

If the third letter is R instead of C, the treatment is the same as for C, except that the roles of A and I are reversed, so the IO is considered to be to the left of the AC in the combined register.
For example, LZR = LZAVLCI

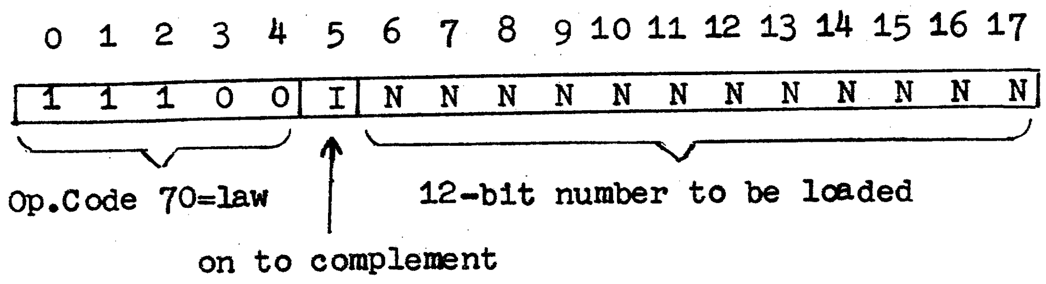
EXAMPLES

OPERATION	A	I	AI	IA
shift left	LSA	LSI	LSC	LSR
shift right	RSA	RSI	RSC	RSR
rotate left	LRA	LRI	LRC	LRR
rotate right	RRA	RRI	RRC	RRR
logical left	LZA	LZI	LZC	LZR
logical right	RZA	RZI	RZC	RZR

count bits in I	RZN 18.
normalize I	LSN 17.
reverse 36 bits of AI	RVAVLVI 18.
reverse 36 bits of AI	RVAVLVI 18.

1.5.3 The law Instruction

law and law i ("load accumulator with") make it possible to load the accumulator with 12 bit positive or negative numbers in one cycle.



law first loads A from the address part of the instruction and then, if the i-bit is on, complements A. Thus, law 3 puts three into A and law i 3 puts 777774 into A.

lan ("load accumulator with negative") is defined as law 7777. Thus, lan n ($1 \leq n \leq 10000$) will load A with -n.

1.5.4 The Operate Class

Each bit of the address part of an opr instruction enables an operation in the processor.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	1	1	1	1	0	E	E	E	E	E	E	E	E	E	E	E	E
						cli	lat	cma	csa	cla	cmi	lai	lia	stf			

$\underbrace{\text{bits } 0\text{–}5}_{\text{Op.Code 76 = opr}}$ $\underbrace{\text{bits } 6\text{–}17}_{\text{operation enabling bits}}$

Op.Code 76 = opr operation enabling bits

Several operations can be enabled by turning on more than one bit in the address part. The order in which the various operations on A and I occur is — first, clear A and clear I; second, OR in the test word; third, negate A; fourth, complement A and complement I; and last, switch data between A and I.

Mnemonic	Op.Code	Name	Function
opr	760000		
nop	760000	no operation	one cycle time delay
stf n	76001n	set flag n (1≤n≤7)	set one of the six program flags. Set all if n=7. Set none if n=0
clf n	76000n	clear flag n (1≤n≤7)	clear selected flag. Clear all if n=7. Clear none if n=0
lia	760020	load I from A	make I the same as A
lai	760040	load A from I	make A the same as I
swp	760060	swap A and I	exchange the contents of A and I
cmi	760100	complement I	invert all the bits of I
cla	760200	clear A	put 0 into A
csa	760400	complement and step A	negate A. If A contained 400000, OVF is set.

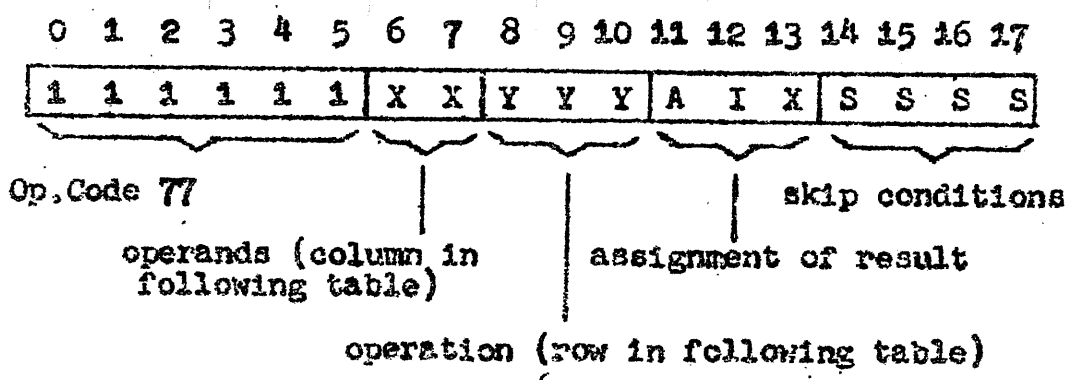
cma	761000	complement A	invert all the bits of A
clc	761200	clear and complement A	put 777777 in A
csc	761400	complement, step, and complement A	subtract 1 from A. If A contained 400000, OVF is set.
	762000	OR test word to A	
lat	762200	load test word	copy test word switches into A
cli	764000	clear I	put 0 into I

Although the stf and clf instructions are usually used to manipulate the program flags, there are also the following two instructions --

Mnemonic	Op.code	Name	Function
lpf	770051	load program flags	Top 2 bits of I to Address mode bits. Bottom 6 bits of I to program flags.
rpf	770050	read program flags	Address mode to top 3 bits of I. Flags to low 6 bits of I.

1.5.7 The Micro-program Class

The micro-program instruction class has the following instruction format --


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
1 1 1 1 1 1 X X Y Y Y A I X S S S S
Op.Code 77
operands (column in following table)
operation (row in following table)
assignment of result
skip conditions

Symbolic Specification of Micro-program Operations

YYY \ XX	00	01	10	11	
000	SPECIAL INSTRUCTIONS	TI	TA	TX	T (Test, or transmit, then clear)
001		NI	NA	NX	N (Negate)
010		A→I	X→A	X→I	exchange (see explanation below)
011		AMI	XMA	XMI	M (arithmetic minus)
100	Z	AVI	XVA	XVI	V (inclusive or)
101	SA	AAI	XAA	XAI	A (bitwise and)
110	SI	A~I	X~A	X~I	~ (exclusive or)
111	SX	A+I	X+A	X+I	+ (arithmetic plus)

Z = zero
S = step, i.e. contents of register + 1

Like the shift class, the names of these instructions are micro-coded, i.e. made up from certain characters and groups of characters. The instructions are written entirely in UPPER CASE characters with no imbedded blanks. (This is why subtraction is specified with the letter M). Since ID has many upper case commands, these instructions must be preceded by a single quote (') when typing them into ID. The assembler has no such problem.

The name of a micro-program instruction is composed of an operation name (an entry in the above table) followed by optional characters to specify register assignment and skip conditions.

The 28 operations in the above table compute a result. Except for the "T" and "→" operations, this computation does not by itself modify any of the three registers. (It may set overflow, however.) The result will be stored in the AC, IO, or XR if bits 11, 12, or 13, respectively, are on. These are specified by the letters A, I, and X immediately following the operation name. (Note that the letters in the preceding table are not assignment letters. To assign the result of "SA" to the AC, use "SAA".) Any or all registers may be specified. Independently of the register assignment, the result is tested, categorized as being >0, <-1, 0, or -1, and the instruction skips if the corresponding skip bit (bits 14 to 17) is on.

The operations are as follows ---

Z	zero.
SA, SI, SX	contents of register plus one. Note that the register is not actually modified unless the result is assigned back to that register, e.g. SAA. SAX sets XR = AC + 1 without changing AC. Stepping 377777 will set overflow.
TA, TI, TX	The register is cleared, and its old contents are the value of the instruction. The value may, of course, be assigned back to the same register, e.g. TAAX sends AC to XR. TAX sends AC to XR and then clears AC.
NA, NI, NX	Negate. The result is the negative (two's complement) of the register. The register is not changed unless the result is assigned back to it. Negation of -400000 will set overflow.
A→I etc.	The indicated register transfer takes place, and the result is the original contents of the register on the right. If the result is assigned back to the register on the left, the two registers will simply be exchanged. X→AX exchanges AC and XR. There are no I→A, A→X, or I→X instructions.
AMI etc.	The result is AC - IO etc. Overflow may be set. There are no IMA, AMX, or IMX instructions.

AVI, AAI, A~I,
A+I, etc. Overflow may be set during addition. These instructions, being commutative, may be written in either order, e.g. I+A or A+I.

The instruction will skip if a skip bit (bits 14 to 17) is on and the corresponding condition is true.

bit	condition
14	>0 (sign bit off but not all bits off)
15	<-1 (sign bit on but not all bits on)
16	0 (all bits off)
17	-1 (all bits on)

These bits may be specified with one or more of the following characters ---

character	bit value	meaning
>	1000	result > 0
<	0101	result < 0
=	0010	result = 0
M	0001	result = -1
_ (underbar)	0010	used with < or >, changes meaning to ≤ or ≥
(vertical bar)	1111	inverts the sense of any skip condition

If more than one character is written, their bit values will be exclusive or'ed. Hence "=" has value 1101 and skips if the result is ≠ 0. " " always skips. "=M" skips if the result is 0 or -1. "<M" (which could be read "less than minus one") has value 0100 and skips if the result is < -1.

All micro-program instructions require one cycle.

SAMPLE MICRO-PROGRAM INSTRUCTIONS

symbolic	octal	action
I+A	773600	computes sum of A and I and does nothing at all with it (but will set OVF if the addition overflows).
A+IA	773700	the sum of A and I is put into A.
A+IAIX	773760	the sum of A and I is put into A, I, and X.
ZAIX	771160	A, I, and X are cleared.
SA>	771210	skip if A plus one is > 0.
SAA=	771302	add one to (step) A and skip if it is 0.
TXM	776001	skip if X is -1 and clear X.
TXXM	776021	skip if X is -1.
TAXI	774060	transfer the contents of A into I and X, then clear A.
TAAXI	774160	same as TAXI, but A is not cleared.
A→IA	772500	exchange the contents of A and I.
X→A>=	774412	transfer the contents of X into A, skip if the previous A is positive.
SAM	771201	skip if A plus one is -1.
A+I<=	773607	skip if the sum of A and I is ≤ 0.
A+I<=|	773610	skip if the sum of A and I is > 0.
A+IX≥	773632	the sum of A and I is put into X. If the sum is ≥ 0, the instruction will skip.
NX=	776202	skip is -X is 0, i.e. if X is 0.