XDS 940

REFERENCE MANUAL

EE 466

BCC 500 Computer

Department of Electrical Engineering
University of Hawaii

Issued September 15, 1977

1.0 INTRODUCTION

The XDS 940 is a medium-sized, general purpose computer accessed from terminals in a time-sharing mode. The 940 is a good machine for a study of assembly-language programming techniques and machine organization. It is sufficiently large that a user does not have to confuse himself with various tricks required to avoid the hardware limitations of typical minicomputers; yet it is simple enough that the user can write, assemble, load and run a small program without having to learn a huge number of details. (This is not true of the IBM 370, for example.)

The 940 is at the upper end of a family of "midi" computers built beginning in 1961 by Scientific Data Systems (SDS), which later became Xerox Data Systems (XDS). The family, not produced since 1968, consists of the 910, the 920, the 930, and the 940. These machines are made of discrete components and are thus physically large and expensive compared to their capability (and with machines which are being built today.) Several hundred were built and marketed over a ten-year period, however, and many are still in use.

In terms of computing power, the machines are very little better, if any, than present-day minicomputers. In fact, it would be no problem to make them today as minicomputers and sell them at competitive prices. Where they differ from the minis, however, is in their longer word length (50% longer than the standard 16 bits). This shows up not just in terms of numerical precision and storage efficiency, but in certain simplicities in the instruction set and addressing modes which result from instructions not having to be encoded into shorter lengths.

The 940 is a time-shared version of the 930. It is equipped with a special operating system (hardware and software) which provides to several users simultaneously the services of a virtual machine. A virtual machine appears to the user to be a complete machine over which he has full control, even though he is in fact time-sharing slightly different hardware with other users. The user may load the virtual machine, examine its state, start it, stop it, and do any operation he could do if he had his hands on a physical equivalent.

The means by which the user communicates with the machine and exercises his control over it is a Teletype or similar on-line terminal. The terminal may be used for input and output when the user's programs are running; and it is used for the inspection of memory locations, register contents, etc. which on an actual machine usually involves lights or indicators. The terminal permits the user to reset or clear his (virtual) machine, start it at a certain location, stop it, etc. Instead of pushing actual buttons and switches, the user types special commands on his terminal. The terminal may also be used to prepare input for a program in advance of its operation, much like the use of a keypunch when preparing program source language or data; and it can display (perhaps selected portions of) the program's output after its termination. The terminal, then, is a multi-use device and assumes different roles at different times and in different contexts.

Aside from its virtues of simplicity and straightforwardness, we use the 940 in this course because the BCC 500 system, as a special feature, emulates the 940 system with reasonable efficiency. We can thus provide a number of students the opportunity to use an on-line system simultaneously. This is advantageous to the students not only in terms of how much work can be done but also in providing some experience in on-line computing.

2.0 940 ORGANIZATION

From his BCC 500 terminal, the user (if he wishes) sees a 940 system, i.e., a complete computer system consisting of memory, processor, input/output device(s), and operating controls and indicators. Figure 1 is a diagram of the machine, with emphasis on its registers and data paths.

2.1 Memory

The memory contains 16384 addressable cells in which information consisting of instructions or data can be stored. (16384 is often referred to as "16K," where 1K = 1024.) Each cell contains 24 bits -- binary zeros or ones -- of storage. It is up to the user to determine where in the memory and in what form the information is to be stored. There are essentially no restrictions as to how the memory may be used (although there are some conventions).

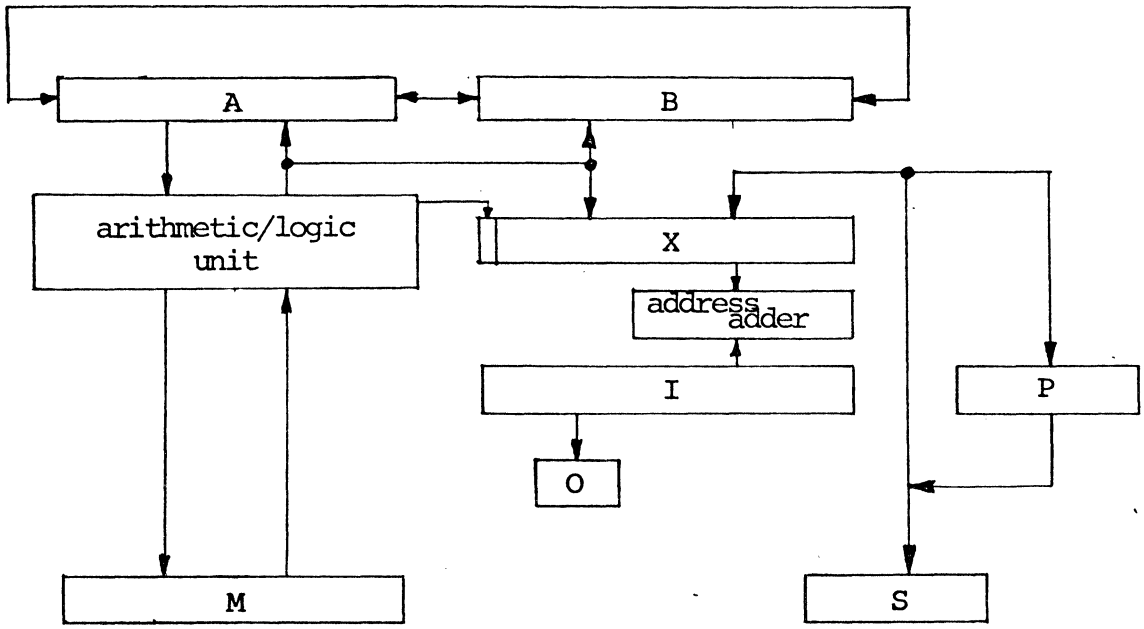


FIGURE 1. 940 Processor.

Figure 2 gives the format of and bit naming conventions for a memory cell in the machine. Each location is identified by a unique address, which is a number ranging from 0 through 16383. As seen in the figure, the bits in a cell are numbered from the most significant to the least beginning with 0.

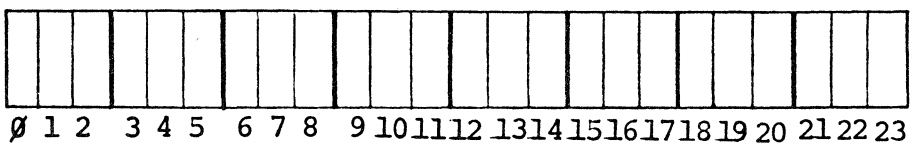


FIGURE 2. Memory Cell Format.

Since one octal digit is readily convertible into three bits, octal notation is used to represent the contents of a cell. Where it might otherwise be ambiguous we use the convention of identifying octal numbers by terminating them with the letter "B." A cell whose contents are as follow:

0 1 1	0 0 0	1 0 1	1 1 0	0 1 1	1 1 0	1 1 1	0 1 0

is said to contain 30563672B.

The memory permits both storing into and retrieving information from any of its cells. Storing involves copying a new value into the cell; retrieving copies the current contents out without modifying the contents (i.e., non-destructively). Either of these operations is called a memory reference or a memory access. In undergoing such a reference the memory must be provided with an address and with a store or a fetch command, as shown in Figure 3. The memory is designed to operate rapidly, at speeds matching those of the processor. The operating speed, or memory cycle time, is independent of the address; and so the memory is termed random access memory (RAM). The (strictly hardware) memory commands are prepared and issued by the in its role of executing the user's program. The user, then, is never explicitly concerned with operating the memory, but it is important for each user at the very outset to correctly visualize the memory, the appearance of its contents, and the way it works.

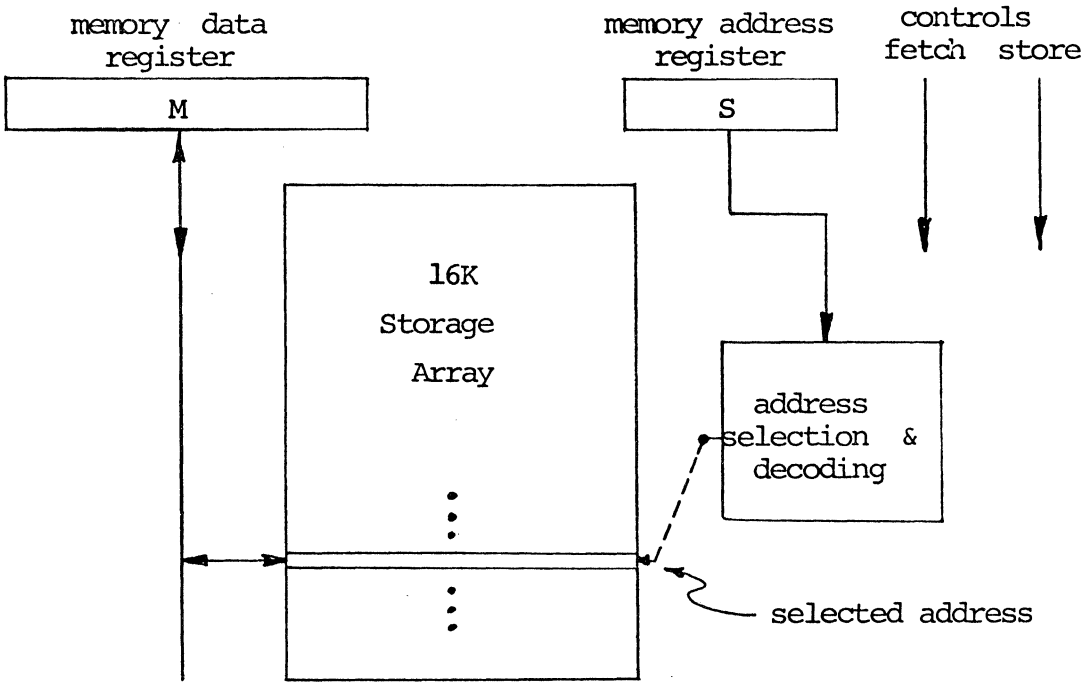


FIGURE 3. Detail of Memory Organization.

2.2 Processor

The processor is the entity which performs the operations required to execute the program. It fetches the machine-language program steps (called operations or instructions) from memory and performs the indicated actions, which include further fetching or the storing of operands. The processor is thus divided into two parts called the control section and the arithmetic section. The control section iteratively:

1) fetches the next instruction,
2) interprets and executes it, (hands over control to the arithmetic section)
3) determines the location of the next instruction.

The arithmetic section:

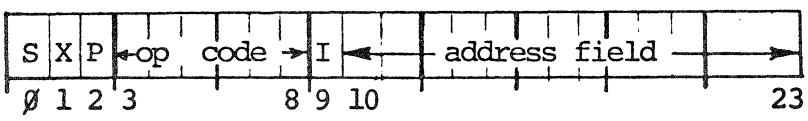
1) determines the address of any operand(s),
2) performs an operation on it (them).

From the user's point of view the control section performs its work with no specific attention other than an awareness on the user's part from time to time of the contents of a register called the program register or P. (A register is identical in structure to the memory cell, except that it is a constituent of the processor and plays a specific role in the processor's operation. Since the register is dedicated to certain functions it does not have to be addressed in the since of the many "registers" of the memory.) P is used to hold the address of the instruction currently being executed; at the end of the current instruction P will be modified to contain the address of the next instruction. P may be thought of as a roving pointer which ranges over the program as it executes and shows at any point of interruption the instruction address to be next executed. P thus provides the address required by the control section when it goes to the memory to fetch an instruction.

940 instructions each occupy one memory cell and are connected together in sequence by the obvious expedient of placing sequences of instructions into sequences of cells, i.e., cells with numerically increasing addresses. Since P most often increments in content value as the program runs, it is frequently referred to as the "P counter." (This counting action may be overridden by the use of branch or control transfer instructions which serve to change the contents of P altogether). As instructions can be fetched only from the 16K memory, P is only a 14-bit register. If the machine should attempt to fetch an instruction located in the next cell from 16383, P will overflow and the fetch will be made from location 0 instead.

2.3 Instruction Format

The format of an instruction is shown below. Each instruction is divided into portions called fields which indicate various aspects of the instruction.



The fields are:

- Bit Ø: System Call.
  If Bit Ø = 1 and Bit 2 = 1, the instruction is a system call, i.e., a type of instruction which causes a branch into a specific entry point in the operating system (see Section 2.9). Input/output, for example, is performed by means of system calls. If Bit 2 = Ø, the Bit Ø field is meaningless; it may have either value.

- Bit 1: Index Designator.
  If Bit 1 = 1, the address calculation known as indexing is to be done. Indexing is described later. It is applicable only to certain instructions.

- Bit 2: Programmed Operator Designator.
  If Bit Ø = Ø and Bit 2 = 1, the instruction is of a special type known as a programmed operator. This is described in Section 2.8.

- Bits 3 - 8: Operation Code.
  This field holds a six-bit number designating one of 64 possible instructions. The 940 does not use all of these combinations. A few are thus termed illegal instructions. The field is also used in conjunction with programmed operators to designate which one of 64 possible operators is being invoked.

- Bit 9: Indirect Address Designator.
  If Bit 9 = 1, a different mode of addressing called indirect addressing, or indirection, is invoked.

- Bits 10 - 23: Operand Field.
  This field contains 14 bits and, like the P counter, is capable of naming any one of the 16K memory locations. The field is most frequently used to refer to the address of an operand in memory. Some instructions, however, use it to hold the operand itself; and some do not use it at all.

2.4 Processor Registers

The arithmetic section of the processor contains three registers labeled A, B, and X. These registers play unique roles in the processor and are addressed implicitly in the instructions. The user must maintain awareness of their contents, however, since it is he who manages the use of these registers within the program.

A is called the accumulator. It is used by almost all the arithmetic and logical instructions and is central to the operation of any program. B is the auxiliary accumulator, used with A in a few arithmetic instructions and in shifting. The X register is called the index register and is used to hold a quantity--termed the index--for offsetting the operand address. Although the indexing operation is an address calculation--a calculation on a 14-bit quantity--X also contains 24 bits.

2.5 Overflow Indicator

The overflow indicator in the computer permits the ready detection and signaling of overflow conditions which might otherwise go undetected or require additional software overhead to detect during arithmetic operations in the execution of a program. The overflow indicator is set to 1 (turned on) if any of the following occurs:

1. A sum or difference resulting from an addition or subtraction cannot be contained within the A register.

2. Multiplication of 40000000B (also written 4B7) by itself. (The A and B registers cannot contain this product.)

3. A division with the absolute value of the numerator equal to or greater than the absolute value off the denominator. (The A register cannot contain this quotient.)

4. An arithmetic left shift changes the value of the bit in the sign position of the A register.

5. Bit 14 of the index register is not equal to Bit 15 of the index register when the instruction RECORD EXPONENT OVERFLOW (ROV) is executed.

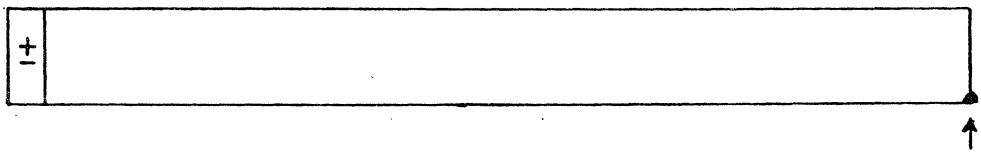
The 940 instruction set contains instructions to reset, test, or test and reset the state of the overflow indicator (see Section 3, "Overflow Instructions").

2.6 Data Formats

The 940 has various instructions which are designed to work on data assumed to be in different formats as follow:

2.6.1 Integers

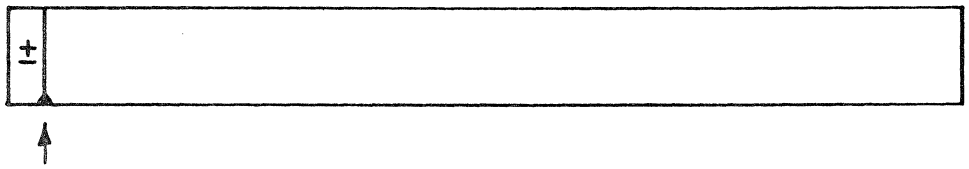
Integers are represented as 2↑24's complement numbers having the format:



Bit 0 indicates the sign of the number, negative numbers having a 1 bit and positive numbers having a 0 bit in this position. The assumed binary point is to the right of Bit 23, the least significant bit. In this form the range of representation is from -2↑23, or -8,388,608, to +2↑23-1, or 8,388,607. All of the arithmetic instructions except multiply (MUL) and divide (DIV) can be used on integer quantities.

2.6.2 Fixed-point Fractions

The arithmetic instructions are designed primarily to operate on fixed-point fractions haveing the following appearance:



The assumed binary point is between Bits 0 and 1 at the more significant end. Negative numbers are handled as complements with respect to 2 (two's complements). The range of representation is from -1.0 to +1-2↑(-23). These numbers have the equivalent of more than 6 decimal digits of accuracy. Fixed-point scaling (a forgotten programming art) is used in working with such numbers during computation.

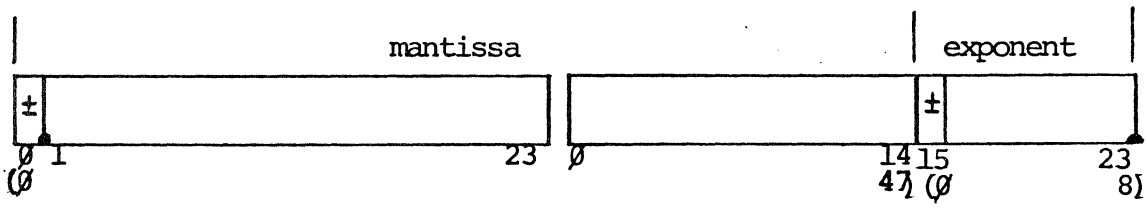
2.6.3 Extended-precision Fixed-point Numbers

Several instructions greatly facilitate the use of multiple precision data. None, however, operate on such data directly. A double-precision fraction, for example, would look like:

In memory such a datum would be stored in two consecutive memory locations.

2.6.4 Floating-point (Real) Numbers

While not having true floating-point instructions, the 940 has several (rather odd) instructions designed to greatly reduce the software overhead of subroutines to perform calculations on reals. These instructions assume the following real-number format:



The mantissa is a 39-bit, two's complement, normalized fixed-point fraction (giving about 11 decimal digits of accuracy). The exponent is a 9-bit, 512's complement integer, permitting an exponent range of 2↑-256 to 2↑255, or about 10↑-77. In memory, the real number is stored in two consecutive memory locations and is addressed by the former (i.e., smaller) address. The virtual 940 (the basic in-

struction set augmented by system calls -- see Section 2.9) does have arithmetic "instructions" which deal directly with reals.

2.6.5 Character Strings

The virtual machine adds other capabilities not found in the hardware instruction set. An important one is the ability to fetch and store individual 8-bit bytes from memory, according to the following format:

T	H	I
S	Ø	I
S	Ø	A
Ø	S	T
R	I	N
G	Ø	Ø

This ability makes the machine well suited to deal with character strings -- variable length sequences of bytes. For this purpose it is imagined that all of memory can be byte addressed, as well as word addressed. Since there are three bytes/word, the byte address is roughly three times in value the address of the word in which it is stored. The precise correspondence is

word addr = byte addr / 3, (0, 1, or 2 remaining)

and the byte position within the word is

Ø	1	2

Byte memory thus looks like the following:

Word Ø:	Byte Ø	1	2
1:	3	4	5
2:	6	7	8
3:	9	1Ø	11
4:	12	13	14

2.7 Address Modification Rules

Most machines provide some means for modifying at execution time the effective address of an instruction from that which it actually contains. This is done a) to reduce the run-time overhead of programs dealing with simple data structures and/or b) to avoid the program's having to modify itself. The 940 provides indexing and indirection (indirect, or deferred, addressing) for these purposes. The two features may be used jointly or singly in the same instruction.

2.7.1 Indexing

The machine contains an index register (X register) for address modification, the use of which does not increase execution time. If Bit 1 in an instruction which addresses memory (some don't) is 1, the 940 adds Bits 10-23 of the X register to the address field of the instruction to produce a different effective address (the address actually referenced). The addition is done strictly modulo 2^14, completely ignoring any overflows which may occur. If Bit 1 is a zero the X register is not added; the effective address is merely the address found in the instruction.

The instruction set provides instructions for modifying and testing the X register.

2.7.2 Indirection

When Bit 9 of an instruction (which permits it) is 1, indirection is invoked. The machine fetches the contents of the address found in the instruction (or the address offset by Bits 10-23 of the X register if the instruction word's Bit 1 = 1) and begins the entire address modification cycle again using Bits 1 and 9 of the newly-fetched location as a guide to further action. This process can repeat many times, depending on the contents of memory.

2.7.3 Simultaneous Indexing and Indirection

It is correct to say that for each instruction executed an effective address is always calculated, the results depending on the X and I bits according to the following algorithm executed by the hardware:

In the following, P is the 14-bit program register, S the 14-bit memory address register, M the 24-bit memory data register, I the 24-bit instruction register, O the 6-bit operation code register, and X the 24-bit index register.

The algorithm is expressed in terms of an informal programming language.

```
*  940 EFFECTIVE ADDRESS CALCULATION:
*  FIRST WE HAVE TO FETCH THE INSTRUCTION.

START: S←P & FETCH;

*  AT THE END OF THE MEMORY CYCLE THE FETCHED DATA IS IN M.

       O←M(3,8);            /*CAPTURE THE OP CODE BITS*/
       FOREVER DO;
          I←M;              /*ADDRESS CALC BEGINS HERE*/
          I←(I+X)MOD 2↑14 IF I(1)=1;
          GOTO DONE IF I(9)=0;
          S←I(10,23) & FETCH;  /*DO INDIRECT STEP*
       ENDFOR;

DONE:  Q←I(10,23);          /*Q IS THE EFFECTIVE ADDR*/
```

(The reader will note that this algorithm accurately describes the behavior of the machine for all four combinations of the X and I bits.)

2.8 Programmed Operators

Most arithmetic machine instructions require in some way three addresses: those of two operands and that of the result. The 940, like most one-address machines, addresses the A register by implication for the first operand and for the result. Its instructions, then, explicitly address only the second operand.

It is not infrequent that a similar situation develops when a programmer is designing a subroutine: the subroutine is to perform some operation on two 24-bit quantities and return a single result. The problem is how to convey to the subroutine the two arguments and receive the result. The obvious choice for a machine of this type is to use A for the first operand and for the result. But the address field of the subroutine call instruction is occupied with the address of the subroutine, forcing some other choice (such as the use of B, perhaps). This is not really bad, but it makes the use of the subroutine a little awkward, especially if we would like to apply address modification to the second operand.

The 940 Programmed Operator (POP) feature permits a programmer to pack into a single instruction both which subroutine is to be entered and a 14-bit address of an operand. The subroutine can with great efficiency and ease retrieve this address and apply the same address modification rules as the bare hardware uses. This makes the POP subroutine look for all subsequent programming purposes very much like a machine instruction.

The basis of the POP is as follows: An instruction is either a POP, or it is not. Therefore only one bit is required in the instruction word to specify whether the feature is to be used. Bit 2 = 1 is used for this purpose. The remaining 6 bits of the operation code field are used to specify the subroutine entry point. 6 bits cannot, of course, directly point to an arbitrary 14-bit address. But the field can direct the machine to an arbitrary location through a 64-word linkage table.

When the 940 fetches a new instruction and detects a 1 in Bit 2 of that instruction (and a 0 in Bit 0), it does not interpret Bits 3-8 as an opcode. Instead it:

1. Stores current value of overflow indicator in Bit 0 of memory location 0.

2. Resets the overflow indicator.

3. Stores zeros in Bits 1-8 of memory location 0 and a 1 in Bit 9.

4. Stores current contents of P register into Bits 10-23 of memory location 0.

5. Loads Bits 2-8 of the instruction word into P register.

The machine does not apply the address modification rules to a POP, nor does it refer to Bits 10-23 of the POP instruction.

The effect of the steps just outlined is to store a normal (except that Bit 9 is always set) subroutine return link (see BRM instruction in Section 3) in memory location 0 and to transfer control to a memory address in the range 100B - 177B. There it is expected that the programmer will have placed an unconditional control transfer to the actual subroutine entry point. A given program may include up to 64 (100B) such subroutines.

The subroutine can access the operand specified back in the POP instruction, along with any address modification specified in the POP, merely by referring to memory location Ø indirectly. Because of Bit 9's previously having been set, the indirect reference is propagated one more level and the effective address is then formed as if the POP had been a machine instruction. This means that any POP can use indexing and/or indirection for any meaningful purposes.

2.9 System Calls

An operating system such as that required in time-sharing cannot permit the user to execute every instruction known to the hardware. Some instructions, such as I/O instructions for example, would bring the (independent) users into serious conflict with each other and with the system. Instead the system must perform the I/O on the user's behalf with due regard for checking his authorization for such I/O, for scheduling considerations, device allocation, etc. The user communicates his wishes to the system (obtains/gives data from/to the system in the case of I/O) by means of system calls, transfers of control through carefully protected entry points of the system software.

The system software is placed in a different area of memory from that addressable by the user. This is made possible by the 940 virtual memory features, not discussed here. Since the user cannot address this memory, there is no way he can fetch improper information (such as someone else's password) or store data into it, thereby possibly destroying or altering the system. All he can do is enter it, and then only at known locations with valid parameters.

The POP mechanism is ideal for this purpose since it provides for protected entry (e.g., only through the POP transfer vector, or linkage table) and makes parameter retrieval so natural. If the 940 detects a 1 in Bit 2 of an instruction word and also sees a 1 in Bit Ø, then before proceeding to perform the steps detailed in Section 2.8 above it first shifts memory addressing to include the system code. When the link return word is saved in memory location Ø, it is placed in the system's location Ø; and when the branch is made to the POP transfer vector in 100B - 177B, it is to the system's transfer vector in the system's 100B - 177B. POPs with Bit Ø set to 1 thus all branch to memory invisible to the user and are termed SYSPOPs.

Because of their great resemblance to machine instructions (now not even requiring the loading of a subroutine into visible memory and the placing of the correct branch into the visible transfer vector), SYSPOPs are indistinguishable from machine instructions, except that they may take a little longer to execute. In effect there are 64 new "instructions" now available to a user.

Through this means all of the instructions denied a user because their execution might bring him in conflict with someone else (the priviledged instructions) have been replaced. In addition, a great number of subroutines which might be called frequently by a typical programmer have been installed in the system and are immediately available via SYSPOPs. This reduces considerably the necessity for a user to have to retrieve a simple library subroutine and install it in his program. It is already there (in system space); all he has to do is call it.

Of the various system calls, many fall into the category ideally suited to the POP: a single parameter (and possibly the A register) is involved. Accordingly such SYSPOPs look like normal machine instructions, and each is assigned its own position in the transfer vector and has its own mnemonic code for use with assembly language. Others, however, either take no parameter or take several. These cases all use the same SYSPOP code, BRS (branch to system); and use the address field to further specify which action to take. Hence it is possible to have many more than 64 system calls.

3. MACHINE INSTRUCTIONS

This section contains a description of SDS 940 instructions, grouped by functional category. With the description of each instruction is a diagram representing the format of the instruction. Preceding this diagram is the assembler mnemonic code that identifies the instruction and the name of the instruction.

Within the instruction diagram, the following conventions are used.

1. The letter "X" in bit position 1 indicates that the instruction invokes indexing if bit position 1 contains a 1 (indexing adds no additional time to instruction execution). If the diagram contains a 0 in bit position 1, indexing does not apply to the instruction and an unpredictable operation occurs if indexing is attempted.
2. Bit positions 3-8 contain a 2-digit octal number that is the operation code of the instruction.
3. The letter "I" in bit position 9 indicates that the instruction invokes indirect addressing if bit position 9 contains a 1 (indirect addressing adds 1 memory cycle for each level). If the diagram contains a 0 in bit position 9, indirect addressing does not apply to the instruction and an unpredictable operation occurs if indirect addressing is attempted.

Following the description of the instruction is a symbolic list of all registers, indicators, and memory locations that can be affected by the instruction. The following symbols are used:

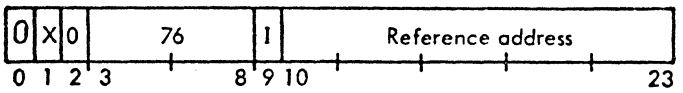
A	A register
B	B register
AB	Combined A and B registers
X	Index register
P	P (program counter) register
Of	Overflow indicator
EL	Effective location

Parentheses are used to denote "contents of". For example, "(A)" denotes "contents of the A register". The contents of registers and the addresses and contents of memory locations are expressed, in this manual, as octal numbers followed by the letter "B". All numbers (except in instruction diagrams) not followed by the letter "B" are decimal base.

Subscripted numbers identify inclusive bit positions. For example, "$(A)_{0-11}$" indicates "the contents of bit positions 0 through 11 of the A register".

LOAD/STORE INSTRUCTIONS

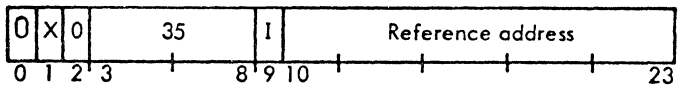
LDA LOAD A



LDA loads the effective word into the A register.

Affected: (A) Timing: 2

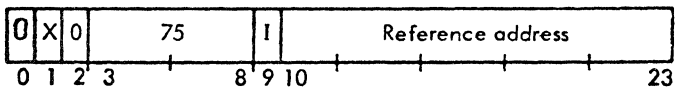
STA STORE A



STA stores the contents of the A register in the effective location.

Affected: (EL) Timing: 3

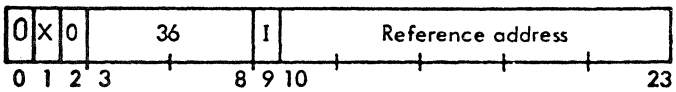
LDB LOAD B



LDB loads the effective word into the B register.

Affected: (B) Timing: 2

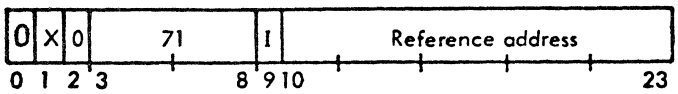
STB STORE B



STB stores the contents of the B register in the effective location.

Affected: (EL) Timing: 3

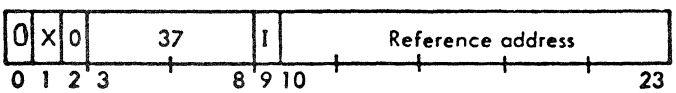
LDX LOAD INDEX



LDX loads the effective word into the index register.

Affected: (X) Timing: 2

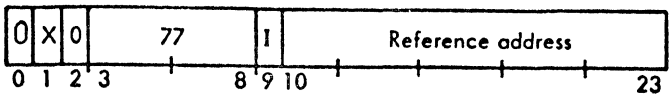
STX STORE INDEX



STX stores the entire contents of the index register in the effective location.

Affected: (EL) Timing: 3

EAX COPY EFFECTIVE ADDRESS INTO INDEX

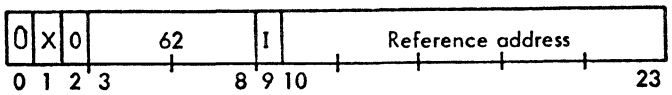


EAX copies the effective virtual address into bit positions 10-23 of the index register. The ten most significant bits of the index register (0-9) are unaffected in the normal and user modes.

The process of computing an effective address for this instruction operates as in a LOAD A instruction, except that instead of obtaining the contents of the actual location, the effective virtual address is used as the operand. For example, if execution of this instruction occurs with a zero indirect address bit and a zero in the index field, then the actual bit configuration in the address field of EAX is copied into bit positions 10-23 of the index register.

Affected: $(X)_{0,\ 10\text{-}23}$ Timing: 2

XMA EXCHANGE MEMORY AND A

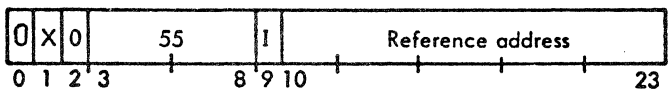


XMA loads the effective word into the A register and, simultaneously, stores the contents of the A register in the effective location.

Affected: (A), (EL) Timing: 3

ARITHMETIC INSTRUCTIONS

ADD ADD



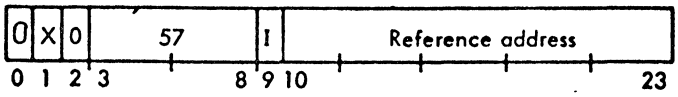
ADD algebraically adds the effective word to the contents of the A register and loads the sum into the A register.

After execution of ADD, bit position 0 of the index (X) register contains the carry from bit position 0 of the 24-bit adder. Therefore, the programmer should be careful when attempting to hold a full word quantity in X while performing an ADD.

If both operands have the same sign but the sign of the sum is different, overflow has occurred, in which case the computer sets the overflow indicator; otherwise, the overflow indicator is unaffected.

Affected: (A), $(X)_0$, Of Timing: 2

ADC ADD WITH CARRY



ADD WITH CARRY is used to perform multiprecision addition. Using the instruction ADD, the program adds the 24 low-order bits of the numbers (ADD automatically retains the carry in the sign position of the X register). Then, the program adds the next 24 bits of the numbers, using ADC, which also adds the carry bit (previously generated) into the low-order position of the adder. The program then continues with as many ADC instructions as are necessary to add the numbers.

After execution of ADC, bit position 0 of the index (X) register contains the carry from bit position 0 of the 24-bit adder. Therefore, the programmer should be careful when attempting to hold a full word quantity in X while performing an ADD WITH CARRY.

If both operands have the same sign but the sign of the sum is different, an overflow has occurred, in which case, the computer sets the overflow indicator to 1; otherwise, the computer resets the overflow indicator to 0.

Affected: (A), $(X)_0$, Of Timing: 2

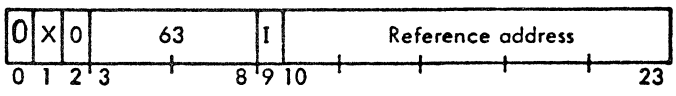
Example:

Assume the A and B registers contain a double-precision number to which the double-precision number in locations M (15034166B) and N (12300000B) is to be added. The less significant halves of the numbers are in the B register and in location N.

The program is:

Instruction	(A, B)	$(X)_0$
(Prior to execution)	20314624, 71510426B	-
XAB[†]	71510426, 20314624B	-
ADD N	04010426, 20314624B	1
XAB	20314624, 04010426B	1
ADC M	35351013, 04010426B	0

ADM ADD A TO MEMORY



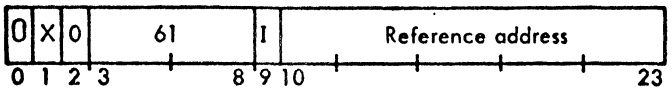
ADM adds the contents of the A register to the effective word and stores the result in the effective location.

If both operands have the same sign but the sign of the result is opposite, an overflow has occurred, in which case the computer sets the overflow indicator to 1; otherwise, the overflow indicator is unaffected.

Affected: (EL), Of Timing: 3

[†]XAB is the mnemonic for the instruction EXCHANGE A AND B (see "Register Change Instructions").

MIN MEMORY INCREMENT

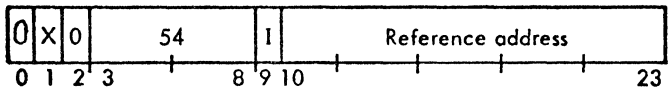


MIN adds 1 to the value of the effective word and stores the resulting sum in the effective location.

Overflow occurs with this instruction if and only if the effective word is 37777777B before execution, in which case 40000000B is the result in the effective location and the overflow indicator is set to 1. If no overflow occurs, the overflow indicator is unaffected.

Affected: (EL), Of **Timing: 3**

SUB SUBTRACT



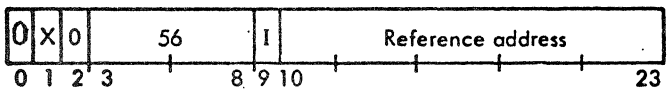
SUBTRACT inverts (forms the one's complement of) the effective word, adds the inverted word plus 1 to the contents of the A register, and loads the result into the A register.

After execution of SUB, bit position 0 of the index (X) register contains the carry from bit position 0 of the 24-bit adder. Therefore, the programmer should be careful when attempting to hold a full word quantity in X while performing a subtraction.

If the sign of the value in A is equal to the sign of the inverted word but the sign of the result is different, overflow has occurred, in which case, the computer sets the overflow indicator to 1; otherwise, the overflow indicator is unaffected.

Affected: (A), $(X)_0$, Of **Timing: 2**

SUC SUBTRACT WITH CARRY



SUBTRACT WITH CARRY is used to perform multiple-precision subtractions. The program uses the instruction SUBTRACT to subtract the low-order 24 bits of the numbers first (SUB automatically retains the carry in the sign position of the X register). The program then subtracts the next 24 bits of the numbers, using SUC, which also adds the carry bit (previously generated in the sign position of the X register) into the low-order bit position of the adder. The program then continues with as many SUC instructions as are necessary to subtract the numbers.

After execution of SUC, bit position 0 of the index (X) register contains the carry from bit position 0 of the 24-bit adder. Therefore, the programmer should be careful when attempting to hold a full word quantity in X while performing a SUBTRACT WITH CARRY.

If the sign of the value in A is equal to the sign of the inverted word but the sign of the result in A is opposite, overflow has occurred, in which case the computer sets the overflow indicator to 1; otherwise, the computer resets the overflow indicator to 0.

Affected: (A), $(X)_0$, Of **Timing: 2**

Example:

Assume that registers A and B and memory location M contain a triple-precision number from which the triple-precision number in locations L, L+1, and L+2 is subtracted.

(A, B, M)

36142070B, 31567000B, 10000001B

(L, L+1, L+2)

14236213B, 46120000B, 10000000B

The sign of one triple-precision number is in A_0, while its 71 binary digits are in A_{1-23}, B_{0-23}, and M_{0-23}. The sign of the other number is in L_0, and its 71 digits are in L_{1-23}, $L+1_{0-23}$, and $L+2_{0-23}$.

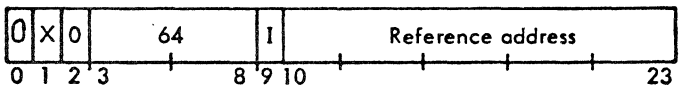
Execution:

Instruction	(A, B) after execution	$(X)_0$
XMA M	10000001, 31567000B	-
SUB L+2	00000001, 31567000B	0
XMA M	36142070, 31567000B	0
XAB	31567000, 36142070B	0
SUC L+1	63447000, 36142070B	1
XAB	36142070, 63447000B	1
SUC L	21704654, 63447000B	0

Answer:

21703654, 63447000, 00000001B

MUL MULTIPLY



MULTIPLY multiplies the contents of the A register by the effective word and loads the fraction product into the A and B registers, with the more significant portion in A. The original contents of B do not affect the operation of the MULTIPLY instruction and are destroyed. The sign of the product is in A_0; the bit in B_0 is part of the product, not treated as a sign bit. Since the product contains at most 46 significant bits, the content of B_{23} is zero.

If the multiplier and multiplicand are both considered integers (i.e., with a binary point to the right of bit position 23), the binary point of the product is to the right of bit position 22 of the B register; thus, the entire result must be shifted 1 bit position to the right to obtain the correct integer product.

If the multiplier and multiplicand both have the value 40000000B, overflow occurs and the computer sets the overflow indicator to 1; otherwise, the overflow indicator is not affected.

Affected: (A), (B), Of **Timing: 4**

Example, multiplication of 3 by 3:

		Before exectuion	After execution
(A, B)	=	00000003, xxxxxxxxB	00000000, 00000022B
EW	=	00000003B	00000003B

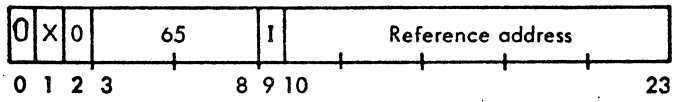
Note that

00000000, 00000011B scaled at 47

is equal to

00000000, 00000022B scaled at 46

DIV DIVIDE



DIVIDE divides the contents of the A and B registers, treated as a double-precision number, by the effective word, loads the fractional quotient into the A register, and loads the fractional remainder into the B register.

During execution of the DIV instruction, the contents of the A and B registers (dividend) taken as a double-precision number are divided by the single-precision contents of the effective location (divisor). If the dividend is a single-precision number, the program should clear the B register prior to executing DIV, or erroneous results may occur. Although a double-length dividend is used, DIV is a single-precision operation; it should not be confused with a double-precision divide operation that uses a double-length divisor and produces a double-length quotient.

After execution of DIV, the single-precision quotient replaces the contents of the A register, and the remaining portion of the dividend that has not been divided (undivided remainder) replaces the contents of the B register. The quotient is signed in accordance with algebraic convention, that is, positive if dividend and divisor signs are alike, but negative otherwise. However, DIV generates only 23 magnitude bits and, if the magnitude of the quotient is so small as to require more than 23 bits to resolve, DIV may produce a zero quotient regardless of the required sign; but the remainder reflects the undivided portion of the original dividend. The binary scaling of the quotient is equal to the dividend scale factor minus the divisor scale factor.

The undivided remainder replaces the contents of the B register and has the same sign as the original dividend. It is scaled, in B, at dividend scaling minus 23.

No overflow occurs if $-1 \leq \frac{(A, B)}{EW} < 1$ (if the quotient is greater than or equal to minus one but strictly less than plus one). If the quotient exceeds these boundaries, overflow occurs and the computer sets the overflow indicator to 1. In this latter case, the results are not arithmetically correct.

Affected: (A, B), Of Timing: 10

Example 1:

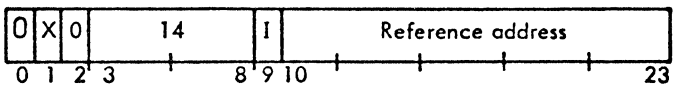
		Before execution	After execution
(A, B)	=	00000000, 00000016B	00000002, 00000001B
EW	=	00000003B	00000003B
Of	=	x	x

Example 2:

(A, B)	=	37777777, 00000002B	40000000, 00000001B
EW	=	44433343B	44433343B
Of	=	x	1

LOGICAL INSTRUCTIONS

ETR EXTRACT



ETR performs a logical AND between corresponding bits of the A register and the effective word and loads the result into A. This instruction performs the operation (bit by corresponding bit) according to the following table:

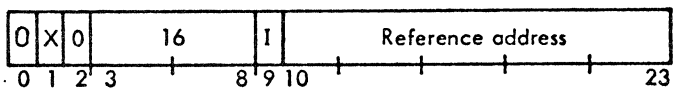
A_i	EW_i	Result in A_i
0	0	0
0	1	0
1	0	0
1	1	1

Affected: (A) Timing: 2

Example:

		Before execution	After execution
(A)	=	64231567B	00231400B
EW	=	00777600B	00777600B

MRG MERGE



MRG performs a logical inclusive OR between corresponding bits of the A register and the effective word and loads the result into A. This instruction performs the operation (bit by corresponding bit) according to the following table:

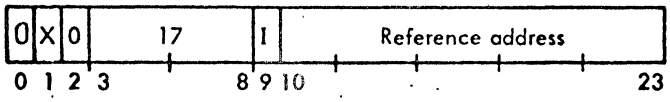
A_i	EW_i	Result in A_i
0	0	0
0	1	1
1	0	1
1	1	1

Affected: (A) Timing: 2

Example:

		Before execution	After execution
(A)	=	06445254B	06746756B
EW	=	02340712B	02340712B

EOR EXCLUSIVE OR



EOR performs a logical exclusive OR between corresponding bits of the A register and the effective word and loads the result into A. This instruction performs the operation (bit by corresponding bit) according to the following table:

A_i	EW_i	Result in A_i
0	0	0
0	1	1
1	0	1
1	1	0

Affected: (A) **Timing: 2**

Example:

	Before execution	After execution
(A) =	34165031B	44112010B
EW =	70077021B	70077021B

The proper memory word configuration logically inverts selected bit positions of the A register. If the effective word is 77777777B, a one's complement of A results.

REGISTER CHANGE INSTRUCTIONS

The facility to operate on and exchange data between the A, B, and index registers is available within the set of micro-instructions in the register change group.

All instructions in the group use the same operation code, 46B. Bit positions 1 and 14 through 23 of the address field specify the function to be performed by each micro-instruction. The programmer may specify combinations of address bits to perform simultaneous operations.

If the selected bits specify that the computer copy two registers into a third during one operation, a merge of the former two registers into the latter results. If the selected control bits specify that the computer copy into a register and clear that same register, the clear operation has no effect. The function of each address bit is:

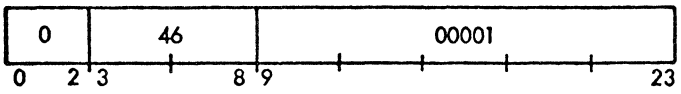
Bit	Function
1	Clear X
14	Copy -(A) into A
15	Copy (A) into X
16	Copy (X) into A
17	Bits 15-23 only[†]
18	Copy (X) into B
19	Copy (B) into X
20	Copy (B) into A
21	Copy (A) into B
22	Clear B
23	Clear A

[†]See STORE EXPONENT, LOAD EXPONENT, and EXCHANGE EXPONENTS

Indirect addressing and indexing do not apply to these instructions.

These instructions require one machine cycle regardless of the number of functions performed. As an aid to the programmer, the most useful combinations have mnemonic designations assigned to them that are recognized by standard SDS 940 programming systems.

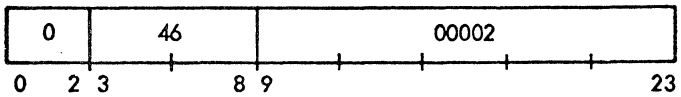
CLA CLEAR A



CLA clears the contents of the A register to zero.

Affected: (A) **Timing: 1**

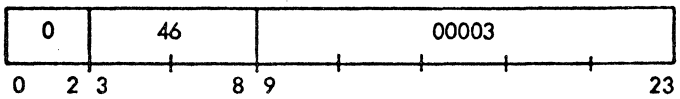
CLB CLEAR B



CLB clears the contents of the B register to zero.

Affected: (B) **Timing: 1**

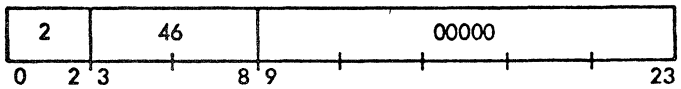
CLAB CLEAR AB



CLAB clears the contents of both the A and B registers to zero.

Affected: (A), (B) **Timing: 1**

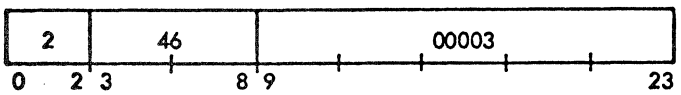
CLX CLEAR INDEX



CLX clears the contents of the index (X) register to zero.

Affected: (X) **Timing: 1**

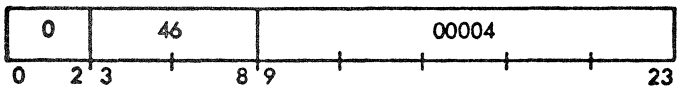
CLEAR CLEAR A, B, AND X



CLEAR clears the contents of the A, B, and index (X) registers to zero.

Affected: (A), (B), (X) **Timing: 1**

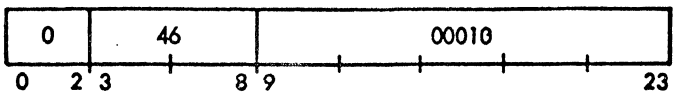
CAB COPY A INTO B



CAB copies the contents of the A register into the B register.

Affected: (B) **Timing: 1**

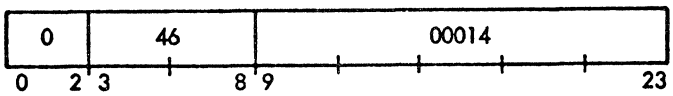
CBA COPY B INTO A



CBA copies the contents of the B register into the A register.

Affected: (A) Timing: 1

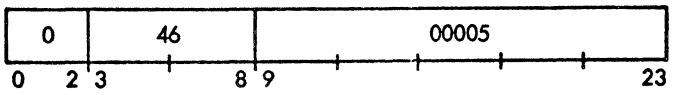
XAB EXCHANGE A AND B



XAB copies the contents of the A register into the B register and, simultaneously, copies the contents of the B register into the A register.

Affected: (A), (B) Timing: 1

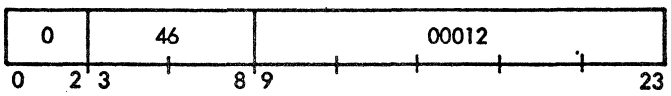
ABC COPY A INTO B, CLEAR A



ABC copies the contents of the A register into the B register and then clears the A register to zero.

Affected: (A), (B) Timing: 1

BAC COPY B INTO A, CLEAR B



BAC copies the contents of the B register into the A register and then clears the B register to zero.

Affected: (A), (B) Timing: 1

CAX COPY A INTO INDEX

CAX copies the contents of the A register into the index register.

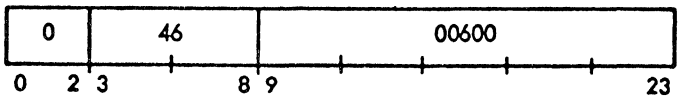
Affected: (X) Timing: 1

CXA COPY INDEX INTO A

CXA copies the contents of the index register into the A register.

Affected: (A) Timing: 1

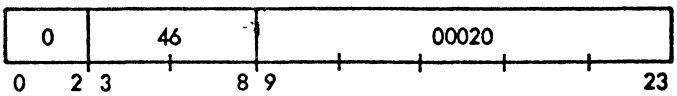
XXA EXCHANGE INDEX AND A



XXA copies the contents of the index register into the A register and, simultaneously, copies the contents of the A register into the index register.

Affected: (A), (X) Timing: 1

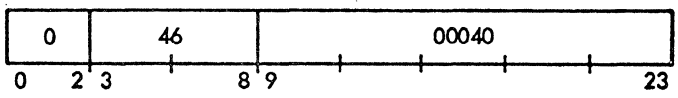
CBX COPY B INTO INDEX



CBX copies the contents of the B register into the index register.

Affected: (X) Timing: 1

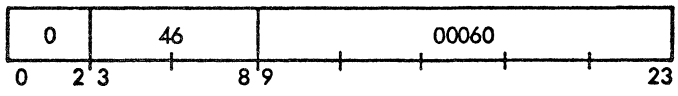
CXB COPY INDEX INTO B



CXB copies the contents of the index register into the B register.

Affected: (B) Timing: 1

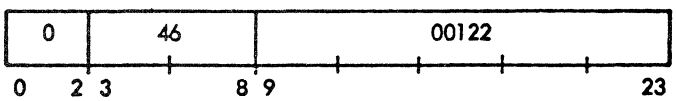
XXB EXCHANGE INDEX AND B



XXB copies the contents of the index register into the B register and, simultaneously, copies the contents of the B register into the index register.

Affected: (B), (X) Timing: 1

STE STORE EXPONENT



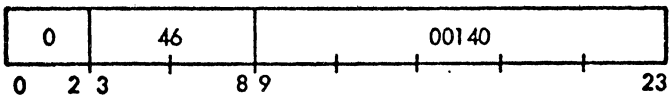
STE copies the 9 least significant bits of the B register into the 9 least significant bit positions of the index register, extends bit 15 of the index register (the sign of the exponent) into bit position 0 of the index register, and then clears the 9 least significant bit positions of B.

Affected: $(B)_{15-23}$, (X) Timing: 1

Example:

	Before execution	After execution
(B) =	64152713B	64152000B
(Index) =	---	77777713B

LDE LOAD EXPONENT



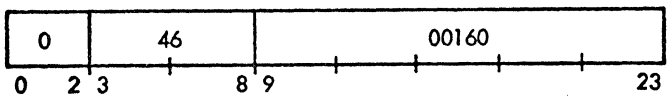
LDE copies the 9 least significant bits of the index register into the 9 least significant bit positions of the B register. The 9 least significant bit positions of B are cleared prior to the transfer.

Affected: $(B)_{15-23}$ Timing: 1

Example:

	Before execution	After execution
(B) =	34765712B	34765151B
(Index) =	00000151B	00000151B

XEE EXCHANGE EXPONENTS



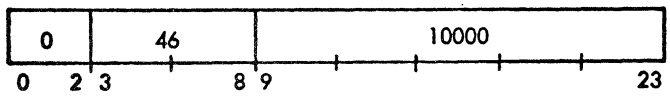
XEE exchanges the 9 least significant bits of the B register with the 9 least significant bits of the index register. The exchange loses no information. The new bit 15 of the index register (the sign of the exponent) is then extended into bit position 0.

Affected: $(B)_{15-23}$, (X) Timing: 1

Example:

	Before execution	After execution
(B) =	67142355B	67142133B
(Index) =	77777133B	00000355B

CNA COPY NEGATIVE INTO A



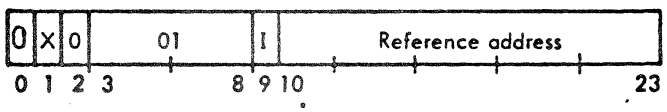
CNA copies the two's complement of the contents of the A register into the A register.

Affected: (A) Timing: 1

BRANCH INSTRUCTIONS

Branch instructions conditionally or unconditionally change the course of the program by altering the contents of the program counter. The programmer should note that these instructions branch to locations determined by the effective address; this means that the branch can operate with all levels of indirect and indexed addressing.

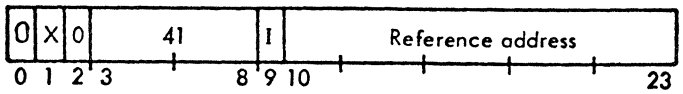
BRU BRANCH UNCONDITIONALLY



BRU takes the next instruction from the location determined by the effective address.

Affected: (P), highest-priority active interrupt level Timing: 1

BRX INCREMENT INDEX AND BRANCH



BRX adds 1 to the contents of the index register. If the resultant index register value contains a 1 in bit position 9, the computer transfers control to the effective location. If not, it takes the next instruction in sequence.

If a BRX instruction is indexed, any transfer of control is to the effective address determined by the value of the index immediately prior to the execution of BRX. The test for transfer is on the incremented value of the index register, just as if the BRX instruction were not indexed.

The 9 most significant bits of the index register (bits 0-8) have no effect on the execution of the instruction, but may be affected by it.

Affected: (X), (P) Timing: 1, if branch
2, if no branch

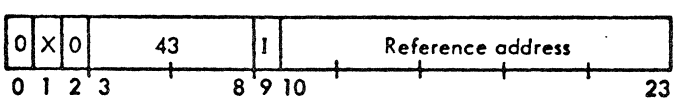
Example:

Location	Instruction	(X Register)
0777B	STA 1500B	77777776B
1000B	BRX 1006B	77777777B
1001B	LDA 2000B	
.	.	
.	.	
.	.	
1006B	BRX 1001B	00000000B
1007B	LDA 2100B	00000000B

The execution of these instructions is in the following order as given by their locations:

0777B
1000B
1006B
1007B

BRM MARK PLACE AND BRANCH



MARK PLACE AND BRANCH performs the following operations:

1. stores the state of the overflow indicator in bit position 0 of the effective location

2. stores the contents of the P register (the address of the BRM instruction) in bit positions 10-23 of the effective location

BRM loads the value of the effective address plus 1 into the P register; thus, the next instruction is taken from the next location after effective location. If the BRM instruction is executed as the operand of an EXECUTE instruction (see page 30), the stored P register value is the address of the initial EXECUTE instruction rather than the address of the BRM instruction.

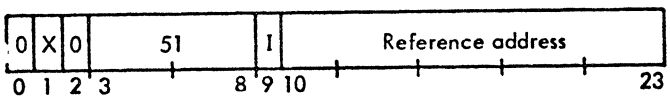
BRM is used to enter subroutines where a return to the main program is desired after completing the subroutine. The subroutine can return program control to the main program by executing a BRI instruction.

Affected: (EL), (P) Timing: 2

Example: BRM 1517B

		Before execution	After execution
(P)	=	522B	1518B
(EM3)	=	3	3
(EM2)	=	2	2
(Of)	=	1	1
(1517B)	=	xxxxxxxxB	53200522B
Mode	=	user	user

BRR RETURN BRANCH



RETURN BRANCH performs a logical OR between bit 0 of the effective word and the overflow indicator, places the result in the overflow indicator, and then loads the P register with a value equal to 1 plus the contents of bit positions 10-23 of the effective location.

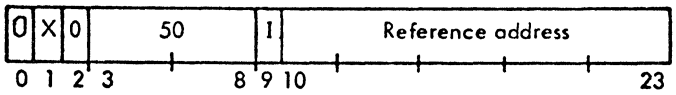
Affected: Of, (P) Timing: 2

Example: BRR 1517B

		Before execution	After execution
(P)	=	1540B	523B
(Of)	=	0	1
(1517B)	=	53200522B	53200522B

TEST AND SKIP INSTRUCTIONS

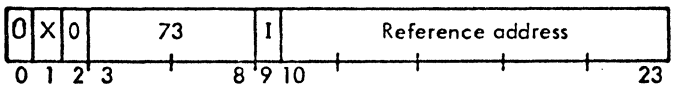
SKE SKIP IF A EQUALS MEMORY



SKE compares the contents of the A register with the effective word. If the contents of A equal the effective word, the computer skips the next instruction in sequence and executes the following instruction. If the contents of A do not equal the effective word, the computer executes the next instruction in sequence.

Affected: (P) Timing: 2, if no skip
3, if skip

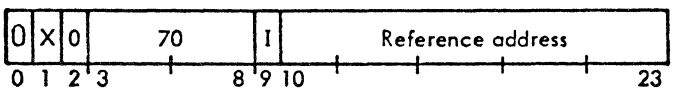
SKG SKIP IF A GREATER THAN MEMORY



SKG algebraically compares the contents of the A register with the effective word. If the contents of A are greater than the effective word, the computer skips the next instruction in sequence and executes the following instruction. If the contents of A are less than or equal to the effective word, the computer executes the next instruction in sequence.

Affected: (P) Timing: 2, if no skip
3, if skip

SKM SKIP IF A EQUALS MEMORY ON B MASK



SKM compares selected bits of the A register with corresponding bits of the effective word. If the selected bits in A are all identical to corresponding bits of the effective word, the computer skips the next instruction in sequence and executes the following instruction. If the selected bits in the A register are not all identical to corresponding bits of the effective word, the computer executes the next instruction in sequence.

The programmer selects the bits in A to be compared by placing ones in the corresponding bit positions of the B register and zeros in the remaining bit positions of B.

SKM treats the contents of A, B, and the effective location to be unsigned, 24-bit, nonnumeric quantities, and does not alter them.

Affected: (P) Timing: 2, if no skip
3, if skip

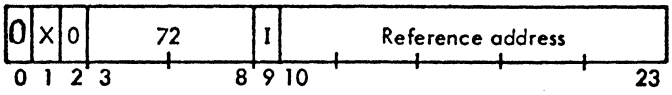
Example:

(A)	(B)	(EL)
00043007B	00177000B	57643240B

Since SKM compares bit positions 8-14 only (as determined by B), and (A) = (EL) in these positions, a skip occurs. Note

that if (B) = 0, a skip occurs regardless of (A) and (EL). Note also that if (B) = 77777777B, the operation of SKM is identical to that of the instruction SKE.

SKA SKIP IF A AND MEMORY DO NOT COMPARE ONES



SKA compares the contents of the A register, bit by bit, with the effective word. If the contents of the A register and the effective word do not have ones in any corresponding bit positions, the computer skips the next instruction in sequence and executes the following instruction. If the contents of the A register and the effective word do have ones in at least one corresponding bit position, the computer executes the next instruction in sequence.

The instruction logically ANDs corresponding bits in A and the effective word, based on the following table:

(A)	EW	Result
0	0	0
0	1	0
1	0	0
1	1	1

If the result produces a 1 in any bit position, a skip does not occur.

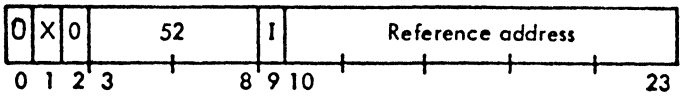
Note: Different configurations of the effective word result in a wide variety of conditional operations for use by the programmer. Some representative configurations are:

Effective word configuration	Operation
40000000B	Skip if (A) is positive
77777777B	Skip if (A) = 0
00000001B	Skip if (A) is even

Contents of A register	
40000000B	Skip if effective word is positive
77777777B	Skip if effective word = 0
00000001B	Skip if effective word is even

Affected: (P) Timing: 2, if no skip
3, if skip

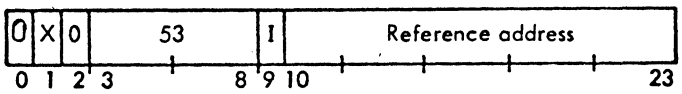
SKB SKIP IF B AND MEMORY DO NOT COMPARE ONES



The operation of SKB is identical to that of SKA, but uses the contents of the B register instead of the contents of the A register.

Affected: (P) Timing: 2, if no skip
3, if skip

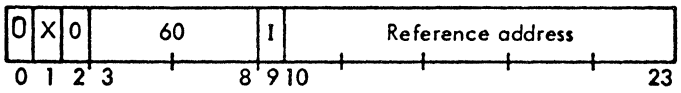
SKN SKIP IF MEMORY NEGATIVE



If the effective word is a negative value (i.e., bit 0 of the effective word is a 1), the computer skips the next instruction in sequence and executes the following instruction. If the effective word is a positive or zero value, the computer executes the next instruction in sequence.

Affected: (P) Timing: 2, if no skip
3, if skip

SKR REDUCE MEMORY, SKIP IF NEGATIVE

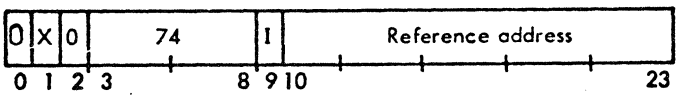


SKR reduces the value of the effective word by one, places the result in the same location, and then tests the effective word for being a negative value. If the effective word is a negative value after being reduced, the computer skips the next instruction in sequence and executes the following instruction. If the effective word is a positive or zero value after being reduced, the computer executes the next instruction in sequence.

An overflow occurs if the initial value of the effective word is 40000000B, in which case the resulting effective word is 37777777B, and the overflow indicator is set. If no overflow occurs, the overflow indicator is unaffected.

Affected: (EL), Of, (P) Timing: 3

SKD DIFFERENCE EXPONENTS AND SKIP



SKD subtracts bits 15 through 23 of the effective word from bits 15 through 23 of the B register, and stores the absolute magnitude of the difference in the X register. If the 9 low-order bits of the effective word are less than or equal to the 9 low-order bits of the B register, the computer executes the next instruction in sequence; otherwise, the computer skips the next instruction in sequence and executes the following instruction.

Affected: $(X)_{15-23}$ Timing: 2, if no skip
3, if skip

SHIFT INSTRUCTIONS

The shift instructions operate on the contents of the A and B registers and offer a complete facility for right and left shifting, cycling, and normalizing the contents of these two registers. The A and B registers, in combination, form a double-length register whose double-length contents can be shifted, cycled, or normalized. This double-length register is named "AB".

When the contents of the AB register shift right, bits from bit position 23 of the A register shift into bit position 0 of the B register. When the AB register shifts left, bits from bit position 0 of the B register shift into bit position 23 of the A register.

The 48-bit contents of the AB register may be cycled using the shift instructions. When the contents of the AB register cycle, the bits that shift from one end of the one register copy into the other end of the other register.

These instructions use the instruction code to determine the direction of shift (66 = right; 67 = left); bits 10-11 (octal position 3) of the instruction address determine the method of shifting as follows:

Bits 10, 11	Function
00	AB shift
10	AB cycle
01	Normalize (left only)

Since the type of shift and number of shifts are determined by bits 10 through 23 of the effective virtual address, indirect addressing and indexing drastically alter the action specified in a shift instruction. When computing the effective virtual address for a shift instruction,

14-bit indexing is performed with all indirectly addressed operands, and

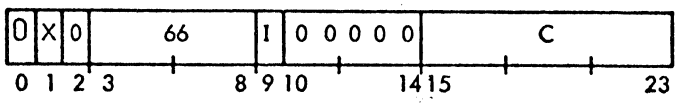
9-bit indexing is performed with all directly addressed operands.

That is, indexing with a direct address can affect only the 9-bit shift count.

When the computer decodes a shift instruction, bit positions 15 through 23 of the effective address of the instruction determine the amount of the shift. The computer treats these nine bits as an unsigned count. If the initial count is equal to zero, no shifting occurs. If the initial count is greater than 48, it is set to 48 prior to shifting. Once the shift begins, the count is reduced by 1 for each position shifted, until it reaches zero. The count C in the following instructions indicates the number of places to be shifted. Shift timing is:

Left shift and normalize count	Cycles	Right shift count
0 - 6	2	0 - 3
7 - 26	3	4 - 14
27 - 46	4	15 - 25
47 - 48	5	26 - 36
	6	37 - 47
	7	48

RSH RIGHT SHIFT AB



RSH shifts the contents of the AB register (that is, A and B registers) right the number of places specified by bits 15 through 23 of the effective address. The bit in the sign position of A does not shift, but its value is copied into the vacated bit positions of the shifted number. The bit in the sign position of B is shifted as a magnitude bit. Bits shifted out of A_{23} shift into B_0. Bits shifting past B_{23} are lost.

Affected: (AB) **Timing: 2-7**

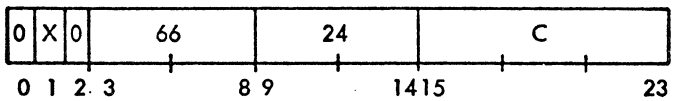
Example:

The instruction is: RSH 18

	Before execution	After execution
(A, B) =	45261237, 27651260B	77777745, 26123727B

Note: This instruction may be used to perform scaling of floating-point numbers by use of indexing, where the difference of the exponents is in the index register as a positive quantity.

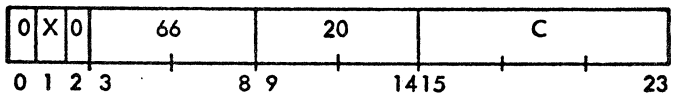
LRSH LOGICAL RIGHT SHIFT AB



LRSH shifts the contents of AB right the number of places specified by bits 15 through 23 of the effective address. The bits in the sign position of A and the sign position of B shift with the rest of the number. Vacated bit positions on the left are filled with zeros. Bits shifting out of A_{23} shift into B_0. Bits shifting past B_{23} are lost.

Affected: (AB) **Timing: 2-7**

RCY RIGHT CYCLE AB



RCY shifts the contents of the AB register right the number of places specified in bits 15 through 23 of the effective address. The bits in the sign positions of A and B shift like any other bits in the number. Bits shifting out of A_{23} shift into B_0. Bits shifting out of B_{23} shift into A_0. The computer treats the double-length register as if it were circular and cycles it onto itself; it loses no bits.

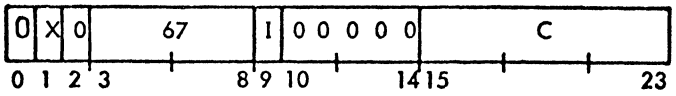
Affected: (AB) **Timing: 2-7**

Example:

The instruction is: RCY 15

	Before execution	After execution
(A, B) =	61235703, 41537701B	37701612, 35703415B

LSH LEFT SHIFT AB



LSH shifts the contents of the AB register left the number of places specified in bits 15 through 23 of the effective address. Bits shift left through the sign position of A, but when a bit, different in value from the original sign, shifts into the sign position, the computer sets the overflow indicator. Bits shifting out of B_0 shift into A_{23}. Bits shifting past position 0 in A are lost. Zeros fill the vacated bit positions on the right end of the B register.

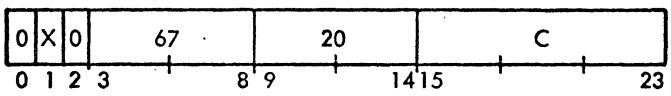
Affected: (AB), Of Timing: 2-5

Example:

The instruction is: LSH 18

	Before execution	After execution
(A, B) =	46712370, 64132711B	70641327, 11000000B

LCY LEFT CYCLE AB



LCY shifts the contents of the AB register left the number of places specified in bits 15 through 23 of the effective address. The bits in the sign positions of A and B shift like any other bits in the number. Bits shifting out of B_0 shift into A_{23}. The instruction copies bits that shift from bit position 0 of A into bit position 23 of B. The computer treats the double-length register as if it were circular and cycles it onto itself; it loses no bits.

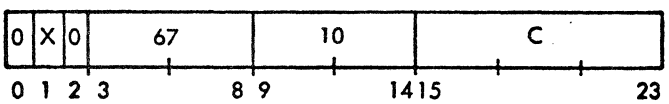
Affected: (A, B) Timing: 2-5

Example:

The instruction is: LCY 9

	Before execution	After execution
(A, B) =	71432560, 34156723B	32560341, 56723714B

NOD NORMALIZE AND DECREMENT X



NOD shifts the contents of the AB register left until (1) a bit appears in position 1 of A that is not equal to the bit in the sign position of A, or (2) until C shifts occur. The computer keeps count of the number of places shifted and when the normalize operation is completed, it subtracts the count from the contents of the index register and places the result back into the index. If, in the attempt to normalize, shifting exceeds 48 places, the contents of the AB register were initially zero. In this case, the computer subtracts 48 from the index register. Zeros fill the vacated positions.

The number C, placed in address bit positions 15 through 23, is an upper limit for the number of left shifts that will occur. The programmer must ensure that C is sufficiently large to permit a complete normalization.

Affected: (A, B), (X) Timing: 2-5

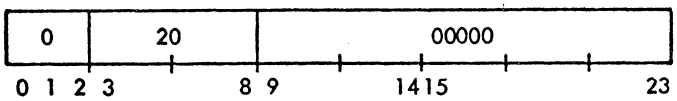
Example:

The instruction is: NOD 30

	Before execution	After execution
(A, B) =	00004632, 76124035B	23153705, 20164000B
(X) =	00000000B	77777765B

CONTROL INSTRUCTIONS

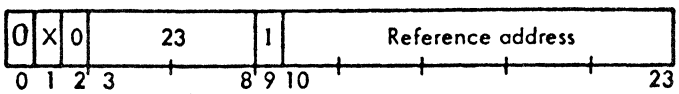
NOP NO OPERATION



Executing NOP does not affect the A register, B register, X register, or memory. Indirect addressing and indexing do not apply to this instruction.

Affected: None Timing: 1

EXU EXECUTE



EXU causes the effective word to be executed as an instruction without altering the contents of the program counter. If the effective word is not a branch, skip, or another EXECUTE instruction, the computer executes the next instruction, after it executes the effective word.

If the effective word is a branch instruction, program control goes to the effective address of the branch and not to the next instruction in sequence following the EXECUTE instruction.

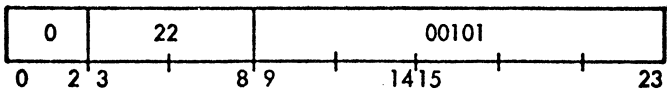
If the effective word is a skip instruction, then, depending on the skip decision, program control returns to the next instruction, or the next instruction plus one, following the EXECUTE instruction.

If the effective word is another EXECUTE instruction, the above process continues identically, with the normal return being the location of the initial EXECUTE instruction plus one. This process can cascade indefinitely.

Affected: Determined by executed instruction Timing: 1 + executed instruction

OVERFLOW INSTRUCTIONS

OVT OVERFLOW INDICATOR TEST AND RESET



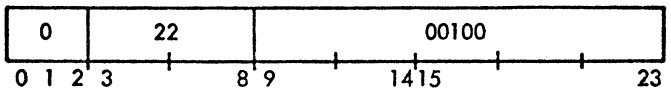
This instruction tests the status of the overflow indicator, skips or not accordingly, and turns the indicator off. If the indicator is off, the computer skips the next instruction in sequence and executes the following instruction. If the indicator is on, the computer turns the indicator off and then executes the next instruction in sequence.

~~In the normal and monitor modes~~, the instruction SKS 20001B may be used to test and reset the overflow indicator.

Affected: (P), Of

Timing: 1, if no skip
2, if skip

OTO OVERFLOW INDICATOR TEST ONLY (940 only)

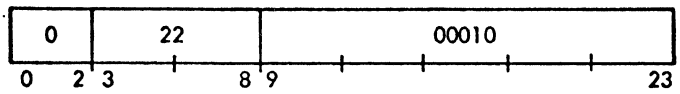


This instruction tests (but does not change) the status of the overflow indicator. If the overflow indicator is on, the computer executes the next instruction in sequence; however, if the overflow indicator is off, the computer skips the next instruction in sequence and executes the following instruction.

Affected: (P)

Timing: 1, if no skip
2, if skip

REO RECORD EXPONENT OVERFLOW



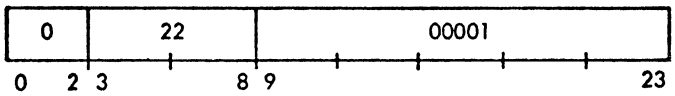
This instruction causes the overflow indicator to be turned on if the content of bit 14 of the index register is not equal to the content of bit 15 of the index register; otherwise, the overflow indicator is not affected.

In the normal and monitor modes, the instruction EOM 20100B may be used to record exponent overflow.

Affected: Of

Timing: 1

ROV RESET OVERFLOW INDICATOR



Affected: Of

Timing: 1