Computer Systems Department

AN/GSA-51A PROGRAMMING MANUAL

February 1968

Keesler Technical Training Center
Keesler Air Force Base, Mississippi

Designed For ATC Course Use

ABOUT THE STUDENT TEXT

STUDENT TEXTS are authorized by the Air Training Command as student training publications for use in training situations peculiar to courses in this Command. They contain specific information required by the student to achieve the learning objectives. It contains the necessary information which is not suitable for student study in other available publications.

STUDENT TEXTS are designed for ATC COURSE USE ONLY. Every effort is made to keep in-use student texts current with technical orders and other directives. Students are cautioned not to use them in preference to technical orders or other authorative documents. When students are authorized to retain student texts they must keep in mind that these publications will not remain current.

AN/GSA-51A PROGRAMMING MANUAL

This student text provides student study and reference material in support of the AN/GSA-51A programming blocks of instruction in Course 2OSR0123-3, BUIC III Computer Programmer.

CONTENTS

CONTENTS (Cont'd)

CHAPTER 1

THE AN/GSA-51A EQUIPMENT

This chapter provides an introduction to the computer equipment used in the BUIC III system. The basic configuration of the AN/GSA-51A is listed. The computer, core memory, and input/output are discussed in detail. Brief descriptions of the remaining pieces of equipment are given. Each will be covered in more detail in the AN/GSA-51A Input/Output programming block of instruction.

AN/GSA-51A EQUIPMENT CONFIGURATION

The commercial name for the computer equipment used by the BUIC system is Burroughs D-825 Modular Processor. The D-825 is a solid-state, internally stored program computer that is based on the principle of total modularity. Total modularity allows the equipment to be freely organized and expanded by the use of combinations of standard computer modules, memory modules, and input/output modules.

The D-825 is organized into a specific configuration for the BUIC III system. The central data processing configuration has been given the name AN/GYK-10, and is comprised of the following:

- 2 computer modules
- 8 core memory modules
- 4 input/output modules
- 2 message processors
- 3 magnetic storage drums

The full configuration of equipment used for the BUIC system is given the name AN/GSA-51A. It includes the AN/GYK-10 equipment and the following terminal devices:

- 10 or 11 data display consoles
- 4 magnetic tape drive units
- 1 tape drive control unit
- 1 status display console
- 1 Flexowriter
- 1 on-line printer
- 1 card reader
- 1 simulator group

DESCRIPTION OF EQUIPMENT

COMPUTER MODULES

The AN/GSA-51A has two computer modules, each being a central processing element for AN/GSA-51A operations. The main function of the computer module is to decode program instructions and to provide the control and logic circuitry and the arithmetic registers necessary for performing the decoded instructions.

Each computer module performs its functions independently of the other computer module. However, each can be programmed to interrupt or begin operation of the other one. Because the computer modules operate independently, two separate programs can be operating simultaneously on the AN/GSA-51A.

There are three functional areas within each computer module. The ARITHMETIC UNIT contains registers and circuitry for performing operations specified by the instructions. The CONTROL UNIT contains circuitry for controlling the operation of instructions and program timing. The SET OF THIN FILM REGISTERS contains 128 registers which are used for data storage and program control.

Each computer module also has an external control panel which is used for performing manual operations necessary to start a program, for performing maintenance operations, and for debugging programs.

CORE MEMORY MODULES

The eight core memory modules are used to hold program instructions and data. Each memory module contains 4096_{10} (or 10000_8) ferrite core locations. Each location has 51 cores (bits) for storing an instruction or data word. Forty-eight of these bits are information bits, one bit is a parity bit, and the remaining two bits are spares. Throughout this document, reference will be made to "48-bit core memory words" because the programmer is concerned only with the 48 information bits in the core memory location and also because this is common terminology among AN/GSA-51A programmers.

All core memory locations may be accessed randomly. Information is transferred between the memory modules and the computer modules and between the memory modules and the input/output modules. Other pieces of equipment which require transfer of information to or from core memory must access the information through the use of the input/output module. Transfer of data in and out of core memory requires 4.33 microseconds.

The eight core memory modules can be thought of as one large memory area. Each memory word is given its own absolute address which distinguishes its location from all other memory locations. The locations are given addresses in consecutive order with octal numbers. They begin with $\emptyset$ and end with 77777_8. The memory modules themselves are numbered beginning with 1. (Reference Figure 1-1).

INPUT/OUTPUT MODULES

The four input/output (I/O) modules provide the control circuitry necessary for data transfer between the core memory modules and the I/O terminal devices. An input/output operation is defined and initiated by computer module action but then proceeds independently under the control of one of the I/O modules. All four I/O modules can be handling separate I/O operations simultaneously.

Compatibility and connection between the I/O modules and the terminal devices is provided by an I/O EXCHANGE. The I/O exchange permits data flow between an I/O module and the terminal device being used. It consists physically of circuitry found in the I/O modules and in the terminal devices.

MESSAGE PROCESSORS

The message processors have three basic functions:

1. To provide a temporary storage for messages accumulated from radar sites, other BUIC NCCs, SAGE DCs, and other related facilities.

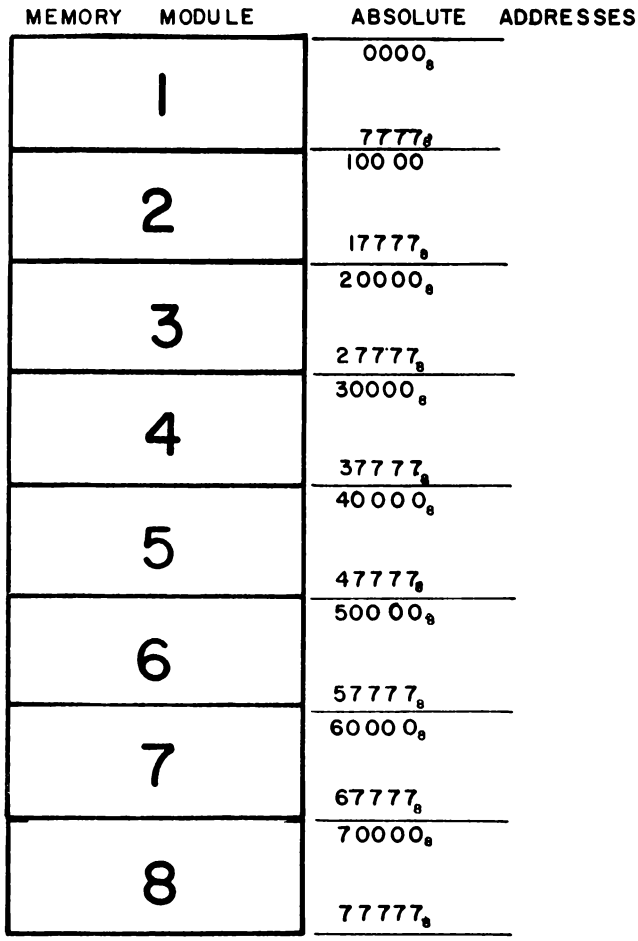


Figure 1-1

2. To change the format of information sent from other facilities to 48-bit words usable by the BUIC system.

3. To change the format of information sent to other facilities from the 48-bit word format to the format used by them.

MAGNETIC STORAGE DRUMS

Each of the first two magnetic storage drums provides $65{,}536_{10}$ words of storage. Some of the words ($39{,}936_{10}$) on each of the two drums may be used for bulk storage of program instructions and data. The remaining $25{,}600_{10}$ on these two drums are used to provide automatic readout and transfer of display data to the data display console. The third magnetic drum is used entirely for bulk storage.

When a display is prepared by the computer and memory modules, it is sent to the magnetic storage drums through the I/O modules. The drums then automatically begin sending the display to the data display consoles and will continue to send it until the display information is erased from the drums.

DATA DISPLAY CONSOLES

The data display consoles are used to display radar and tracking information on cathode-ray type scopes and also to input information into the AN/GYK-10 to be processed by the BUIC programs. The console operators input such information by pressing buttons on keyboard panels on the data display consoles. They may also use a light gun on the main scope to aid in sending positional information to the computer.

STATUS DISPLAY CONSOLE

The status display console provides a means for monitoring the operational status of all AN/GSA-51A equipment, provides a way of applying and removing power for itself and for the rest of the AN/GSA-51A, and assists in the repair of faulty modules.

FLEXOWRITER

The Flexowriter is an electric typewriter and a paper tape punch and reader combined into one unit. It is used for the exchange of information between the operator and the operating program. The paper tape punch and reader are run at the option of the operator. However, at all times, the electric typewriter provides a hard-copy record of the exchange of information regardless of whether the exchange was made through the typewriter or through the paper tape punch and reader.

PUNCH CARD READER

The punch card reader reads 12-row, 80-column punch cards into central processing modules of the AN/GSA-51A. It is capable of reading punch cards at a maximum rate of 200 cards per minute. Information being read is sensed by photoelectric cells at a read station. The information is transferred to the card reader control which translates the card codes into AN/GSA-51A character codes. The character codes are sent, one character at a time, to the I/O module that is controlling the input operation. The I/O module then sends the information to one of the memory modules so that it will be available for use by the controlling program.

ON-LINE PRINTER

The on-line printer is used to print alphanumeric data directly from core memory. It prints a maximum of fifteen words (120 characters) per line at a maximum rate of 600 lines per minute.

MAGNETIC TAPE DRIVE UNITS

The magnetic tape drive units are used to process magnetic tapes which provide bulk storage for large quantities of program instructions and data. Some of the operations of the tape drive units are performed under the control of the tape drive control unit. Those operations which the tape drive units can perform without the use of the control unit are rewind, backspace, advance, load, and unload.

TAPE DRIVE CONTROL UNIT

The one magnetic tape drive control unit controls all four magnetic tape drives. Under its control, the tape drive units perform read, write, advance, backspace, rewind, and erase operations.

SIMULATION GROUP

The simulator group is used to provide the central data processing equipment with manually composed messages that simulate inputs from two interceptor pilots to effect equipment testing and personnel training.

CHAPTER 2

GENERAL CODING FEATURES

This chapter provides a general introduction which details the basic operations of the computer and provides the background necessary to understand the coding instructions and their usage.

WORD STRUCTURES

Programs for the AN/GSA-51A require two basic types of words -- INSTRUCTION WORDS and DATA WORDS. These two types of words differ in function as well as format. However, both types of words are stored in core memory. In some programs, the instruction words and data words are grouped separately and stored into different areas of memory. In other programs, the instruction words and data words are grouped into one continuous set of words which are stored together in core memory. The following paragraphs describe the function and format of the two types of words.

INSTRUCTION WORDS

Instruction words hold the machine language code which gives the computer module most of the information necessary for performing the operations of the program. When a memory location is used to hold an instruction word, the 48 information bits are organized into four 12-bit groups called syllables. (See the diagram below.) Each syllable is used to specify part of the total operation to be performed by an instruction. An instruction word does not necessarily contain one single instruction. It can contain one entire instruction, part of an instruction, or parts of more than one instruction.

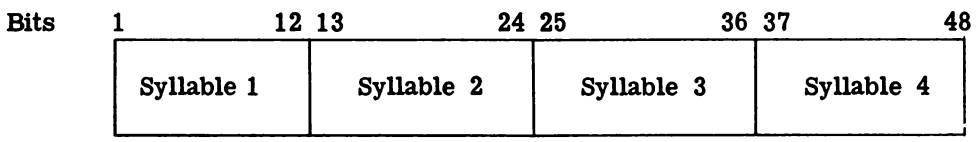


Figure 2-1

DATA WORDS

Data words contain the constant and variable data which are manipulated by the instructions of the program. There are three types of data words: alphanumeric, fixed-point, and floating-point. Each will be discussed individually.

ALPHANUMERIC DATA WORD. The alphanumeric data word is used in performing alphanumeric functions such as input/output operations. The 48-bit word is divided into 8 six-bit bytes. Each of these bytes contains the binary representation of one of the numbers, letters, or symbols used in the AN/GSA-51A. The binary representations follow a 6-bit Hollerith code which is shown in Figure 2-3. The alphanumeric data word does not contain a sign bit.

An important point to remember is that whenever the AN/GSA-51A is commanded by the input/output instruction to print a word from memory, the word will be printed in an alphanumeric format. For example, the word in Figure 2-2 will be printed ABC/123 and not 2122236101020360_8.

BYTES	0	1	2	3	4	5	6	7
BINARY WORD	010001	010010	010011	110001	000001	000010	000011	110000
OCTAL VALUE OF WORD	21	22	23	61	01	02	03	60
CHARACTER REPRESENTED	A	B	C	/	1	2	3	SPACE

Figure 2-2

6 BIT HOLLERITH - - PRINTER & FLEX

	ØØ	1Ø	2Ø	3Ø	4Ø	5Ø	6Ø	7Ø
Ø	Φ	8	+ / ∧	H	-	Q		Y
1	1	9	A	I	J	R	/	Z
2	2	= / ←	B	° / +	K	° / ¬	S	° / ≠
3	3	° / ;	C	.	L	$ / CR	T	,
4	4	' / ×	D	)	M	* / ↑	U	(/ V
5	5	° / (	E	° /]	N	° / [	V	° / :
6	6	° / =	F	° / <	O	° / >	W	° / "
7	7	° / ≧	G	° / ≦	P	° / Δ	X	° / DL

DOUBLE SYMBOL — PRINTER / FLEX

Figure 2-3

FIXED-POINT ARITHMETIC DATA WORD. The fixed-point arithmetic data word is used in performing all fixed-point arithmetic operations. The first bit of the 48-bit word is a sign bit and the remaining 47 bits contain a number in binary form. If the first bit is set to Ø, the number is positive. If the bit is set to 1, the number is negative.

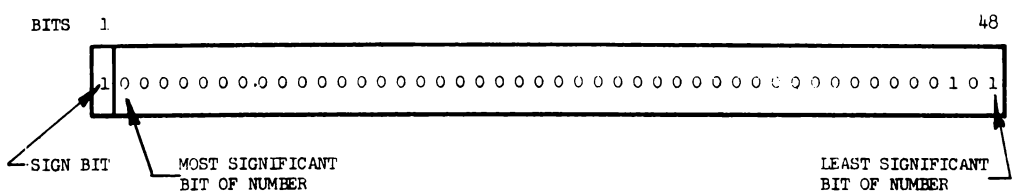


Figure 2-4. Fixed-Point Arithmetic Data Word

The sample fixed-point word contains a -5_8 (which is also a -5_{10}). In the AN/GSA-51A, the binary point is usually considered to be on the left, between bits 1 and 2. However, the actual location of the binary point is determined by the programmer because there is no physical binary point within the word. For example, the number in the diagram on the previous page may actually be -2.5_{10} to the programmer. In this case, the programmer would consider the binary point to be between bits 47 and 48.

FLOATING-POINT ARITHMETIC DATA WORD. The floating-point data word is used in performing all floating-point arithmetic operations. Floating-point words enable the AN/GSA-51A to work with much larger numbers than can be contained in the normal fixed-point data word and greatly reduce the amount of work the programmer must do for scaling.

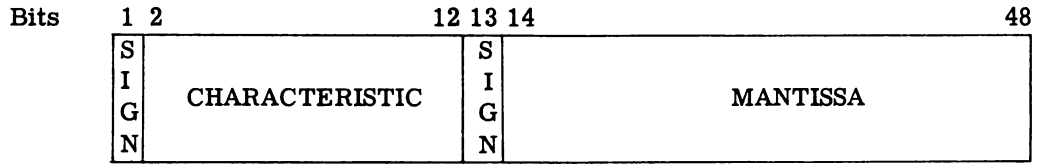


Figure 2-5

A floating-point word consists of two parts, a mantissa and a characteristic. The mantissa contains a binary quantity and the characteristic indicates the number of times the mantissa is raised by a power of two. In other words, the characteristic indicates the actual position of the binary point in the mantissa. The sign bit in bit 1 indicates whether the mantissa is to be raised by a positive or negative power of two. The sign bit in bit 13 indicates whether the mantissa is algebraically positive or negative.

FLOW OF INFORMATION WITHIN THE COMPUTER

The operation of the AN/GSA-51A is controlled by an internally stored program located in core memory. All data flow occurs between I/O modules and memory modules and between computer modules and memory modules. There is no direct data transfer between computer and I/O modules

Information flows directly between computer modules and memory modules over data transfer buses. Each computer has its own computer bus, Computer BUS 1 for computer module one and Computer BUS 2 for computer module two. The information being transferred will either be program instructions going to the computer to be decoded and operated, or data words being fetched from memory for operands, or the results of an operation going to memory to be stored.

Information flows directly between I/O modules and memory modules over data transfer buses called I/O BUS A and I/O BUS B. I/O BUS A is time-shared by I/O modules one and two. I/O BUS B is time-shared by I/O modules three and four. If both I/O modules on one I/O BUS simultaneously request access to core memory, the I/O module with the lower number will be granted access first, unless the program specifies otherwise.

All core memory modules are accessible by all computer and I/O buses. Therefore, the core memory is totally shared. Conflicts will arise if more than one computer or I/O BUS simultaneously request access to the same memory module. A SWITCHING INTERLOCK is provided to resolve these conflicts by handling the requests in order according to priority levels inherent in each request. The inherent priorities are as follows (from highest to lowest):

1. I/O BUS A
2. I/O BUS B
3. Computer BUS 2
4. Computer BUS 1

While a computer or I/O bus is accessing a particular memory module, all other computer and I/O buses are denied access to that memory module until the data transfer is complete. During this time, however, another memory module may be accessed by any other I/O or computer bus.

The switching interlock consists physically of a portion of the circuitry in each of the computer, memory, and I/O modules.

NORMAL AND CONTROL MODE

Operation of the BUIC program is frequently interrupted by messages from outside sources such as radar stations, by switch actions taken at the data display consoles, and by internal AN/GSA-51A equipment conditions. In order to handle these interruptions efficiently, two modes of AN/GSA-51A operation are provided. They are called normal mode and control mode.

NORMAL MODE

While operating in the normal mode, the AN/GSA-51A will recognize all interrupt conditions and when an interruption occurs, will in most cases automatically transfer operation to the control mode and to instructions which handle the particular interrupt condition. All necessary control data is stored so that the interrupted program can resume at the appropriate place after the interrupt condition is processed.

There are three AN/GSA-51A instructions which will NOT operate in the normal mode. These are the TIO, LSR, and IRR instructions. Loading the Interrupt Base Address Register (IAR) with a LTF instruction also will NOT operate in the normal mode.

CONTROL MODE

While operating in the control mode, the AN/GSA-51A will only recognize a select few interrupt conditions. Most of the interrupt conditions are ignored until the AN/GSA-51A operation returns to the normal mode. This allows the computer to process an interrupt condition with little likelihood of being interrupted further.

ALL AN/GSA-51A instructions can operate in the control mode. Also, the system will react differently to the HLT instruction depending whether the control or normal mode is functioning.

PROGRAM INSTRUCTIONS

The program instructions for the AN/GSA-51A have two distinctive characteristics. They can have up to three memory addresses and are variable length.

MULTI-ADDRESS FEATURE

Instructions for the AN/GSA-51A can specify from zero to three core memory addresses. The capability of having three addresses is convenient because many fundamental arithmetic operations involve three factors and therefore may need three storage locations; one each for the two operands and one for storage of the result. The number of addresses in an instruction depends on the particular operation to be performed.

VARIABLE LENGTH FEATURE

Instructions that have been assembled and stored in memory are comprised of strings of from one to seven 12-bit syllables which specify the individual parts of the total operation to be performed. There are five types of instruction syllables. These will be discussed briefly before launching into a discussion of the organization of syllables within instructions.

1. INSTRUCTION SYLLABLES

OPERATION SYLLABLE	The operator syllable is always the first syllable of an instruction. It specifies the fundamental operation to be performed by use of a 6-bit code and indicates the number of syllables to follow in the syllable string by use of three 2-bit codes.
MEMORY ADDRESS SYLLABLE	The memory address syllable is used to specify the relative address of data to be fetched from or stored in core memory.
BRANCH ADDRESS SYLLABLE	A branch address syllable is used to specify the relative address of the instruction word which should be operated next. Normally, the instruction words are operated in consecutive order, but a branch syllable may change the order of operation.
INDEX SYLLABLE	The index syllable, which is inserted in the syllable string immediately preceding the syllable to be indexed, contains the addresses of from one to three index registers whose contents are to be added to a memory address syllable, branch address syllable, or special syllable.
SPECIAL SYLLABLES	There are fourteen special syllables which are used to specify control data and instruction variations essential to the execution of certain instructions. The individual special syllables are discussed in Chapter 5.

2. ORGANIZATION OF SYLLABLES WITHIN INSTRUCTIONS

An instruction consists of an operator syllable followed by as many as six other syllables or by as few as none. The number of syllables present in a particular instruction depends on:

a. The type of instruction being used.
b. Use of indexes in the instruction.
c. Use of the operand stack. (Referencing the stack does not require a syllable.)

Since each instruction may vary in length from one to seven syllables, some instructions are longer or shorter than the core memory word length.

To avoid wasting core space, the assembler PACKS instructions into memory. Packing means that whenever possible, the assembler uses all four syllable locations in each memory instruction word of the program. An instruction may begin in any of the four syllables of an instruction word, can end with the same or any succeeding syllable of an instruction word, or can continue on into as many as two succeeding instruction words. The first instruction in a program always begins in the left-most syllable of the first program instruction word.

Assume that a program contains four instructions and the instructions have these types of syllables:

Instruction 1 OPERATOR, MEMORY, MEMORY
Instruction 2 OPERATOR, SPECIAL, SPECIAL, BRANCH
Instruction 3 OPERATOR, INDEX, MEMORY, MEMORY, BRANCH
Instruction 4 OPERATOR

This program would be packed into memory in the following manner:

WORD Ø	OPERATOR	MEMORY	MEMORY	OPERATOR
WORD 1	SPECIAL	SPECIAL	BRANCH	OPERATOR
WORD 2	INDEX	MEMORY	MEMORY	BRANCH
WORD 3	OPERATOR			

Figure 2-6

The unused portion of WORD 3 would contain zeros.

INSTRUCTION EXECUTION

The instructions of a program are stored in core memory in blocks of contiguous instruction words. The instruction words are automatically fetched by the computer module one at a time as needed during the execution of the program. When an instruction word is fetched from memory, the actual word remains in its memory location and an image of the word is placed in a 48-bit thin film register called the PROGRAM STORAGE REGISTER (PSR). While the duplicate instruction word is in the PSR, it is decoded by the computer module which then sends the necessary commands to the logic circuitry so that the operation can be accomplished.

During the execution of the program, another thin film register keeps track of the absolute address of the instruction word currently being executed (which means that the instruction word's duplicate is in the PSR). This 16-bit register is called the PROGRAM COUNT REGISTER (PCR). After an instruction word has been completely decoded and the computer module is ready for a new instruction word, a one is added to the contents of the PCR. This changes the contents of the PCR to the absolute address of the next instruction word in sequence. Using this new address, the computer module then fetches the next instruction word and places it in the PSR.

The action of fetching an instruction word and placing it in the PSR is called a NORMAL FILL. When execution of an instruction requires a (relatively) large amount of time, the computer module automatically fetches the next instruction word into a second PSR and begins to decode this even before it has finished using the instruction word in the first PSR. This is called an OVERLAP FILL.

Instruction words are not always fetched in consecutive order. There are many instructions in the AN/GSA-51A which request that a specific instruction word be fetched next. The instruction word which is specified to be fetched next may be the next word in sequence, but most often it is a word somewhere else in the group of instruction words. This method of changing the order in which the instruction words are to be fetched and operated is known as BRANCHING. Once a branch has been taken, subsequent instructions will be taken from consecutive instruction words following the instruction to which the branch was taken until another instruction which specifies branching is encountered.

THIN FILM REGISTERS

Each computer module contains a set of 128 thin film registers. Some of the thin film registers are used by the computer module logic circuitry during the execution of program instructions. Other registers are used to provide an indexing capability. Still others are used as a small data storage area called the operand stack.

Each thin film register contains 24 bits, but only 12 or 16 bits of each register are used. A map of the thin film registers is given on the following page to show which registers use 16 bits and which use 12.

The thin film registers are used individually as 12- or 16-bit registers or in multiple. When a programmer specifies that thin film registers are to be used in multiple, three adjacent 16-bit registers or four adjacent 12-bit registers will be used to create a 48-bit register. The computer module control circuitry uses adjacent 12-bit registers in multiple as 24-, 36-, and 48-bit registers. It also uses 16-bit registers in multiple as 32-, 48-, and 64-bit registers.

OPERAND STACK

The operand stack is a data storage area located in the thin film registers of each computer module. It contains the equivalent of four 48-bit words and is used for temporary storage of program data. The following paragraphs are devoted to the purpose, structure, and operation of the stack.

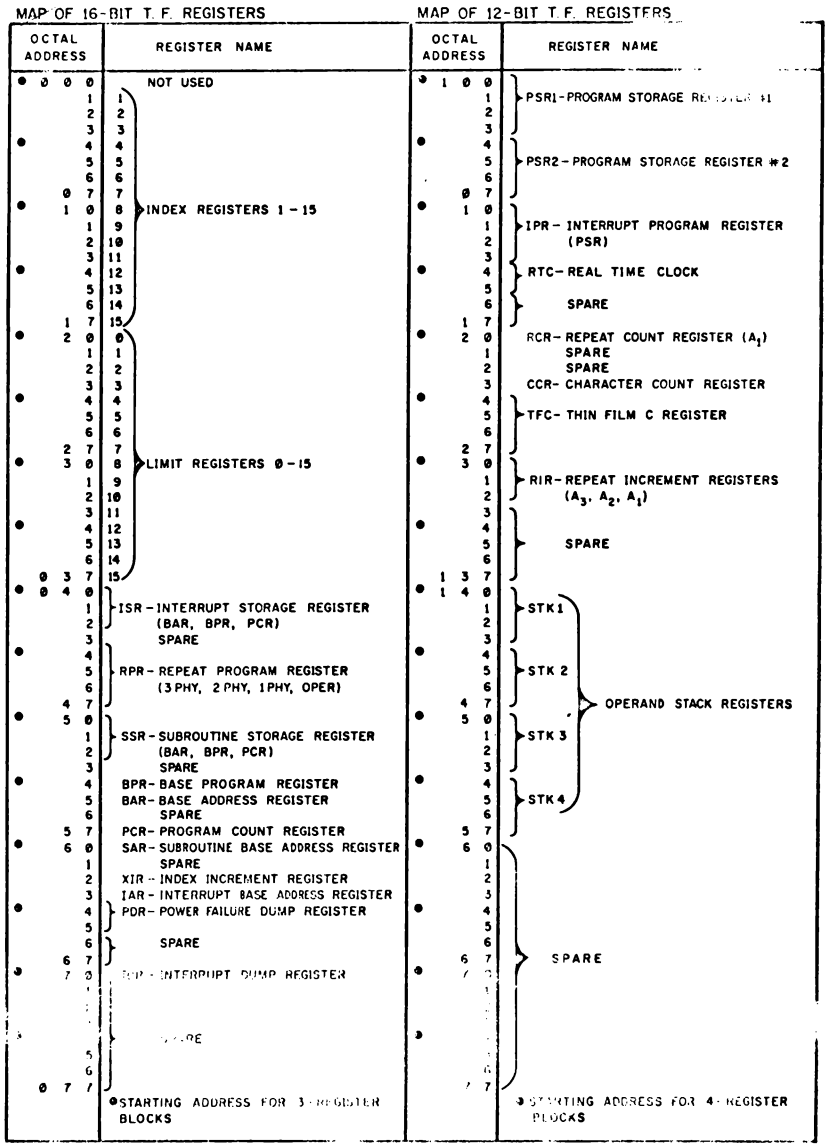


Figure 2-7

PURPOSE OF THE STACK

The stack provides faster instruction operation and memory economy.

FASTER INSTRUCTION OPERATION. Access time for the stack is more than twice as fast as access time for core memory. Therefore, use of the stack as a memory location in an instruction will increase the speed at which the instruction operates.

CORE MEMORY ECONOMY. When a program is assembled, all core memory references require a 12-bit memory syllable. References to the stack do not require a syllable because all stack references are completely specified within the tag bits of the operator syllable. Therefore, each time the stack is referenced in an instruction, the number of syllables required for that instruction is decreased by one, thereby conserving core memory space when the program is packed into memory at assembly time.

STRUCTURE OF THE STACK

The stack consists of sixteen contiguous 12-bit thin film registers which are organized into four 48-bit words as shown in the diagram below.

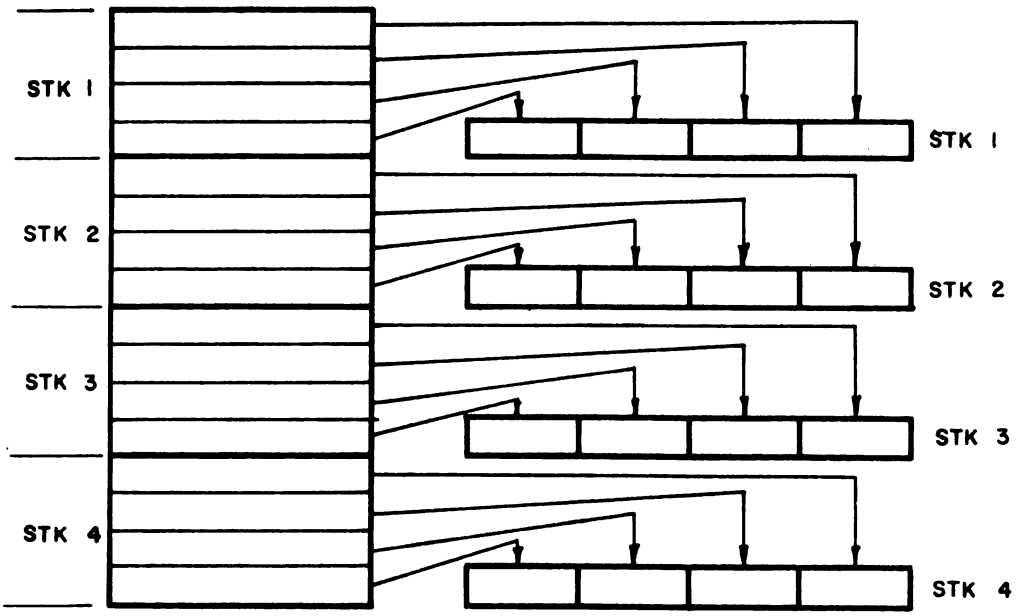


Figure 2-8

If, for example, STK 1 contained the values in Figure 2-9, the extended 48-bit STK 1 word would be 7777555533331111_8.

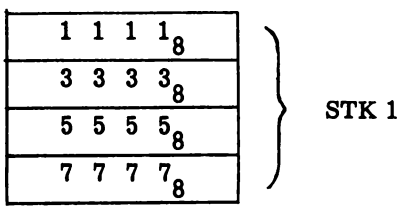


Figure 2-9

OPERATION OF THE STACK

Operation of the stack is most clearly understood if the stack is thought of as a four word "circular memory" like those pictured in Figures 2-10 and 2-11. One of the 48-bit words is always being "pointed at" by a "read-write head". This word is known as the "top of the stack". As the "read-write head" changes to an adjacent stack word, the "top of the stack" changes with it. In Figure 2-10 and 2-11, the "read-write head" moved from STK 2 to STK 1 thereby making STK 1 the new "top of the stack".

With this conceptual picture of the stack, the stack APPEARS to rotate and the "read-write head" APPEARS to remain stationary. Within the computer, no physical movement takes place but the "read-write head" electronically changes from one word to the next.

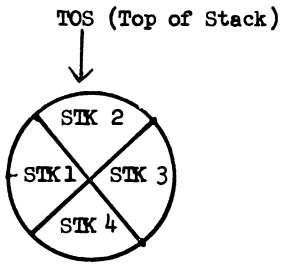


Figure 2-10

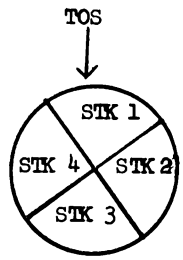


Figure 2-11

These two conceptual figures show the operand stack stepping in the clockwise direction. (Stepping is synonymous with rotating.) The stack can also be stepped in the counter-clockwise direction. In fact, there are two instructions which do nothing but step the stack in one direction or the other. The operand stack can also be stepped within program instructions which use the stack as a data memory reference. Each time data is to be fetched from or stored into the "top of the stack" for the operation of an instruction, the programmer has the choice of stepping the stack or not stepping the stack. NORMAL operation is defined as stepping the stack at the time of stack access. Not stepping the stack at the time of stack access is called HOLDING the stack.

In a normal stack reference, the direction that the stack will be stepped and the sequence of stepping are determined by whether the stack reference is one in which information is being fetched or stored. In fact, it is common terminology to call the two direction of movement the FETCH DIRECTION and the STORE DIRECTION. Usually, the fetch direction is considered to be counter-clockwise and the store direction is considered to be clockwise. However, there is an instruction (RVS) which will reverse the movement pattern of the "read-write head" (and consequently the conceptual picture of stack rotation). After this instruction is operated, the fetch direction is considered to be clockwise and the store direction is considered to be counter-clockwise. This condition will remain until the instruction RVS is executed again.

If the stack reference is a NORMAL FETCH reference, the information will be fetched from the "top of the stack", the stack will be stepped once in the fetch direction, and the rest of the instruction will continue. If the stack reference is a NORMAL STORE reference, the stack will be stepped once in the store direction and then the information will be stored into the "top of the stack". In a HOLD FETCH or HOLD STORE stack reference, the stack is not stepped.

The rules given in the paragraph above are summarized below:

HOLD FETCH	data fetched from top of stack; stack not stepped
HOLD STORE	data stored into top of stack; stack not stepped
NORMAL FETCH	data fetched from top of stack; stack stepped once in fetch direction
NORMAL STORE	stack stepped once in store direction; data stored into top of stack.

When information is fetched from a stack word, the information in the stack word is not destroyed. However, when information is stored into a stack word, the previous contents of that stack word are destroyed.

The programmer is not concerned with which physical stack word (1, 2, 3, or 4) is being accessed. All rules of operation of the stack apply no matter which word the "read-write head" is located at. Generally, the programmer does not know which physical stack words are being accessed throughout his program. As a program is written which uses the operand stack, the programmer need only keep track of the relative location of the information that is stored in the four sections of the "circular memory" and which section is currently the "top of the stack".

RELATIVE AND ABSOLUTE ADDRESSING

When a program is created for the AN/GSA-51A, the instructions are coded according to a standard format and are punched onto 80 column cards to be read into the computer. When the cards are read in, they are converted by the assembler to the binary machine language which the computer module decodes at the time of operation.

After the program is assembled, it may be stored anywhere in memory for operation. It makes no difference whether the program is stored toward the beginning, toward the end, or in the middle of core memory. Operation of the program will be performed in exactly the same manner. Therefore, programs for the AN/GSA-51A are floatable and may be moved from one area of core memory to another as the need arises. To understand why this is possible, it is necessary first to understand the concepts of relative and absolute addresses, the base address register-BAR, and the base program register-BPR.

ABSOLUTE ADDRESSES

Each of the $1ØØØØØ_8$ words in core memory is given an octal address which distinguishes its location from all other core memory locations. This address never changes and is known as an ABSOLUTE address.

RELATIVE ADDRESSES

Relative addresses are given to the instruction words and data words of a program. That is, each instruction word is given an address which is relative to an absolute address which is to be stored in the Base Program Register (BPR). Generally, the first instruction word has an address of $Ø$ and the instruction words following are given consecutive octal addresses beginning with 1_8. These addresses are assigned at assembly time. In the same manner, the data words are given addresses which are relative to an absolute address which is to be stored in the Base Address Register (BAR). See EXAMPLE A in Figure 2-12.

It is possible to combine the instruction words and data words into one continuous set of words in which case the entire set of words will be given addresses which are relative to the same absolute address.

In EXAMPLE B, the instruction words and data words are separated. They may be intermixed as long as the programmer makes certain that data words will not be accidentally operated as instruction words. This is accomplished by branching around the data words.

EXAMPLE A

Ø	First prog. inst. word
1	2nd " " "
2	3rd " " "
3	4th " " "
4	5th " " "
5	6th " " "
Ø	First prog. data word
1	2nd " " "
2	3rd " " "
3	4th " " "

EXAMPLE B

Ø	First prog. inst. word
1	2nd " " "
2	3rd " " "
3	4th " " "
4	5th " " "
5	6th " " "
6	First prog. data word
7	2nd " " "
1Ø	3rd " " "
11	4th " " "

Figure 2-12

BASE ADDRESS REGISTER

The Base Address Register (BAR) is a 16-bit register located in thin film which holds the ABSOLUTE core memory address that is added to relative DATA addresses located in memory syllables when the program is executed.

BASE PROGRAM REGISTER

The Base Program Register (BPR) is a 16-bit register located in thin film which holds the ABSOLUTE core memory address that is added to relative instruction word addresses located in BRANCH syllables when the program is executed.

If a program is organized so that the instruction words and data words are one continuous set of words, the BPR and BAR may contain the same absolute address. This can be true only if the relative data addresses are not greater than 3777_8 and the relative instruction word addresses are not greater than 3777_8.

CONVERSION OF RELATIVE ADDRESSES TO ABSOLUTE ADDRESSES

When a symbolic program is assembled into machine language, there are many references to instruction words and data words which require that memory addresses be specified. Relative addresses, not absolute addresses, are specified at this time. For example, an instruction which designates the contents of memory location "A" to be added to contents of memory location "B" and the answer to be stored in memory location "C" will require three memory address syllables as part of the instruction. The three addresses in the syllables will be relative to the BAR.

When the assembled program is stored into core memory for operation, the program still has the RELATIVE memory address references but the instruction words and data words are stored into core memory locations which have ABSOLUTE addresses. Therefore all relative memory address references must be converted to absolute addresses. This conversion

process is an inherent function of the AN/GSA-51A and is performed automatically just before each instruction is executed by adding the contents of the BAR to data word relative address references found in memory syllables and the contents of the BPR to instruction word relative address references found in branch syllables. The following formulas summarize this process.

Relative address of data word + BAR = absolute address of data word.

Relative address of instruction word + BPR = absolute address of instruction word.

This, then, is why programs are floatable in the AN/GSA-51A. An assembled program may be stored anywhere in core memory for operation simply by placing it in the desired locations and storing the appropriate addresses in the BAR and BPR.

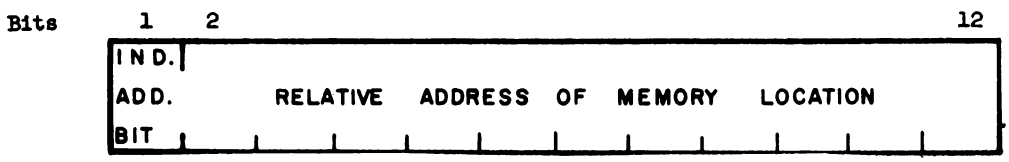
DIRECT AND INDIRECT ADDRESSING

Core memory locations for fetching and storing data are specified in program instructions through the use of memory syllables. Each memory syllable may contain either a direct address or an indirect address.

Direct address: a relative address to which the contents of the BAR are added to obtain the absolute core memory address for fetching or storing data.

Indirect address: an address of a core memory location, the contents of which contains the address of another core memory location. An indirect address is always a relative address if it is found in a memory syllable and the contents of the BAR must be added to obtain the absolute address. If the indirect address is found in a program data word, it is an absolute address.

The format of the 12-bit memory syllables is as follows:



Bits 2 through 12 always specify a relative data address. Bit one is used for indirect addressing. If there is a Ø in bit one, bits 2 through 12 of the memory syllable contain a DIRECT address. If there is a 1 in the first bit, bits 2 through 12 contain an INDIRECT address.

Direct addressing is used most frequently in AN/GSA-51A programs. Direct addressing limits the number of words which may be addressed to the first $2{,}048_{10}$ words immediately following the location specified by the BAR. (Eleven bits allows a maximum of $2{,}048_{10}$ words with addresses beginning at Ø and ending at 3777_8.) However, this limitation is not important because the programmer can use indexes to extend beyond this area.

Indirect addressing can also be used to specify addresses beyond the $2,048_{10}$ words following the BAR. If the first bit of a memory syllable is a one, the address is indirect and the least significant 16-bits of the address specified is then the absolute address of the instruction operand or another indirect (but absolute) address. If the seventeenth least significant bit of the location specified is the original indirect address is a one, it indicates that the indirect address chain is to be continued and that the absolute address specified by the sixteen least significant bits is another indirect address. Indirect addressing may be continued in this manner as many times as the programmer wishes.

If the seventeenth least significant bit of the location specified by an indirect address is a $\emptyset$, the address specified in the sixteen least significant bits of this location is the direct address of the operand to be used by the original instruction.

The following is an example of an indirect address chain. The BAR is set to 1000_8:

LOCATION	BIT SETTING	OCTAL EQUIVALENT OF ABSOLUTE ADDRESS
0100 - Third Syllable (Memory Syllable of Instruction)	1 1 0 0 0 0 0 0 1 1 1 0 0	0034 + 1000 (BAR) = 1034
1034 (Core Data Area)	00000000000000000000000000001000000010100000000	1200
1200 (Core Data Area)	00000000000000000000000000000000000001100101000	1450
1450 (Core Data Area)	Contains value of operand to be used in the instruction	

It is important to remember that indirect addressing is a function of the programmer and it is up to the programmer to see that the indirect address bit is set by the assembler and that the locations specified by indirect addresses contain the appropriate information.

TAGS

A tag is a combination of alphanumeric character which is used in place of an octal address to specify a memory location for data reference or branching. Tags are used by the programmer when he writes the symbolic language code for his program. The assembler assigns a relative location in memory for each tag and then uses the address of this location to convert the tag to a branch or memory syllable.

If the programmer knows the memory address for a branch location or a data reference, he can use the octal address in his symbolic language code. However, his task is much easier if he uses tags and lets the assembler supply the addresses.

A tag can be any combination of letters and numerals as long as it does not exceed five characters, contains no leading or embedded blanks, and has at least one letter.

Each tag is used to specify one memory location only. If it is used for more than one location, the program will not assemble correctly because the assembler is given no way of distinguishing the different locations.

USE OF TAGS IN BRANCHING

When one instruction is to branch to another, IDENTICAL tags are used in coding each of the instructions.

One of the tags is placed to the left of the operation code of the instruction to which the branch is to be taken. This tag will cause the assembler to store the relative address of the instruction word which contains the beginning syllables of the instruction, but will not generate a program syllable itself. The relative address is stored in a "dictionary" which contains all program tags and their addresses and other program information. This tag will also cause the assembler to left-justify the instruction in an instruction word.

The other tag is located within the instruction that will cause the branching. This tag will generate a branch syllable which will contain the relative address of the instruction to which the branch is to be taken. The assembler finds the address by looking in its "dictionary".

More than one instruction may branch to a particular instruction. However, no two instructions may be referenced by the same tag.

USE OF TAGS FOR DATA REFERENCES

Throughout a program, references may be made to data memory locations by using tags. Before the assembler can convert these tags to memory syllables, it computes the relative addresses of the locations specified by the tags and stores them in the same "dictionary" that is used for branch tags. As memory syllables are generated, the assembler finds the correct relative address for the tags by looking in this "dictionary".

The assembler cannot compute the relative address for a data reference tag unless the tag is defined by the programmer. Data reference tags are defined by using a declarative code to create a 48-bit data word with some value in it and by placing the tag to the left of the declarative code.

If the tag is used to specify a table, the first word in the table will be generated by a declarative code with the tag on its left. All succeeding words will be generated by declarative codes which do not have tags on their left. The assembler will store the first address of the table in its "dictionary" and this address will be placed in the memory syllable. All other words in the table may be accessed through the use of index registers.

There are a few special cases in the use of tags for data references and these will be discussed in the paragraphs that follow:

INDIRECT ADDRESSING. Each data tag within an instruction generates one memory syllable. The tag may be used to generate a direct address or an indirect address. If indirect addressing is to be used, the tag must be preceded by a prime mark which will cause the assembler to place a one in the first bit of the memory syllable.

EXAMPLE:

```
ABC2 will generate a direct address
'ABC2 will generate an indirect address
```

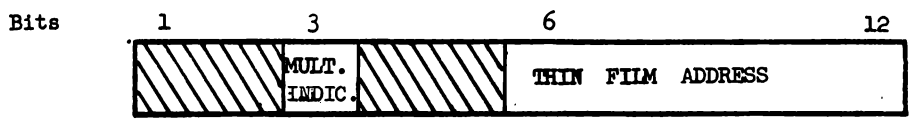
OPERAND STACK. The operand stack may be used for data reference in an instruction. NORMAL stack usage is access of the stack with rotation (stepping) and is designated by the tag N. HOLD stack usage is access without rotation and is designated by the tag H. An N or an H in an instruction is all that is needed to reference the operand stack.

THIN FILM IDENTIFIERS. Many thin film registers have tags which are known as thin film identifiers. A chart of the thin film identifiers and the octal addresses associated with them is given below.

OCTAL ADDRESSES OF THIN FILM IDENTIFIERS

X1	001	L3	023	PCR	057
X2	002	L4	024	SAR	060
X3	003	L5	025	XIR	062
X4	004	L6	026	IAR	063
X5	005	L7	027	PDR	064
X6	006	L8	030	IDR	070
X7	007	L9	031	PSR1	100
X8	010	L10	032	PSR2	104
X9	011	L11	033	IPR	110
X10	012	L12	034	RTC	114
X11	013	L13	035	RCR	120
X12	014	L14	036	CCR	123
X13	015	L15	037	TFC	125
X14	016	ISR	040	RIR	130
X15	017	RPR	044	S1	140
L0	020	SSR	050	S2	144
L1	021	BPR	054	S3	150
L2	022	BAR	055	S4	154

A thin film identifier can be used in an instruction which accesses thin film and the assembler will automatically supply the thin film address. To access registers which do not have identifiers, the octal thin film address must be used in the symbolic coded program because the programmer cannot create new tags for thin film registers. The address is placed in one of the special syllables, the T syllable, which has the following format:



If bit 3 is set to Ø, only the thin film register specified by the address in bits 7 through 12 is to be used. If bit 3 is a 1, adjacent registers are to be used in multiple to create a 48-bit word.

To specify the use of multiple thin film registers, an M and space must precede the thin film identifier in the sumbolic coding. (An example is M X4.) The M will generate a one in the third bit of the T syllable. If an octal address is used rather than a thin film identifier, a one precedes the three digit address and it generates the one in the third bit.

When the third bit is set to one, the computer module ignores the last two bits of the T syllable, causing them to be effective zeros. This gives the film address an effective Ø or 4 for the last octal digit.

This modified address is used to specify the register which contains the least significant 12 or 16 bits of the total 48-bit word, and the rest of the 48-bit word consists of three 12-bit registers or the two 16-bit registers which immediately follow this register.

INDEXING

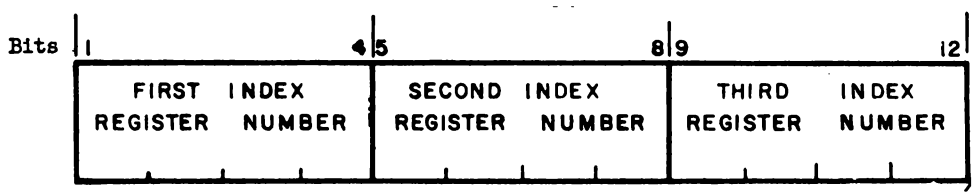
Indexing provides for the automatic modification of an instruction syllable by the addition of the contents of one, two, or three index registers. Most instruction syllables can be indexed. Those which cannot be modified by indexes are operator syllables, index syllables, and all syllables of an instruction to be repeated through the use of another instruction which specifies that repeating should be done.

When a syllable is to be modified by indexing, the contents of each index to be used are added to the contents of that syllable. If the syllable that is to be modified contains an indirect address, the contents of the index register or registers will be applied to the last level address only. The addition takes place in the computer module in special registers used for instruction execution at the time the instruction is operated. The actual core memory contents of the syllable being indexed remains unchanged.

Index registers are always added to instruction syllables. If subtraction is desired, the index must contain the two's complement of the amount to be subtracted. In this way, the addition will be an effective subtraction.

There are fifteen index registers. They are located in thin film and are numbered one through fifteen. Each index register contains sixteen unsigned bits. If a memory syllable is indexed, any core location may be addressed because only fifteen bits are required to specify the maximum core memory address (77777_8).

When symbolic language coding is converted to machine language code, all of the three or fewer index registers whose contents are to be added to a program syllable are specified in one index syllable. The index syllable always precedes the syllable it modifies. The format of the index syllable is as follows:



The twelve bits are divided into three 4-bit portions. Each portion specifies the number of an index register. If fewer than three indexes are used, the unused 4-bit portions will contain zeros. If an instruction syllable is not indexed, an index syllable will not be generated for it.

In addition to the fifteen index registers, sixteen limit registers are provided to implement the indexing capability. The limit registers contain sixteen unsigned bits and are numbered Ø through 15.

The limit registers are only used in conjunction with index registers. As a program progresses, the value of an index register may be increased or decreased by program instruction. If the programmer wishes to continue the modification of the index registers until a certain value has been reached, he may place that value in one of the limit registers. The XLC instruction is provided so that the index register can be compared with the limit register to see if the limit has been reached. If it has been reached, the program will continue at a specified instruction. If it has not been reached, the program will continue at another specified instruction. Any index register can be compared with any limit register.

Limit register zero is somewhat special because it may be loaded with any value but will always be an effective zero when used in this comparison-type instruction. Therefore, it is normally used as the limit register when the lower limit is zero and the index register is being decremented.

CHAPTER 3

BUIC III ASSEMBLER

ASSEMBLY LANGUAGE PROGRAMMING

An assembler is a programming tool designed to alleviate part of the effort required in coding programs. Assemblers accept symbolic languages[1] as input and convert this input automatically into the internal machine language[2]. The term "symbolic" refers to the use of mnemonic[3] codes in place of machine-language codes and the use of symbolic tags in place of absolute addresses. For example, the augend for a binary add might be located in memory at the symbolic address VALUE, which has the actual machine address of 3000_8. By using an assembler language for coding, the programmer is not burdened with keeping track of the memory location used, their addresses, or the actual machine language for the instruction he is using. Since an assembler will do this, and more, the programmer is free to code programs at a greater speed. In most instances, the speed of program-coding will be limited only by the programmer's ability to solve the problem at hand.

The assembler reads the symbolic code[4] from punched cards or magnetic tape and converts it to machine instruction codes in a one to one ratio with the symbolic instructions. That is, one machine instruction code is generated for every symbolic instruction. As an example, for every BAD symbolic input, a 65_8 machine instruction is generated. In addition, the assembler provides hard-copy documentation (program listings) of the program for the programmer's use in debugging his program. This documentation consists of a symbolic listing and machine-code listing of the instructions and constants in the program along with comments inserted by the programmer. On the following page is an example of a program listing.

The assembler also interprets special control and declarative codes which assist the programmer in preparing his program. Control codes are used to inform the assembler of the assembly origin (the address of the first instruction of his program), base register settings (BPR, BAR) and so forth. Other control codes reserve blocks of memory locations for data storage. No machine-code instructions are produced for these operations.

Declarative codes do not generate machine instructions either. However, they do generate data. Two examples of generated data are - absolute addresses and data constants in octal, hollerith, or decimal forms.

---

[1] An example of a SYMBOLIC language or instruction is any of the 62 instructions listed on your programmer's card-BAD, BSU, TRS, etc.

[2] MACHINE language is the numerical (binary or octal) format of the symbolic language. The binary numbers 110101_2 (binary add), 110100_2 (binary subtract), 011101_2 (transfer) are examples of MACHINE language.

[3] The symbols - BAD, BSU, etc., in a symbolic language are called MNEMONIC codes.

[4] SYMBOLIC CODE is symbolic language.

PROGRAM LISTING

	SYMBOLIC LISTING			MACHINE-CODE LISTING				
	IDT	PROG						
			THE NAME OF THIS PROGRAM IS PROG					
	BSU	AA,BB,CC	CC = AA - BB = -9	00000	6452	0004	0005	0006
	BAD	CC,D(+10),BB	BB = CC + 1Ø = +1 NOTICE THAT THE ASSEMBLER HAS CREATED A DATA WORD AT LOCATION 7 FOR THE RC WORD	00001	6552	0006	0007	0005
	CLA	AA	THIS INSTRUCTION CONSUMES ONLY TWO SYLLABLES IN CORE	00002	2040	0004		
*	CLA	CC	NOTICE THAT THE ASTERISK CAUSES THIS INSTRUCTION TO BE LEFT JUSTIFIED	00003	2040	0006		
	HLT		NOTICE THAT THE HLT INSTRUCTION IS PACKED RIGHT AFTER THE PREVIOUS INSTRUCTION IN CORE	00003			0100	
AA	DEC	-8	NOTICE THE SIGN BIT IN THE DATA WORD	00004	4000	0000	0000	0010
BB	OCT	1		00005	0000	0000	0000	0001
CC	OCT	Ø		00006	0000	0000	0000	0000
	END							

Figure 3-1

There are two terms often used when speaking of an assembler. They are SOURCE programs and OBJECT programs. A source program is the symbolic code used as the INPUT to the assembler. An object program is the binary machine language program that is the final OUTPUT.

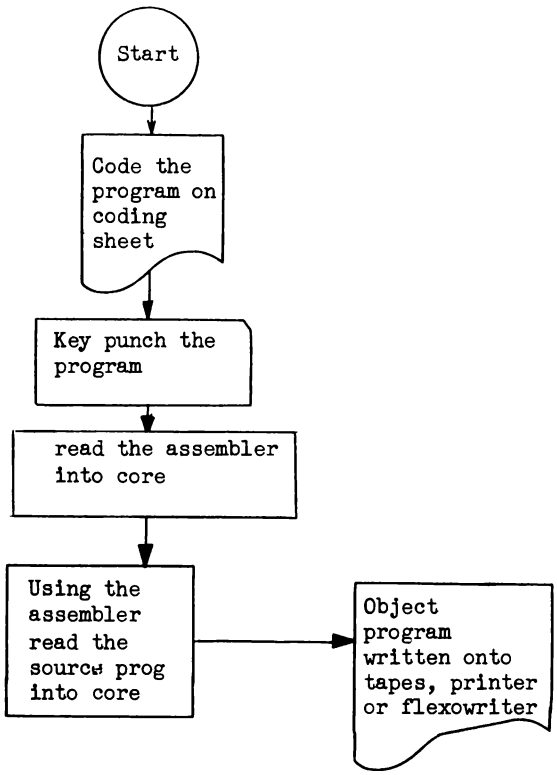


Figure 3-2. A macro flow showing the sequence of events from hand-written program to computer output.

BUIC III ASSEMBLER - PSA, PSB

The BUIC III language assembler is a two-pass assembler. Its official name is PSA, PSB (Pass A and Pass B). The first pass reads symbolic inputs from the card reader or tape, converts the symbolic codes to binary, sets up relative program and data references, generates the data region, creates the dictionary[5], assigns system index registers, and extracts the required compool[6] information. The partially processed data and other required information is made available to the second pass for final processing.

The second pass of the assembler generates the binary words, converts the relative program and data references to binary addresses, and adds required data from the compool. The side by side listing, Figure 3-1, is generated from the binary and the partially processed symbolic codes.

---

[5]A DICTIONARY is a listing and description of all items used by the program. It is generated as part of the symbolic output.

[6]COMPOOL - (COMMUNICATION POOL) is an area of core containing items and tables that can be referenced by more than one program.

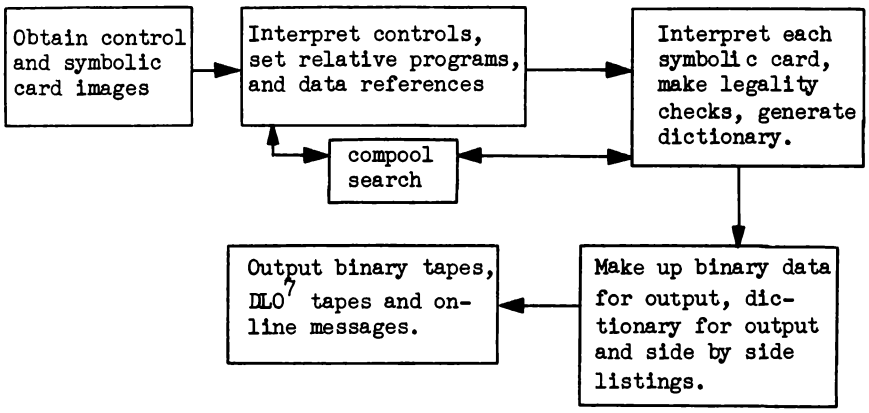


Figure 3-3. A Macro Flow of PSA, PSB Operations

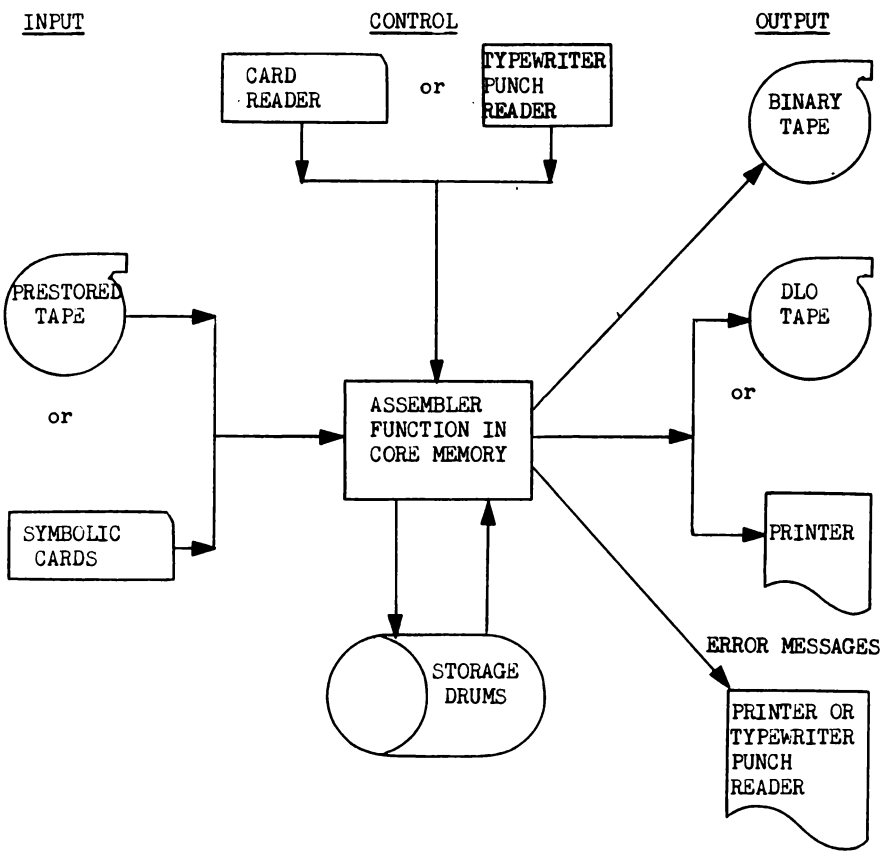


Figure 3-4. A System Flow of PSA, PSB

[7]DLO - (DELAYED OUTPUT) When an immediate printout of the object program is not desired the information is written onto a tape. The tape is later used to dump the information on the line printer.

The assembler function requires the following equipment of the AN/GSA-51A data processing set:

1 controller comparator (IOCU)

1 computer module

5 memory modules for operation of this function

1 magnetic tape drive or 1 card reader for symbolic input

1 magnetic tape drive for binary output

1 drum for storage

1 additional drum for storage (for programs in excess of 5ØØØ symbolic cards)

1 card reader or 1 typewriter-punch-reader for input control

1 magnetic tape drive for storage (for programs in excess of 1Ø,ØØØ symbolic cards)

BUIC III UTILITY SYSTEM

PSA, PSB is one program from the BUIC III Utility System. A utility system is used to perform miscellaneous or utility functions such as - tape searching, memory dumps, tape dumps, tape maintenance-making new tapes, and duplicating tapes, clearing drums, clearing core memory, data conversions, etc. The official name for the BUIC III Utility System is UCP (Utility Computer Program). Don't let the word program throw you. It really is a SYSTEM. UCP contains more than 30 separate programs and is so large that it can't all be contained in core memory at one time. So the system is read from the UCP Master tape and placed on drums. Such a huge system needs an executive control program that knows which of the 30 programs the programmer needs and where it is located on the drums. UCP's master executive program is CUE (Control Utility Executive). When a program is desired, CUE reads it in from the drums. Figure 3-5 shows one possible UCP system configuration.

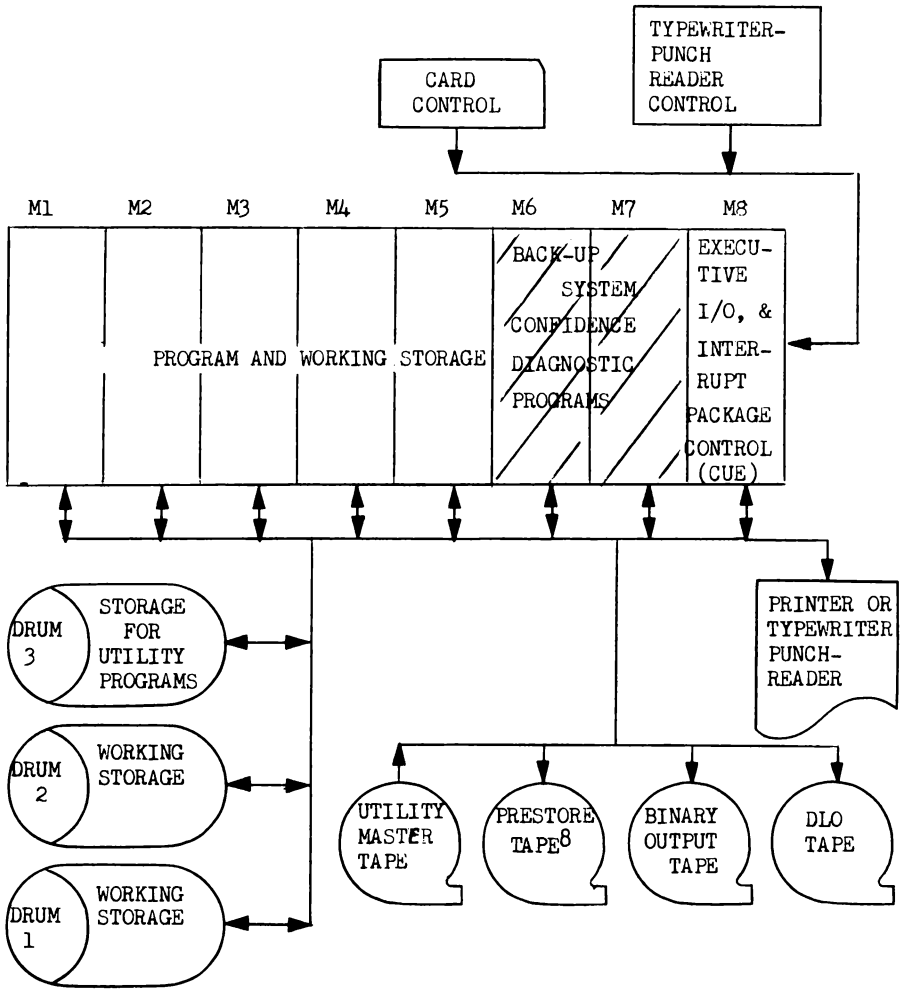


Figure 3-5. A UCP Configuration

[8] PRESTORE - is a term used to describe a tape that has information written on it in 6-bit Hollerith. It is created by reading a symbolic program (12-bit Hollerith) into core through card reader. The 12-bit Hollerith is converted to 6-bit Hollerith and the program is written directly out onto a tape. NO assembly process has occurred. This tape is then used at a later date as the input to the assembler.

CHAPTER 4

BUIC III ASSEMBLER INPUTS

THE AN/GSA-51A CODING SHEET

FORMAT OF CODING SHEET

The AN/GSA-51A coding sheet is used to give the programmer's symbolic statements the format necessary for use by the assembler. The format of the coding sheet is basically the same as the format of the 80 column punch cards on which symbolic statements are punched for input into the computer.

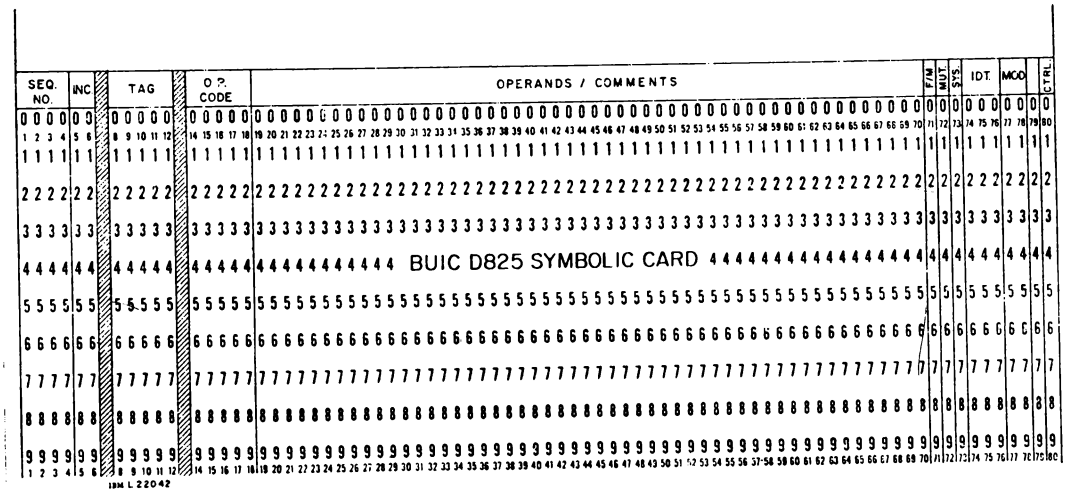


Figure 4-1. BUIC D825 Symbolic Card

COLS 1-4 are for reference only and may contain any notation the programmer wishes to use. The columns are generally used to sequence the program deck with octal or decimal numbers. Although the assembler does not use columns 1 through 4, any information in them will be printed out on the assembly listing.

COLS 5-6 are used for incrementing the reference numbers given in columns 1 through 4. This is especially helpful when inserting a card into the middle of the deck after the deck has been given sequence numbers. These columns are not used by the assembler, but they are printed on the assembly listing.

COL 7 is not used.

COLS 8-12 contain symbolic tags (addresses) assigned to the instructions to which branching is performed in the program. These identifying tags are also used for defining data within the program.

COL 13 is not used.

COLS 14-18 may contain:

1. AN/GSA-51A instruction operation codes which specify the type of instruction to be performed. This type of code will generate an operator syllable in the assembled program.

2. Declarative codes which generate 48-bit data words.

3. Control codes which provide program control information to the assembler.

4. Pseudo Codes which generate AN/GSA-51A instruction operation codes.

COLS 19-64 are used to specify:

memory reference for operands
indexes
branching locations
codes which generate special syllables
comments

All memory syllables, branch syllables, index syllables, and special syllables are generated from information given in these columns. Comments are used by the programmer to help keep track of what the program is doing. They do not generate machine language code, but are printed out on the assembly listing exactly as written. COMMENTS MUST FOLLOW THE SYMBOLIC INSTRUCTION CODES AND MUST BE SEPARATED FROM THEM BY AT LEAST TWO BLANK COLUMNS. Any card that does not have codes in columns 14 through 18 may be used entirely for comments in columns 19 through 64.

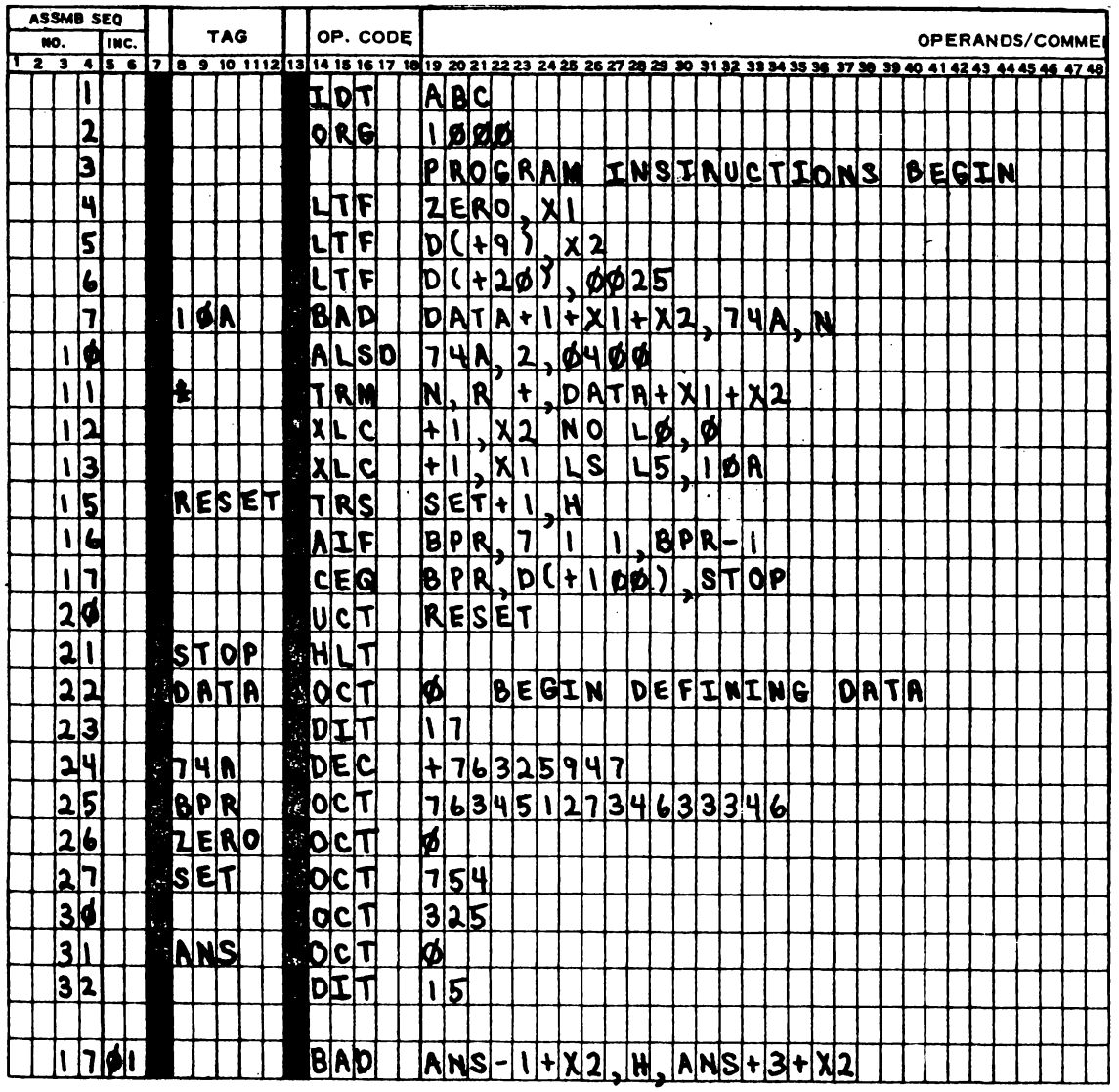
COLS 65-70 are used for sequencing cards when program coding is done is JOVIAL, a language not covered by this manual. These columns are ignored by the BUIC assembler.

COLS 71-80 are used for reference information for filing a deck of cards that is to become part of a larger group of cards which make up a system. These columns are ignored by the BUIC assembler.

USE OF CODING SHEET

The coding sheet, shown on the following page, gives examples of many different types of instructions to show the formats acceptable by the assembler. In preparing this coding sheet, no attempt was made to give logic to the program as a whole. Emphasis is on instruction format only. Each line of code would be punched onto one 80 column card to be read into the AN/GSA-51A and processed by the assembler. Details of this coding sheet are discussed in the pages that follow.

CODING SHEET



ASSMB SEQ NO.	INC.	TAG	OP. CODE	OPERANDS/COMME
1 2 3 4	5 6	8 9 10 11 12	14 15 16 17	19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
1			IDT	ABC
2			ORG	1ØØØ
3				PROGRAM INSTRUCTIONS BEGIN
4			LTF	ZERO, X1
5			LTF	D(+9), X2
6			LTF	D(+2Ø), ØØ25
7		1ØA	BAD	DATA+1+X1+X2, 74A, N
1Ø			ALSD	74A, 2, Ø4ØØ
11		*	TRM	N, R +, DATA+X1+X2
12			XLC	+1, X2 NO LØ, Ø
13			XLC	+1, X1 LS L5, 1ØA
15		RESET	TRS	SET+1, H
16			AIF	BPR, 7 1 1, BPR-1
17			CEQ	BPR, D(+1ØØ), STOP
2Ø			UCT	RESET
21		STOP	HLT	
22		DATA	OCT	Ø BEGIN DEFINING DATA
23			DIT	17
24		74A	DEC	+76325947
25		BPR	OCT	7634512734633346
26		ZERO	OCT	Ø
27		SET	OCT	754
3Ø			OCT	325
31		ANS	OCT	Ø
32			DIT	15
17Ø1			BAD	ANS-1+X2, H, ANS+3+X2

RULES FOR USING CODING SHEETS

The following paragraphs are a discussion of the AN/GSA-51A coding sheet with specific references made to the coding sheet shown above.

1. SEQUENCING - Col 1-6

a. Although octal numbers are used on this coding sheet, sequencing may be done in any number system adaptable to the numerals Ø through 9.

b. Errors in sequencing will not be reflected in the program, but MAY cause problems if the program deck is sorted on a card sorter. Although the number 14 was skipped in this deck, the deck does not need to be renumbered because the numbers are in order and the cards can be sorted correctly.

c. Leading zeros in the sequence numbers need not be written on the coding sheet nor punched into the cards.

d. Card $17\emptyset1_8$ is to be placed between card 17_8 and $2\emptyset_8$ after the program has been punched onto cards.

2. SYMBOLIC TAGS

a. All tags can be from one to five characters in length and must contain no leading or embedded blanks.

b. Each tag can have any combination of letters and numerals as long as there is AT LEAST ONE LETTER somewhere in the tag.

c. N and H by themselves should NOT be used as symbolic tags because they are used to reference the operand stack. See lines 7, 11, 15, and 17Ø1 of the sample coding sheet.

d. Tags in cols 8-12 are always left-justified. See lines 7, 15, 21, 22, 24, 25, 26, 27 and 31 of the sample coding sheet.

e. All data area tags must be referenced somewhere in the program area. The programmer must make certain that the data tags are located at a place in the program listing where they won't be operated as instruction by mistake. In this sample program, the data tags were located after the halt (HLT) instruction. Therefore, the computer will have stopped operation just before it reaches the data area.

f. An asterisk can be placed in column 8. The asterisk is used to left justify an instruction in core. See line 11.

3. OPERATOR CODES - Cols 14-18

a. ALL operator codes are LEFT-JUSTIFIED in the op code field.

b. The op code field may contain instruction operator codes, declarative codes, and control codes. IDT, ORG, and DIT are sample control codes. OCT and DEC are sample declarative codes.

4. OPERANDS AND COMMENTS - Cols 19 and following

a. The operands always begin in column 19.

b. ALL operands in an instruction are separated from one another by commas.

c. Index register codes follow the operand they are modifying and are linked to that operand with PLUS signs. See line 11.

d. Memory references may be incremented or decremented by a DECIMAL number. See lines 15 and 16.

e. If a memory reference is incremented or decremented AND indexed, the index register codes follow the decimal number that specifies the increment or decrement amount. See lines 7 and 17Ø1.

f. A blank is not allowed in the operand syllable unless it is a "legal blank" peculiar to a particular instruction. Lines 11, 12, 13, and 16 are examples of instructions with "legal blanks' in the operand syllable.

g. Memory syllables may be symbolic tags, octal addresses, or RC words. Line 1Ø contains an octal address (Ø4ØØ) for a memory syllable. Lines 5 and 6 contain RC words.

h. When a thin film register is being referenced, the references may be the thin film symbolic tag or an octal thin film address. In lines 4 and 5, index registers one and two are referenced by their thin film symbolic tags (X1 and X2). In line 6, limit register five is specified by its octal address (ØØ25).

i. Thin film symbolic tags may be used as tags in the program or data areas. Tags are recognized as thin film mnemonics ONLY when they are in certain syllables of instructions, which are specifically used for accessing thin film. Although BPR is used throughout the coding sheet as a core memory tag, the assembler will not confuse it with the thin film identifier BPR.

j. Some instructions do not have operands. See line 21.

k. All comments follow the operands by at least two spaces. See line 22.

l. Any card that does not have codes in columns 14 through 18 may be used entirely for comments in columns 19 through 64. See line 3.

SYLLABLE STRUCTURES

There are 18 syllables associated with the BUIC III instructions and each syllable has a unique format. An alphabetical listing of all syllables is found on page 36 . Following this listing, an example is given for each syllable. Included in each example is: (1) a short discussion and diagram of the syllable, (2) the syllable's octal configuration, and (3) representative coding formats. The BUIC III programmer's card should be a handy reference for the syllable diagrams. The User's Manual, TM 2780/004/00, Chapter 2 will be an additional reference for assembler coding formats.

SYLLABLE PACKING

The machine language format of the AN/GSA-51A instructions can vary in length from ONE to SEVEN 12-bit syllables. Therefore, many instructions are shorter or longer than the core memory word length of four syllables. To CONSERVE core memory space, the assembler PACKS the syllables into core memory using the following rules:

1. The first syllable of the first instruction in a program is always placed in the leftmost syllable of the first register.

2. When the assembler starts packing a new instruction, it usually continues with the next available syllable without necessarily changing to a new register.

3. Packing begins in the left-most syllable for each register and moves toward the right. After all four syllables of a register have been used, the assembler continues with the next instruction word.

4. Any instruction that has a tag or an asterisk in the tag field will be packed into memory beginning with the left-most syllable of the next available register. Any unused syllables in the previous register will contain zeros.

5. An index syllable always precedes the syllable it modifies.

The instructions below have been placed in instruction words using the rules given on this page. Actually, the symbolic coding is changed to its BINARY MACHINE LANGUAGE equivalent before being placed in memory, but it has NOT been changed in this example in order to facilitate an understanding of instruction PACKING. NOTE THAT STACK REFERENCES DO NOT REQUIRE A SYLLABLE.

	TAG	OP. CODE	
		BSU	DATA1,DATA2+X5,ANS
		TRS	ANS,H
		CEQ	COUNT,TEN,STOP
	1ØA	BAD	COUNT,ONE,COUNT
		BAD	N,DATA3,DATA1
		CEQ	DATA1+X3,DATA2+X4,1ØA
	STOP	HLT	

REGISTER	SYLLABLE 1	SYLLABLE 2	SYLLABLE 3	SYLLABLE 4
Ø	BSU	DATA1	X5	DATA2
1	ANS	TRS	ANS	CEQ
2	COUNT	TEN	STOP	(zeros)
3	BAD	COUNT	ONE	COUNT
4	BAD	DATA3	DATA1	CEQ
5	X3	DATA1	X4	DATA2
6	1ØA	(zeros)	(zeros)	(zeros)
7	HLT	(zeros)	(zeros)	(zeros)

SYLLABLE ABBREVIATIONS

In defining the syllable layout the following syllable abbreviations are used:

B	Branch address syllable
C	Character syllable
F	Field syllable
Ia	Index increment amount syllable
IO	Input/Output syllable
Iv	Index increment variant syllable
Ja	Subroutine jump address syllable
Ji	Subroutine jump increment syllable
L	Logical syllable
M	Memory address syllable or stack reference
O	Operator syllable
Rc	Repeat count syllable
Ri	Repeat increment syllable
S	Shift syllable
T	Thin film address syllable
Vs	Special register and computer interrupt variant syllable
Vt	Transmit variant syllable
X	Index syllable

B BRANCH SYLLABLE

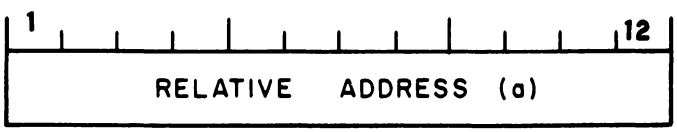


Figure 4-2

All twelve bits of the branch syllable are used to specify the relative address of the location to which the branch is taken. The contents of the BPR are added to the branch syllable to obtain the absolute branch address. Indirect addressing CANNOT be used with a branch syllable.

An octal Integer or an internal tag may be used as a branch syllable. The TAG may be incremented or decremented by a decimal integer. Either the tag or the octal integer can be indexed.

ASSEMBLER CODING FORMATS:

1. UCT 45
   2240 0045 0000 0000

2. UCT Ø1A
   2240 0367 0000 0000

3. UCT Ø1A+X2
   2240 1000 0367 0000

C CHARACTER SYLLABLE

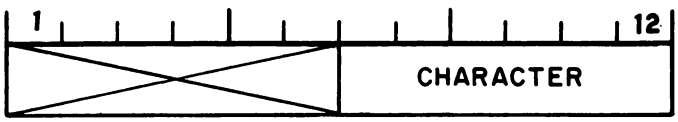


Figure 4-3

The character syllable is defined by a character or by two octal digits enclosed by brackets. The 6-bit Hollerith code or the two octal digits are inserted in the syllable. The , (comma) cannot be used as a character.

ASSEMBLER CODING FORMATS: CSE M, C, B

1. CSE KTSAA,D,NS77
   3526 1217 0030 1030

2. CSE KTSA,(24),NS77
   3526 1217 0030 1030

F FIELD SYLLABLE

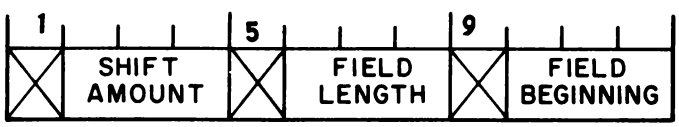


Figure 4-4

SHIFT AMOUNT = number of character shifts

FIELD LENGTH = 001 thru 111 = 7 characters; 000 = 8 characters

FIELD BEGINNING = 000 thru 111 starting character 0 - 7 respectively.

The Field syllable is coded using three decimal integers - amount of shift (0 - 7), followed by field length (0 - 8 with 0 or 8 indicating a full word) and the beginning field (0 - 7).

The field syllable may also be defined by using compool items which exactly fit into bytes (fields). To use a compool defined item in a field syllable, use two item names or a decimal integer (0-7). The number indicates the least significant field position. If two item names are given, and the item are of unequal length, the least significant fields are aligned.

ASSEMBLER CODING FORMATS: SAF M,F,M

1. AIF CMAND+15,6 3 3,CMAND+15
   4052 0122 3063 0122

2. SAF CMESG,CMESG 7,H
   4151 1642 5025

3. CEF EEPN+X9,EEPN 7,BA13
   5272 0011 4323 7420
   0172 0000 0000 0000

4. SAF EDCY,EDCY EDCY,EDCY
   4073 6400 1504 4062
   6400 1504

I_a INDEX INCREMENT SYLLABLE

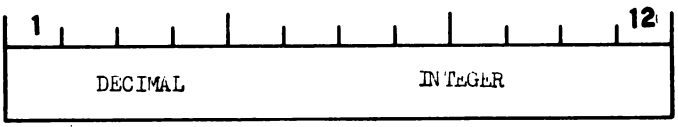


Figure 4-5

The index increment syllable is a signed decimal integer. The sign is coded as the first bit in the I_v SYLLABLE.

I_v INDEX INCREMENT VARIANT SYLLABLE

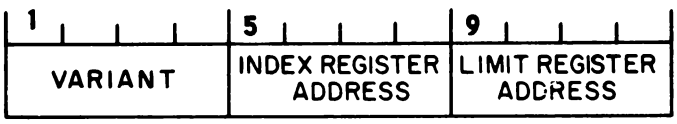


Figure 4-6

Bit 1 0 = increase; 1 = decrease
Bit 2 = 1 X > L
Bit 3 = 1 X < L
Bit 4 = 1 X = L

Index increment variant syllable contains three sets of characters which must always be present. FIRST, the index (bit 5-8) is indicated as Xn where n is a decimal integer from 0-15. SECOND, the branch condition (bits 1-4) is coded as:

BR - Branch unconditionally

EQ - Index = Limit
GR - Index > Limit
GQ - Index ≥ Limit
LQ - Index ≤ Limit
LS - Index < Limit
NQ - Index ≠ Limit
NO - No branch

THIRD, the limit (bits 9-12) is indicated as Ln where n is a decimal integer from 0-15.

ASSEMBLER CODING FORMAT: XLC I_a,I_v,B

1. XLC +3,X7 LS L1,ADØ36
   1252 0003 2161 0062

2. XLC -Ø+X3,X11 NO LØ,Ø
   1272 0003 0000 4260
   0000

3. XLC -1,X3 NQ LØ,BA3
   1252 0001 7060 0420

IO I/O SYLLABLE

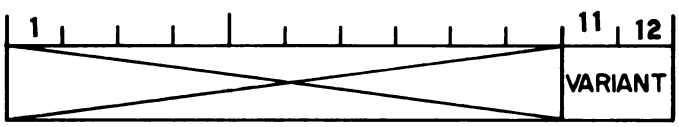


Figure 4-7

Bit 11 12

0	0	conditional descriptor bus A
0	1	unconditional descriptor bus A
1	0	conditional descriptor bus B
1	1	unconditional descriptor bus B

The input/output syllable is defined by indicating the bus desired by an A or a B and the type of descriptor by a U for unconditional and a C for conditional.

ASSEMBLER CODING FORMAT: TIO IO,M,B

1. TIO A U,SETUP,Ø
   1672 0001 0461 0000

2. TIO A C,COMD,BAT
   1672 0000 0462 0615

3. TIO B U,RELSE,Ø
   1672 0003 0463 0000

4. TIO B C,COMD,Ø2D
   1672 0002 0462 0023

J_a SUBROUTINE JUMP ADDRESS SYLLABLE

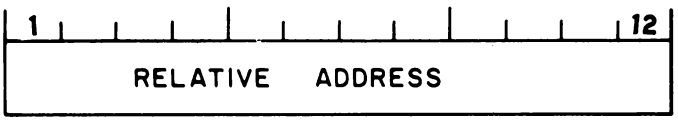


Figure 4-8

The subroutine jump address may be defined either as a decimal integer, a compool program tag which designates a subroutine, or an internal tag. When a compool tag for a subroutine is coded in the Ja syllable the index value under the SAR will be set in the syllable at assembly time. The value C(SAR+Ja) is loaded into the BPR and PCR.

J_i SUBROUTINE JUMP INCREMENT SYLLABLE

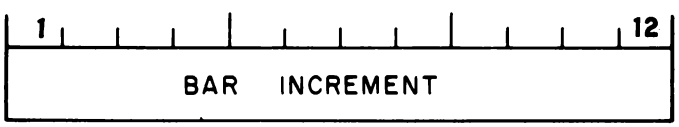


Figure 4-9

The subroutine jump increment syllable is defined by a decimal integer or an internal tag. The value in the J_i syllable is added to the present value in the BAR and the sum is then loaded into the BAR.

ASSEMBLER CODING FORMAT: SRJ J_a,J_i

1. SRJ REC,Ø
   1450 0320 0000 0000

L LOGICAL SYLLABLE

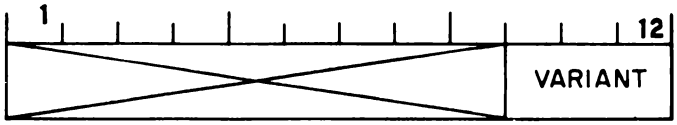


Figure 4-10

Bit 10 11 12

0	0	1	test POV (program overflow)
0	1	0	test PUN (program underflow)
1	0	0	test PNN (not normalized)

Any combination of more than one of the machine conditions may be specified in the same instruction by supplying the appropriate codes separated by a blank.

ASSEMBLER CODING FORMAT: BRC L,M

1. BRC POV,TOM
   1150 00001 1432 0000

2. BRC PUN,BOB
   1150 0002 0736 0000

3. BRC PNN,SKP
   1150 0004 1002 0000

4. BRC POV PNN PUN,ABC
   1150 0007 0010 0000

M MEMORY ADDRESS SYLLABLE

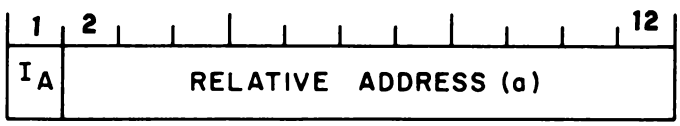


Figure 4-11

Bits 2 through 12 contain the relative address of a core memory location. The contents of the BAR are added to this address to obtain the absolute memory address. If bit 1 is set to Ø, the relative address is a direct address. If bit 1 is set to 1, it is an indirect address.

A memory syllable may be coded as-

1. an octal integer. (The number is NOT modified by the BAR at assembly time.)

2. a compool item or table tag. (If a system table is referenced, the syllable is NOT modified by the BAR at assembly time.)

3. an internal tag

4. a register containing word (RC word)

5. a temporary register. (A T followed by a decimal number references the TREGS. Leading zeros are ignored by the assembler.)

6. a stack reference- N for normal stack or H for hold stack.

ASSEMBLER CODING FORMATS: CLA M

1. CLA KIT
   2040 4322 0000 0000

2. BSU H,0(1),H
   6431 0407 0000 0000

3. LOR N,H,'CØØ2+X1Ø
   5507 0012 5337 0000

4. BAD N,T3,N
   6510 0512 0000 0000

5. CLA KIT+4
   2040 4326 0000 0000

O OPERATOR SYLLABLE

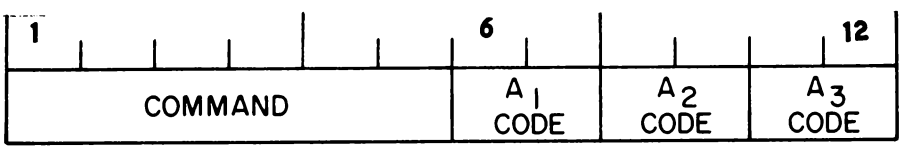


Figure 4-12

The first syllable of all instructions is an operator syllable. The first six bits indicate the fundamental operation to be performed by the instruction. Each of the instruction mnemonics has an octal number associated with it. The assembler converts the mnemonics to the binary equivalent of the associated number and places this equivalent in these first six bits. For example, a BAD (binary add) instruction would have $11Ø1Ø1_2$ (which is 65_8) in the first six bits of the operator syllable.

The second six bits contain three 2-bit address identification codes which, combined with the basic structure of the particular instruction, indicate the number and types of syllables that are to follow the operator syllable.

The 2/bit codes are:

00 - normal stack reference or no syllable (no syllable will be generated)

01 - hold stack reference (no syllable will be generated)

10 - unindexed memory, branch, or special syllables (one syllable will be generated

11 - indexed memory, branch or special syllable (two syllables will be generated; one for the index and one for the memory, branch, or special syllable)

ASSEMBLER CODING FORMAT:

1. UCT SA1Ø
   2240 1615 0000 0000

2. BSU NØØ4,SØ14,NØØ4
   6452 3501 3567 3501

3. AIF CTSA+X8,Ø 1 Ø,CTSA+X8
   4073 0010 0741 0020
   0010 0741

R_c REPEAT COUNT SYLLABLE

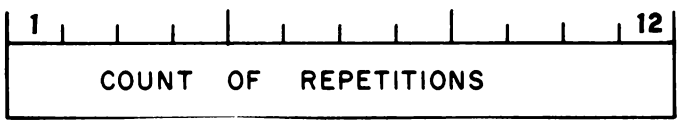


Figure 4-13

Repeat count syllable is defined by a decimal integer.

R_i REPEAT INCREMENT SYLLABLE

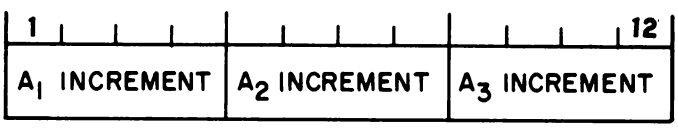


Figure 4-14

The repeat increment syllable is defined by three decimal integers ranging between Ø and 15.

ASSEMBLER CODING FORMAT: RPT R_c,R_i,B

1. RPT 42,1 1 Ø,CA4Ø
   1052 0052 0420 0102

S SHIFT SYLLABLE

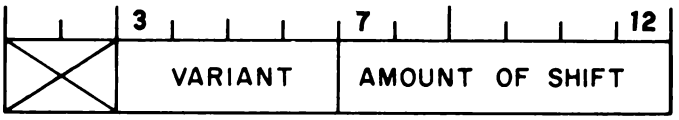


Figure 4-15

Bit 3 = 1 = double; 0 = single
Bit 4 = 1 = right; 0 = left
Bit 5 = 1 = logical; 0 = arithmetic
Bit 6 = 1 = end off; 0 = end around

The shift syllable may be coded as a decimal integer-n. (n $<$ 49)

```
FLCD N,39,N
3610 1247 0000 0000
```

To implement compool sensitivity on the shift instruction, the code CYC is used. The shift syllable may be two compool item tags or an item tag and a decimal integer (ranging from 1-48). The compool item tag may be followed by a slash (/) and a number (EPUN/5). The number must be smaller than the number of bits in the item. If two elements are used in the shift syllable, (EPUN/5 is one element) the two elements are separated by a blank. The first item tag or integer indicates the initial least significant bit position of the value. The second item tag or integer indicates the least significant bit position after cycling has occurred. If the item tag is followed by a slash and a number, the bit position indicated by the number is considered the item's least significant bit position.

ASSEMBLER CODING FORMAT: SHF M,S,M CYC M,S,M

1. CYC N,ITER 29,N
   3610 0613

2. CYC N,29 ITER,N
   3610 0645

3. CYCL N,ITER ITEM,N
   3610 0645

4. CYCL N,ITER/2 29,N
   3610 0625

5. CYC N,ITEM/3 ITER/7,N
   3610 0607

ITER is a 12 bit item starting in bit 7. ITEM is a 4-bit item starting in bit 4.

T THIN FILM ADDRESS SYLLABLE

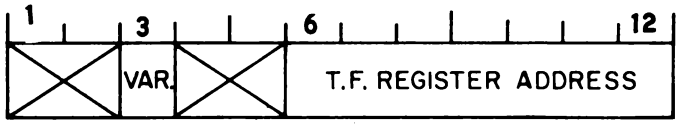


Figure 4-16

Bit 3 - Ø = 1 register; 1 = more than 1 register
Bit 6 - Ø = 16 bit register; 1 = 12 bit register

The T syllable may be a thin film register alphanumeric tag or the octal address of the thin film register.

ASSEMBLER CODING FORMAT: LTF M,T STF T,M

1. LTF H,1134
   3030 1134 0000 0000

2. STF PCR,H
   1544 0057 0000 0000

3. LTF H,BPR
   3030 0054 0000 0000

4. STF M PSR1,H
   1544 1100 0000 0000

V_t TRANSMIT VARIANT SYLLABLE

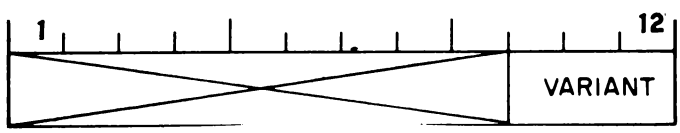


Figure 4-17

Bit 10	11	12	
1	0	0	R round
0	1	1	C change sign
0	0	1	+ make the sign positive
0	1	0	- make the sign negative

ASSEMBLER CODING FORMAT: TRM M,V_t,M

1. TRM ABCD,+,ABCD
   3452 0015 0001 0015

2. TRM ABCD,-,ABCD
   3452 0015 0002 0015

3. TRM ABCD,C,ABCD
   3452 0015 0003 0015

4. TRM ABCD,R,ABCD
   3452 0015 0004 0015

5. TRM ABCD,R +,ABCD
   3452 0015 0005 0015

V_s SPECIAL VARIANT SYLLABLE

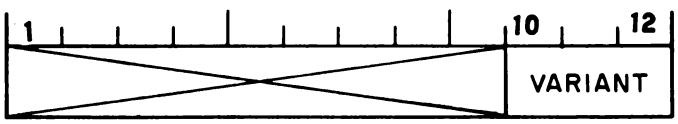


Figure 4-18

Bit 10	11	12	
1	0	0	Interrupt computer N
0	1	0	Upper/lower limit register to be loaded
0	0	1	Mask register to be loaded

The special or computer interrupt syllable may contain MASK (bit 10 = 1), BOUND (bit 11 = 1), and/or INTER (bit 12 = 1) as elements.

ASSEMBLER CODING FORMATS: LSR M,V_s

1. LSR Ø1A,MASK
   3150 0350 0001 0000

2. LSR Ø3A,BOUND
   3150 0400 0002 0000

3. LSR Ø4A,INTER
   3150 0420 0004 0000

X INDEX SYLLABLE

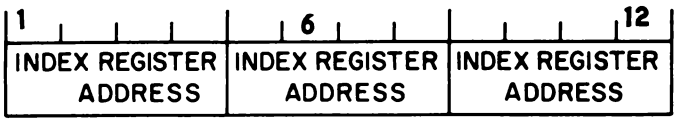


Figure 4-19

The 12 bits of an index syllable are divided into three 4-bit portions. Each portion specifies the number of an index register (1_{10} through 15_{10}). The index register number is the same as the index register address. If fewer than three indexes are used, the unused 4-bit portions will contain zeros.

The index syllable may NOT be used with the following: an operator syllable, references to RC words, the index syllable itself, within an instruction that is being repeated by a RPT instruction, and with any reference to the stack-N or H. Up to three index registers may modify one syllable. However, if the syllable is a compool defined MEMORY SYLLABLE, the assembler automatically allocates one index register which leaves the programmer only two index registers to use. For compool defined PROGRAMS the system limits the programmer in his choice of index registers. In BUIC III, X15 is the system index register.

ASSEMBLER CODING FORMAT:

1. UCT GAØØ+X1Ø
   2260 0012 0421 0000

2. CEF TSTA+X8+X9,Ø 5 Ø,HATD
   5272 0211 4363 0120
   0763 0000 0000 0000

ASSEMBLER PSEUDO CODES

Pseudo codes or instructions perform certain operations or functions. However, they do NOT have and do NOT GENERATE OCTAL codes such as a BAD (65_8) machine instruction. Pseudo codes are never executed by the computer.

There are two types of pseudo codes - GENERATIVE and NON-GENERATIVE.

1. Generative Pseudos - a pseudo instruction is defined as generative if ONE or MORE words appear in the OBJECT program as the result of its use.

2. Non-Generative Pseudos - a pseudo instruction is defined as non-generative if NO word appears in the OBJECT program as the result of its use.

DECLARATIVE CODES

Declarative codes are generative pseudo codes. FOR EACH DECLARATIVE CODE THE ASSEMBLER GENERATES ONE 48-BIT WORD OF INFORMATION. The one exception is the DIT code, which may cause more than one word of binary information to be generated. A list of the BUIC III Declarative Codes is provided in Table I.

TABLE I. BUIC III DECLARATIVE CODES

Declarative Code Columns 14-18	Declarative Code Name	Specific Layout Columns 19 and Following
ADR	Address	Internal tag or compool table tag or compool item tag or octal integer
ADRA	Address relative to BAR.	Internal tag or compool table tag or compool item tag or octal integer
ADRP	Address relative to BPR.	Same as ADRA
CMK	Complement Mask	Compool item tag
DEC	Decimal	Signed decimal integer or signed decimal fraction
DIT	Ditto	Decimal integer
DRA	Drum address	Compool program tag or compool table tag
FLT	Floating	Decimal integer and exponent
HOL	Hollerith	Eight characters
LNG	Length	Compool table tag
MSK	Mask	Compool item tag
OCT	Octal	Octal integer
SKP	Skip	Decimal integer
TIX	Table index	Block I of Compool table tag
VAL	Value	Compool item tag w/ corresponding value

ADR. The address declarative code causes the assembler to generate a binary address for the tag in the column 19 and following. The tag may be (1) an internal tag, (2) an octal integer, (3) compool table tag and decimal block number, (4) or compool item tag. The address generated is relative to the ORG card or if there is no ORG card, it is relative to zero. The address generated is right justified in the register. The ADR card is usually identified by a data tag in columns 8 and following. If an internal tag is used (in column 19 and following) the address associated with the tag in the tag table shall be used. Script an apostrophe if indirect addressing is desired. Bit 32 is set to a one in the 48 bit word generated. An indirect address bit must be used when reference is made to a compool defined system table so that an entry is made in the absolute address list. See TM-2780/004/00 Chapter 2 for more detailed information.

EXAMPLE:

```
Columns   0 0 1 1 1 1 1 1 1 1 1 1 2 2 2 2
          8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
          R A 1 2     A D R     ' A 1 2

Code:     c(RA12) = 0000 0000 0020 0054
```

ADRA. Declaring and scripting an ADRA code is identical to the ADR code. THE ADDRESS GENERATED IS RELATIVE TO THE BAR. At assembly time the address generated is decremented by the last SET BAR card value in the symbolic deck, unless an octal integer is coded in columns 19 and following.

EXAMPLE:

```
Columns   1 1 1 1 1 1 2 2 2 2 2
          4 5 6 7 8 9 0 1 2 3 4

          A D R A   A 1 2
```

A12 is associated with 54_8 in the tag table. The last "SET BAR,7" card caused a BAR value of 7.

```
Code:     0000 0000 0020 0045
```

ADRP. Declaring and scripting an ADRP code is identical to the ADR code. THE ADDRESS GENERATED IS RELATIVE TO THE BPR. At assembly time the address generated is decremented by the last SET BPR card value in the symbolic deck, unless an octal integer is coded in columns 19 and following.

EXAMPLE:

```
Columns   1 1 1 1 1 1 2 2 2 2
          4 5 6 7 8 9 0 1 2 3

          A D R P   A 1 2
```

A12 is associated with 54_8 in the tag table. The last "SET BPR, 17" card caused a BPR value of 17_8.

```
Code:     0000 0000 0000 0035
```

CMK. The complement mask declarative code generates a 48-bit binary complement mask containing 0's in all the bit positions occupied by the compool item and 1's in all other bits of the word. The item is defined in columns 19-22.

EXAMPLE:

```
Columns   1 1 1 1 1 1 2 2 2 2 2
          4 5 6 7 8 9 0 1 2 3 4

          C M K     I T E M
```

Item starts in bit 4 and is 4 bits long.

```
Code:     7037 7777 7777 7777
```

DEC. The decimal declarative code causes a signed decimal integer or a signed decimal fraction to be converted to a 48-bit binary word. Up to 8 characters are converted.

EXAMPLE:

```
Columns   1 1 1 1 1 1 2 2 2 2 2
          4 5 6 7 8 9 0 1 2 3 4

          D E C     + 2 9
          D E C     - . 5

Code:     0000 0000 0000 0035
          6000 0000 0000 0000
```

DIT. The ditto declarative code causes the last binary word which was generated as a result of a declarative code to be repeated the number of times specified by the decimal integer.

EXAMPLE:

```
Columns   1 1 1 1 1 1 2 2 2
          4 5 6 7 8 9 0 1 2

          D E C     + 9
          D I T     9

Code:     Will be 10 words containing
          0000 0000 0000 0011
```

DRA. The drum address declarative code DRA generates a 48-bit binary word containing the compool defined drum address for the table or program in bits 3-18, the sum of the A and B lengths if a program in bits 19-36, and the drum number in bits 43-48.

EXAMPLE:

```
Columns   1 1 1 1 1 1 2 2 2 2
          4 5 6 7 8 9 0 1 2 3

          D R A     A T R
          D R A     A L P 1
```

```
Code:     1773 0700 5050 0001
          1507 5000 0000 0001
```

FLT. The floating declarative code causes a signed decimal fraction with a signed exponent to be converted to a 48-bit floating point number. Bit 1 is the sign of the exponent, bits 2-12 are exponent, bit 13 is the sign of the mantissa and bits 14-48 are the mantissa. The decimal fraction must be signed and must have a magnitude less than one, and may contain up to 8 digits. The exponent, representing a power of 10, must be a signed decimal number with a maximum value of 618.

EXAMPLE:

```
Column    1 1 1 1 1 1 2 2 2 2 2 2 2 2
          4 5 6 7 8 9 0 1 2 3 4 5 6 7

          F L T     + . 5 0 E - 1

Code:     4001 2000 0000 0000
```

HOL. The Hollerith declarative code causes the characters in columns 19-26 to be converted to 6-bit Hollerith coding.

EXAMPLE:

```
Column    1 1 1 1 1 1 2 2 2 2 2 2 2 2
          4 5 6 7 8 9 0 1 2 3 4 5 6

          H O L     G T

Code:     2763 6060 6060 6060
```

LNG. The length declarative code generates a 48-bit binary word containing the number of blocks in the table in bits 1-6, the number of words per block in bits 21-30 and the total length of the table in bits 35-48.

EXAMPLE:

```
Column    1 1 1 1 1 1 2 2 2 2 2 2
          4 5 6 7 8 9 0 1 2 3 4

          L N G     B A 0 1

Code:     0200 0000 0500 0012
```

MSK. The mask declarative code generates a 48-bit binary mask containing 1's in the bit positions occupied by the compool item. All other bits are zero. The item is defined in columns 19-22.

EXAMPLE:

```
Column    1 1 1 1 1 1 2 2 2 2 2 2
          4 5 6 7 8 9 0 1 2 3 4

          M S K     I T E M
```

ITEM starts in bit 4 and is 4 bits long.

Code: 0740 0000 0000 0000

OCT. The octal declarative code causes the unsigned octal integer starting in column 19 to be converted to a right justified binary word. Leading zeros may be suppressed and the maximum number of digits is 16.

EXAMPLE:

```
Column    1 1 1 1 1 1 2 2 2 2
          4 5 6 7 8 9 0 1 2 3

          O C T     6 0 7 4 1

Codes:    0000 0000 0006 0741
```

SKP. The skip declarative code shall cause the assembler to increase the relative location counter by the amount of the decimal integer appearing in column 19 and following. No binary information is generated.

EXAMPLE:

```
Column    1 1 1 1 1 1 2 2 2 2 2
          4 5 6 7 8 9 0 1 2 3 4

          S K P     10
```

TIX. The table index declarative code generates a 48-bit binary word. This word contains the 3 letter table tag in bits 1-18, bit 24 contains a "1" if entry is in the mixed section of the compool and contains "0" if it is in the table section, the compool index value for the table is in bits 31-42 and the table type is in bits 43-48.

EXAMPLE:

```
Column    1 1 1 1 1 1 2 2 2 2 2 2
          4 5 6 7 8 9 0 1 2 3 4 5

          T I X     R S T 1

Code:     5165 2600 0004 2201
```

VAL. The value declarative code generates a 48-bit binary word containing the desired value for the item positioned to the item. The item desired is in columns 19-22. The value appears as a signed decimal number (8 or less digits) or as an unsigned octal number (8 or less digits) starting in column 24.

EXAMPLE:

```
Column    1 1 1 1 1 1 2 2 2 2 2 2 2 2
          4 5 6 7 8 9 0 1 2 3 4 5 6 7

          V A L     I T E M   - 6
          V A L     I T E M   + 6
```

ITEM starts in bit 4 and is 4 bits long.

Code: 0700 0000 0000 0000
0300 0000 0000 0000

There is one group of pseudo instructions which deal with shifting and cycling, all of which cause the assembler to generate an SHF instruction. These will be covered in a later chapter with the rest of the machine instructions.

RC WORD

One of the most useful and common operands is the "RC" word. RC means "Register Containing." RC words may be used in memory syllables. The Assembler generates the appropriate binary word at the end of the program and inserts its address into the M syllable.

	TYPE	SCRIPT	
1.	OCTAL	0(X)	Where X is an octal integer from 1-16 digits.

EXAMPLE: LCM 0(1), H
CODE: 2333 0407 0000 0000 c(0407) = 0000 0000 0000 0001

2. HOLLERITH H(Q) Where Q is up to 8 keypunch characters, and the) may not be one of the 8 characters.

EXAMPLE: TRS 8H(CONTROL), H
CODE: 3544 0434 0000 0000 c(0434) = 2346 4563 5146 4360

3. DECIMAL D(+X) Where X is a decimal integer or a decimal fraction (less than one) from 1-8 digits. A sign must be included.

EXAMPLE: TRS D(+12) ,H
CODE: 3544 0502 0000 0000 c(0502)= 0000 0000 0000 0014

4. MASK M(ITEM) Where ITEM is a compool defined item name.

EXAMPLE LOR ECCI,M(ECCI),ECCI (ECCI) is compool defined. It starts in bit 13 for one bit)
CODE: 5573 6400 2770 0405
6400 2770 c(0405) = 0000 4000 0000 0000

5. COMPLEMENT MASK C(ITEM) Where ITEM is a compool defined item name.

EXAMPLE: LAN EPUN,C(EPUN),H (EPUN is compool defined. It starts in bit 43 for 5 bits).
CODE: 5671 6400 1506 0406

c(0406) = 7777 7777 7777 7701

6. VALUE V(ITEM Y) Where ITEM is a compool defined item name and Y is a signed decimal number, an unsigned octal number, or a status for a status item.

EXAMPLE: LOR H,(EPUN +1), EPUN (EPUN is compool defined. It starts
CODE: 5533 0410 6400 1506 in bit 43 for 5 bits).

c(0410) = 0000 0000 0000 0002

7. FLOATING F(±.XE+Y) (Where X is a decimal fraction from 1-8 digits. Y is a power of 10 expressed as a decimal integer with a maximum value of 618. Both signs, the decimal point and the E are required.

EXAMPLE: BAD 100,F(-.8E+1) ,200
CODE: 6552 0100 0277 0200
c(0277) = 0004 6000 0000 0000

RC words may NOT be indirectly addressed, indexed, incremented nor decremented. Duplicate RC words reference the same register.

CONTROL CODES

A control code provides control information to the Assembler. Control Codes are non-generative with the EXCEPTION of the DATA and END cards. The tag field must be blank on all control cards except a comment card. A list of the BUIC III control codes is provided in Table II.

TABLE II. BUIC III CONTROL CODES

Control Code Columns 14-18	Control Code Name	Specifications Layout Columns 19 and Following
Blank	Comment	Information for Listing
DATA	Data	Not Used.
DRUM	Drum Location	Compool program tag of drum number and drum address
END	End	Internal tag or octal integer
IDT	Identification	Three character program tag and optional two character modification
ORG	Origin	One octal integer, two octal integers or compool program tag
PRGL	Program Length	One decimal integer or two decimal integers
REL	Relative Address	One octal integer, compool program tag or data
SET	Set Relative Base Address (BPR/BAR)	Base address type, internal tag or octal integer.
TREG	Temporary Register	Decimal integer

BLANK Columns 14-18. The comment card is used for symbolic information which is to appear on the symbolic delayed output (DLO) tape or printer listing.

DATA. A DATA control code indicates the start of the information which the programmer wishes to address under the BAR. The programmer groups all words which are used as data at the end of the program and heads this group with a DATA card. The assembler will save five words for control information.

The format for the five control words is shown in Figure 4-20. An example is given to illustrate the control words. In the example, the number of RC words is 223; the relative address of the RC words is 5147; the total length of the data area is 1772. What was the value of the ORG card? If no ORG card is in the deck, then the values for the BAR and BPR are zero.

	0 0 0 1 6 7	1 1 8 9	3 3 0 1	4 8
Word 1	Number of RC words	Number of instruction Area* Spares	Relative Address of Instruction Area Spares	
Word 2	Relative Address of RC words	Number of Data Area Spares	Relative Address of Data Area Spares	
Word 3	Relative Address of Instruction Area Spares	Total Length of Data Area	Relative Address of Absolute Address List	
Word 4			Absolute value of BPR from ORG card	
Word 5			Absolute Value of BAR from ORG card.	

*The INSTRUCTION AREA contains all binary information generated before the DATA card.

Figure 4-20. Five Control Words

SYMBOLIC LISTING			MACHINE CODE LISTING					
UPSBB	STF	L11,N	03435	1540	0033			
	LTF	N,PCR	03435			3010	0057	
	REL	3500						
	DATA		03500	0002	2300	4200	3436	five control words
			03501	0051	4701	0000	5372	
			03502	0034	3617	7200	1434	
			03503	0000	0000	0000	0000	
			03504	0000	0000	0000	0000	
N001	OCT	0	03505	0000	0000	0000	0000	

If a DATA card is inserted, DATA is considered as an internal tag and is listed in the tag table.

If NO DATA card is included in the deck, the five control words will still be generated. However, they will be labeled CONTROL WORDS and appear after the declarative codes and before the internally generated RC words. An example is given below:

	02754				0000
	02755	0000	0000	0000	0000
	02756	0000	0000		
CONTROL WORDS	02757	6000	6707	2300	3055
	02760	0027	6600	0001	4000
	02761	0030	5540	0000	0005
	02762	0000	0000	0000	0000
	02763	0000	0000	0000	0000
GENERATED CONSTANTS	02766	0000	0000	0000	0000
	02767	0000	0000	0000	0050
	02770	0000	0000	0004	7777

DRUM. A DRUM control code indicates the drum number and drum address of the following binary information. All binary information generated after the drum card is output in drum storage tape records. If no drum card is used, binary information is output in core storage tape records. Column 19 and following may be scripted in the following ways: (1) a compool program tag of three letters. The drum number and drum address from the compool are used. (2) Script a drum number where the number is 1-3. Leave one blank and add the drum address where the address is an octal integer from 0-177777.

END. The END control code indicates the last card is a symbolic program deck. An octal integer or an internal program tag may be coded in column 19 and following. If a tag is used, its address is decreased by the value on the last SET BPR,X card. The octal number or modified tag address is considered to be the relative starting address for operation of the program. This address is inserted in the second word of the core operate tape record. The octal number can range 00000 to 77777.

IDT. The identification card is the first card in every deck prepared for assembly. Columns 19-21 contain three alpha characters to identify the program name. Column 22-23 contain two alphanumeric characters, the modification (mod) of the program.

ORG. The origin card contains the BPR and BAR values for the program at run time. When a program is read into core from tapes, the BPR value on the ORG card is used as the first address for the program area. The BAR value is used as the first address for the data area.

```
            1 1 1 1 1 1 2 2 2 2
Columns     4 5 6 7 8 9 0 1 3 4

Example:    O R G       X , Y
```

The specifications field may be scripted in the following ways:

1. The BAR and BPR may be scripted from 0 - 77777_8. When only one number is used, both the program area and the data area are given the same base address value.

```
            1 1 1 1 1 1 2 2 2 2 2
Columns     4 5 6 7 8 9 0 1 2 3 4

Example:    O R G       1 0 0 0 0
```

C(BPR) = 10000
C(BAR) = 10000

2. Script the BPR where the octal address is from one to five digits 0-77777. Add a comma. Script the BAR with an octal address between 0 - 77777.

```
            1           1 2 2 2 2 2 2 2 2 2 2
Columns     4           9 0 1 2 3 4 5 6 7 8 9

Example:    O R G       4 0 0 0 0 , 4 3 0 0 0
```

C(BPR) = 40000
C(BAR) = 43000

3. Script a compool program name of three letters. The program core address is used as the BPR's value. The program core address plus the length of the program area is used as the value for the BAR.

```
            1 1 1 1 1 1 2 2 2 2 2
Columns     4 5 6 7 8 9 0 1 2 3 4

Example:    O R G       R A P
```

C(BPR) = value of compool program area.
C(BAR) = BPR + program area length.

PRGL. The program length control card indicates the lengths for the program instruction area and the data area. These lengths override previously defined lengths. This card must appear before the DATA card in the symbolic deck. The specifications field is mandatory and may be optionally scripted in one of the following ways:

1. Script program area length where the decimal length is from 0 - 20000. When only one number is used, the data area length is set to zero.

2. Script program area length where the decimal length is from 0 - 20000. Add a comma. Script data area length where the decimal length is from 0 - 2048.

REL. The relative address control card indicates the relative location of the binary information generated after the REL card. The location is relative to the origin card values. If no REL card is used, the relative location is assumed to start at zero. A REL card can be used as often as desired within a program and shall reset the assembler's counter that generates relative addresses.

The specifications field (column 19 and following) may be coded in the following ways:

1. Script the relative address with an octal number between 0 - 77777.

2. Script a three letter compool program tag. The program core address from the compool is used as the address for the following code.

3. Script DATA. The core address of the program entry in the dictionary plus the program area length is used as the relative address for the following code.

SET. The set BPR/BAR/SAR control code is used at ASSEMBLY time ONLY. The SET card tells the ASSEMBLER what the BPR and BAR values SHOULD be set to at run time. The assembler then computes all memory and branch syllables for these particular BAR and BPR values. BAR,BPR or SAR goes into columns 19-21. This is followed by a comma and an octal number, DATA or an internal program tag. Any number of SET cards may appear in a deck. If no SET cards are included, all addresses are relative to zero.

SET BAR,X — Where X is an octal number, DATA, or any internal program tag. All memory syllables except those generated for compool defined system tables and for octal integers shall have the numerical value of X subtracted from their numerical value.

SET BPR,X — Where X is defined as above. All branch syllables except those generated for an octal number shall have the numerical value for X subtracted from their numerical value.

SET SAR,X — Where X is an octal number, DATA or any internal program tag. All Ja syllables shall have the numerical value of X subtracted from their numerical value.

TREGS. The temporary register control code reserves the indicated number of words for temporary storage.

```
1 1 1 1 1 1 2 2 2
4 5 6 7 8 9 0 1 2
T R E G   N          where   1 ≤ N ≤ 1000
```

The TREGS are used in memory syllables.

```
BAD T1,T2,T3      is the same as
BAD T01,T02,T03   is the same as
BAD T001,T002,T003
```

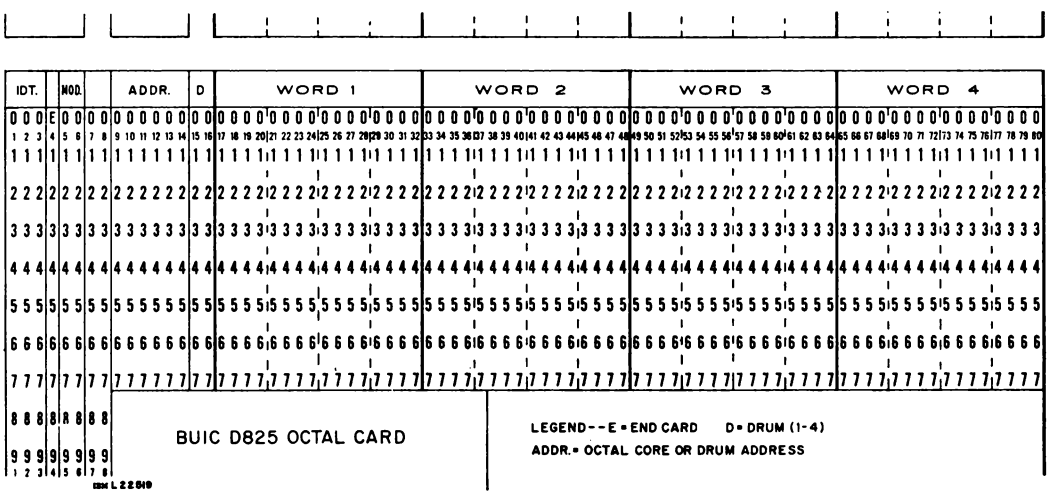
If your program happens to have an internal tag called T1, and a temporary register called T1, the program will reference the internal tag. There is only ONE TREG card in a deck and it can be inserted anywhere between the IDT and END card.

The following section of a program illustrates ORG, DIT, OCT, HOL, TREG, and REL CARDS.

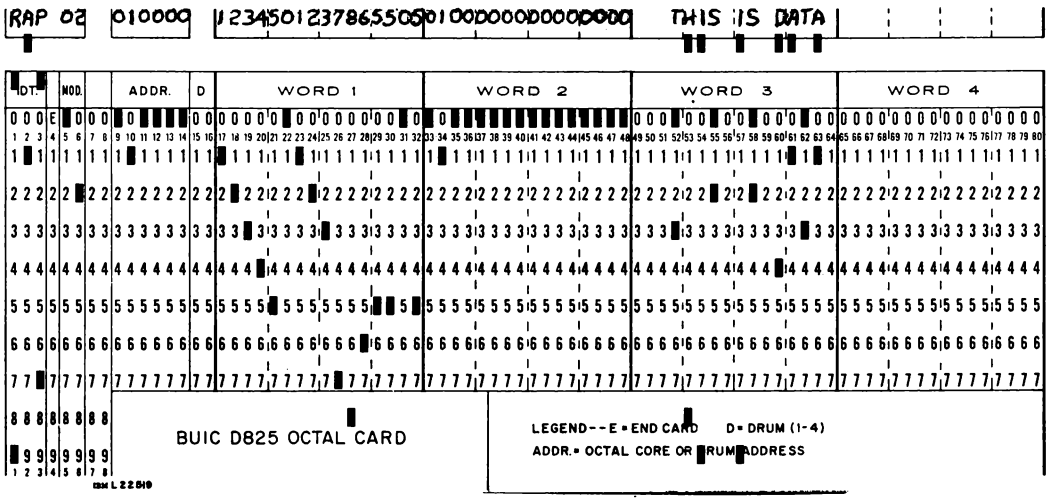
```
        ORG   0
*       NOP                                                    00000 0000
DLIST   UCT   START                                            00001 2240 2000
        OCT   0                                                00002 9999 0000 0000 0000
        DIT   1000                                             00003 0000 0000 0000 0000
SET     OCT   200021                                           01753 0000 0000 0020 0021
RELSN   OCT   20                                               01754 0000 0000 0000 0020
CMD1    OCT   17000100000301   Teleprinter One Line Output     01755 0017 0001 0000 0301
CMD2    OCT   40000100000010      Flex 40 Octal Words          01756 0040 0001 0000 0010
CUE     OCT   70010                                            01757 0000 0000 0007 0010
CONTL   HOL   CONTROL                                          01760 2346 4563 5146 4360
        HOL   MODE                                             01761 4446 2425 6060 6060
        TREG  40
        REL   2000
START   SRJ   CUE,CONTL                                        02000 1450 1757 1760
*       CLA   H                                                02001 2020
        LTF   H,M X1                                           02001      3030 1001
        LTF   H,M X14                                          02001           3030
                                                               02002 1016
```

OCTAL CORRECTOR CARDS

Octal correctors are used to make changes or corrections to program areas and data areas in core or on drums. (Octal correctors are called octal correctors because the information on the IBM card is coded in octal digits.) The following card format is now in use as a BUIC Octal Card.



CODING CONVENTIONS FOR OCTAL CORRECTOR CARD

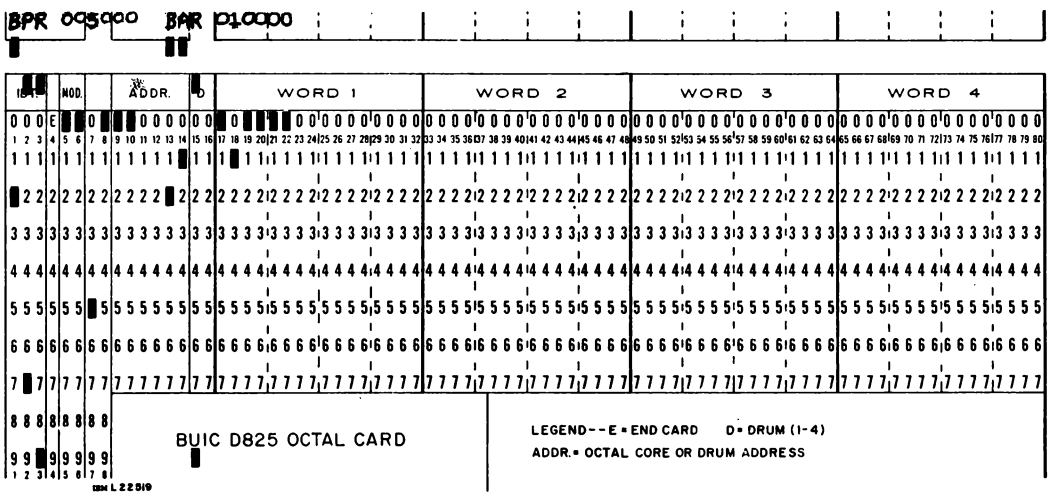


COLUMNS	SCRIPT
1-3	Identification of the program being corrected.
5-6	Model number of the program.
9-14	Relative address of data or instructions to be corrected. A "P" indicates that the address in cols 9-14 is to be considered as RELATIVE to the BPR. An "A" indicates that the address in cols 9-14 is to be considered as relative to the BAR. A blank indicates that the address in cols 9-14 is to be considered as ABSOLUTE. The address will NOT BE ADDED to the BPR or BAR values on the BPR/BAR card.
16	Script a drum number 1-3. If a drum number appears in column 16, the address in cols 9-14 will be interpreted as a drum address. If column 16 is left blank, the octal corrector is to be loaded into core memory.
17-32	First 16 digit word.
33-48	Second 16 digit word.
49-64	Third 16 digit word.
65-80	Fourth 16 digit word.

CODING CONVENTIONS FOR BPR/BAR CARD

The octal corrector deck is normally PRECEDED by a BPR/BAR card. The purpose of this card is to tell the loader program that all octal correctors following this card are to be loaded into memory relative to the value specified on the BPR/BAR card.

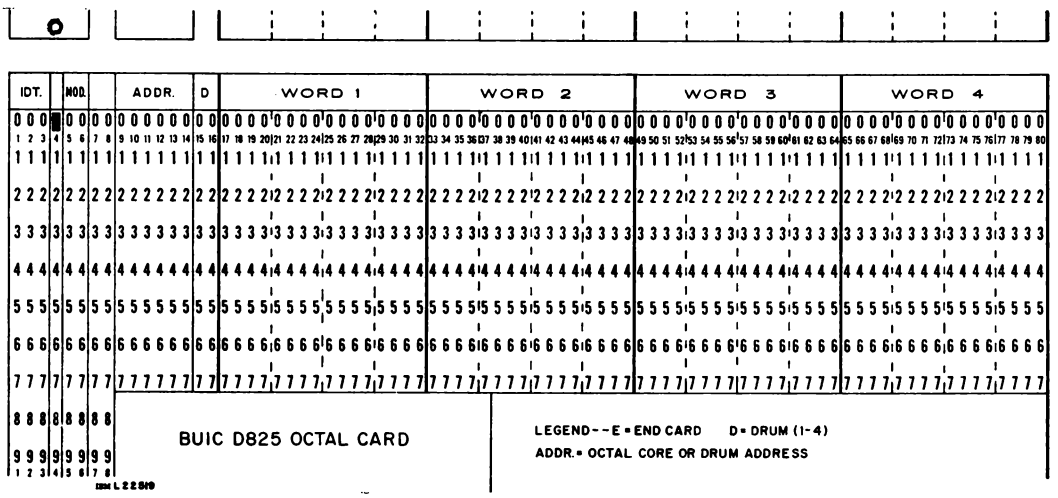
The CORE octal addresses are considered RELATIVE to the BPR or relative to the BAR. DRUM octals are considered ABSOLUTE addresses.



COLUMNS	SCRIPT
1-3	BPR
5-10	ABSOLUTE address of the first instruction in the program area.
13-15	BAR
17-22	ABSOLUTE address of the first data register in the data area.

CODING CONVENTIONS FOR END CARD

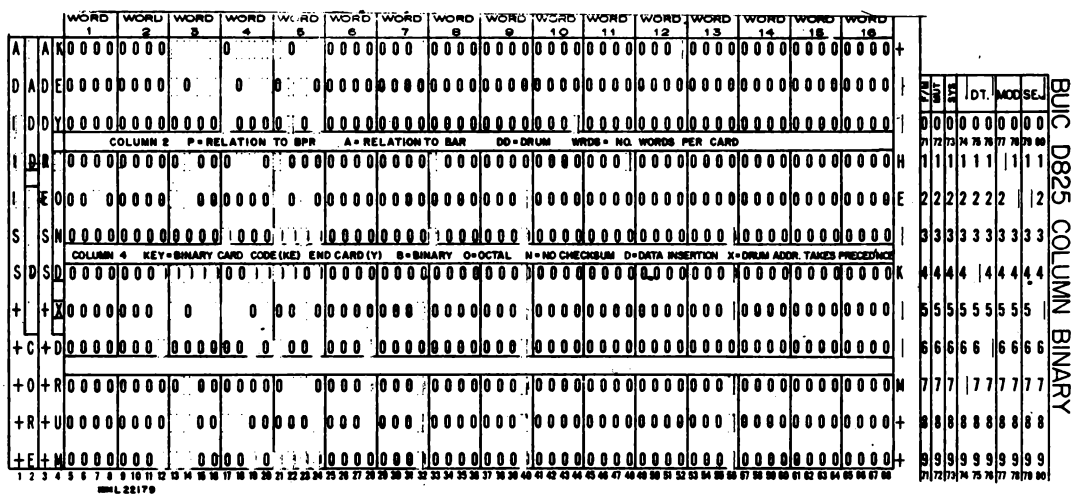
The octal corrector deck is FOLLOWED by an END card which causes termination of the octal load function.



COLUMNS	SCRIPT
4	zero

BINARY CARDS

Binary cards are rarely encountered in the BUIC III system -- mainly because the BUIC III system doesn't have the capability to produce binary cards. To produce binary cards, a card punch must be included as one of the peripheral devices. BUIC III doesn't have a card punch. However there are instances such as startover when binary cards might be used by the computer operator. Below is an example of a binary card.



Binary Card

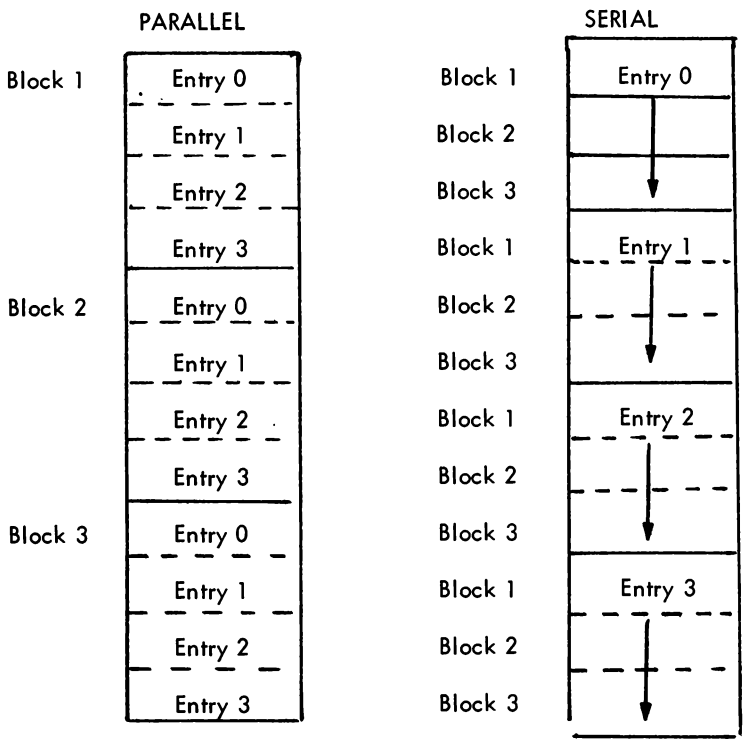
COLUMN	MEANING
1-4	contain control information such as whether the information is for drums, relative to the BAR or BPR, etc.
5-68	16 binary words.
69-80	More control information.

CHAPTER 5

INTERNAL DATA STRUCTURES

TABLES

The two basic structural table types are PARALLEL (or block) and SERIAL (or slot).



PARALLEL TABLES

A parallel table is always of RIGID or FIXED STRUCTURE, i.e., the number of blocks is always the same and items are unique to a certain block of the table. A parallel table could, however, be of VARIABLE LENGTH. If the number of entries in a parallel table is not constant and the table is packed from the top (entry 0 position in above diagram) down, this table might reasonably be considered of variable length.

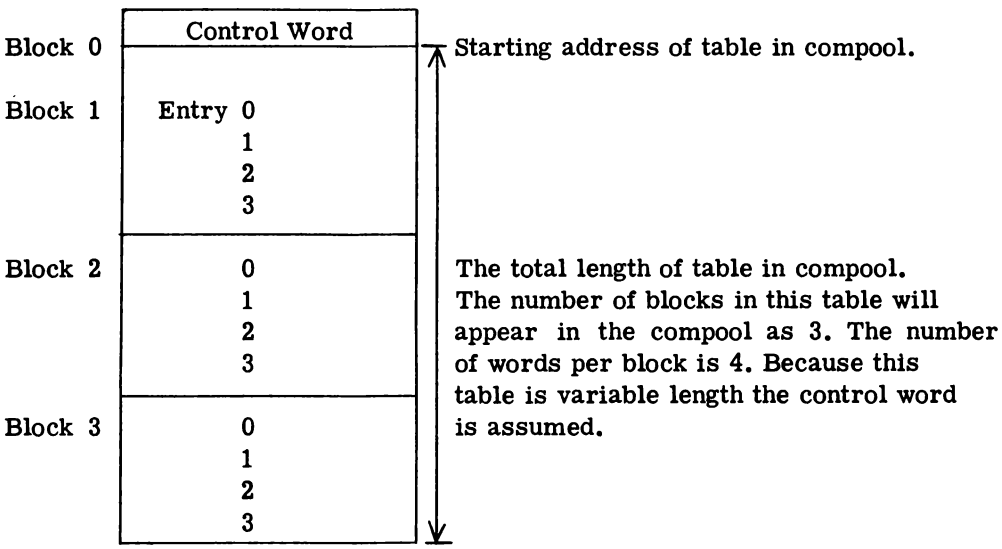
SERIAL TABLES

A serial table may be of RIGID length but it could also be a VARIABLE length table, i.e., the length of the table at any instant would correlate with the current number of entries or full slots in the table.

A serial table could also be of VARIABLE STRUCTURE. One entry might require eight registers, another three registers, and still another fourteen registers. This would mean that the number of registers in each entry is not constant. An example of this might be a recording specification table for BUIC.

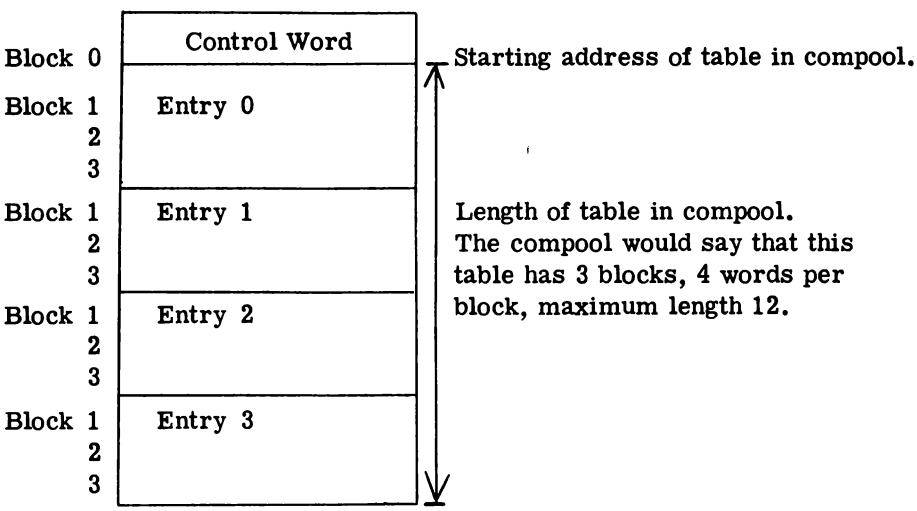
If a parallel or serial table has rigid structure but variable length, the NUMBER OF BLOCKS is equal to the number of words in each entry. The maximum number of entries would be the number of words per block, and the maximum table length would be the product of the two. An additional word, however, is appended to the beginning of the table. This word is maintained by those programs using the table. It contains the number of entries or number of full registers currently in the table. This word is addressed by requesting block 0 of the table. This control information should be in the least significant twelve bits of the word. THE ADDRESS OF THIS TABLE IN THE COMPOOL WOULD POINT TO THE FIRST WORD OF THE FIRST ENTRY (BLOCK 1), NOT TO THE CONTROL WORD (BLOCK 0).

VARIABLE LENGTH PARALLEL TABLE



For a serial table of rigid structure and rigid length the number of words per entry is the number of blocks in the table. The total number of entries this table will hold is the number of words per block. Assume in the example of a serial table given at the beginning of this chapter that this table was 300 registers long. There are three words in each entry (or three blocks). There would then be 100 words per block.

VARIABLE LENGTH SERIAL TABLE



A SERIAL TABLE of VARIABLE STRUCTURE and RIGID LENGTH will be defined in the compool as having one word per block; the number of blocks and the total table length are synonymous. The number of blocks in the compool for this type of table, however, will appear as zero. For a SERIAL table of VARIABLE structure and VARIABLE length the number of words per block should again be equal to ONE and the total (or maximum) table length and the number of blocks are synonymous. In the compool again, however, the number of blocks will appear as ZERO. This table will also have a CONTROL WORD appended to it (Block 0 of the table) which should contain the current number of full registers in this table.

BUIC TABLES

Tables in BUIC are addressed in three ways:

1. DIRECT ADDRESSING for those tables that occur within the range of a program's BAR. These are BAR TABLES.

2. ABSOLUTE ADDRESSING using index registers. These are SYSTEM TABLES.

3. INDIRECT ADDRESSING. If the table is neither a BAR table or a system table it is then an INDIRECTLY ADDRESSED TABLE.

If a table is under the range of a program's BAR, a core address is meaningless. The Assembler will assign the address according to the type of BAR table that it is. The BAR TABLE types are as follows:

a. TYPE A TABLES are those that must be saved from run to run. These tables will be saved on drums after each run of the program so that the next run will have this information again.

b. TYPE B TABLES are those that contain PRESET information but NEED NOT BE SAVED after each run. These tables will be brought into core for each run, may be modified if desired, but the new contents will not be saved. Hence, on the next run the original will be restored.

c. TYPE C TABLES are used as buffer area. These tables are used solely for the purpose of TEMPORARY STORAGE WITHIN ONE PROGRAM. The condition of the table prior to use is not guaranteed nor are the contents saved.

d. TYPE D TABLES are used in special I/O transfers. Programs that GENERATE TABLES FOR DISPLAY or use tables to RECEIVE INPUTS FROM SOME I/O DEVICE, will have this type of table. This will allow these tables to fall under the BAR of a particular program, yet will allow the control program to access them fairly easily.

ITEMS

The following is a brief review of the items used in BUIC.

B = BOOLEAN

A one-bit indicator or a series of one-bit indicators being treated as one item.

V = VALUE

The value of each configuration of bits in the item has a specific meaning, i.e.,

1 = north

2 = east

3 = south

4 = west

U = UNSIGNED

Any unsigned number. If the number is an integer (counters, arithmetic integers, etc.), it need not have scaling associated with it. If it contains fractional bits, then scaling must be associated with it.

S = SIGNED

Any signed number. Again it may or may not have scaling associated with it.

C = CHARACTRON

Charactron code is that code which generates symbols on the situation or tabular display scopes.

D = BINARY CODED DECIMAL

The item is divided into a series of four bits, with each set of four bits representing a decimal digit.

H = HOLLERITH

This Hollerith is assumed to be in Burroughs' format.

T = TRACK NUMBER

Track numbers should be left justified in the assigned bits.

M = MIXED

Items defined as M type are assumed to INCLUDE MORE THAN ONE DISTINCT PIECE OF INFORMATION grouped as one item for programming convenience.

COMPOOL

In OPERATIONAL BUIC programming, tables and items used by more than one program are listed in the compool (Communication Pool). This is nothing more than a convenient reference guide listing the tables, items, and programs in the BUIC ADP system. It gives such information as scaling, length of tables and items, bit positions of items, and core locations. During assembly, the assembler has access to the compool. Therefore, one may reference symbolically one particular table, and the assembler will automatically search the compool and insert the table's RELATIVE core location.

SYSTEM INDEX REGISTERS

When a compool is constructed, the area of core that the tables and programs occupy is divided into blocks of 2048_{10} or 4000_8 registers or half modules. There is a SYSTEM INDEX REGISTER assigned to each half module of core.

In the BUIC ADP system, compool occupies two entire consecutive memory modules. There are then four half modules. Each half module is assigned an index register. Index 11 is assigned the first half module; index 12 is assigned the second half module; index 13 the third half module; index 14 the fourth half module. For example, if compool occupies memory modules three and four - $20{,}000_8$ to $37{,}777_8$ the index registers would be loaded and assigned as follows:

c(X11) = 20,000	assigned to core locations 20,000 to 23,777
c(X12) = 24,000	assigned to core locations 24,000 to 27,777
c(X13) = 30,000	assigned to core locations 30,000 to 33,777
c(X14) - 34,000	assigned to core locations 34,000 to 37,777

When a compool defined table, item, or program is referenced, the assembler GENERATES (in addition to the memory syllable) an INDEX SYLLABLE containing the INDEX REGISTER associated with that section of core. It is the programmer's responsibility then, to be aware of this index register and to maintain its contents.

In the memory syllable a RELATIVE address is generated rather than an absolute address, because only 11 bits in a memory syllable can be used for an address. The twelfth bit is the indirect addressing bit.

A COMPOOL defined table, NPS01, is located at 20,000 for 100_{10} registers. The following reference is made to it.

```
*BAD  NPS01,N,N
```

The instruction would appear in core -

```
6564 4400 0000

(BAD) (X11) (NPS01's FIRST RELATIVE address)
```

Note that the ASSEMBLER AUTOMATICALLY added an index syllable and the address of the assigned index register. Nowhere in the symbolic coding did it specify X11. At run time the following would occur -

c(X11) = 20000

$$c\left(\begin{matrix}\text{memory}\\ \text{syllable}\end{matrix}\right) = \frac{0000}{20000} = \text{ABSOLUTE address of NPS01}$$

COMPOOL CODING CONVENTIONS

When referencing compool items, tables, and programs in a memory syllable, there are certain coding restrictions.

ITEMS = all item names are FOUR alpha characters.

TABLES = all table names are three alpha characters followed by ONE or TWO digits.

NPS01 is the SAME as NPS1. However, a table CANNOT be referenced as NPS.

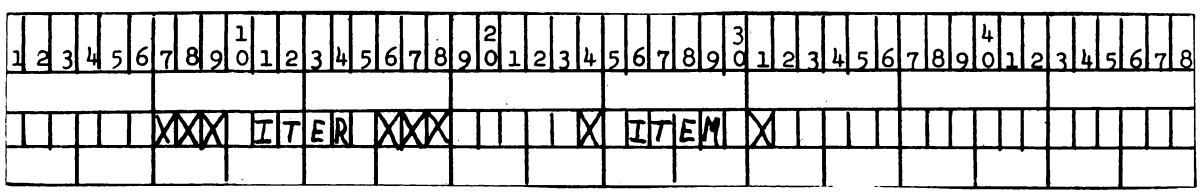
PROGRAMS = all programs are three alpha characters.

COMPOOL USAGE IN THE CYC INSTRUCTIONS

The following are special applications of compool using the CYC pseudo instruction.

To implement compool sensitive coding of the shift instruction, the code CYC in columns 14-16 is used. A 36_8 is generated in the operator syllable and the S (shift) syllable indicates a full right cycle. The syllable layout (starting in column 19) is (M), Y, (M). The Y may be two compool item tags or an item tag and a decimal integer (ranging from 1-48). The compool item tag may be followed by a slash and a number. The number must be smaller than the number of bits in the item. The two elements are separated by a blank. The first item tag or integer indicates the initial least significant bit position of the value. The second item tag or integer indicates the least significant bit position after cycling has occurred. If the item tag is followed by a slash and a number, the bit position indicated by the number is considered as the least significant bit position.

EXAMPLE	SYMBOLIC EQUIVALENT	CODE GENERATED BY THE ASSEMBLER	
CYC N,ITER 29,N	FRC N,11,N	3610	0613
CYC N,29 ITER,N	FRC N,37,N	3610	0645
CYC N,ITER ITEM,N	FRC N,13,N	3610	0615
CYC N,ITER/2 29,N	FRC N,21,N	3610	0625
CYC N,ITEM/3 ITER/7,N	FRC N,35,N	3610	0643



SCALING

A number, as such, is of very little use, and must be related in some way to the physical world to be meaningful. In other words it must represent units of some kind, such as miles, pounds, degrees, etc., and must contain a radix point to indicate which parts of the number are integral and which are fractional. The radix point in the decimal system is known as the decimal point, in the octal system as the octal point, and in the binary system as the binary point.

No digital computer keeps track of the unit represented by numbers within the machine; this is always up to the programmer. The AN/GSA-51A has the capability of keeping track of the position of the radix point within the machine; since this point may move, according to the operation performed, this is known as a FLOATING POINT capability. The AN/GSA-51A also may perform operations without keeping track of the radix point, but treating all numbers as fractions. This is known as a FIXED POINT capability. The technique of keeping track of the radix point in fixed point operations is known as SCALING.

SCALING NOTATION

To facilitate communication between programmers, a type of scaling notation must be arbitrarily adopted. The method to be used in this document will merely indicate the number of bits between the sign bit and the imaginary radix point. If we denote $0.011000\ldots._2$ as being scaled B2, the value has a sign bit, 2 integral bits, and 45 fractional bits. The above then, would represent a value of 1.5_{10}. If the scaling factor is B4, the number would contain a sign bit, 4 integral bits and 43 fractional bits, and would represent a value of 6_{10}.

ADDITION AND SUBTRACTION OF SCALED ITEMS

In addition or subtraction, it is imperative that the radix points be in the same position. That is, both numbers have the same scaling factor. Consider the following example where the scaling factors are not identical.

A = 2 scaled B2 = 0.100000...0.

B = 2 scaled B3 = 0.010000...0.

A + B = 0.110000...0.

In no way does this equal 4. It is necessary to shift registers to obtain identical scaling factors. Here we could shift A one bit to the right, or shift B one bit to the left. If we shift A we have the following:

A = 2 scaled B3 = 0.010000...0.

B = 2 scaled B3 = 0.010000...0.

A + B = 4 scaled B3 = 0.100000...0.

Another consideration remains. One must always provide for the largest possible sum in an addition or subtraction. Consider the preceding example. If we shift B to line up the radix points.

A = 2 scaled B2 = 0.10000...0.

B = 2 scaled B2 = 0.10000...0.

When A is added to B an overflow condition exists.

A + B = 0.00000...1.

Here the programmer provided for 2 integral bits (B2) for the result, and the sum (4) required 3 bits. Though the example was addition, the same considerations apply for subtraction. A final point is in keeping the sign bit of signed items in bit position 1.

MULTIPLICATION OF SCALED ITEMS

Scaling for multiplication is somewhat different than for addition or subtraction. It is not necessary to have identical scaling factors for the two operands. It is however, necessary to place the sign bits of the operands in bit position 1. The scaling factor of the result is equal to the sum of the scaling factors of the operands. Recall from BMU instruction that the TFC register is tied to the A3 syllable to contain the 96 bit product.

EXAMPLE:

A X B, put product into C

A = 0000000000000012 or 12_8 scaled B47

B = 0000000000000005 or 5_8 scaled B47. After BMU A, B, C

C = 0000000000000000 TFC = 0000000000000062

The product is 62_8 scaled B94. (sign bits in A3 and TFC, and 94 integral bits). Rescaling the product to B47 (right justified in memory) can be accomplished most easily by STF M TFC, C.

Another approach to the preceding problem is often used. That is to left justify the operands. It is easy to rescale the product, if necessary, from this standpoint.

DIVISION OF SCALED ITEMS

As in all other operations, the sign bit of the operands must be positioned to bit position 1. The scaling factor of the quotient is obtained by subtracting the scaling factor of the division from the scaling factor of the dividend. Due to the operating characteristics of the division operation, it is necessary to insure that the DIVISOR IS LARGER THAN THE DIVIDEND. This is accomplished, when necessary, by rescaling the operands to meet this requirement.

Given the following Compool:

DOGS	MICO	01	05	S	3.01
CATS	MICO	08	04	S	3.00
RATS	MICO	16	07	S	5.01

Solve: DOGS $\div$ CATS, put the result into RATS.

Without rescaling the operands, the quotient would be scaled BO(B3-B3). The prime consideration is in decreasing the magnitude of DOGS to insure its value is less than CATS and that the result will be scaled B5.

Solution:

```
LAN  CATS, M(CATS), N
FLS  N,7,N              Get sign of Cats in Bit Position 1.
LAN  DOGS, M(DOGS), N
ARS  N,5,N              Rescale DOGS to B8 while maintaining sign bit.
BDV  N,N,N              Result scaled B8-B3 = B5 in top of stack position.
FRS  H,16,H             Result to bit positions of RATS.
LAN  N,M(RATS),N
LOR  N, RATS, RATS      Deposit Quotient into RATS
```

CHAPTER 6

BUIC III ASSEMBLER OUTPUTS

This chapter discusses the conversion of symbolic coded language to binary machine language, a process accomplished by the assembler. Topics covered include assembly listings, error printouts, octal dumps, thin film dump, DLO, and a dictionary.

The programmer codes instructions for his program in a symbolic language. It is the function of the assembler to convert this symbolic language. The assembler also produces an assembly listing in which symbolic language is listed side by side with the octal representation of the binary machine language. Any errors in the symbolic language that are detected by the assembler will be noted.

ASSEMBLY LISTINGS

The following is a sample assembly listing. The columnar headings are NOT included on listings.

SAMPLE ASSEMBLY LISTING

SEQUENCE NUMBERS	TAG FIELD	OP CODE	OPERANDS AND COMMENTS	RELATIVE CORE MEMORY LOCATION	OCTAL REPRESENTATION OF INSTRUCTION WORDS AND PROGRAM DATA IN CORE MEMORY
1		LTF	ZERO,X1 LOAD INDEX	00000	3050 0006 0001
2		LTF	FIVE,L1 LOAD INDEX	00000	3050
				00001	0007 0020
3	AGAIN	BAD	SUM,DATA+X1,SUM	00002	6556 0005 0400 0010
				00003	0005
4		XLC	+1,X1 LS L1,AGAIN	00003	1252 0001 2021
				00004	0003
5		HLT		00004	0100
6	SUM	OCT	0 DEFINE DATA	00005	0000 0000 0000 0000
7	ZERO	OCT	0	00006	0000 0000 0000 0000
8	FIVE	OCT	5	00007	0000 0000 0000 0005
9	DATA	OCT	3	00010	0000 0000 0000 0003
10		DEC	-6	00011	4000 0000 0000 0006
11		OCT	14	00012	0000 0000 0000 0014
12		OCT	7	00013	0000 0000 0000 0007
13		OCT	25	00014	0000 0000 0000 0025

ERRORS RESULTING FROM SYMBOLIC INPUTS

The following error messages are output on the symbolic listing as a result of incorrect symbolic input. These errors do not cause the program to halt.

"ASSEMBLER ERROR XXX": An incompatibility has been found within the assembler or the programmer has exceeded the absolute address list. XXX indicates the operand number.

"ADDRESS ILLEGAL XXX": A negative address or an address exceeding the legal maximum has been encountered. XXX indicates the operand number.

"BAR OVERFLOW": The data area has been exceeded.

"COMPOOL ERROR XXX": A compool tag has been used incorrectly. XXX indicates the operand number.

"CONTROL VALUE ERROR": An error has been encountered in the specifications field on a control code card.

"DECLARATIVE VALUE ERROR": An error has been made in the specifications field on a declarative code card.

"DICT OVERFLOW": The dictionary has overflowed.

"DUPLICATE TAG XXX": A duplicate tag has been encountered in the tag field (XXX = blank) or a duplicate tag has been used in the specifications field. XXX indicates the operand number.

"ILLEGAL INSTRUCTION": An illegal instruction has been encountered in the operation code field.

"ILLEGAL OPERAND XXX": An error has been encountered while processing the operand indicated by XXX.

"ILLEGAL TAG": An error has been encountered while processing the tag field.

"INDEXING ERROR XXX": Too many indexes and/or an illegal index have been encountered while processing the operand indicated by XXX.

"UNDEFINED TAG XXX": A tag encountered in the specifications field is not in the compool and does not appear in the tag field. XXX indicates the operand number.

"VALUE TOO LARGE XXX": The value of an increment or decrement is greater than 4095. XXX indicates the operand number.

The following is a sample assembly listing which contains errors detected during the assembly process.

```
0001            IDT     TEST                                                    AA1
0002                       THE FUNCTION OF THIS PROGRAM IS TO PROVIDE A
0003                       SAMPLE OF ASSEMBLER OUTPUT
000301          DRUM    4 1000
0004            SET     BPR,START
0005            SET     BAR,DATA
000501          TREG    2
0006    START   STF     PCR,N                                   00000  1540 0057            0 001000
0007            LTF     N,BPR                                   00000                3010 0054  0 001000
0008    10A     CLA     T1                                      00001  2040 0006            0 001001
      ILLEGAL LITERAL           2ND                                                          ERROR
0009            BAD     ONE,0(1),H                              00001            6551 0002  0 001001
                                                                00002  0000                  0 001002
      ILLEGAL operand.                                                                       ERROR
0010            ADD     91R,H,12                                00002       0000 0000 0000  0 001002
                                                                00003  0000 0000 0000 0000  0 001003
      ILLEGAL VARIANT           2ND                                                          ERROR
0011    10B     XLC     -3,X1 NO LC,10B                         00004  1252 0003 7020 0004  0 001004
      ILLEGAL OPERAND           3RD                                                          ERROR
0012            BSL     91R,T2,T2 THIS COMMENT IS INCORRECTLY SPACED    00005  6452 0000 0007 0000  0 001005
      ILLEGAL INSTRUCTION                                                                  ERROR
0013            LCR     T1,91R+X10+X9,T2                        00006  0000 0000 0000 0000  0 001006
                                                                00007  0000 0000 0000       0 001007
0014            REL     50
      ILLEGAL operand           1ST                                                          ERROR
0015    10C     LTF     0(+9),X5                                00050  3050 0000 0005       0 001050
0020            XLC     +2,X5 NO L9,10C                         0005C                1252    0 001050
                                                                00051  0002 0131 0050       0 001051
0021            DATA
      ILLEGAL INSTRUCTION                                                                  ERROR
0022    91R     DLC     +999                                    00052  0000 0000 0000 0000  0 001052
                                                                00053  0000 0000 0000       0 001053
0023    ONE     OCT     1                                       00054  0000 0000 0000 0001  0 001054
0024            HOL                                             00055  6060 6060 6060 6060  0 001055
0025            END     10C                                     00056                       0 001056

10A   00001 *   1CB   00004     10C   00050     91R   00052     DATA  00052     ONE   00054     START 00000

* UNREFERENCED TAGS
```

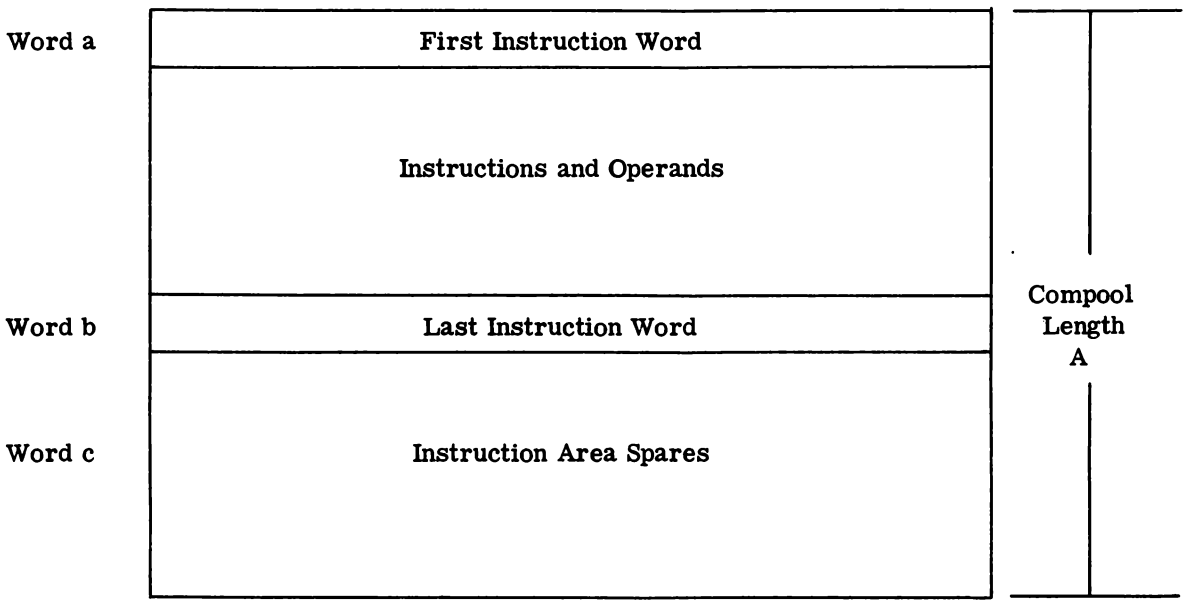
NOTE: When an error is detected by the assembler, it leaves seven blank syllables at the location. This can be seen by looking at the right side of the listing.

BINARY OUTPUT

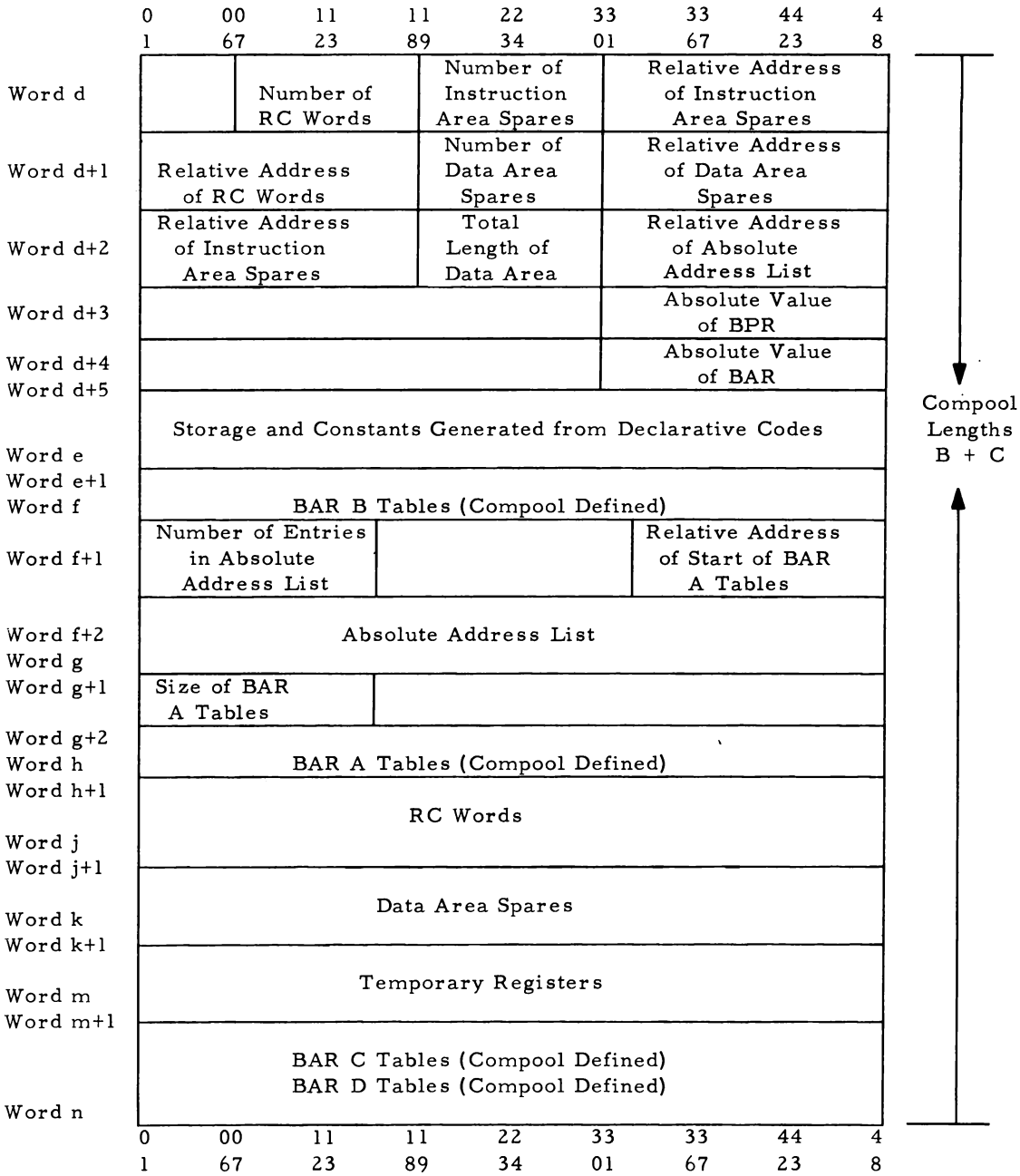
A binary output from the assembler, if requested, is provided on tape. The binary output includes instructions and operands, data, register containing (RC) words, an absolute address list, control words, dictionary and index usage table.

PROGRAM LAYOUT

The instruction area contains all binary information generated before the DATA card is input.



The data area is initiated when a DATA card is input. If no DATA card is used, the data area is assumed to be immediately following the last binary information generated and all binary storage records are relative to the BPR.



WORD	BITS	DESCRIPTION
d	07-18	Number of RC words (word h+1 through word j)
d	19-30	Number of instruction area spares (word b through word c). If no data card (word j+1 through word k).
d	31-48	Address of instruction area spares (word b). If no data card (word j+1).
d+1	01-18	Address of RC words (word h+1).
d+1	19-30	Number of data area spares (word j+1 through word k). (Zero if no data card.)
d+1	31-48	Address of data area spares (word j+1). If no data card (word k+1).
d+2	01-18	Address of instruction area spares. If no data card (word j+1).
d+2	19-30	Total length of data area (Zero if no data card.)
d+2	31-48	Relative address of absolute address list (address of (f+1) minus address of d).
d+3	31-48	Address for BPR thin film register from ORG card.
d+4	31-48	Address for BAR thin film register from ORG card.
d+5 thru e	01-48	Storage and constants generated from declarative codes.
e+1 thru f	01-48	Space for BAR B tables. These tables are defined in the compool. They are read in from drums but not back out on drums.
f+1	01-12	Number of entries in absolute address list (word f+2 through word g).
f+1	31-48	Address of start of BAR A tables (word g+1).
f+2 thru g	25-30	Key

0 = Null
1 = Program
2 = System subroutine
3 = Internal procedure
4 = Compool defined indirectly addressed table or indirectly addressed system table
5 = Table allocated by overlay or simple item allocated by overlay (Not used in assembler)
7 = Item or table block reference in compool defined indirectly addressed table

	KEY: 1	2	3	4	5	7
01-12	Compool Index in program section (if compool defined)	Compool index in program section	Procedure length	Compool index in table section		Compool index in table section for parent table (if in compool defined table)
13-24	A Length		Relative address to start of internal procedures			
25-30	Key = 1	Key = 2	Key = 3	Key = 4	Key = 5	Key = 7
31-48	Absolute core address	Absolute core address	Absolute core address	Absolute core address	Absolute core address	Absolute core address

WORD	BITS	DESCRIPTION
g+1	01-12	Total size of all BAR A tables (word g+2 through h)
g+2 thru h	01-48	BAR A tables. These tables are defined in the compool. They are read in and out from drums.
h+1 thru j	01-48	RC words
j+1 thru k	01-48	Spares in data area (If no data card, spares in instruction area.)
k+1 thru m	01-48	Temporary registers
m+1 thru n	01-48	BAR C tables (compool defined) BAR D tables (compool defined)

DELAYED OUTPUT (DLO)

In addition to a binary output program, PSA,PSB will produce a SYMBOLIC listing on TAPE for the programmer. This tape can then at a later date be dumped on the printer. The format and appearance of a DLO is identical to any symbolic listing directly outputed by the assembler.

OCTAL CORE DUMPS

An octal core dump is when the contents of core have been dumped in their octal configuration. Just one register or all of memory, 77,777 registers, can be dumped.

At the beginning of each octal dump, the absolute core locations dumped are listed. In the example, locations 30,000 to 50,000 have been dumped. In an octal dump there is no way to tell which registers contain data and which contain instructions.

Octal dumps are read from left to right. At the extreme left is the absolute core location of the first register on that line. Looking at the example, the contents of location 30,000 are 3050.0123.0020.0000. A dot is used to separate each 12 bit syllable. Location 30,001 is next. Its contents are 3050.0124.0015.3050, and so forth.

When there are duplicate lines to be printed, the printer prints

*** DUPLICATE LINE/S ***

Notice that 030170 to 030327 are duplicate zeros, so they aren't printed.

CORE 030000 050000

030000	3050.0123.0020.0000	3050.0124.0015.3050	0125.0025.3050.0126	1004.0000.0000.0000
030004	2060.3015.0014.1252	0001.2145.0004.3050	0126.1004.0000.0000	5673.2015.0036.0127
030010	2015.0046.1252.0001	2105.0007.3050.0126	0004.0000.0000.0000	2060.3015.0036.1252
030014	0001.2145.0013.3050	0126.0006.0000.0000	5671.3015.0025.0130	3030.0007.7632.0126
030020	0035.0000.0000.0000	5671.3015.0025.0131	3631.0701.4170.3415	0054.0060.6101.3631
030024	0504.4131.0060.4073	2015.0036.0060.2015	0036.4171.3415.0054	2463.5670.3015.0025
030030	0131.3631.0701.6101	3631.0504.4131.2460	4073.2015.0036.0065	2015.0036.1252.0001
030034	3500.0040.0000.0000	5671.3015.0025.0131	7246.0126.0040.2240	0021.0000.0000.0000
030040	1252.0001.2145.0016	3050.0126.1004.0000	7672.2015.0036.0126	0060.3544.0124.5670
030044	2015.0036.0132.3631	0536.6501.3631.0501	4131.1046.5633.0131	2015.0014.3544.0124
030050	4170.2015.0036.1465	3631.0536.6501.3631	0501.4131.2046.4073	2015.0014.0042.2015
030054	0014.3544.0133.4073	2015.0014.0065.2015	0014.1252.0001.2105	0042.0000.0000.0000
030060	3050.0126.1004.7172	0015.0044.0125.0114	3544.0126.5272.2415	0036.0024.0066.0000
030064	1252.0001.2125.0062	2240.0112.0000.0000	7666.2015.0046.0112	7666.2415.0036.0064
030070	5671.2415.0036.0132	5670.2015.0046.0132	6401.5645.0134.6126	0122.3652.0122.0523
030074	0122.4171.2415.0036	1465.4170.2015.0046	1465.6401.3431.0001	6125.3631.0523.6531
030100	0122.7532.0135.0064	3544.0126.5272.2015	0046.0024.0106.3544	0121.4073.2415.0036
030104	0024.2415.0036.2240	0112.0000.0000.0000	5573.2415.0036.0136	2415.0036.1544.0004
030110	4131.2027.4073.2415	0036.0023.2415.0036	3050.0126.0005.1252	0001.2105.0062.0000
030114	3544.0120.3030.0057	0000.0000.0000.0000	0000.0000.0000.0000	0000.0000.0000.0023
030120	0000.0000.0003.0777	0000.0000.0100.0000	0000.0000.0000.0001	6060.6060.6060.2543
030124	0000.0000.0001.0000	0000.0000.0000.0005	0000.0000.0000.0000	7777.7700.7777.7777
030130	0000.0000.0000.7777	7777.0000.0000.0000	7777.7700.0000.0000	0000.0000.0057.2051
030134	3777.7700.0000.0000	0000.0000.0000.0031	0000.0000.0200.0000	7777.0000.0000.0000
030140	0000.0000.0000.0024	0000.0000.0000.0000	0000.0000.6200.0000	7777.7777.0077.7777
030144	0000.0000.0100.0000	0000.0000.0200.0000	7777.7700.7777.7777	0000.0000.0003.1000
030150	0000.0000.0000.0000	0000.0000.0000.0025	0000.0000.0000.0000	0000.0077.0000.0000
030154	7777.7700.0000.0000	0100.0000.0000.0000	7777.0000.0000.0000	0000.0000.4057.2051
030160	0000.0000.0000.0031	0000.0000.0100.0000	0000.0000.0200.0000	0000.0000.0003.0777
030164	0000.0000.0000.0000	0000.0000.0000.0023	0000.0000.0000.0000	0000.0000.0000.0000
030170	0000.0000.0000.0000	0000.0000.0000.0000	0000.0000.0000.0000	0000.0000.0000.0000
***		DUPLICATE LINE/S		***
030330	0000.0000.0000.0000	0000.0000.0057.2051	0071.6200.0000.0000	5133.0000.0000.0000
030334	0000.3471.0000.0000	0000.0000.0000.0115	0000.0000.0000.0005	0000.0000.0000.0005
030340	0000.0000.0000.0005	0000.0000.0000.0000	0000.0000.0000.0000	0000.0000.0000.0000
030344	0000.0000.0000.0005	0000.0000.0000.0002	0000.0000.0010.7600	0000.0000.0001.0000

Octal Dump

THIN FILM OUTPUT

A thin film dump may be requested as DLO on the tape drive or direct via the flexowriter or printer. In either case the format is the same.

X0	000000	L0	000020	ISR	000040	SAR	000060	PSR1	0103	0102	0101	0100
X1	000001	L1	000021	ISR	000041	61	000061	PSR2	0107	0106	0105	0104
X2	000002	L2	000022	ISR	000042	XIR	000062	IPR	0113	0112	0111	0110
X3	000003	L3	000023	43	000043	IAR	000063	RTC	0117	0116	0115	0114
X4	000004	L4	000024	RPR	000044	PDR	000064	RCR	0123	0122	0121	0120
X5	000005	L5	000025	RPR	000045	PDR	000065	TFC	0127	0126	0125	0124
X6	000006	L6	000026	RPR	000046	66	000066	RIR	0133	0132	0131	0130
X7	000007	L7	000027	RPR	000047	67	000067	1134	0137	0136	0135	0134
X8	000010	L8	000030	SSR	000050	IDR	000070	S1	0143	0142	0141	0140
X9	000011	L9	000031	SSR	000051	71	000071	S2	0147	0146	0145	0144
X10	000012	L10	000032	SSR	000052	72	000072	S3	0153	0152	0151	0150
X11	000013	L11	000033	53	000053	73	000073	S4	0157	0156	0155	0154
X12	000014	L12	000034	BPR	000054	74	000074	1160	0163	0162	0161	0160
X13	000015	L13	000035	BAR	000055	75	000075	1164	0167	0166	0165	0164
X14	000016	L14	000036	56	xxxxxx	76	000076	1170	0173	0172	0171	0170
X15	000017	L15	000037	PCR	xxxxxx	77	000077	1174	xxxx	xxxx	xxxx	xxxx

1. With three exceptions, the names of all of the thin film registers are used; octal addresses are used for the spares. These three exceptions are as follows:

   a. The real time clock (RTC) is actually two 12 bit thin film registers, 114 and 115. Registers 116 and 117 are spares.

   b. The repeat count register (RCR) is actually only one 12 bit register, number 12Ø. Registers 121 and 122 are spares. Register 123 is the character count register (CCR).

   c. The repeat increment registers (RIR) are actually 13Ø, 131, and 132. Registers 133 is a spare.

2. The registers containing X's in the sample will contain invalid information in an actual dump; these registers are used by the program that dumps the thin film.

DICTIONARY

A dictionary is a detailed listing of all items, tables, constants, and tags (compool and non-compool) used in the source program. The dictionary is generated after the last binary information in the program. A sample dictionary is given. An explanation of the headings is given after the dictionary.

ENTRY	NAME	CLAS	DEFN	TYPV	FBIT	CHL1	CHL2	SIZE	DIMN	PACK	IOFL	DRGT	DUPT	SDC	OLAY	SIND	PDAT	DECL	ASGN	DONE	LOCA
0	00000000	0-NULL	0-NUL	0-INT	0	0	0	0	0	0	00	0	0	0	0	0	0	0	0	0	000000
1	00000000	0-NULL	0-NUL	0-INT	0	0	0	0	0	0	00	0	0	0	0	0	0	0	0	0	000000
2	00000000	0-NULL	0-NUL	0-INT	0	0	0	0	0	0	00	0	0	0	0	0	0	0	0	0	000000
3	RIP13	12-PGM	2-COM	0-INT	0	0	0	0	0	0	00	0	0	0	0	0	0	1	0	0	030000
4	AA	1-STAL	6-ASM	24-MIX	0	0	0	48	0	0	00	0	0	0	0	0	1	1	1	0	000003
5	RXLO	5-SITM	7-NSC	0-INT	0	1	29	12	0	1	00	0	0	0	0	0	0	1	1	0	000103
6	NXPO	5-SITM	7-NSC	0-INT	0	5	30	18	0	1	00	3	0	0	0	0	1	1	1	0	000104
7	OXPO	5-SITM	7-NSC	0-INT	0	9	31	12	0	1	00	2	0	0	0	0	0	1	1	0	000105
8	CC	1-STAL	6-ASM	24-MIX	0	0	0	48	0	0	00	0	0	0	0	0	1	1	1	0	000010
9	IRNC	5-SITM	7-NSC	0-INT	0	3	32	12	0	1	00	2	0	0	0	0	0	1	1	0	000106
10	BB	1-STAL	6-ASM	24-MIX	0	0	0	48	0	0	00	0	0	0	0	0	1	1	1	0	000026
11	IAZM	5-SITM	7-NSC	0-INT	36	3	32	12	0	1	00	12	0	0	0	0	0	1	1	0	000106
12	SINE	5-SITM	7-NSC	0-INT	0	11	33	18	0	1	00	17	0	0	0	0	1	1	1	0	000107
13	COSI	0-NULL	0-NUL	24-MIX	0	0	0	48	0	0	00	0	0	0	0	0	1	0	0	0	000000
14	NYPO	5-SITM	7-NSC	0-INT	30	5	30	18	0	1	00	3	0	0	0	0	1	1	1	0	000104

Sample Dictionary

ITEM NAME	ITEM MEANING
ASGN	1 = location assigned indicator
BRGT	Number of fractional bits in item
CHL1	
CHL2	
CLAS	1 = statement label or tag 3 = constant 4 = item 5 = simple item 7 = table 12 = program
DECL	1 = entry was declared in source program
DEFN	2 = compool defined table 6 = internal tag 7 = compool defined item
DIMN	1 = rigid length table 2 = variable length table If entry is a tabular item, value is the word of the entry.
DONE	
DUPT	
ENTRY	The entry number of this item, table, or program in the dictionary table.
FBIT	1 = parallel table 2 = serial table 3 = initial bit position of a simple item or tabular item.
IOFL	This item is the index into the overflow table, if the name is greater than five bytes.
LOCA	Relative core address
NAME	Name of a program, table, item, or tag.
OLAY	1 = overlay declaration
PACK	0 = no packing 1 = medium 2 = dense 3 = program specified packing

ITEM NAME	ITEM MEANING
PDAT	1 = signed item
SDC	1 = entry is a secondary dictionary channel
SIND	1 = preset data
SIZE	If item - number of bits If table - number of words per entry
TYPV	0 = integer 1 = floating point 2 = hollerith 5 = fixed point 6 = boolean 8 = BCD 16 = track coded number item 24 = mixed If entry COMPOOL defined TABLE 0 = BAR A table 1 = BAR B table 2 = BAR C table 3 = BAR D table 4 = Indirect addressed tables 8 = System table

CHAPTER 7

AN/GSA-51A COMPUTER INSTRUCTIONS

This chapter contains the AN/GSA-51A instructions, rules for their usage, and examples thereof. They are arranged in nine groups--signifying in general those instructions which are related to one another. The groups are arranged in the approximate order of importance and usage. Each instruction heading includes the mnemonic code, instruction name, and the octal code. The first line under the heading indicates the syllable layout. Beginning at the left each letter designate, e.g., (M), T, B, is numbered as the A_1 location; the second letter, A_2; the third letter, A_3, respectively. It should be remembered that instructions may vary as to their requirements for locations of operands from zero to three locations. Where no letter designates are shown, only the operator syllable is required for the computer to complete the instruction.

The groups of instruction for the purposes of this chapter are as follows:

1. Fixed-Point Arithmetic
2. Thin Film and Stack
3. Commonly Used Ungrouped Instructions
4. Comparison
5. Logical Operations
6. Cycling and Shifting
7. Field
8. Floating-Point Arithmetic
9. Miscellaneous

The following glossary identifies the letter designates in the mnemonic code used in this chapter. Each letter refers to a distinct instruction syllable type.

LETTER	SYLLABLE TYPE
1. B	Branch address
2. C	Character
3. F	Field
4. Ia	Index increment amount
5. IO	Input/Output
6. Iv	Index increment variant
7. Ja	Subroutine jump address

LETTER	SYLLABLE TYPE
8. Ji	Subroutine jump increment
9. L	Logical machine conditions
10. M	Memory only
11. (M)	Memory or stack (Fetch operation)
12. $\underline{(M)}$	Memory or stack (Store operation)
13. Rc	Repeat count
14. Ri	Repeat increment
15. S	Shift
16. T	Thin film address
17. Vs	Special or computer interrupt variant
18. Vt	Transmit variant

FIXED-POINT ARITHEMTIC INSTRUCTIONS

The fixed-point arithmetic instructions considered in this group involve the algebraic manipulation of the entire 48-bit word. Arithmetic overflow can occur in all the fixed-point arithmetic instructions with the exception of BMU, binary multiplication.

For fixed-point addition and subtraction, overflow occurs when the results of the computation exceed the limit of the 48-bit word, where the first bit is reserved for the sign bit. In the examples shown below for binary addition and subtraction, the leftmost octal number, if 4 or greater, indicates a minus sign bit.

For fixed-point division, overflow occurs when the absolute value of the divisor is less than or equal to the absolute value of the dividend. In order to avoid an overflow situation, therefore, the divisor must always be scaled to meet this condition. The reason for this special condition is derived from the fact that all arithmetic data words are treated as fractional values by the AN/GSA-51A.

In all cases the locations specified by the A_1 and A_2 syllables remain unchanged unless referenced by A_3.

BAD - BINARY ADD - 65_8

BAD (M),(M),(M)

BAD performs fixed-point addition. The contents of the location specified by A_1 and the contents of the location specified by A_2 are algebraically added together. The resulting sum is placed in A_3.

If the address results in zero, it will always be a positive zero. A negative zero will be changed to positive zero.

INDICATORS

The POV (Program OVerflow) indicator flip-flop is set if overflow occurs and the result of the overflow is placed in the location specified by A_3. The true sum, scaled 2^{-1}, may be determined from the overflow result by subtracting a one and shifting right one position. The answer is in effect equal to one-half of the correct answer.

Note in the following examples that the first bit of the 48-bit word denotes the sign.

Ex: (1) BAD AUG1,ADD1,SUM1

AUG1 = 377777777777777_8 (positive value)
ADD1 = 4000000000000043_8 (negative value)
SUM1 = 377777777777734_8

POV Indicator Not Set

(2) BAD AUG2,ADD2,N

AUG2 = 377777777777732_8 (positive value)
ADD2 = 0000000000000065_8 (positive value)
Top of stack = 0000000000000020_8

POV Indicator Set*

*Adding the two octal numbers gives a preliminary result of 3000000000000017 with a left-over 1, which cannot be added to the sign bit. It is therefore carried over and added to 17, thereby producing the 20 and causing the POV indicator to be set.

BSU - BINARY SUBTRACT - 64_8

BSU (M),(M),(M)

BSU performs fixed-point subtraction. The contents of the location specified by A_2 are algebraically subtracted from the contents of the location specified by A_1, and the difference is placed in the location specified by A_3.

If the difference is zero, a positive zero (0000000000000000_8) will be placed in the location specified by A_3. A negative zero will be changed to a positive zero.

INDICATORS

POV (Program OVerflow) indicator flip-flop is set if overflow occurs and the result of the overflow is placed in A_3. The true difference, scaled 2^{-1}, may be determined from the overflow result by subtracting one in bit position 48, shifting right one position and LORing a one in bit position 2.

Note in the following examples that the first bit of the 48-bit word denotes the sign.

Ex: (1) BSU MIN1,SUB1,DIFF1

MIN1 = 0000000000000001_8 (positive value)
SUB1 = 4000000000000004_8 (negative value)
DIFF1= 0000000000000005_8

POV Indicator Not Set

(2) BSU MIN2,SUB2,N

MIN = 3776000000000000_8 (positive value)
SUB = 4007770000000000_8 (negative value)
Top of Stack = 0005770000000001_8

POV Indicator Set**

**Subtracting the negative octal number from the positive octal number gives a preliminary result of 0005770000000000 with a carry-over of 1, which cannot be added to the sign bit. The one is therefore added to the difference to produce the entire result.

BMU - BINARY MULTIPLY - 61_8

BMU (M),(M),(M)

BMU performs fixed-point multiplication. The contents of the location specified by A_1 are algebraically multiplied by the contents of the location specified by A_2. The least significant half of the double word length product is placed in TFC (Thin Film C - 124_8 - 127_8); the most significant half is placed in the location specified by A_3.

The sign of the TFC is the same as the sign of A_3 with the following exception: If the most significant half of the product is zero, a positive zero (0000000000000000_8) is placed in the location specified by A_3. TFC, however, may contain a negative zero (4000000000000000_8).

Note in the following examples that the first bit of the 48-bit word denotes the sign bit.

Ex: (1) BMU MULT1,MULD1,PROD1

MULT1 = 4000000000000010_8 (negative value)
MULD1 = 0000000000000001_8 (positive value)

PROD1 = 0000000000000000_8
TFC = 4000000000000010_8

(2) BMU MULT2,MULD2,N

MULT2 = 0000000000000100_8 (positive value)
MULD2 = 0100000000000001_8 (positive value)

Top of Stack = 0000000000000002_8
TFC = 0000000000000000_8

BDV - BINARY DIVIDE - 60_8

BDV (M),(M),(M)

BDV performs fixed-point division. The contents of the location specified by A_1 are algebraically divided by the contents of the location specified by A_2. The resulting quotient is placed in A_3, the remainder (which takes its sign from the dividend A_1), in TFC (Thin Film C - 124_8 - 127_8).

If the quotient has a value of zero, a positive zero (0000000000000000_8) is placed in the location specified by A_3; TFC may contain a negative zero (4000000000000000_8).

Note in the following examples that the first bit of the 48-bit word denotes the sign bit.

INDICATORS

POV, the Program OVerflow flip-flop, will be set by the resulting quotient overflow if the absolute value of the contents of A_1 is greater than that of A_2.

Ex: (1) BVD DVND1,DVSR1,QUOT1

DVND1 = 0000000000000001_8 (positive value)
DVSR1 = 0000000000000003_8 (positive value)

QUOT1 = 1252525252525252_8
TFC = 0000000000000002_8

(2) BDV DVND2,DVSR2,N

DVND2 = 4000000000000001_8 (negative value)
DVSR2 = 0000000000000002_8 (positive value)

Top of Stack = 6000000000000000_8
TFC = 4000000000000000_8

THIN FILM AND STACK INSTRUCTIONS

The instructions described in this group are those which make special reference to either the thin film or stack registers. As noted earlier, the operand stack is located in 16 registers of thin film, although direct access to the stack can be made by specifying H or N in the appropriate operand location without using a thin film instruction. The special stack instructions outlined in this group refer only to the rotation and direction of rotation of the stack.

The following is a list of thin film registers providing a description, symbolic name, number of bits and octal address of location. Either the symbolic name or the octal address may be used in coding to gain access to the thin film registers.

SYMBOLIC NAME	DESCRIPTION	# BITS	$LOCATION_8$
BAR	Base Address Register	16	055
BPR	Base Program Register	16	054
CCR	Character Count Register	12	123
IAR*	Interrupt Base Address Register	16	063
IDR	Interrupt Dump Register	16	070
IPR	Interrupt Program Register	48	110-113
ISR	Interrupt Storage Register	48	040-042
LØ & L1-L15**	Limit Registers Ø - 15	16 ea	020-037
PCR	Program Count Register	16	057

*All thin film registers may be addressed in both modes except for IAR (Interrupt Address Register - 063_8) which can be loaded in the control mode only. The IAR can be stored, however, in either mode.

**Values may be stored in LØ, but LØ in any index/limit compare (XLC) instruction has an effective value of zero.

SYMBOLIC NAME	DESCRIPTION	# BITS	LOCATION$_8$
PDR	Power Failure Dump Register	32	064-065
PSRI1	Program Storage Register 1	48	100-103
PSR2	Program Storage Register 2	48	104-107
RCR	Repeat Count Register	12	120
RIR	Repeat Increment Registers	48	130-132
RPR	Repeat Program Register	48	044-047
RTC	Real Time Clock	24	114-115
SAR	Subroutine Base Address Register	16	060
SSR	Subroutine Storage Register	64	050-053
STK1 or S1	Stack Register 1	48	140-143
STK2 or S2	Stack Register 2	48	144-147
STK3 or S3	Stack Register 3	48	150-153
STK4 or S4	Stack Register 4	48	154-157
TFC	Thin Film C	48	124-127
XIR	Index Increment Register	16	062
X1-X15	Index Registers 1 - 15	16 ea	001-017
Not used		16	000
Spares		16	043
		16	053
		16	056
		16	061
		16 ea.	066-067
		16 ea.	071-077
		12 ea.	116-117
		12 ea.	121-122
		12 ea.	133-137
		12 ea.	160-166

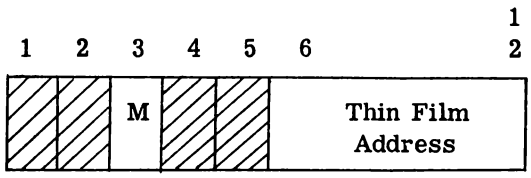
LTF - LOAD THIN FILM - 30_8

LTF (M),T

LTF takes the contents of the location specified by A_1 and places as many of the least significant bits as indicated by A_2 into the thin film register(s) specified by A_2.

Multiple thin film registers loading has the following two peculiarities: the final two bits of the Thin Film address are treated as if both were zeros, whether or not this is the case, and the 48 bits of data are loaded so that the first thin film register receives the LEAST significant set of 12 or 16 bits and the others successively more significant sets.

A_2 Instruction Syllable Structure:



Bit 3 is one to indicate multiple thin film register loading of 48 bits or zero to indicate single Thin Film Register loading of 12 or 16 bits.

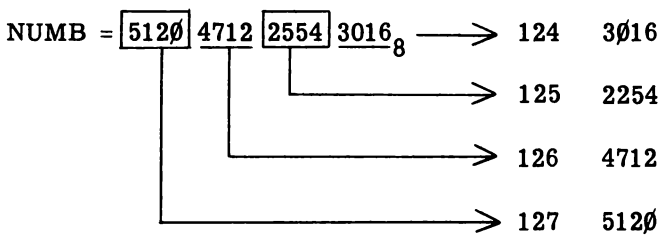
Ex: (1) LTF XFER,PCR

XFER = 4705213655400017_8

PCR = 000017_8

Note that only the 16 least significant bits of XFER are loaded in the PCR register, since PCR is a 16-bit register.

(2) LTF NUMB,M TFC



Note further that the least significant set of 12 bits is loaded in the first register. TFC can nevertheless be symbolically understood to contain the entire value of NUMB in its original order.

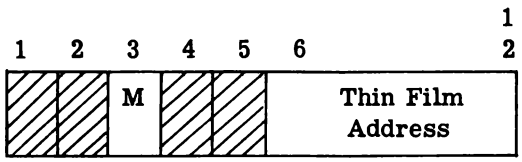
STF - STORE THIN FILM - 15_8

STF T,(M)

STF takes the contents of the thin film register(s) specified by A_1 and places as many of the least significant bits as indicated by A_1 into the location specified by A_2. Either 12 or 16 bits are transferred right justified. The remaining bits of A_2 are set to zero.

Multiple thin film register storing has the following two peculiarities: the final two bits of the thin film address are treated as if both were zeros, whether or not this is in fact the case. Secondly, the contents of the first thin film register to be stored is placed right justified in the location specified by A_2. Each succeeding register is stored so that the most significant bits, which are contained in the last thin film register, are stored left justified in the location specified by A_2.

A_1 Instruction Syllable Structure:

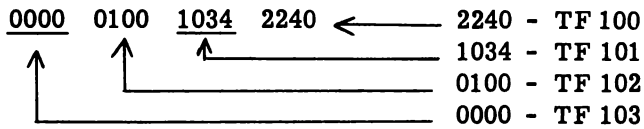


Bit 2 is one to indicate multiple thin film register loading of 48 bits, or zero to indicate single thin film register loading of 12 or 16 bits.

Ex: (1) STF X5,SAVE

X5 = 154023_8
SAVE = 0000000000154023_8

(2) STF 11Ø2,TEMP (Multiple thin film storing using an octal thin film address)



Note that the computer in the above example ignored the final two bits of the thin film address, and commenced storing from $1ØØ_8$.

XLC - INDEX/LIMIT COMPARE - 12_8

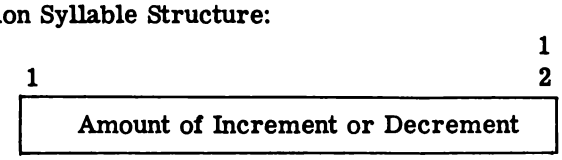
XLC Ia,Iv,B

XLC alters the contents of the Index Register specified by bits 5-8 of A_2 in a manner specified by bit 1 of A_2 by the amount specified in A_1. The new* value is compared to the contents of the Limit Register specified by bits 9-12 of A_2; if the condition(s) specified in bits 2-4 of A_2 is (are) met, control will continue to the first syllable of the location specified in A_3, otherwise control will continue to the next instruction in sequence.

If Limit Register zero is specified, comparison will be made to the value zero.

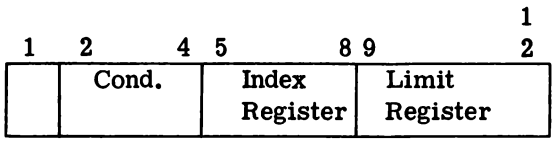
Decrementing is symbolically indicated by a minus Ia value, incrementing by a positive value.

A_1 Instruction Syllable Structure:



*NOTE: The index register is altered first, THEN the comparison is made.

A_2 Instruction Syllable Structure:



Bit 1 is zero to increment, one to decrement.

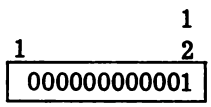
Conditions specified in bits 2-4 of A_2:

VALUE	MEANING	Iv SYMBOL
0	No transfer - index will be altered only	NO
1	Index = limit	EQ
2	Index > limit	GR
3	Index ≥ limit	GQ
4	Index < limit	LS
5	Index ≤ limit	LQ
6	Index ≠ limit	NQ
7	Unconditional Transfer	BR

In coding the A_2 syllable, the condition is separated from the index and limit registers by blanks. The increment (or decrement) amount is coded in DECIMAL notation.

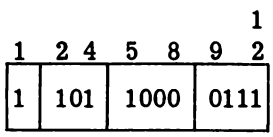
(1) XLC -1,X8 LQ L7,END

A_1 syllable



$= 0001_8$

A_2 syllable



$= 6607_8$

Note the amount of decrement is stored in the A_1 syllable and the minus sign is stored in Bit 1 of the A_2 syllable.

Before - X8 = 000005_8
L7 = 000004_8

After - X8 = 000004_8; therefore, program will branch to location END, which must contain an appropriate instruction.

Ex: (2) XLC +1,X3 BR LØ,FIXIT

Before - X3 = 000006_8
After - X3 = 000007_8; program will unconditionally branch to FIXIT, which must contain an appropriate instruction. Note that a Limit Register must be indicated although no comparison is made. This instruction provides for incrementing an index register and unconditional branching.

Ex: (3) XLC +1Ø,X9 NO LØ,Ø

Before - X9 = 000012_8
After - X9 = 000024_8; program will go on to the next instruction in sequence. Although no comparison or branching is specified, a limit register and a branch address must be specified.

Ex: (4) XLC +Ø,X2 EQ L2,OUT

Before - X2 = 000006
After - X2 = 000006; L2 = 00002 program will not branch. The next sequential instruction will be operated. This instruction provides for comparing an index register without altering the index register.

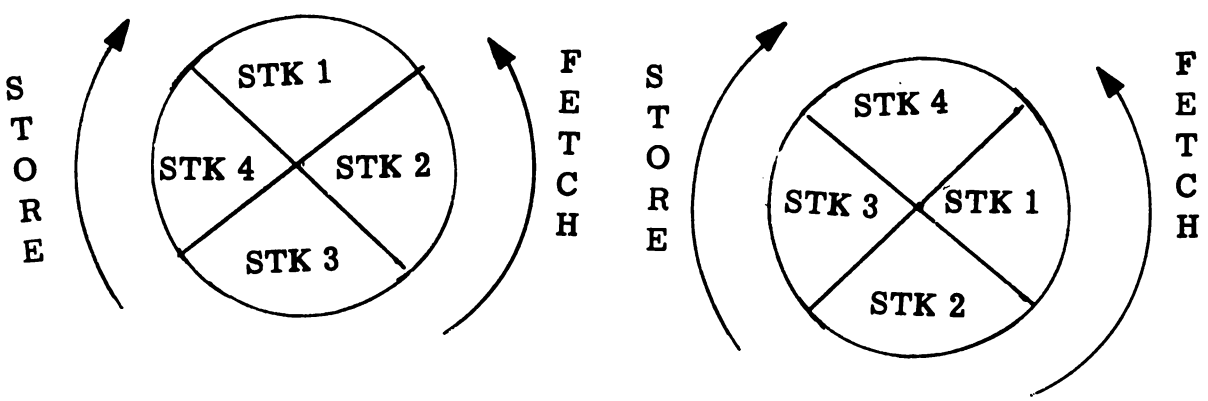
SSD or SSS - STEP STACK DOWN - 03_8

SSD (or SSS)

SSD (or SSS) moves the stack in the store direction one position but transfers no data.

Ex: SSD

The arrows indicate the direction of stack movement.



BEFORE INSTRUCTION AFTER INSTRUCTION

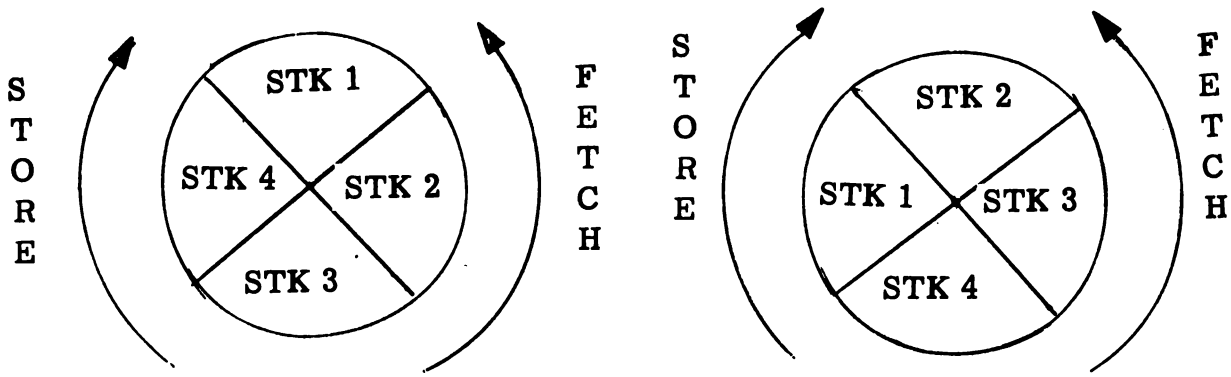
SSU or SSF - STEP STACK UP - 02_8

SSU (or SSF)

SSU (or SSF) moves the stack in the fetch direction one position but transfers no data.

Ex: SSU

The arrows indicate the direction of stack movement.



BEFORE INSTRUCTIONS AFTER INSTRUCTIONS

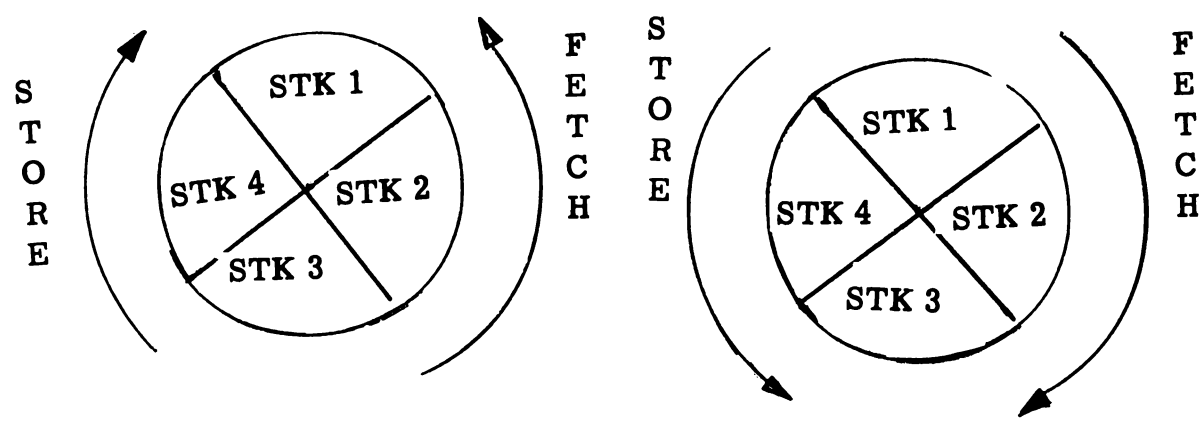
RVS - REVERSE STACK - 06_8

RVS

RVS reverses the direction of stack movement to make what was previously the fetch direction subsequently the store direction, and vice versa, for succeeding stack references. No data is transferred and the top of the stack remains unchanged.

Ex: RVS

The arrows indicate the direction of stack movement.



BEFORE INSTRUCTIONS AFTER INSTRUCTIONS

COMMONLY USED UNGROUPED INSTRUCTIONS

The instructions considered in this group differ greatly in function and type. Their importance and frequent usage warrants consideration at this point rather than grouping them with Miscellaneous Instructions, which is the last group treated in this document.

BRB - BRANCH ON BIT - 26_8

BRB (M),(M),B

BRB takes the contents of the location specified by A_1 performs a right, logical, end-around shift of 1 bit and places the result in A_2. If the low order bit of the INITIAL contents of the location specified by A_1 is one, control will transfer to the first syllable of the location specified by A_3; otherwise, control continues to the next instruction in sequence.

Ex: BRB CHECK,N,BRNCH

Before - Check = 0000000000000777_8

After - Check = 0000000000000777_8

Top of Stack = 4000000000000377_8

Program will branch to location BRNCH which must contain an appropriate instruction left justified in the memory word. Note that the contents of CHECK remain unchanged after the instruction. The end-around shifted data is stored in the location specified by the A_2 syllable, which in this example is the top of the stack.

CLA - CLEAR - 20_8

CLA (M)

CLA clears all 48 bits at the location specified by A_1 to zero.

Ex: CLA H

Before - Top of stack = 4000000000000000_8

After - Top of stack = 0000000000000000_8

HLT - HALT - 01_8

HLT

HLT affects computer operation in one of two ways depending on whether the computer is operating in the normal or the control mode.

In the normal mode, the computer is interrupted, that is, switched to the control mode and control is transferred to the 11th location greater than that specified in the IAR (Interrupt Address Register - 063_8). An automatic storage of significant thin film register contents takes place, as described under the Interrupt System. (See IRR Instruction.)

In the control mode, the computer halts, and all sequencing ceases; bits 7-12 of the operator syllable may be used for identification of the stop.

Ex: HLT 77

When the computer halts at this instruction in the control mode, 0177_8 will be indicated in the COMMAND register display on the computer control panel.

NOP - NO OPERATION - 00_8

NOP

NOP continues the normal sequencing without further fuss.

The NOP instruction is used to provide an instruction syllable of all zeros. This will be done automatically by the assembler when it is necessary that the next instruction in sequence be left justified in the next memory location and the present memory location contains unused syllables.

RPT - REPEAT - 10_8

RPT Rc,Ri,B

RPT causes the instruction at the location specified by A_3 to be executed the number of times specified by A_1. After each execution, the effective addresses of the repeated instruction will be incremented by the corresponding three values specified by A_2 to produce new effective addresses for the next execution.

A_2 Structure:

1 4	5 8	9 1 2
1st Add. Incr.	2nd Add Incr.	3rd Add. Incr.

The increments of A_2 are applied to the 1st, 2nd, and 3rd Physical Address Syllable following the repeated instruction's Operator Syllables. This does NOT necessarily correspond to A_1-A_3 of the instruction description, since use of the stack by the repeated instruction could result, for instance, in the A_3 syllable being the 1st Physical Syllable after the Operator Syllable because A_1 and A_2 are stack references. The 1st address increment refers to the first Physical Syllable in A_3 and the numbering continues in that order.

The number of times an instruction is to be repeated may vary from 0 through 4095. If the value is zero, the instruction will not be operated at all.

The Operator Syllable of the instruction to be repeated must be the leftmost syllable at the location specified by A_3.

RPR (Repeat Program Register - 044_8-047_8) contains an image of the repeated instruction during the repetition process; 047_8 contains the Operator Syllable, 046_8 the 1st Physical Syllable, 045_8 and 2nd Physical Syllable and 044_8 the 3rd Physical Syllable.

RCR (Repeat Count Register - 120_8) contains the counter (initially the contents of A_1) for the number of repetitions.

The location of the instruction to be repeated is calculated relative to the BPR (Base Program Register - 054_8) rather than the BAR (Base Address Register - 055_8) --- that is, A_3 is a BRANCH rather than a DATA reference.

RPT may use indexing but not indirect addressing for A_3. The instruction being repeated may use indirect addressing, but no indexing is allowed.

The repetition process may be terminated in two ways--when RCR reaches zero, or when the instruction being repeated executes a branch. In case of the RCR reaching zero, the next instruction to be operated will be the one after the RPT instruction. In the second case, RCR will have been decremented the number of times the instruction has been repeated, including the branch, and the three address syllables of RPR will contain the EFFECTIVE address values for the final repetition.

There are certain instructions which because of their nature cannot be executed by means of the RPT instruction. These are: RPT (Repeat) itself, SJR (Subroutine Jump), SRR (Subroutine Return), IRR (Interrupt Return), UCT (Unconditional Transfer), and XLC (Index, Limit-Compare).

In coding the A_2 syllable of the repeat instruction a blank must be inserted between the values of each of the three physical syllables being incremented. The values in integer form for the A_1 and A_2 syllables in the RPT instruction are in decimal notation. Both syllables may, of course, be modified by indexing.

```
Ex:     RPT     8,Ø Ø Ø,ODD

        ODD = BRB   T1,T1,BRANCH
          T1 = 0000000000007550(8)
```

After First Execution: T1 = 0000000000003764_8
RCR = 0007_8

After Second Execution: T1 = 0000000000001772_8
RCR = 0006_8

After Third Execution: T1 = 0000000000000775_8
RCR = 0005_8

After Fourth Execution: T1 = 4000000000000376_8
RCR = 0004_8

The next instruction executed is taken from location BRNCH. If RCR had reached 0000 without a branch (e.g., T1 = 0000000000070000_8 at the start), the next instruction to be executed would have been the next instruction in sequence following the RPT instruction.

TRS - TRANSMIT - 35_8

TRS (M),(M)

TRS takes the contents of the location specified by A_1 and places them in the location specified by A_2.

Ex: TRS DATA,N

Before - Data = 0000460137020000_8

After - Data = 0000460137020000_8

Top of Stack = 0000460137020000_8

Caution: With the exception of the 4 stack registers, the TRS instruction cannot be used to access Thin Film registers. Of course, as in the example above, when accessing the stack, H or N are the symbols required in coding.

TRM - TRANSMIT MODIFIED - 34_8

TRM (M),V_t,(M)

TRM takes the contents of the location specified by A_1, modifies them as specified in A_2, and places the result in the location specified by A_3.

A_2 Structure:

1 ... 9	10	11	12
x x x x x x x x x	R	M	P

If bit 10 of A_2 is one, the modification will include rounding which proceeds as follows:

If bit 2 of TFC (Thin Film C - 124_8 - 127_8) is a one, a fixed point one (0000000000000001_8) is algebraically added to the image of the contents of A_1. If overflow occurs, POV (Program OVerflow) will be set.

If bits 11 - 12 of A_2 are 10, the modification will include setting the sign to minus.

If bits 11 - 12 of A_2 are 01, the modification will include setting the sign to plus.

If bits 11 - 12 of A_2 are 11, the modification will include changing the sign.

The contents of A_1 syllable are not changed by the TRM instruction unless the A_3 syllable denotes the same location as the A_1 syllable.

In coding a TRM instruction where both rounding and setting of the sign are combined, a blank must separate the rounding symbol from the sign modifying symbol. (When both are used, the rounding symbol must come first.) Symbols for modification in coding are:

- \- set the sign bit in A_3 to minus.
- \+ set the sign bit in A_3 to plus.

C change the sign bit of A_1 and insert the changed sign in A_3.

R modification includes rounding.

Ex: TRM H,R -,TEMP

Before - Top of stack = 0034567123456667_8
TFC = 2000000000000000_8

After - Temp = 4034567123456670_8
Top of stack = unchanged

UCT - UNCONDITIONAL TRANSFER - 22_8

UCT B

UCT will transfer control unconditionally to the first syllable at the location specified by A_1, which must be a valid operator syllable.

Ex: UCT NTRY

NTRY = BAD ADD1,AUG1,SUM1

COMPARISON INSTRUCTIONS

This group will only consider those instructions which compare 48-bit words to determine if conditions for branching have been met. Comparison of characters or bytes within a word will be considered in the Field Instructions. Comparison of index to limit registers has been outlined previously under the XLC instruction.

There are two kinds of comparison instructions treated in this group: the first kind deals with an alphanumeric comparison of the entire 48-bit word; the second kind is concerned with an algebraic comparison of the 48-bit word in which the first bit is treated as a sign bit. In an alphanumeric comparison the first bit is treated as a value bit. Given the following example, an alphanumeric comparison would yield different results from an algebraic comparison.

$A_1 = 4000777777777777_8$
$A_2 = 0000777777777777_8$

Compared algebraically, A_2, with a positive sign bit, is greater than A_1. However, in an alphanumeric comparison, with the sign bit treated as a value, A_1 is greater than A_2. In a comparison of equality, however, the same result occurs whether the comparison is algebraic or alphanumeric: A_1 is not equal to A_2.

ACE - ALPHANUMERIC COMPARE EQUAL - 72_8

ACE (M),(M),B

ACE compares the contents of the location specified by A_1 as a 48-bit unsigned value to the contents of the location specified by A_2. If the A_1 contents are equal to the A_2 contents, control branches to the first syllable of the location specified by A_3. Otherwise, control continues to the next instruction in sequence. The location specified by A_3 must contain left justified a valid operator syllable.

Ex: ACE ITEM,MATCH,BRNCH

ITEM = 1234567012345670_8
MATCH = 1234567012345670_8

The next location to be executed will be taken from location BRNCH.

ACG - ALPHANUMERIC COMPARE GREATER - 71_8

ACG (M),(M),B

ACG compares the contents of the location specified by A_1 as a 48-bit unsigned value to the contents of the location specified by A_2. If the A_1 contents are greater than the A_2 contents, control branches to the first syllable of the location specified by A_3. Otherwise, control continues to the next instruction in sequence. The location specified by A_3 must contain left justified a valid operator syllable.

Ex: ACG ITEM,N,BRNCH

ITEM = 3777777777777777_8
Top of Stack = 4777777777777777_8

The next instruction to be executed will be the one following the ACG instruction in sequence.

ACL - ALPHANUMERIC COMPARE LESS - 70_8

ACL (M),(M),B

ACL compares the contents of the location specified by A_1 as a 48-bit unsigned value to the contents of the location specified by A_2. If the A_1 contents are less than the A_2 contents, control branches to the first syllable of the location specified by A_3. Otherwise, control continues to the next instruction in sequence. The location specified by A_3 must contain a valid operator syllable left justified.

Ex: ACL ITEM,SMALL,BRNCH

ITEM = 3777777777777777_8
SMALL = 4777777777777777_8

The next instruction to be executed will be taken from location BRNCH.

CEQ - COMPARE EQUAL - 76_8

CEQ (M),(M),B

CEQ takes the contents of the location specified by A_1 and the contents of the location specified by A_2 and performs an algebraic, 48-bit signed comparison between the two. If the contents of A_1 are equal to the contents of A_2, control will transfer to the first syllable of the location specified by A_3. Otherwise, control continues to the next instruction in sequence. The location specified by A_3 must contain a valid operator syllable left justified.

Ex: CEQ ITEM,MATCH,BRNCH

ITEM = 1234567012345670_8
MATCH = 1234567012345670_8

The next instruction to be executed will be taken from location BRNCH.

CGR - COMPARE GREATER - 75_8

CGR (M),(M),B

CGR takes the contents of the location specified by A_1 and the contents of the location specified by A_2 and performs an algebraic, 48-bit signed comparison between the two. If the contents of A_1 are greater than the contents of A_2, control will transfer to the first syllable of the location specified by A_3. Otherwise, control continues to the next instruction in sequence. The location specified by A_3 must contain a valid operator syllable left justified.

Ex: CGR ITEM,N,BRNCH

ITEM = 4777777777777777_8
Top of stack = 3777777777777777_8

The next instruction to be executed will be the one following the CGR instruction in sequence. Note that the 4 in ITEM denotes a minus sign bit and two zero bits in the numeric value of the word.

CLS - COMPARE LESS - 74_8

CLS (M),(M),B

CLS takes the contents of the location specified by A_1 and the contents of the location specified by A_2 and performs an algebraic, 48-bit signed comparison between the two. If the contents of A_1 are less than the contents of A_2, control will transfer to the first syllable of the location specified by A_3. Otherwise, control will continue to the next instruction in sequence. The location specified by A_3 must contain a valid operator syllable left justified.

Ex: CLS ITEM,SMALL,BRNCH

ITEM = 3777777777777777_8
SMALL = 4777777777777777_8

The next instruction to be executed will be the one following in sequence the CLS instruction. Note that the 3 in ITEM denotes a plus sign bit and two one bits in the numeric value of the word.

The logical operations that can be performed on the AN/GSA-51A are AND, OR, Exclusive OR, and Complement. These operations can be performed on a full 48-bit word or upon a partial word (character or byte) which is field defined. The group considered here concerns only logical operations upon the full 48-bit word. Field logical instructions will be treated later, in the section on Field Instructions.

One important function of logical instructions is to isolate or modify particular bits in a 48-bit word. Many data words are so designed in the BUIC system as to accommodate more than one piece of information. The use of arithmetic instructions for modifying parts of a word is limited, however, since arithmetic instructions operate on the entire word and treat the first bit as a sign bit. In order to modify parts of words only, it becomes necessary to isolate the part from the rest of the word, perform any necessary changes, and reinsert the changed bits in their appropriate position. The logical instructions described here facilitate these operations.

LAN - LOGICAL AND - 56_8

LAN (M),(M),(M)

LAN takes the contents of the location specified by A_1 and the contents of the location specified by A_2, performs a logical AND operation on all 48-bit positions, and places the result in the location specified by A_3. The AND operation compares bits of the same bit position and produces a zero when either or both are zeros and a one when they are both ones. The contents of A_1 and A_2 remain unchanged unless A_3 denotes one of these locations.

Ex: (1) LAN DATA,MASK1,N

BEFORE - DATA = 4321765013476034_8
MASK1 = 0000000000000007_8

AFTER - Top of stack = 0000000000000004_8

Note that this operation has isolated the three least significant bits in DATA and placed them in the A_3 location.

Ex: (2) LAN DATA,CMSK1,DATA

BEFORE - DATA = 4321765013476034_8
CMSK1 = 7777777777777770_8

AFTER - DATA = 4321765013476030_8

Note that this operation has cleared (set to zero) the three least significant bits of DATA.

LOR - LOGICAL OR - 55_8

LOR (M),(M),(M)

LOR takes the contents of the location specified by A_1 and the contents of the location specified by A_2, performs a logical OR operation on all 48-bit positions, and places the result in the location specified by A_3. The OR operation compares bits of the same bit position and produces a zero where they are both zero and a one when either or both are ones. The contents of A_1 and A_2 remain unchanged unless A_3 denotes one of these locations.

Ex: LOR DATA,SEVEN,DATA

BEFORE - DATA = 4777666655550432_8

SEVEN = 0000000000007000_8

AFTER - DATA = 4777666655557432_8

Note that this operation has set three bits beginning with bit position 37 to one. A binary add instruction with DATA and SEVEN would not accomplish the same result since the computer would treat DATA as a negative value and SEVEN as a positive value.

LXR - LOGICAL EXCLUSIVE OR - 54_8

LXR (M),(M),(M)

LXR takes the contents of the location specified by A_1 and the contents of the location specified by A_2, performs a logical EXCLUSIVE OR operation on all 48-bit positions, and places the result in the location specified by A_3. The EXCLUSIVE OR operation compares bits of the same bit position and produces a zero when they are identical (either both ones or both zeros) and a one when they are different. The contents of A_1 and A_2 remain unchanged unless A_3 denotes one of these locations.

Ex: LXR DATA,CHANG,N

BEFORE - DATA = 7373737373737373_8

CHANGE = 6565656565656565_8

AFTER - Top of Stack = 1616161616161616_8

LCM - LOGICAL COMPLEMENT - 24_8

LCM (M),(M)

LCM takes the contents of the location specified by A_1, performs a logical COMPLEMENT operation on all 48-bit positions, and places the result in the location specified by A_2. The COMPLEMENT operation reverses the settings of bits, so that ones become zeros and zeros become ones. The contents of A_1 remains unchanged unless A_2 denotes the A_1 location.

Ex: LCM DATA,COMP

BEFORE - DATA = 0102030405060700

AFTER - COMP = 7675747372717077

CYCLING AND SHIFTING INSTRUCTIONS

This group of instructions provides for the shifting of the contents of the location specified in the A_1 syllable and in some cases in combination with the TFC register. A total of 16 different types of shifting are possible, depending upon the following four alternatives: (1) either right or left; (2) either single or double; (3) either logical or arithmetic; (4) either circular (cycling) or end-off.

Right or left indicates the desired direction in which the shifting is to take place. In a double shift, the contents of the TFC register are shifted as well as the contents of the location specified by A_1. In a single shift only the contents of the A_1 location are shifted. Logical shifts are performed on bits 1 through 48 of the contents of the location specified in the A_1 syllable and, if a double shift is specified, on bits 1 through 48 of the TFC register also. On the other hand, arithmetic shifts are performed on bits 2 through 48 of the contents of the location specified by A_1 and, if a double shift is indicated, on bits 2 through 48 of the TFC register also. In cycling shifts, bits are shifted end-around in the direction specified. If cycling is not specified, the bits are shifted end-off and zeros are inserted into the opposite end of the data word.

Although the octal code for the shifting instructions (35_8) remains the same irrespective of the alternatives above, the mnemonic pseudo code varies as to the type of alternatives selected. This variation of mnemonic code affects the structure of the A_2 syllable, which denotes the kind and amount of shifting requested. The A_2 syllable of this instruction is a special syllable and is composed as follows:

A_2 Instruction Syllable Structure

1 2	3	4	5	6	7 1 2
Not Used	A	B	C	D	Amount of Shift

A = 0 for single
1 for double

B = 0 for left shift
1 for right shift

C = 0 for arithmetic
1 for logical

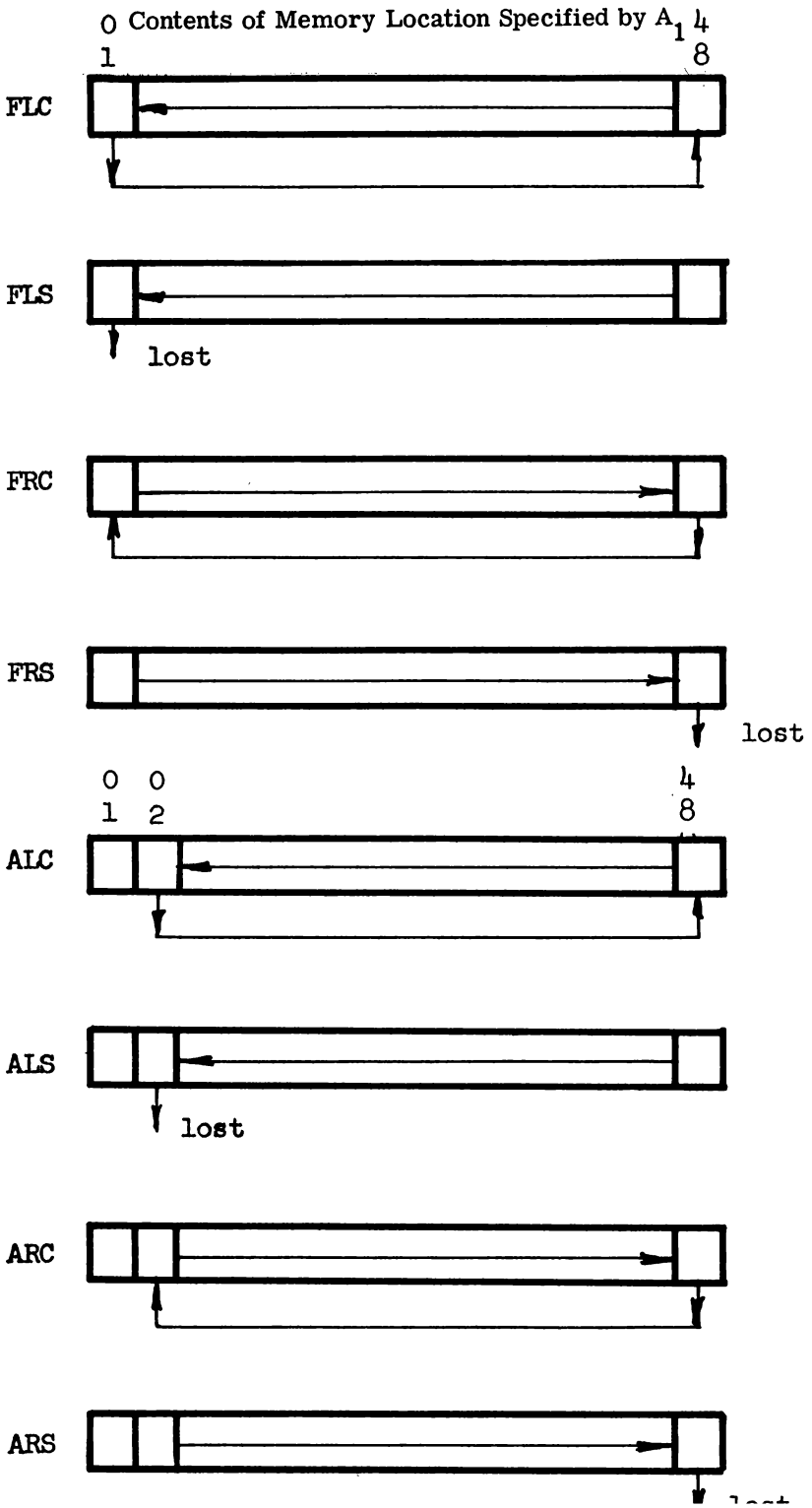
D = 0 for cycle
1 for shift (end-off)

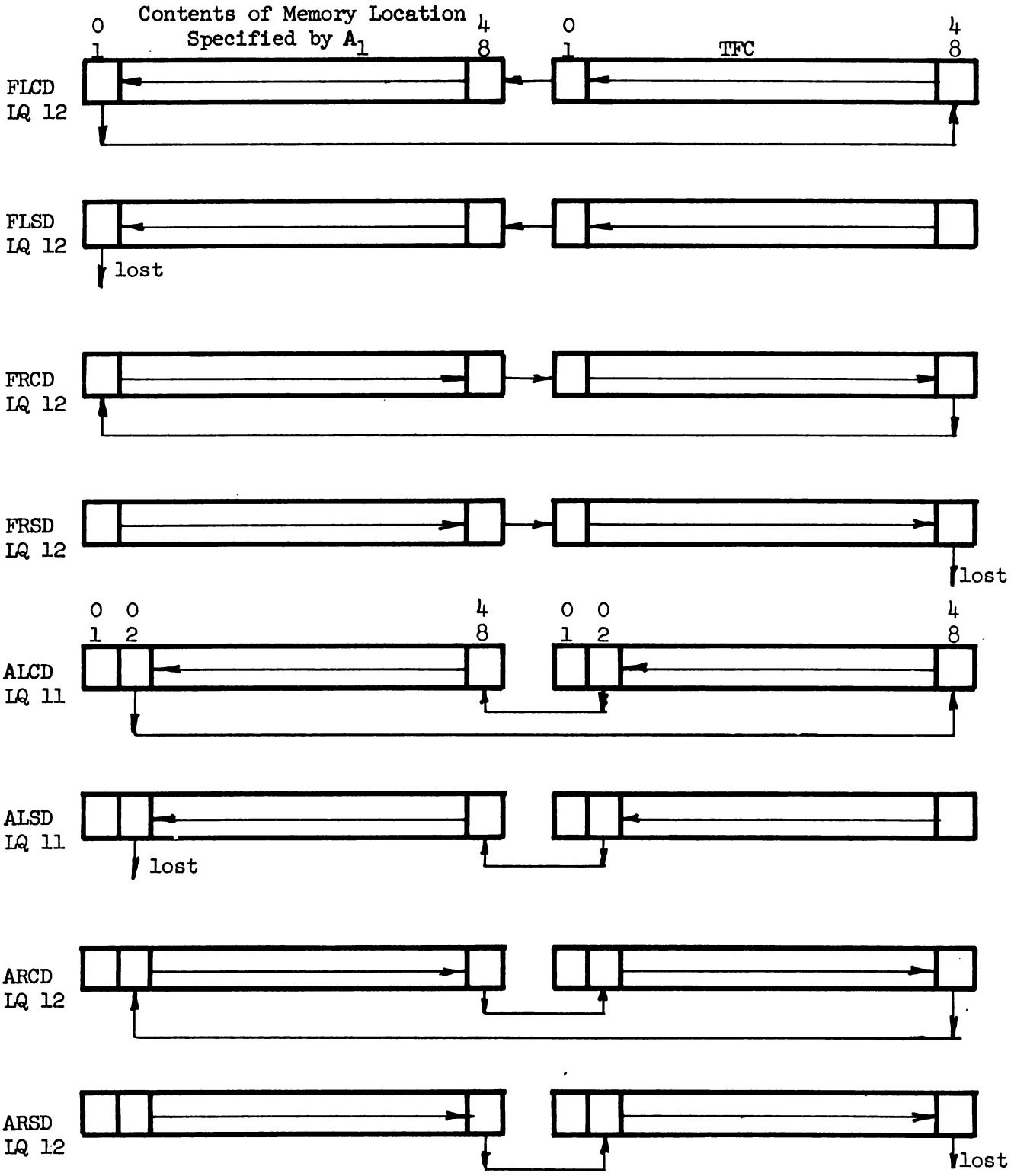
In coding using the pseudo instruction, only the number of bits to be shifted is inserted on the coding sheet for the A_2 syllable. The number of bits is a decimal number.

The following diagrams illustrate the shifting as specified by the appropriate mnemonic pseudo code. The first letter "F" denotes a logical (full) shift; The "A" signifies an arithmetic shift. "L" and "R" denote left and right shift, respectively. "S" and "C" signify an end-off shift and end-around cycle, respectively. If the last letter is a "D", a double shift is indicated; otherwise a single shift will be performed. A more detailed explanation with examples follows the diagrams.

The amount of bits to be shifted for the single shift instruction cannot exceed 63_{10}, as that is the largest number that can be inserted in the A_2 syllable. However, there are fixed limits for the double shift instructions as defined in the diagrams below. The limit of number of bits to be shifted for each type of instruction is listed below each relevant pseudo code.

SHIFT (SHF) PSEUDO CODES



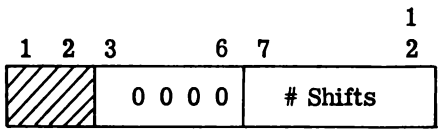

Contents of Memory Location Specified by A_1
TFC
FLCD
IQ 12
FLSD
IQ 12
lost
FRCD
IQ 12
FRSD
IQ 12
lost
ALCD
IQ 11
ALSD
IQ 11
lost
ARCD
IQ 12
ARSD
IQ 12
lost

ALC - ARITHMETIC LEFT CYCLE - 36_8

ALC (M),S,(M)

ALC is a variation of the Shift Op Code, 36_8, which takes the contents of the location specified by A_1, rotates bits 2 - 48 left the number of bit positions specified in bits 7 - 12 of A_2 and places the result in the location specified by A_3. Bits which pass thru bit position 2 are reinserted at bit position 48. The sign bit (bit 1) is unaffected by the execution of this instruction.

A_2 Structure:



3-6 contain the Variation Code.
7-12 contain the number of bit positions to be shifted.

Ex: ALC VALUE,42,PACK

BEFORE - VALUE = 4000000000000036_8

AFTER - PACK = 7600000000000000_8

VALUE = 4000000000000036_8

This instruction packs a signed, fixed-point number into the 6 most significant bits.

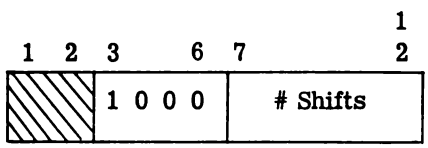
ALCD - ARITHMETIC LEFT CYCLE DOUBLE - 36_8

ALCD (M),S,(M)

ALCD is the variation of the Shift Op Code, 36_8, which takes the contents of the location specified by A_1 in combination with the contents of TFC (Thin Film C - 124_8 - 127_8) as a low order extension. Bits 2-48 of the A_1 image and bits 2-48 of TFC are rotated left the number of bit positions specified in bits 7-12 of A_2 and the 48 high order bits of the resulting double image are placed in the location specified by A_3. Bits which pass thru bit position 2 of the A_1 image are reinserted at bit position 48 of TFC, and those which pass thru bit position 2 of TFC continue into bit position 48 of the A_1 image. Neither sign bit is affected.

The number of bit positions to be shifted must not exceed 11.

A_2 Structure:



3-6 contain the Variation Code.
7-12 contain the number of bit positions to be shifted.

Ex: ALCD VALUE,6,PACK

BEFORE - VALUE = 4012345670123456_8

TFC = 7654321001234567_8

AFTER - TFC = 5432100123456700_8

PACK = 5234567012345675_8

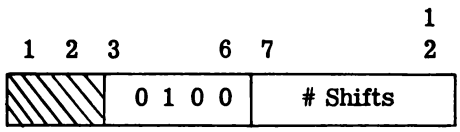
VALUE = 4012345670123456_8

ARC - ARITHMETIC RIGHT CYCLE - 36_8

ARC (M),S,(M)

ARC is a variation of the Shift Op Code, 36_8, which takes the contents of the location specified by A_1, rotates bits 2-48 right the number of bit positions specified in bits 7-12 of A_2 and places the result in the location specified by A_3. Bits which pass thru bit position 48 are reinserted at bit position 2. The sign bit (bit 1) is unaffected by the execution of this instruction.

A_2 Structure:



Ex: ARC PACK,42,VALUE

BEFORE - PACK = 7600000000000000_8

AFTER - VALUE = 4000000000000036_8

PACK = 7600000000000000_8

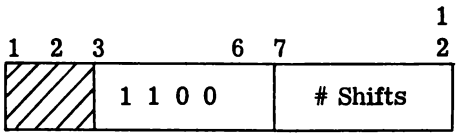
ARCD - ARITHMETIC RIGHT CYCLE DOUBLE - 36_8

ARCD (M),S,(M)

ARCD is the variation of the Shift Op Code, 36_8, which takes the contents of the location specified by A_1 in combination with the contents of TFC (Thin Film C - 124_8 - 127_8) as a low order extension. Bits 2-48 of the A_1 image and bits 2-48 of TFC are rotated right the number of bit positions specified in bits 7-12 of A_2, and the 48 high order bits of the resulting double image are placed in the location specified by A_3. Bits which pass thru bit position 48 of the A_1 image continue into bit position 2 of TFC, and those which pass thru bit position 48 of TFC are reinserted at bit position 2 of the A_1 image. Neither sign bit is affected.

The number of bit positions to be shifted must not exceed 12_{10}.

A_2 Structure:



3-6 contain the Variation Code.
7-12 contain the number of bit positions to be shifted.

Ex: ARCD PACK,12,VALUE

BEFORE - PACK = 5675306420024713_8

TFC = 6700247135602471_8

AFTER - TFC = 6345670024713560_8

VALUE = 5234567530642002_8

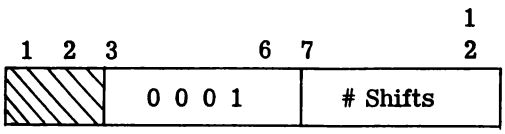
PACK = 5675306420024713_8

ALS - ARITHMETIC LEFT SHIFT - 36_8

ALS (M),S,(M)

ALS is a variation of the Shift Op Code, 36_8, which takes the contents of the location specified by A_1, shifts bits 2-48 left the number of bit positions specified in bits 7-12 of A_2 and places the result in the location specified by A_3. Bits which pass thru bit position 2 are lost and zeros are inserted at bit position 48. The sign bit (bit 1) is unaffected by the execution of this instruction.

A_2 Structure:



3-6 contain the Variation Code.
7-12 contain the number of bit positions to be shifted.

Ex: ALS VALUE,42,PACK

BEFORE - VALUE = 7654321001234567_8

AFTER - PACK = 6700000000000000_8

VALUE = 7654321001234567_8

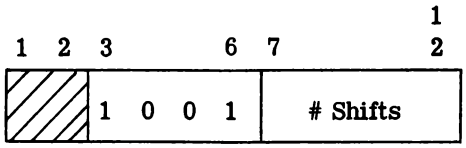
ALSD - ARITHMETIC LEFT SHIFT DOUBLE - 36_8

ALSD (M),S,(M)

ALSD is the variation of the Shift Op Code, 36_8, which takes the contents of the location specified by A_1 in combination with the contents of TFC (Thin Film C - 124_8 - 127_8) as a low order extension. Bits 2-48 of the A_1 image and bits 2-48 of TFC are shifted left the number of bit positions specified by A_3. Bits which pass thru bit position 2 of the A_1 image are lost, those which pass thru bit position 2 of TFC continue into bit position 48 of the A_1 image, and zeros are inserted at bit position 48 of TFC. Neither sign bit is affected.

The number of bit positions to be shifted must not exceed 11_{10}.

A_2 Structure:



3-6 contain the Variation Code.
7-12 contain the number of bit positions to be shifted.

Ex: ALSD VALUE,6,PACK

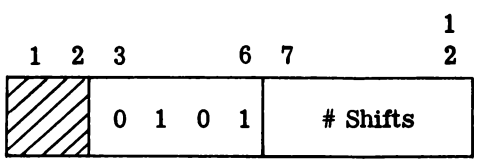
BEFORE - VALUE = 0123456776543210_8
TFC = 7766554433221100_8
AFTER - TFC = 6655443322110000_8
PACK = 2345677654321077_8
VALUE = 0123456776543210_8

ARS - ARITHMETIC RIGHT SHIFT - 36_8

ARS (M),S,(M)

ARS is a variation of the Shift Op Code, 36_8, which takes the contents of the location specified by A_1, shifts bits 2-48 right the number of bit positions specified in bits 7-12 of A_2 and places the result in the location specified by A_3. Bits which pass thru bit position 48 are lost and zeros are inserted at bit position 2. The sign bit (bit 1) is unaffected by the execution of this instruction.

A_2 Structure:



3-6 contain the Variation Code.
7-12 contain the number of bit positions to be shifted.

Ex: ARS PACK,42,VALUE

BEFORE - PACK = 7700000000000000_8

AFTER - VALUE = 4000000000000037_8

PACK = 7700000000000000_8

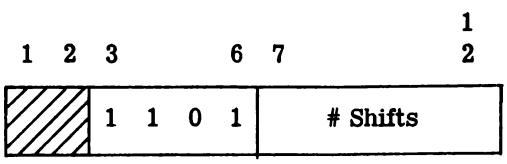
ARSD - ARITHMETIC RIGHT SHIFT DOUBLE - 36_8

ARSD (M),S,(M)

ARSD is a variation of the Shift Op Code, 36_8, which takes the contents of the location specified by A_1 in combination with the contents of TFC (Thin Film C - 124_8 - 127_8) as a low order extension. Bits 2-48 of the A_1 image and bits 2-48 of TFC are shifted right the number of bit positions specified in bits 7-12 of A_2, and the 48 high order bits of the resulting double image are placed in the location specified by A_3. Bits which pass thru bit position 48 of the A_1 image continue into bit position 2 of TFC, those which pass thru bit position 48 of TFC are lost and zeros are inserted at bit position 2 of the A_1 image. Neither sign bit is affected.

The number of bit positions to be shifted must not exceed 12_{10}.

A_2 Structure:



3-6 contain the Variation Code.
7-12 contain the number of bit positions to be shifted.

Ex: ARSD PACK,6,VALUE

BEFORE - PACK = 1077553310664422_8

TFC = 4000000000000000_8

AFTER - TFC = 5100000000000000_8

VALUE = 0010775533106644_8

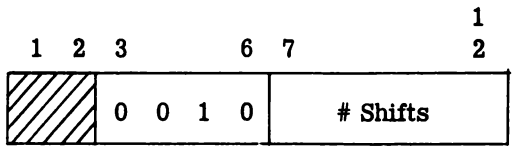
PACK = 1077553310664422_8

FLC - FULL LEFT CYCLE - 36_8

FLC (M),S,(M)

FLC is a variation of the Shift Op Code, 36_8, which takes the contents of the location specified by A_1, rotates bits 1-48 left the number of bit positions specified in bits 7-12 of A_2 and places the result in the location specified by A_3. Bits which pass thru bit position 1 are reinserted at bit position 48. The sign bit (bit 1) is included in the execution of this instruction.

A_2 Structure:



3-6 contain the Variation Code.
7-12 contain the number of bit positions to be shifted.

Ex: FLC BITS,42,NEW

BEFORE - BITS = 0011223344556677_8

AFTER - NEW = 7700112233445566_8

BITS = 0011223344556677_8

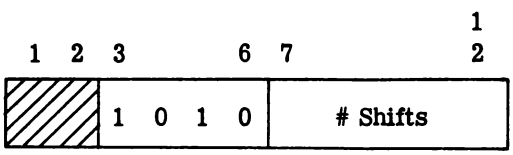
FLCD - FULL LEFT CYCLE DOUBLE - 36_8

FLCD (M),S,(M)

FLCD is a variation of the Shift Op Code, 36_8, which takes the contents of the location specified by A_1 in combination with the contents of TFC (Thin Film C - 124_8-127_8) as a low order extension. Bits 1-48 of the A_1 image and bits 1-48 of TFC are rotated left the number of bit positions specified in bits 7-12 of A_2, and the 48 high order bits of the resulting double image are placed in the location specified by A_3. Bits which pass through bit position 1 of TFC continue into bit position 48 of the A_1 image, and those which pass through bit position 1 of the A_1 image are reinserted at bit position 48 of TFC. Both sign bits are included in the execution of this instruction.

The number of bit positions to be shifted must not exceed 12_{10}.

A_2 Structure:



3-6 contain the Variation Code.
7-12 contain the number of bit positions to be shifted.

Ex: FLCD BITS,6,FUDGE

BEFORE - BITS = 0123456776543210_8

TFC = 0011223344556677_8

AFTER - TFC = 1122334455667701_8

FUDGE = 2345677654321000_8

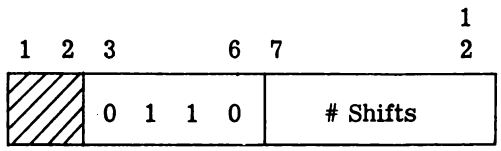
BITS = 0123456776543210_8

FRC - FULL RIGHT CYCLE - 36_8

FRC (M),S,(M)

FRC is a variation of the Shift Op Code, 36_8, which takes the contents of the location specified by A_1, rotates bits 1-48 right the number of bit positions specified in bits 7-12 of A_2 and places the result in the location specified by A_3. Bits which pass thru bit position 48 are reinserted at bit position 1. The sign bit (bit 1) is included in the execution of this instruction.

A_2 Structure:



3-6 contain the Variation Code.
7-12 contain the number of bit positions to be shifted.

Ex: FRC BITS,42,NEW

BEFORE - BITS = 0123765445673210_8

AFTER - NEW = 2376544567321001_8

BITS = 0123765445673210_8

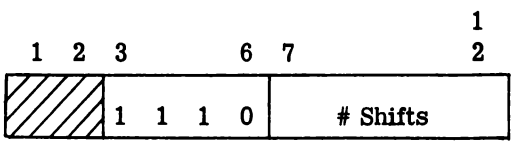
FRCD - FULL RIGHT CYCLE DOUBLE - 36_8

FRCD (M),S,(M)

FRCD is a variation of the Shift Op Code, 36_8, which takes the contents of the location specified by A_1 in combination with the contents of TFC (Thin Film C - 124_8-127_8) as a low order extension. Bits 1-48 of the A_1 image and bits 1-48 of TFC are rotated right the number of bit positions specified in bits 7-12 of A_2, and the 48 high order bits of the resulting double image are placed in the location specified by A_3. Bits which pass thru bit position 48 of the A_1 image continue into bit position 1 of TFC, and those which pass thru bit position 48 of TFC are reinserted at bit position 1 of the A_1 image. Both sign bits are included in the execution of this instruction.

The number of bit positions to be shifted must not exceed 12_{10}.

A_2 Structure:



Ex: FRCD BITS,9,FUDGE

BEFORE - BITS = 1023475675643120_8

TFC = 1001211232234334_8

AFTER - TFC = 1201001211232234_8

FUDGE = 3341023475675643_8

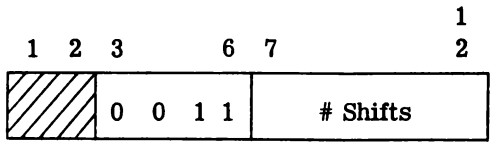
BITS = 1023465675643120_8

FLS - FULL LEFT SHIFT - 36_8

FLS (M),S,(M)

FLS is a variation of the Shift Op Code, 36_8, which takes the contents of the location specified by A_1, shifts bits 1-48 left the number of bit positions specified in bits 7-12 of A_2 and places the result in the location specified by A_3. Bits which pass thru bit position 1 are lost and zeros are inserted at bit position 48. The sign bit (bit 1) is included in the execution of this instruction.

A_2 Structure:



3-6 contain the Variation Code.
7-12 contain the number of bit positions to be shifted.

Ex: FLS BITS,27,FUDGE

BEFORE - BITS = 0011223344556677_8

AFTER - FUDGE = $\overline{4556677000000000_8}$

BITS = 0011223344556677_8

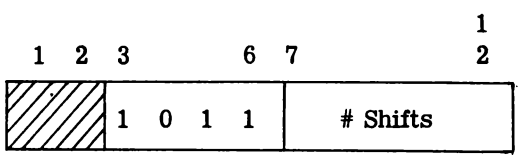
FLSD - FULL LEFT SHIFT DOUBLE - 36_8

FLSD (M),S,(M)

FLSD is a variation of the Shift Op Code, 36_8, which takes the contents of the location specified by A_1 in combination with the contents of TFC (Thin Film C - 124_8-127_8) as a low order extension. Bits 1-48 of the A_1 image and bits 1-48 of TFC are shifted left the number of bit positions specified in bits 7-12 of A_2, and the 48 high order bits of the resulting double image are placed in the location specified by A_3. Bits which pass thru bit position 1 of the A_1 image are lost, those which pass thru bit position 1 of TFC are reinserted at bit position 48 of the A_1 image, and zeros are inserted at bit position 48 of TFC. Both sign bits are included in the execution of this instruction.

The number of bit positions to be shifted must not exceed 12_{10}.

A_2 Structure:



3-6 contain the Variation Code.
7-12 contain the number of bit positions to be shifted.

Ex: FLSD BITS,9,FUDGE

BEFORE - BITS = 7654321001234567_8

TFC = 0011223344556677_8

AFTER - TFC = 1223344556677000_8

FUDGE = 4321001234567001_8

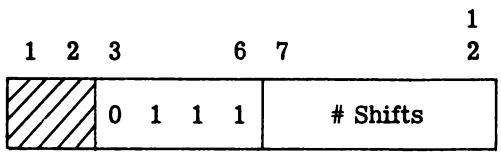
BITS = 7654321001234567_8

FRS - FULL RIGHT SHIFT - 36_8

FRS (M),S,(M)

FRS is a variation of the Shift Op Code, 36_8, which takes the contents of the location specified by A_1, shifts bits 1-48 right the number of bit positions specified in bits 7-12 of A_2, and places the result in the location specified by A_3. Bits which pass thru bit position 48 are lost and zeros are inserted at bit position 1. The sign bit (bit 1) is included in the execution of this instruction.

A_2 Structure:



3-6 contain the Variation Code.
7-12 contain the number of bit positions to be shifted.

Ex: FRS BITS,33,FUDGE

BEFORE - BITS = 1111222233334444_8

AFTER - FUDGE = 0000000000011112_8

BITS = 1111222233334444_8

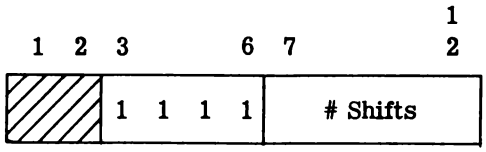
FRSD - FULL RIGHT SHIFT DOUBLE - 36_8

FRSD (M),S,(M)

FRSD is a variation of the Shift Op Code, 36_8, which takes the contents of the location specified by A_1 in combination with the contents of TFC (Thin Film C - 124_8-127_8) as a low order extension. Bits 1-48 of the A_1 image and bits 1-48 of TFC are shifted right the number of bit positions specified in bits 7-12 of A_2, and the 48 high order bits of the resulting double image are placed in the location specified by A_3. Bits which pass thru bit position 48 of the A_1 image continue into bit position 1 of TFC, those which pass thru bit position 48 of TFC are lost, and zeros are inserted at position 1 of the A_1 image. Both sign bits are included in the execution of this instruction.

The number of bit positions to be shifted must not exceed 12_{10}.

A_2 Structure:



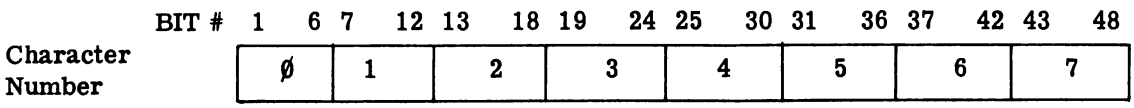
3-6 contain the Variation Code.
7-12 contain the number of bit positions to be shifted.

Ex: FRSD BITS,12,FUDGE

BEFORE - BITS = 7654321001234567_8
TFC = 4455667700112233_8
AFTER - TFC = 4567445566770011_8
FUDGE = 0000765432100123_8
BITS = 7654321001234567_8

FIELD INSTRUCTIONS

Field instructions are provided to manipulate fixed portions of a data word. For the purpose of field instructions, the 48-bit data word is divided into 8 characters or bytes. Each character is numbered from 0 thru 7 as shown below, and contains a set of six adjacent bits. Manipulation of any one or more characters in a word necessarily entails all six bits in the specified character(s). To effectively use field instructions, the position of the data to be accessed must coincide with one or more character portions in the 48-bit word.

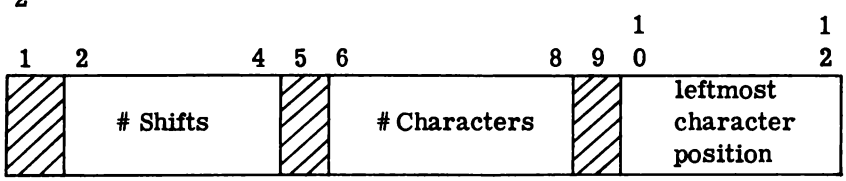


Where the desired data to be used shares a character with other data, the logical instructions rather than the field instructions must be used.

Field instructions encompass a wide variety of manipulation: extraction or insertion of field values, simple arithmetic, comparison and branching, and logical operations.

The A_2 syllable of each field instruction is a special field syllable which is used to indicate the character or characters upon which the instruction operates.

A_2 Field Syllable Structure



Bits 2 thru 4 contain the number of bytes or characters to be shifted. Shifting in field instructions is always right, logical and end-around. Each single shift is always understood as a rotation of the entire character -- all six bits.

Bits 6 thru 8 contain the number of characters, i.e., the length of the field, upon which the instruction operates. If these bits contain zeros, a FULL WORD is indicated.

Bits 10 thru 12 contain the leftmost character position of the field. It should be noted that a field cannot be specified in an end-around fashion. For example, if bits 6 thru 8 contain a 4 and if bits 10 thru 12 contain a 5, the computer will only operate on three characters, those in character positions 5, 6, and 7.

In most of the arithmetic and logical field instructions, the stack is an implied operand. The fields for manipulation or computation are taken from the contents of the stack and the location specified by A_1. After the operation, the result is inserted into the cleared field in an image of A_1. This result is then shifted the amount specified and stored in the A_3 location. In the compare/branch field instructions, the field extracted from the contents specified by A_1 are first shifted as specified and then compared to the contents of the top of the stack. Two of the field instructions do not use the stack as an implied operand. The SAF (Strip and Adjust Field) and the LCF (Logical Complement Field). However, the top of the stack can be specified in all field instructions in the A_1 or A_3 syllables.

In coding the field instructions, blanks are necessary between the three values of the A_2 syllable.

For special field instruction coding, referencing items which are compool sensitive, separate coding definitions are given.

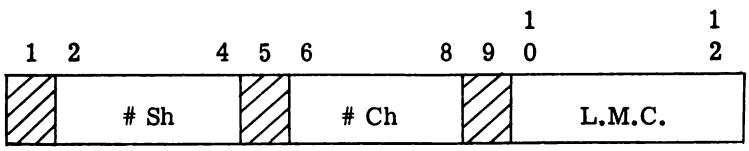
SAF - STRIP AND ADJUST FIELD - 41_8

SAF (M),F,(M)

SAF uses the field specification of A_2 to strip* an image of the contents of the location specified by A_1. The result is right end-around shifted the number of times specified in A_2 and placed in the location specified by A_3.

The contents of the location specified by A_1 will not be changed by the instruction unless referenced by A_3.

A_2 Structure:



2-4 contain the number of shifts (0-7 characters).
6-8 contain the number of characters, i.e., length of the field (0-7 characters) with zero indicating 8 characters or a full word.
10-12 contain the leftmost character position of the field (0-7 position).

*STRIPPING - is the process applied to a word image by which all bits lying within the field area specified by A_2 retain their original value and all others are set to zero.

Ex: SAF SET,6 2 3,PART

BEFORE - SET = 2256321256432144_8

AFTER - SET = 2256321256432144_8

AFTER - PART = 0012560000000000_8

INTERMEDIATE IMAGE OF A_1

After stripping 0000001256000000_8

COMPOOL CODING:

SAF A_1, N ITEM, A_3

Strip an amount of bytes of length "ITEM," and whose least significant byte position is "N". Adjust its position so that the least significant byte position corresponds to the least significant byte position of ITEM.

SAF A_1, ITEM N, A_3

Strip an amount of bytes of length "ITEM," and whose least significant byte position is the least significant byte position of ITEM. Adjust its position so that its least significant byte position corresponds to byte position N.

SAF A_1, ITEM 1 ITEM 2, A_3

Strip an amount of bytes of length "ITEM 1", and whose least significant byte position is the least significant byte position of ITEM 1. Adjust its position so that its least significant byte position corresponds to the least significant byte position of ITEM 2.

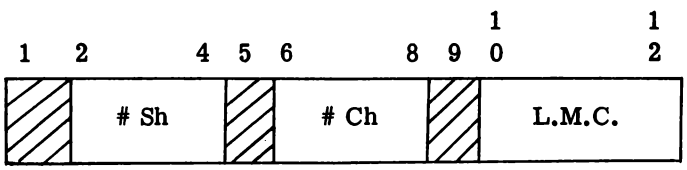
AIF - ADJUST AND INSERT FIELD - 40_8

AIF (M),F,(M)

AIF using an image of the contents of the location specified by A_1, it clears that area of the image as defined by the field specification of A_2; rotates an image of the contents of the top of the stack right end-around the number of character positions specified by A_2; ORs the two 48-bit images together and places the result in the location specified by A_3.

Neither the position or contents of the stack, nor the contents of the location specified by A_1 will be changed by AIF unless referenced by A_3.

A_2 Structure:



2-4 contain the number of shifts (0-7 characters).
6-8 contains the number of characters, i.e., length of the field (0-7 characters) with zero indicating 8 characters or a full word.
10-12 contain the leftmost character position of the field (0-7 position).

Ex: AIF BYTES,7 1 1,HOLES

BEFORE - BYTES = $77\underline{6}6554433221100_8$

BEFORE - Top of stack = 0000570000000000_8

AFTER - HOLES = $77\underline{5}7554433221100_8$

AFTER - Top of stack = 0000570000000000_8

INTERMEDIATE IMAGES OF A_1

After clearing 7700554433221100_8

After stack rotation 0057000000000000_8

Note that if any other fields in the stack contained non-zero values other than those specified above, these values would also be ORed into HOLES, although not in character positions specified by the A_2 syllable.

COMPOOL CODING:

AIF A_1,ITEM N,A_3

Clear an amount of bytes of length "ITEM," and whose least significant byte position is "N". Adjust the top of the stack so that the least significant byte positions of ITEM corresponds to byte position N.

AIF A_1,N ITEM, A_3

Clear an amount of bytes of length "ITEM", and whose least significant byte position is the least significant byte position of ITEM. Adjust the top of the stack so that byte position N corresponds to the least significant byte position of ITEM.

AIF A_1,ITEM 1 ITEM 2,A_3

Clear an amount of bytes of length "ITEM 2", and whose least significant byte position is the least significant byte position of ITEM 2. Adjust the top of the stack so that the least significant byte position of ITEM 1 corresponds to the least significant byte position of ITEM 2.

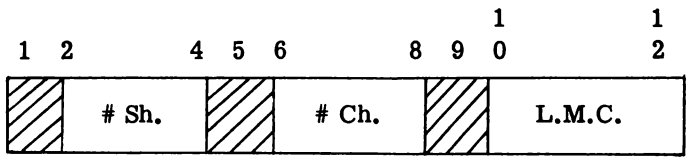
CEF - COMPARE EQUAL FIELD - 52_8

CEF (M),F,B

CEF uses the field specification of A_2 to strip an image of the contents of the location specified by A_1, and shifts the results right end-around the number of character positions specified in A_2. A logical, 48-bit unsigned comparison is made between the rotated image and the contents of the top of the stack. If the image is equal to the stack, control will transfer to the first syllable of the location specified by A_3, otherwise control continues to the next instruction in sequence.

Neither the position or contents of the stack, nor the contents of the location specified by A_1 will be changed by CEF unless referenced by A_3.

A_2 Structure:



2-4 contain the number of shifts (0-7 characters).
6-8 contain the number of characters, i.e., length of the field (0-7 characters) with zero indicating 8 characters or a full word.
10-12 contain the leftmost character position of the field (0-7 position).

Ex: CEF WORD+X9,4 1 5,MATCH

BEFORE - Top of stack = 0021000000000000_8
WORD+X9 = 3266224413210600_8
AFTER = Program branches to instruction contained in MATCH.

INTERMEDIATE IMAGES OF A_1

After stripping = 0000000000210000_8
After rotation = 0021000000000000_8

Note that, if the top of the stack contained non-zero values in other field locations, the result would not be to branch, since the comparison is made with the entire 48-bit word.

COMPOOL CODING:

CEF A_1,N ITEM,A_3

Strip an amount of bytes of length "ITEM", and whose least significant byte position is byte position N. Adjust its position so that byte position N corresponds to the least significant byte position of ITEM.

CEF A_1,ITEM 1 ITEM 2,A_3

Strip an amount of bytes of length "ITEM", and whose least significant byte position is the least significant byte position of ITEM 1. Adjust its position so that the least significant byte position of ITEM 1 corresponds to the least significant byte position of ITEM 2.

CEF A_1,ITEM N,A_3

Strip an amount of bytes of length "ITEM", and whose least significant byte position is the least significant byte position of ITEM. Adjust its position so that the least significant byte position of ITEM corresponds to byte position N.

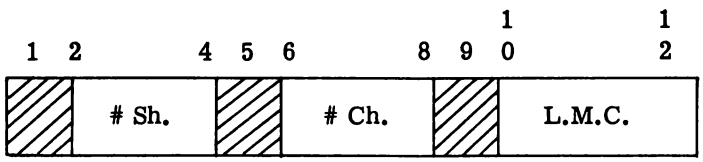
CGF - COMPARE GREATER FIELD - 51_8

CGF (M),F,B

CGF uses the field specification of A_2 to strip an image of the contents of the location specified by A_1, and shifts the results right end-around the number of character positions specified in A_2. A logical, 48-bit, unsigned comparison is made between the rotated image and the contents of the top of the stack. If the image is greater than the stack, control will transfer to the first syllable of the location specified by A_2, otherwise control will continue to the next instruction in sequence.

Neither the position or contents of the stack, nor the contents of the location specified by A_1 will be changed by CGF unless referenced by A_3.

A_2 Structure:



2-4 contain the number of shifts (0-7 characters).
6-8 contain the number of characters, i.e., length of the field (0-7 characters) with zero indicating 8 characters or a full word.
10-12 contain the leftmost character position of the field (0-7 position).

Ex: CGF DATA+X5,5 2 4,MOST

BEFORE - Top of stack = 0077776000000000_8
DATA+X5 = 2165265377752274_8

AFTER - program will continue to the next instruction in sequence.

INTERMEDIATE IMAGES OF A_1

After stripping = 0000000077750000_8
After rotation = 0077750000000000_8

Note that, if the stack contained non-zero values in the other field locations, the result might be changed, since the comparison is made with the entire 48-bit word.

COMPOOL CODING: Handled in same manner as CEF.

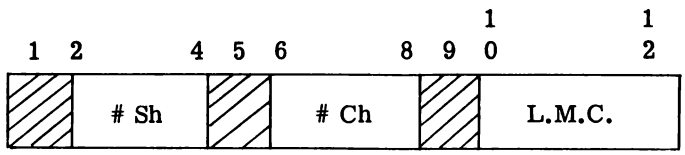
CLF - COMPARE LESS FIELD - 50_8

CLF (M),F,B

CLF uses the field specification of A_2 to strip an image of the contents of the location specified by A_1, and shifts the result right end-around the number of character positions specified in A_2. A logical, 48-bit unsigned comparison is made between the rotated image and the contents of the top of the stack; if the image is less than the stack, control will transfer to the first syllable of the location specified by A_3; otherwise control will continue to the next instruction in sequence.

Neither the position or contents of the stack, nor the contents of the location specified by A_1 will be changed by CLF unless referenced by A_3.

A_2 Structure:



2-4 contain the number of shifts (0-7 characters).
6-8 contain the number of characters, i.e., length of the field (0-7 characters) with zero indicating 8 characters or a full word.
10-12 contain the leftmost character position of the field (0-7 position).

Ex: CLF WORDS+X2+X3+X4,7 1 5,LEAST

BEFORE - Top of stack = 0000000001000000_8

WORDS+X2+X3+X4 = 2342612566004331_8

AFTER - Program will branch to instruction contained in location LEAST.

INTERMEDIATE IMAGES OF A_1

After stripping = 0000000000000000_8

After rotation = 0000000000000000_8

COMPOOL CODING: Handled in the same manner as CEF.

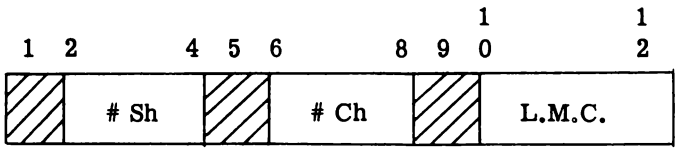
BAF - BINARY ADD FIELD - 43_8

BAF (M),F,(M)

BAF uses the field specification of A_2 to strip first an image of the contents of the location specified by A_1 and then an image of the contents of the top of the stack, and performs a 48-bit unsigned addition with the results. The sum is stripped in turn, an image of the contents of the location specified by A_1 has its field area specified by A_2 cleared to zero, and the two images are ORed together. The result is right end-around shifted the number of times specified in A_2 and placed in the location specified by A_3.

Neither the position or contents of the stack, nor the contents of the location specified by A_1 will be changed by BAF unless referenced by A_3.

A_2 Structure:



2-4 contain the number of shifts (0-7 characters).
6-8 contain the number of characters, i.e., length of the field (0-7 characters) with zero indicating 8 characters or a full word.
10-12 contain the leftmost character position of the field (0-7 position).

Ex: BAF ITEM+X10,0 1 1,SUM

BEFORE - Top of stack = 7622532161742160_8

ITEM+X10 = 6263214463125432_8

AFTER - SUM = 6205214463125432_8 (NOTE OVERFLOW BIT IS LOST)

INTERMEDIATE IMAGES

Stripped Image of A_1	-	0063000000000000_8
Stripped Image of Stack	-	0022000000000000_8
Unsigned Addition	-	0105000000000000_8
Stripped Image of Addition	-	0005000000000000_8
Cleared Image of A_1	-	6200214463125432_8

COMPOOL CODING:

BAF A_1,ITEM N,A_3

Strip an amount of bytes of LENGTH "ITEM" and whose least significant byte position is the least significant byte position of ITEM. Strip the corresponding bytes in the top of the stack. Adjust the resulting sum so that the least significant byte position of ITEM corresponds to byte position "N".

BAF A_1,N ITEM,A_3

Strip an amount of bytes of length "ITEM" and whose significant byte position is "N". Strip the corresponding bytes in the top of the stack. Adjust the resulting sum so that byte position N corresponds to the least significant byte position of ITEM.

BAF A_1,ITEM 1 ITEM 2,A_3

Strip an amount of bytes of length "ITEM 1," and whose least significant byte position is the least significant byte position of ITEM 1. Strip the corresponding bytes in the top of the stack. Adjust the resulting sum so that the least significant byte position of ITEM 1 corresponds to the least significant byte position of ITEM 2.

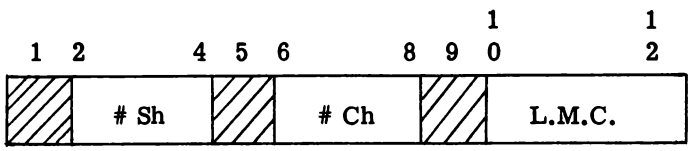
BSF - BINARY SUBTRACT FIELD - 42_8

BSF (M),F,(M)

BSF uses the field specification of A_2 to strip first an image of the contents of the location specified by A_1 and then an image of the contents of the top of the stack, and performs a 48-bit unsigned subtraction with the results which gives the ABSOLUTE difference of the two. The difference is stripped in turn, an image of the contents of the location specified by A_1 has its field area as specified by A_2 set to zero, and the stripped difference is inserted into the cleared image. The result is right end-around shifted the number of times specified in A_2 and placed in the location specified by A_3.

Neither the position or contents of the stack, nor the contents of the location specified by A_1 will be changed by BSF unless referenced by A_3.

A_2 Structure:



2-4 contain the number of shifts (0-7 characters).
6-8 contain the number of characters, i.e., length of the field (0-7 characters) with zero indicating 8 characters or a full word.
10-12 contain the leftmost character position of the field (0-7 position).

Ex: BSF ITEM+X10,4 1 4,DIF

BEFORE - Top of stack = 7622532161742160_8
ITEM+X10 = 6263214463125432_8
AFTER - DIF = 0212543262632144_8

INTERMEDIATE IMAGES

Stripped Image of A_1	-	0000000063000000_8
Stripped Image of Stack	-	0000000061000000_8
Absolute Difference	-	0000000002000000_8
Cleared Image of A_1	-	6263214400125432_8
Insertion of Difference	-	6263214402125432_8

COMPOOL CODING: Handled in the same manner as BAF.

LAF - LOGICAL AND FIELD - 47_8

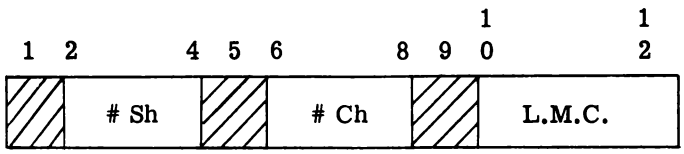
LAF (M),F,(M)

LAF uses the field specification of A_2 to strip first an image of the contents of the location specified by A_1 and then an image of the contents of the top of the stack, and then logically ANDs the results together. This logical combination is stripped in turn, an image of the contents of the location specified by A_1 has its field area as specified by A_2 cleared to zero, and the stripped combination is inserted into the cleared image. The result is right end-around shifted the number of times specified in A_2 and placed in the location specified by A_3.

The AND operation compares bits of the same bit position and produces a zero when either or both are zeros and a one when they are both ones.

Neither the position or contents of the stack, nor the contents of the location specified by A_1 will be changed by the instruction unless referenced by A_3.

A_2 Structure:



2-4 contain the number of shifts (0-7 characters).
6-8 contain the number of characters, i.e., length of the field (0-7 characters) with zero indicating 8 characters or a full word.
10-12 contain the leftmost character position of the field (0-7 position).

Ex: LAF WORD+X1,1 2 4,COMBO

BEFORE - WORD+X1 = 2132562244312614_8

Top of Stack = 2653416124151206_8

AFTER - COMBO = 1421325622041126_8

INTERMEDIATE IMAGES

Stripped Image of A_1	-	0000000044310000_8
Stripped Image of Stack	-	0000000024150000_8
Result of AND operation	-	0000000004110000_8
Cleared Images of A_1	-	2132562200002614_8
Insertion of AND result	-	2132562204112614_8

COMPOOL CODING: Handled in the same manner as BAF.

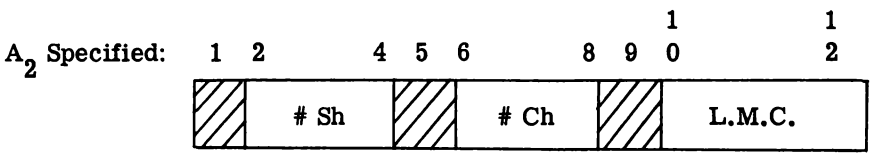
LOF - LOGICAL OR FIELD - 44_8

LOF (M),F,(M)

LOF uses the field specification of A_2 to strip first an image of the contents of the location specified by A_1 and then an image of the contents of the top of the stack, and then logically ORs the results together. This logical combination is stripped in turn, an image of the contents of the location specified by A_1 has its field area as specified by A_2 cleared to zero, and the stripped combination is inserted into the cleared image. The result is right end-around shifted the number of times specified in A_2 and placed in the location specified by A_3.

The OR operation compares bits of the same bit position and produces a zero when they are both zeros and a one when either or both are ones.

Neither the position or contents of the stack, nor the contents of the location specified by A_1 will be changed by the instruction unless referenced by A_3.



2-4 contain the number of shifts (0-7 characters).
6-8 contain the number of characters, i.e., length of the field (0-7 characters) with zero indicating 8 characters or a full word.
contain the leftmost character position of the field (0-7 position).

Ex: LOF BITS,1 5 0,FILL

BEFORE - BITS = 3556754255264105_8
Top of Stack = 5621423522375241_8
AFTER - FILL = 0577777777772641_8

INTERMEDIATE IMAGES

Stripped Image of A_1	-	3556754255000000_8
Stripped Image of Stack	-	5621423522000000_8
Result of OR Operation	-	7777777777000000_8
Cleared Image of A_1	-	0000000000264105_8
Insertion of OR Result	-	7777777777264105_8

COMPOOL CODING: Handled in the same manner as BAF.

LXF - LOGICAL EXCLUSIVE OR FIELD - 45_8

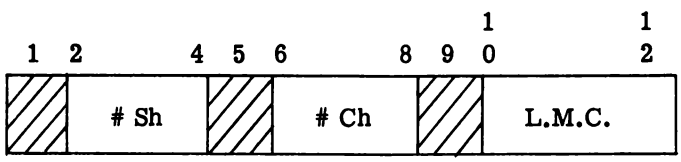
LXF (M),F,(M)

LXF uses the field specification of A_2 to strip first an image of the contents of the location specified by A_1 and then an image of the contents of the top of the stack, and then logically EXCLUSIVE ORs the results together. This logical combination is stripped in turn, an image of the contents of the location specified by A_1 has its field area as specified by A_2 cleared to zero, and the stripped combination is inserted into the cleared image. The result is right end-around shifted the number of times specified in A_2 and placed in the location specified by A_3.

The EXCLUSIVE OR operation compares bits of the same bit position and produces a zero when they are identical (either both ones or both zeros), and a one when they are different.

Neither the position or contents of the stack, nor the contents of the location specified by A_1 will be changed by the instruction unless referenced by A_3.

A_2 Structure:



2-4 contain the number of shifts (0-7 characters).
6-8 contain the number of characters, i.e., length of the field (0-7 characters) with zero indicating 8 characters or a full word.
10-12 contain the leftmost character position of the field (0-7 position).

Ex: LXF BITS,0 6 1,COMBO

BEFORE - BITS = 2153667152443122_8
Top of stack = 4155345026423144_8
AFTER - COMBO = 2106522174060022_8

INTERMEDIATE IMAGES

Stripped Image of A_1	-	0053667152443100_8
Stripped Image of Stack	-	0055345026423100_8
Result of Exclusive OR Operation	-	0006522174060000_8
Cleared Image of A_1	-	2100000000000022_8
Insertion of EXCLUSIVE OR Result	-	2106522174060022_8

COMPOOL CODING: Handled in the same manner as BAF.

LCF - LOGICAL COMPLEMENT FIELD - 46_8

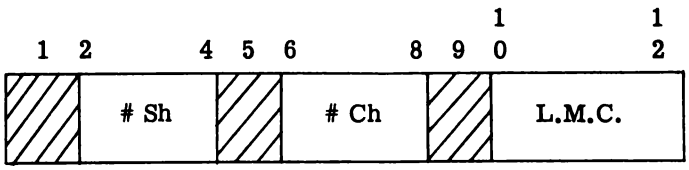
LCF (M),F,(M)

LCF uses the field specification of A_2 to strip an image of the contents of the location specified by A_1 and logically COMPLEMENTS the result. This complemented field is stripped in turn, an image of the contents of the location specified by A_1 has its field area as specified by A_2 cleared to zero, and the two images are ORed together. The result is right end-around shifted the number of times specified in A_2 and placed in the location specified by A_3.

The COMPLEMENT operation reverses the settings of bits so that ones become zeros and zeros become ones.

The contents of the location specified by A_1 will not be changed by the instruction unless referenced by A_3.

A_2 Structure:



2-4 contain the number of shifts (0-7 characters).
6-8 contain the number of characters, i.e., length of the field (0-7 characters) with zero indicating 8 characters or a full word.
10-12 contain the leftmost character position of the field (0-7 position).

Ex: LCF BITS, 3 1 6,HOLD

BEFORE - BITS = 2132554161253243_8

AFTER - HOLD = 2545432132554161_8

INTERMEDIATE IMAGES

Stripped Image of A_1	-	0000000000003200_8
Result of COMPLEMENT Operation	-	7777777777774577_8
Result Stripped	-	0000000000004500_8
Cleared Image of A_1	-	2132554161250043_8
Insertion of COMPLEMENT Result	-	2132554161254543_8

COMPOOL CODING:

LCF A_1,ITEM N,A_3

Strip an amount of bytes of length "ITEM," and whose least significant byte position is the least significant byte position of ITEM. Adjust the complemented result so that the least significant byte position of ITEM corresponds to byte position N.

LCF A_1,ITEM 1 ITEM 2,A_3

Strip an amount of bytes of length "ITEM 1 ', and whose significant byte position is the least significant byte position of ITEM 1. Adjust the complemented result so that the least significant byte position of ITEM 1 corresponds to the least significant byte position of ITEM 2.

FLOATING-POINT ARITHMETIC INSTRUCTIONS

This group of instructions treats the four basic floating-point arithmetic instructions and includes the instruction for converting a fixed-point arithmetic data word to its equivalent floating-point format (CBF). In all floating-point arithmetic instructions, it is assumed that the contents specified in the A_1 and A_2 syllables are in floating-point format.

Floating-point format is defined as follows: the most significant 12 bits represents a 12-bit signed exponent. The least significant 36 bits represents a 36-bit signed mantissa. The mantissa indicates a binary quantity, and the exponent denotes the number of times the mantissa is raised by the power of 2. As a result, the exponent indicates the actual position of the binary point in the mantissa.

Ex:	positive exponent and negative mantissa	$0005623400000000_8 = -22.34_8$
	negative exponent and positive mantissa	$4005223400000000_8 = +.01116_8$

Note that the first bit of the mantissa is the sign bit and the imaginary binary point in a floating-point word is understood to be just right of the mantissa sign bit.

The use of floating-point computation facilitates operation with much larger numbers than would be possible with only a fixed-point arithmetic capability. It also reduces considerably the amount of scaling which must be done by the programmer.

Normalization is an important condition of a floating-point word. In this condition all leading zeros have been shifted out of the mantissa and the most significant data bit (bit 14) is equal to 1. Depending upon the particular arithmetic instruction, various results ensue if either or both operands are not normalized. In general, the result of a floating-point instruction will not be normalized if its operands are not normalized.

FAD - FLOATING ADD - 67_8

FAD (M),(M),(M)

FAD performs floating-point addition. The contents of the location specified by A_2 (the addend) are algebraically added to the contents of the location specified by A_1 (the augend). The result is then stored in the location specified by A_3. It is assumed

that the A_1 and A_2 operands of this instruction are in floating-point format. The resultant data word that is stored in A_3 is normalized floating-point number, except as noted below. The contents of A_1 and A_2 remain unchanged unless A_3 denotes one of these locations.

The orders of magnitude of the mantissas of the A_1 and A_2 contents are aligned by automatically computing the algebraic difference between the exponents of the two operands and shifting the smaller mantissa (the mantissa with the algebraically smaller exponent) right the required number of places.

Ex: FAD AUG1,ADD1,SUM1

BEFORE - AUG1 = 0007346234321000_8
ADD1 = 0003102120120000_8 —— Aligned to = 0007004105005000_8

AFTER - SUM1 = 0007352341326000_8

If the difference between the exponents of the two operands is 35 or greater, but less than 2047, the operand that has the algebraically greater exponent is first added to a value of positive zero. Normalization, if required, is then performed. This requires the removing of leading zeros in the mantissa by an appropriate number of left shifts and algebraically adding to the exponent a negative value equal to the number of bits shifted. The result is stored in A_3.

Ex: FAD AUG2,ADD2,SUM2

BEFORE - AUG2 = 5252525252525252_8
ADD2 = 4210421042104210_8
AFTER - SUM2 = 4213610421042100_8

Note that a three-bit normalization of ADD2 was required and the exponent 4210_8 changed accordingly.

If the signs of the exponents of the two operands are unlike, and the sum of the absolute values of the two exponents is greater than 2047, the add operation is not performed, and the operand that has the positive exponent is stored in A_3.

Ex: FAD AUG3,ADD3,SUM3

BEFORE - AUG3 = 2525123456707654_8 (positive exponent)
ADD3 = 6525765432100000_8 (negative exponent)
AFTER - SUM3 = 2525123456707654_8

Note that in this case normalization of the result is not performed.

If the execution of this instruction results in mantissa overflow, the computer corrects the result by shifting the mantissa right, arithmetically, one place, inserting the most significant bit lost by overflow, and then adjusting the exponent by algebraically adding +1. However, if exponent overflow occurs as a result of this adjustment, the correct mantissa, along with an overflow exponent of 0001_8 is stored in A_3, and the program (POV) Flip-flop is set. Because the adjustment is always a +1, no overflow of this type will occur with a negative exponent.

Ex: FAD AUG4,ADD4,SUM4

BEFORE - AUG4 = 377777777777777_8
ADD4 = 377777777777777_8

AFTER - SUM4 = 000177777777777_8
POV indicator on

Note that the preliminary addition of the two mantissas yielded 77777777777_8 with an overflow which is corrected by the shift right and insertion.

If the execution of this instruction produces a mantissa that is equal to zero, "floating-point zero" is stored in A_3, and the program underflow (PUN) flip-flop is set. Floating-point zero is the smallest possible positive number, $0x2^{-2047}$, and appears as a negative exponent of all 1's with a positive mantissa of all zeros.

Ex: FAD AUG5,ADD5,SUM5

BEFORE - AUG5 = 7773124544631400_8 aligned to = 7765001245446314_8 (negative mantissa)
ADD5 = 7765401245446314_8 (positive mantissa)

AFTER - SUM5 = 7777000000000000_8
PUN indicator on

If, as a result of normalization, there is exponent underflow (negative value added to exponent exceeds -2047), the program underflow (PUN) flip-flop is set, and floating-point zero is stored in the location specified by A_3.

Ex: FAD AUG6,ADD6,N

BEFORE - AUG6 = 7774124544631477_8 (positive mantissa)
ADD6 = 7774520000000000_8 (negative mantissa)

AFTER - Top of stack = 7777000000000000_8
PUN indicator on

The preliminary result before normalization is 7774004544631477_8. As a result of normalization, a -5 is added to the exponent 7774_8, which results in underflow.

FSU - FLOATING SUBTRACT - 66_8

FSU (M),(M),(M)

FSU performs floating-point subtraction. The contents of the location specified by A_2 (the subtrahend) are algebraically subtracted from the contents of the location specified by A_1 (the minuend). The result is then stored in the location specified by A_3. It is assumed that A_1 and A_2 operands of this instruction are in floating-point format. The resultant data word is a normalized floating-point number, except as noted below. The contents of A_1 and A_2 remain unchanged unless A_3 references one of these locations.

The characteristics of the FSU instruction are the same as those described on the previous page for the FAD (Floating Add) instruction with the following exception. During the alignment of mantissas, if the difference between the two exponents is 35 or greater and the algebraically greater exponent is the subtrahend, then the sign of its mantissa will be complemented.

Ex: FSU MIN1,SUB1,DIFF1

BEFORE - MIN1 = 6525765432104567_8
SUB1 = 2525452525252525_8
AFTER - DIFF1 = $\overline{2525052525252525}_8$

The above example involves operands with unlike signs and results in an exponent difference of greater than 2047. The subtract operation is not performed and the operand with the positive exponent is stored in A_3. Normalization of the result is not performed either. Note that the sign of the mantissa in SUB1 has been changed prior to transfer because SUB1 has the algebraically greater exponent.

Ex: FSU MIN2,SUB2,DIFF2

BEFORE - MIN2 = 5252525252525252_8
SUB2 = 4210421042104210_8
AFTER - DIFF2 = $\overline{4213210421042100}_8$

The above example involves operands with an exponent difference of less than 2047 and greater than 34. The operand with the algebraically greater exponent is added to a value of +0, normalized, and stored in A_3.

In this example a 3-bit normalization of the result has been performed, and the exponent is adjusted accordingly. Note that the sign of the mantissa is SUB2 has been changed during the alignment because SUB2 has the algebraically greater exponent.

Ex: FSU MIN3,SUB3,DIFF3

BEFORE - MIN3 = 0002311000000000_8 (positive mantissa)
SUB3 = 0002710400000000_8 (negative mantissa)
AFTER - DIFF3 = 0003310600000000_8

In the above example, subtraction resulted in mantissa overflow. A 1-bit right shift of the mantissa was performed, the overflow bit inserted in the most significant bit position of the mantissa, and the exponent increased by a +1.

Ex: FSU MIN4,SUB4,DIFF4

BEFORE - MIN4 = 7773524544631400_8 aligned to = 7765401245446314_8
SUB4 = 7765401245446314_8
AFTER - DIFF4 = 7777000000000000_8
PUN indicator on

In the above example, the FSU operation resulted in a mantissa equal to zero. As a consequence, floating-point zero is stored in A_3.

FMU - FLOATING MULTIPLY - 63_8

FMU (M),(M),(M)

FMU performs floating-point multiplication. The contents of the location specified by A_1 are multiplied by the contents of the location specified by A_2. The most significant portion of the double precision product is normalized, if other than zero, and placed in the location specified by A_3. The least significant portion, whose mantissa has the same sign as that of the most significant portion, is placed in TFC (Thin Film C - 124_8-127_8). The resulting exponent is the algebraic sum of the original exponents of the two operands.

When the product has been normalized one bit position, the TFC exponent will be one greater than the A_3 exponent and the high order bit of the TFC mantissa will be identical to the low order bit of A_3. When the product has not been normalized one bit position, the A_3 and TFC exponents will be identical and the high order bit of the TFC mantissa will not necessarily be the same as the low order bit of A_3. The contents of the TFC register are not changed during the normalization operation. A normalized product will result for an FMU operation only when the original operands are normalized.

It is assumed that both the A_1 and A_2 operands are in the floating-point format. The contents of A_1 and A_2 remain unchanged unless A_3 references these locations.

If either the A_1 or A_2 operand contains floating-point zero or a mantissa that is not normalized, the program not normalized (PNN) flip-flop is set prior to performing the multiply operation.

If the sum of two positive exponents of the two operands is greater than 2047, the program overflow (POV) flip-flop is set and the absolute value of the mantissa portion of A_1, along with the overflow exponent, is stored in A_3.

Ex: FMU MULT1,MULD1,PROD1

BEFORE - MULT1 = 3011252525252525_8
MULD1 = 2777777777777777_8
AFTER - PROD1 = 2011252525252525_8
POV indicator on

Note that the left-most 1 bit resulting from the overflow of the exponent addition is carried foward and added to the exponent before being stored in PROD1.

If the sum of two negative exponents of the two operands is less than -2047, the program underflow (PUN) flip-flop is set and floating-point zero is stored in A_3. Note in the example below that the PNN indicator also will be set because MULT2 is not normalized.

Ex: FMU MULT2,MULD2,PROD2

BEFORE - MULT2 = 5252525252525252_8
MULD2 = 6767676767676767_8
AFTER - PROD2 = 7777000000000000_8
PNN and PUN indicators on

If the mantissa portion of the most significant half of the product is equal to zero, floating-point zero is stored in A_3, and the program overflow (PUN) flip-flop is set. The least significant half of the product in the TFC register remains unchanged. Note in the example below that the PNN indicator also will be set because MULD3 is not normalized.

Ex: FMU MULT3,MULD3,PROD3

BEFORE - MULT3 = 2461765432102345_8
MULD3 = 0000000000000001_8
AFTER - PROD3 = 7777000000000000_8
TFC = 2461765432102345_8
PUN and PNN indicators on

FDV - FLOATING DIVIDE - 62_8

FDV (M),(M),(M)

FDV performs floating-point division. The contents of the location specified by A_1 (the dividend) is divided by the contents of the location specified by A_2 (the divisor). The 35-bit quotient with sign and the resulting exponent are stored in the location specified by A_3. The resulting exponent is the algebraic difference between the original exponents of the two operands. A 35-bit register right justified. The

The exponent of the remainder is always equal to zero. The contents of A_1 and A_2 remain unchanged unless A_3 references one of these locations. It is assumed that both the A_1 and A_2 operands are in the floating-point format.

If either the A_1 or A_2 operand contains floating-point zero or a mantissa that is not normalized, the program not normalized (PNN) flip-flop is set prior to performing the divide operation. If both of the original operands of an FDV operation are normalized, the result will also be normalized.

If the difference in the values of the exponents is greater than 2047 (exponent signs unlike and sign of divide exponent positive), the program overflow (POV) flip-flop is set. Under these conditions, the absolute value of the mantissa portion of A_1 with the overflow exponent, is stored in A_3. Note in the example below that the PNN indicator is set because DVSR1 is not normalized.

Ex: FDV DVDN1,DVSR1,QUOT1

BEFORE - DVND1 = 377777777777777_8
DVSR1 = 5252525252525252_8
AFTER - QUOT1 = 1252377777777777_8
PNN and POV indicators on

Note above that as a result of exponent overflow, a +1 is algebraically added to the exponent stored in QUOT1.

If the difference in the values of the exponents is less than -2047 (exponent signs unlike and sign or dividend exponent negative), the program underflow (PUN) flip-flop is set, and floating-point zero is stored in A_3.

Ex: FDV DVND2,DVSR2,QUOT2

BEFORE - DVND2 = 7654321012345670_8
DVSR2 = 2525252525252525_8
AFTER - QUOT2 = 7777000000000000_8
PUN indicator on

If a quotient mantissa equal to zero is produced as the result of the divide operation, a floating-point zero is stored in A_3. The remainder in TFC remains unchanged. It should be noted that this condition will never occur if both of the original operands are normalized.

If normalized operands are used and the execution of this instruction results in mantissa overflow, the overflow condition is automatically corrected by saving the most-significant mantissa bit that would be lost by overflow and then adjusting the exponent by algebraically adding +1. If exponent overflow occurs as a result of this one bit adjustment, the correct mantissa with an overflow exponent of 0001_8 is stored in A_3, and the program overflow (POV) flip-flop is set.

Ex: FDV DVND3,DVSR3,QUOT3

BEFORE - DVND3 = 377777777777777_8
DVSR3 = 0000377777777777_8
AFTER - QUOT3 = 0001600000000000_8
POV indicator on

Note above that mantissa overflow occurred because the absolute value of the divisor does not appear greater than the dividend.

Mantissa overflow resulting from not-normalized operands will cause the POV flip-flop to be set. In this case, all overflow bits are lost, and adjustment of the exponent does not take place.

Ex: FDV DVND4,DVSR4,QUOT4

BEFORE - DVND4 = 0000200000000000_8
DVSR4 = 0000100000000000_8
AFTER - QUOT4 = 0000000000000000_8
TFC = 0000000000000000_8

As noted earlier, the exponent of the remainder is always equal to zero (in TFC). However, the following equation may be used to determine the correct exponent for the remainder:

Remainder exponent = (2 x dividend exponent) -
(42 + divisor exponent + quotient exponent)

CBF - CONVERT BINARY TO FLOATING-POINT - 25_8

CBF (M),(M)

CBF treats the contents of the location specified by A_1 as a signed 48-bit, fixed-point number with an implied exponent of zero. Normalization (or the removal of leading zeros) is performed on this number, and, as each bit of normalization occurs, the exponent for the resulting floating-point is increased by a -1. When the normalization has been completed, the most significant 35-bits of the result with the original sign form the mantissa. The exponent is equal to 2^{-n}, where n represents the number of bits of normalization that have occurred. The resulting exponent and mantissa of the floating-point equivalent are stored in the location specified by A_2.

If fewer than 12 bits of normalization are required during this operation, a corresponding number of least significant bits of the original contents of A_1 will be lost. If the contents of A_1 are equal to zero, floating-point zero (7777000000000000_8) is inserted into the location specified by A_2.

Ex: CBF CONVT,STORE

BEFORE - CONVT = 0004576137615213_8

AFTER - STORE = 4010227705770650_8

Note that there are 8-bits of normalization required. As a result, the four least significant bits of CONVT are lost in the conversion operation, since the twelve most significant bits of STORE are reserved for the exponent.

MISCELLANEOUS INSTRUCTIONS

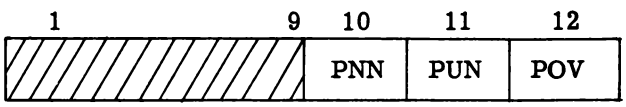
This group of instructions directs a variety of computer operations. Most of these instructions are concerned with special features of the AN/GSA-51A computer including the Interrupt System, Input/Output operations, and subroutine control. Two additional instructions, Branch on Condition (BRC) and Character Search (CSE), are also included.

BRC - BRANCH ON CONDITION - 11_8

BRC L,B

BRC tests the setting of PNN (Program Non Normalized), PUN (Program UNderflow), and POV (Program OVerflow) flip-flop. Indicators as bits 10, 11, and 12 respectively of A_1 are ones. All combinations of the three bits may be used to test all combinations of the indicators. If any bit is one and the corresponding indicator is on, the indicator will be turned off and control will transfer to the first syllable of the location specified in A_2. If no bits are one or all their respective indicators are off, control continues to the next instruction in sequence.

The A_1 syllable is structured as follows:



A one is set in bits 10, 11, or 12 or any combination if testing of any of the three conditions is desired. The coding of A_1 is as follows:

0 = no test	3 = test POV or PUN	6 = test PNN or PUN
1 = test POV	4 = test PNN	7 = test PNN or PUN or POV
2 = test PUN	5 = test PNN or POV	

The following conditions may produce program overflow (POV):

(1) Overflow may result from fixed-point addition, subtraction, or division.

(2) Overflow may result from the execution of the TRM (Rounding) instruction.

(3) Exponent overflow in floating-point arithmetic may result from the addition of two positive exponents.

(4) Quotient overflow may result from a floating-point division using non-normalized operands.

The following conditions may produce program underflow (PUN):

(1) Underflow may result from the addition of two exponents of floating-point numbers.

(2) Underflow will result from a floating-point addition or subtraction that produces a result of floating-point zero.

(3) Underflow will result from a floating-point multiplication using non-normalized operands that result in an answer of floating-point zero.

Program not normalized (PNN) indicator is set during floating-point multiplication or division as a result of using operands which have leading zeros in the mantissa portion.

Ex: (1) BRC 3,BRNCH

BEFORE - Program underflow indicator on. Program overflow indicator not on.

AFTER - The next instruction will be taken from the first syllable in the instruction contained in BRNCH.

(2) BRC 4,CHECK

BEFORE - Program not normalized (PNN) indicator not on.

AFTER - The next instruction will be taken from the one following in sequence.

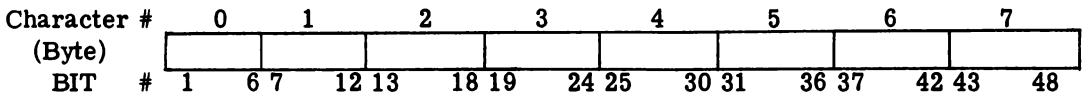
It should be noted that the branch in the program overflow (POV) condition will never be executed when the computer is operating in the normal mode with the overflow mask register bit appropriately set. The computer in this mode will respond to the overflow interrupt condition prior to the execution of the instruction. The POV flip-flop will have been reset when the computer is returned to the normal mode after servicing the interrupt.

CSE - CHARACTER SEARCH - 32_8

CSE (M),C,B

CSE takes the contents of the location specified by A_1 and scans from RIGHT to LEFT, beginning with the character (or byte) indicated by CCR (Character Count Register - 123_8), for the first character with an unsigned, six-bit magnitude equal to the setting of bits 7-12 of A_2. The A_2 syllable is always coded as a single Hollerith character, except the comma (,) or a 2 digit octal number (representing the six bits being checked). If such a character is found, the scan terminated with CCR indicating the character position, and control will transfer to the first syllable of the location specified by A_3. Otherwise, control will continue to the next instruction in sequence with CCR set to zero.

The following diagram represents character numbering in the contents of the A_1 syllable:



Scanning is controlled by the CCR value and cycles as follows:

(1) CCR value is decreased by one. If this is the first cycle of the scan and CCR is initially zero, it will flip to seven. It should be noted that character #7 actually refers to the eighth character, #6 to the 7th, etc. If scanning is desired for the entire word, it may be necessary to precede this instruction with an LTF instruction, loading CCR with a zero.

(2) The contents of the indicated character are compared as an unsigned, 6-bit magnitude with the setting of bits 7-12 of A_2. If the indicated character value is equal to the A_2 setting, control will transfer to the first syllable of the location indicated by A_3 with the CCR value set for the matching character position.

(3) If the above comparison is unsuccessful, CCR is examined. If the value is zero, indicating that the last character has been tested, normal sequencing of instructions continues. Otherwise, the complete cycle is repeated.

Ex: (1) CSE DATA,14,MATCH

BEFORE - CCR = 0000_8

DATA = 3670142626747675_8

AFTER - CCR = 0002_8

The next instruction executed will be from location MATCH. Scanning began with character #7 (or the 8th character).

(2) CSE H,D,BRNCH

BEFORE - Top of Stack = 4354657607102132_8

CCR = 0003_8

AFTER - CCR = 0000_8

The next instruction in sequence is executed. Scanning began with the 2nd character (65_8).

SRJ - SUBROUTINE JUMP - 14_8

SRJ Ja,Ji

SRJ provides a means of communication between programs and subroutines which are to be used many times during the execution of these programs. Through the use of unique storage registers in thin film, any caller program may transfer to a

subroutine by using this instruction. Upon completion of the subroutine a companion instruction, Subroutine Return (SSR), will assure a return to the next instruction in sequence of the caller program. The SRJ instruction transfers program control to the first instruction of a subroutine whose starting location is indirectly addressed by the A_1 syllable.

The following sequence of computer operations is performed AUTOMATICALLY in the execution of this instruction.

(1) The current values of the BAR (Base Address Register-055_8), BPR (Base Program Register-054_8), and PCR (Program Count Register-057_8) of the caller program are stored in the SSR (Subroutine Storage Register-050_8 - 052_8). As a result, the 16 low-order bits of the SSR contain the BAR, the 16 middle bits contain the BPR, and the 16 high-order bits contain the PCR. It should be noted that the low-order bits begin at thin film address 050_8 of the SSR (Subroutine Storage Register) while the high order bits are found in thin film address 052_8. It is this storage of caller program control values that facilitiates return to the main program after completion of the subroutine.

(2) The contents of the SAR (Subroutine Base Address Register - 060_8) are added to the contents of the A_1 syllable to provide the memory location which contains the starting address of the subroutine in the 16 least significant bits of the memory word. The SAR must have been loaded prior to the execution of this instruction. It generally contains the starting address of a subroutine address table and the A_1 syllable indexes a particular entry in that table. The A_1 or Ja (Subroutine Jump Address) syllable therefore can be understood to contain the relative address of this memory location and when added to the SAR indicates the location whose contents specify the starting address of the subroutine. If the A_1 syllable is indexed, the contents of that index would also be added to the SAR plus the contents of the A_1 syllable to determine the location containing the starting address of the subroutine.

Ex: SRJ 2,0

SAR - 000200_8

Memory Location 000200_8 = 0000000000010300_8 (starting address of ABC subroutine)
Memory Location 000201_8 = 0000000000011100_8 (starting address of DEF subroutine)
Memory Location 000202_8 = 0000000000012000_8 (starting address of square root subroutine)

(3) The low order 16-bits of the memory location specified by the addition of the SAR to the A_1 syllable is stored into the BPR and the PCR. This action implements the transfer of control to the starting address of the subroutine. The above example transfers control to the square root subroutine.

(4) The contents of the BAR are increased by the contents of the A_2 syllable (the special Ji syllable). In coding, the number specified in the Ji syllable is decimal. The BAR setting is normally changed to specify the beginning address of data in the caller program which will be used by the subroutine.

(5) Index register X15 is loaded with a quantity equal to the value that has just been loaded into the BPR minus the value that has just been loaded into the BAR. If the result of this subtraction is negative, it will appear in 2's complement form. To gain access to data locations unique to the subroutine, these memory addresses must be indexed with index register X15.

(6) Control will now shift to the first syllable in the memory location specified by the contents of the PCR.

Ex: SRJ 1,64

BEFORE - SAR = 000200_8

Memory location 000201_8 = 0000000000002000_8 (Memory Location = SAR+A_1)
BAR = 001000_8
BPR = 001000_8
PCR = 001030_8
AFTER - SSR = 0041400200001000_8

(Note changes in octal configuration occur because the old BAR, BPR, and PCR are in 16-bit registers.)

New Bar = 001100_8 (64_{10} converts to 100_8)
New BPR = 002000_8
New PCR = 002000_8
X14 = 000700_8 (New BPR less new BAR)

The next instruction executed is taken from location 002000_8.

It is possible to go from one subroutine to another -- a process called nesting. When subroutines are nested, it is necessary to preserve the contents of the SSR (Subroutine Storage Register) which contains the BAR, BPR, and PCR of the caller program. A temporary memory or unused thin film location will serve the purpose. A shift from one subroutine to another using the SRJ instruction will cause the contents of the SSR to be overwritten. When control is shifted back to the parent subroutine, the SSR must be reloaded with appropriate caller program values.

SRR - SUBROUTINE RETURN - 04_8

SRR

SRR provides a means of return from a subroutine which was entered with SRJ; the BAR (Base Address Register - 055_8), BPR (Base Program Register - 054_8), PCR (Program Count Register - 057_8) are loaded from SSR (Subroutine Storage Register - 050_8 - 052_8).

The contents of the PCR will be increased by 1 and the next program word of the caller program will be executed. The first syllable of this word, therefore, must be a valid operator syllable.

Index register X15 will be loaded with a quantity equal to the value that has just been loaded into the BPR minus the value that has just been loaded into the BAR. This action by the computer becomes significant when returning from a nested subroutine.

Ex: SRR

BEFORE - SSR = 0001400000000100_8
BPR = 040503_8
PCR = 040555_8
BAR = 000200_8
X15 = 040303_8

AFTER - New BPR = 000000_8
New PCR = 000030_8
New BAR = 000100_8
New X15 = 177700_8 (BPR minus BAR)

Note above that the least significant 16-bits of the SSR contain the BAR; the middle 16-bits, the BPR; and the most significant 16-bits, the PCR. The next instruction to be executed will be taken from core location 000031_8.

LSR - LOAD SPECIAL REGISTER - 31_8

LSR (M),V_S

LSR has three variations:

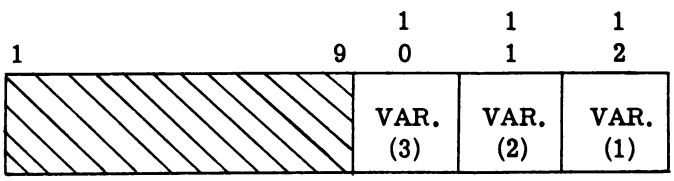
(1) It may be used for loading the mask register.

(2) It may be used for loading the memory bounds register.

(3) It may be used to interrupt a computer or start a computer that is halted.

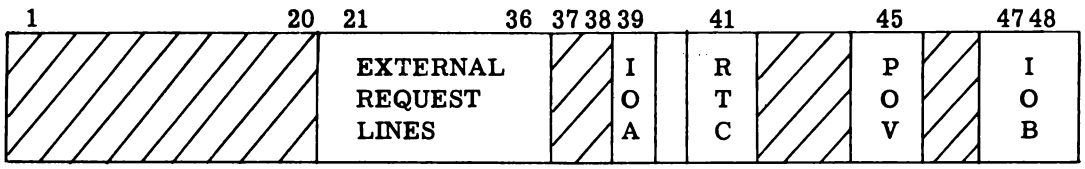
The A_2 (or special V_S) syllable indicates which variation is requested. It should be noted that the LSR instruction can only be executed in the control mode and is considered an illegal instruction if execution is attempted in the normal mode.

A_2 (V_S) Syllable Structure:



(1) MASK REGISTER

If Bit 12 of A_2 is set to one, the Interrupt Mask Register will be loaded with the contents of the location specified by A_1. The purpose of this instruction is to enable the computer to recognize certain interrupt conditions. If the mask bits are not set, the computer will not acknowledge the referenced interrupt conditions. The following structure of the memory location specified by A_1 will indicate which bits must be set for the interrupt to occur.



BIT NUMBER	INTERRUPT CONDITION
21-36	External Request Lines 1-16, Respectively
39	I/O Termination, Bus A
41	Real Time Clock Overflow
45	Arithmetic Overflow
47	Spare
48	I/O Termination, Bus B

The programmer must insert the appropriate values in the location specified by A_1 to insure that the desired bit setting will occur.

Ex: LSR MASK,0001
MASK = 0000001777771210_8

The A_2 syllable indicates a setting of the mask registers. The mask bits will be set for interrupt in the following conditions:

(a) An external request from any of the 16 external request lines generated by an external request when information is to be entered into the system.

(b) The termination of an I/O operation as indicated by the result descriptor list in memory.

(c) The real time clock overflow -- by presetting the RTC register the RTC overflow interrupt can occur at fixed time intervals.

(d) An arithmetic operation has resulted in the program overflow (POV) flip-flop being set.

(2) MEMORY BOUNDS REGISTERS

If Bit 11 of A_2 is set to one, the 8 least significant bits of the contents of the location specified by A_1 will be placed in the Lower Memory Bounds Register, and the 8 next least significant bits in the Upper Memory Bounds Register. This instruction is used in a multi-program system to prevent data stored in memory by one program from being destroyed by another program. A complete memory address within the computer occupies 16 bits. Only the 8 most significant bits of the Bounds Registers are set. The 8 least significant bits, then, are treated as zeros.

Ex:
```
LSR     LIMIT,0002
LIMIT = 0000000000004004(8)
```

The Lower Memory Bounds Register will be set with bits 41 thru 48 of LIMIT, and with eight implied low-order zero bits, the lower memory limit complete address is 2000_8.

The Upper Memory Bounds Register will be set with bits 33 thru 40 of LIMIT and with eight implied low-order zero bits, the upper memory limit equals 4000_8. No program instruction will be executed which would insert data into any core location between addresses 2000_8 and 4377_8. Note that the Computer in testing for limits checks only the 8 most significant bits of a complete memory address and that this instruction can set only these bits. The additional 377_8 represents 255 (decimal) additional memory locations which are not out of bounds.

(3) INTERRUPT COMPUTER N

If Bit 10 of A_2 is set to one, the computer module designated by the three least significant bits of the contents of the location specified by A_1 will be affected by this instruction. It will cause interrupt bit 14 of the interrupt register in the specified computer to be set. The specified computer will recognize that an interrupt condition exists and, if halted, will begin operation in the control mode. An operating computer will recognize an interrupt condition only in the normal mode and this instruction will interrupt such an operating computer and transfer control to the interrupt table to service the interrupt. The interrupt table consists of transfer instructions to routines which will service the various interrupt conditions and is indexed in relation to the I register bit number.

SER - STORE EXTERNAL REQUESTS - 21_8

SER (M)

SER is used to store the input status of the 16 external request lines in bits 33 thru 48 of the location specified by A_1. External request line #1 status is stored in Bit 33; line #2 in Bit 34; and so forth in successive order. In addition, the computer module number is inserted in Bits 29 and 30 of the location specified by A_1.

EXTERNAL REQUEST LINE	FUNCTION	LSR MEMORY WORD BIT	SER MEMORY WORD BIT
1	Flexowriter	21	33
2	Simulator Group	22	34
3	Status Display Console	23	35
4	Spare	24	36
5	Spare	25	37
6	Spare	26	38
7	Message Processor 2 Fill Output Group I (GRIF)	27	39
8	Message Processor 2 Fill Output Group III (GRIIIF)	28	40
9	Message Processor 2 Buffer Empty (CTF)	29	41
10	Message Processor 2 Dump Buffer 1 (EI1)	30	42
11	Message Processor 2 Dump Buffer 2 (EI2)	31	43
12	Message Processor 1 Fill Output Group I (GRIF)	32	44
13	Message Processor 1 Fill Output Group III (GRIIIF)	33	45
14	Message Processor 1 Buffer Emtry (CTF)	34	46
15	Message Processor 1 Dump Buffer 1 (EI1)	35	47
16	Message Processor 1 Dump Buffer 2 (EI2)	36	48

TABLE I. External Requests

This instruction may be used in an interrupt service routine which has been initiated as a result of an external request generated by a terminal device, when information is to be entered into the system. Sixteen terminal device request lines are available to each computer. A terminal device can generate an external request interrupt only if the mask register bit corresponding to the line to which the device is connected is set. The LSR (Load Specified Register) instruction sets the Mask register bit. To determine which terminal device generated the request, this instruction is used.

The setting of A_1, bits 33 thru 48 is determined as follows:

The bit will be set to 1 if a signal is being transmitted over the corresponding request line.

The setting will be 0 if a signal is not being transmitted over the corresponding external request line.

A series of BRB (Branch on Bit) instructions can then be used to detect the external request line which caused the interrupt, so that the service routine can begin to process the new information coming into the system. An external request line signal will usually be terminated only when the device transmitting the signal is accessed.

Ex: SER REQST

Suppose after execution REQST = 0000000001100000_8. The one set in bit 30 indicates that computer module #1 has been interrupted. The one set in bit 33 indicates that external request line #1 has information to be processed by the computer.

IRR - INTERRUPT RETURN - 05_8

IRR

IRR is used to initiate normal mode operation or to restore the conditions necessary for resuming normal execution of a program after the processing of an interrupt has been completed. The IRR is a control mode instruction which provides for the return of program control to the normal mode at a point in the program defined by the interrupt storage registers. Pertinent control data was stored in the following three interrupt storage registers when the interrupt occurred.

(1) ISR - Interrupt Storage Register (TF-040_8-042_8) - the contents of the BAR are stored in the least significant section (TF-040_8); the contents of the BPR are stored in the next higher order section (TF-041_8), and the contents of the PCR are stored in the most significant section (TF-042_8).

(2) IPR - Interrupt Program Register (TF-110_8-113_8) - the contents of the PSR (Program Storage Register) are stored in the IPR. The PSR contains the program instruction word which the computer was operating at the time of the interrupt.

(3) IDR - Interrupt Dump Register (TF-070_8) - the contents of significant control flip-flops are stored in this register. Bit settings in this register indicate which syllable of the instruction word is to be operated upon return to the normal mode program.

During the execution of this instruction the original contents of the BAR, BPR, and PCR are restored to their respective values from the contents of the ISR. The contents of the IPR are loaded into the PSR and the original contents of some of the control flip-flops are restored from the IDR. The next instruction in sequence as determined by the PSR and the syllable indicator of the appropriate flip-flop is then executed. It should be noted that at the time of interrupt the transfer to the control mode is made only at the end of an instruction, which may or may not be the last syllable of an instruction word. Upon execution of IRR the next syllable to be executed will be an operator syllable. The IRR instruction can only be executed in the control mode and is considered an illegal instruction if executed in the normal mode.

Ex: IRR

BEFORE - BAR = 000010_8
BPR = 000010_8
PCR = 000156_8
ISR = 0040000000000647_8
IDR = 042100_8
IPR = 0101224006470000_8

AFTER - PCR = 001000_8
BPR = 000000_8
BAR = 000657_8
PSR1 = 0101224006470000_8

The control flip-flops (PS1, PS2, PS3, RPF, FRP, PF1, POV, PUN, PNN) will be reset in accordance with the contents of the IDR (Interrupt Dump Register). This enables the computer to operate the next instruction in sequence. The following table describes the significance of each bit position in the IDR.

TABLE II. Contents of Interrupt Dump Register

BIT #	DESCRIPTION
1,2,3	Bits represent states of PS1, PS2 and PS3 flip-flops which indicate the address of the next PSR syllable to be operated. This syllable is an operator syllable, since the transfer to control mode can occur only at the end of an instruction. The PSR1 (Program Storage Register 1) syllables are numbered from left to right, 3, 2, 1 and 0. The PSR2 (Program Storage Register 2) syllables are numbered from left to right, 7, 6, 5 and 4.
4	Bit indicates state of RPF flip-flop. This bit is a 1 if a repeated instruction was interrupted.
5	Bit represents state of FRP flip-flop. This bit is a 1 if a repeated instruction was interrupted before execution of the first iteration.
6,7	Bits represent state of PF1 and PF2 flip-flops. These bits contain 1 for each PSR (Program Storage Register) that will contained information after execution of the last instruction before the interrupt was processed. Bit 6 refers to PSR1 and Bit 7 refers to PSR2. If the last syllable of a PSR was used as the last syllable of the instruction before interrupt, this PSR is empty. If the last syllable was not used, this PSR is filled. If overlap has occurred, the other PSR is filled; otherwise it is empty. When bits 6 and 7 are restored to flip-flops PF1 and PF2 and both of the bits are 1's, one of the flip-flops will be reset, since overlap has been lost.
8	Bit indicates state of POV (Program OVerflow) flip-flop.
9	Bit represents state of PUN (Program UNderflow) flip-flop.
10*	Bit represents state of PNN (Program Not Normalized) flip-flop.

*Bits 11-16 of the IDR are not used to restore control flip-flops.

TIO - TRANSMIT INPUT/OUTPUT - 16_8

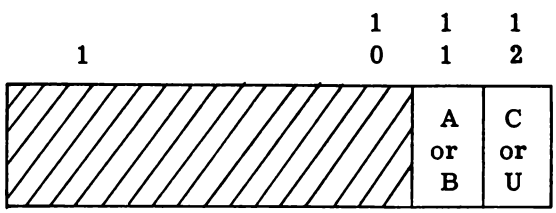
TIO IO,M,B

TIO operates only in the control mode. This instruction has three basic variations and is used to direct and monitor input and output operations of the AN/GSA-51A computer. A series of stages are required to complete an input/output operation. Each of these stages is associated with a particular form of the A_1 syllable and a particular control word (descriptor). When a descriptor is sent by the TIO instruction, its location is specified in the A_2 syllable. When a descriptor is sent by the I/O module to be monitored by the program, it will be located in core memory in a Descriptor List table which is indexed by the I/O module responsible for transmission. The following table outlines seven stages for one of the many possible methods of I/O programming.

The TIO instruction addresses a single I/O BUS and its associated I/O modules only. BUS A is associated with I/O Modules 1 and 2; BUS B, with I/O Modules 3 and 4. For greater program efficiency, more than one I/O module may be accessed by separate Setup descriptors and each In Process descriptor can be checked to find an available I/O module.

The special A_1 syllable indicates which BUS is being accessed and whether the TIO instruction is conditional or unconditional. As noted in Table II on Page 150, TIO instructions with Release and Setup descriptors are unconditional; instructions with a Command Descriptor are conditional.

The following diagram represents the structure of the A_1 syllable:



The BUS is symbolically addressed with "A' or "B" which is interpreted by setting bit 11 of the A_1 Syllable to zero for BUS A or one for BUS B.

The form of the TIO instruction is symbolically specified by "C" or "U" which is interpreted by setting bit 12 of the A_1 syllable to zero for conditional and one for unconditional.

Ex: TIO A U,SETUP,Ø

A_1 syllable = 0001_8

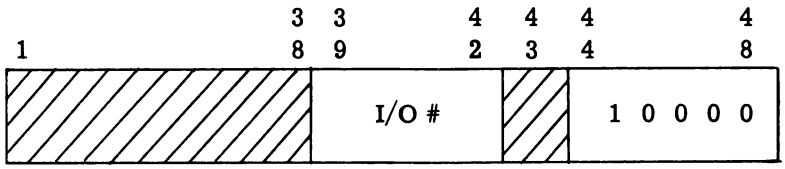
Stage	Purpose	Program Responsibility	I/O Module Responsibility
1. Setup	To transmit address of Descriptor List table to I/O module.	Execute a TIO instruction, unconditional, using a Setup Descriptor.	Initializes its own DBAR Setting with Descriptor List address.
2. 1st In Process	To check if a satisfactory setup descriptor has been transmitted to a non-busy I/O module.	Check appropriate bit settings in Descriptor List table, using relevant computer instructions.	A non-busy I/O module Transmits In Process descriptor to appropriate entry of Descriptor List table.
3. 1st Release	To permit transmission of command descriptor. Every command Descriptor to the I/O module must be preceded by a release I/O module.	Clear appropriate entries of Descriptor List table. Execute A TIO instruction, unconditional, using a Release Descriptor.	Prepares itself for command.
4. Command	To specify type of I/O operation, specific device, core memory referenced, and amount of words and/or records involved.	Execute A TIO operation, conditional, using a Command Descriptor.	Prepares for and commences I/O operation.
5. 2nd In Process	To check whether the operation has begun.	Check appropriate bit settings in Descriptor List table, using relevant computer instructions.	Transmits In Process Descriptor to appropriate entry of Descriptor List Table.
6. Result	To check for completion and success of I/O operation.	Check appropriate bit settings in Descriptor List table, using relevant computer instructions.	Transmits Result Descriptor to appropriate entry of Descriptor List table.
7. 2nd Release	To release I/O module for further operations.	Execute A TIO instruction, using a Release Descriptor.	I/O module on non-busy status.

TABLE III.
Input-Output Operation Stages
For A Sample Method of I/O Programming

RELEASE DESCRIPTORS

Release Descriptors are transmitted by an unconditional TIO to free a specified I/O module. If the specified I/O module is not busy, the Release Descriptor is ignored. If busy and actively engaged in I/O, the I/O module terminates the transmission and returns a Result Descriptor and goes non-busy. Therefore, if more than I/O operation is possible simultaneously, the program must check the In Process Descriptor of a SETUP instruction to ascertain an available I/O module before transmitting a Release Descriptor. If busy, but not actively engaged in I/O, the I/O module goes non-busy. The Release Descriptor is also received by the first non-busy (lowest number) I/O module associated with the BUS specified. If a parity error occurs in the Release Descriptor, the Descriptor will be ignored. However, if the first non-busy I/O module detects the error, it will return an In Process Descriptor indicating the error. With a Release Descriptor an In Process Descriptor is sent only if a parity error is detected. The Descriptor List table should be cleared prior to a Release Descriptor TIO instruction so that appropriate entries can be checked for a parity error.

The Release Descriptor is contained in the location specified by the A_2 syllable and has the following structure:



Bits 39-42 contain the I/O Module number.

Ex: TIO A U,RELSE,Ø

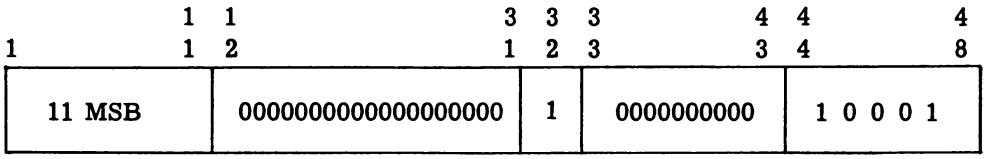
RELSE = 0000000000000120$_8$
This instruction would release I/O module 1.

SETUP DESCRIPTORS

Setup Descriptors are transmitted by an Unconditional TIO to initialize DBAR (Descriptor Base Address Register) settings in the I/O module. The Setup Descriptor sets all I/O modules on the addressed BUS busy and provides them with a location (the DBAR setting of the I/O module) in memory of a table called the Descriptor List. It is to this table that each I/O module will return In Process and Result Descriptors (see below), where they may be examined by the program monitoring the I/O transmission. Only a non-busy I/O module will return an In Process Descriptor following a setup descriptor. Each I/O module will use a unique portion of the Descriptor List: In Process Descriptors will go to the DBAR setting plus twice the I/O module number, Result Descriptors will go to the DBAR setting plus twice the I/O module number plus one (that is, Result Descriptors of a given I/O module will lie at a location one greater than that of the In Process Descriptors of the same I/O module). NOTE: I/O modules are numbered from 1 to a maximum of 5, so the first I/O module returns its In Process Descriptor to the location two greater than the basic DBAR setting.

If the parity of the Setup Descriptor is incorrect, the previous DBAR settings will be used in returning the In Process Descriptor. After power initially comes on, the DBAR setting will be zero.

The Setup Descriptor is contained in the location specified by A_2 and has the following structure:



Bits 1-11 contain the 11 most significant bits of the Descriptor List table location. Since a memory location is specified by 16 bits, the five least significant bits of the address must equal zero. This means that the Descriptor List table must be placed in core memory at a location whose address is a multiple of 32, so that the least significant five bits will equal zero.

Ex: TIO A U,SETUP,0

SETUP = 0400000002000218
The DBAR will be set to 10000_8

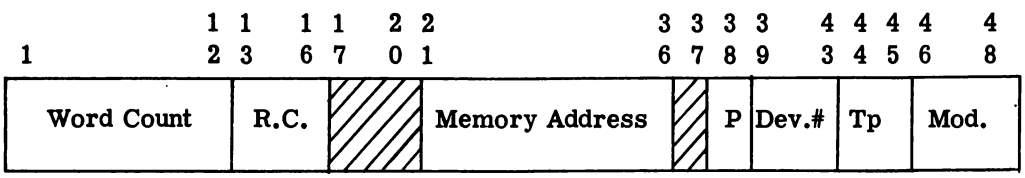
COMMAND DESCRIPTORS

Command Descriptors are transmitted by a conditional TIO to specify and initiate the I/O operation. If all I/O modules are busy, or if parity is incorrect, control will transfer to the first syllable of the location specified by A_3.

The Command Descriptor is accepted by the first, non-busy module. It is important, therefore, that Release and Setup Descriptors have been sent to the first non-busy (the lowest number module) I/O module prior to transmitting the Command Descriptor. In Process and Result Descriptors are transmitted to entries in the Descriptor List table in accordance with the I/O module number sending the transmission. The program must know which I/O module is performing the operation so that it can check the appropriate entry.

If branching occurs in this instruction, it is generally programmed to loop back to the Release Descriptor TIO instruction and to commence the operation again.

The Command Descriptor is contained in the location specified by A_2 and has the following structure:



1-12 contain the Word Count for each operation.
13-16 contain the Record Count (0 means 16 records).
21-36 contain the Memory Address of the first piece of data-incremented during transmission.
38 contains a Priority Bit - 0 Normal Priority
1 Special Priority

39-43 contain the Device Number.
44-45 contain the Type of I/O:
00 one way output device write
10 one way input device read
01 two way device write
11 two way device read
46-48 Modifications (ex: backspace) for device.

To facilitate setting bits in the Command Descriptor, Table IV provides a simple key for determining bits 1-12 (Word Count), 13-16 (Record Count), and 37-48.

TABLE IV. Command Descriptor Bit Setting

COMMAND	BITS 37-48	WORD COUNT	RECORD COUNT
Flexowriter-Write	0010	-N-	00
Flexowriter-Read	0030	-N-	-N-
Flexowriter-Read and Unlock Keyboard	0031	-N-	-N-
Flexowriter-Read and Lock Keyboard	0032	-N-	-N-
Drum 1 Read	2111*	-N-	00
Drum 1 Write	2113*	-N-	00
Drum 1 Erase	0112	0001	00
Drum 2 Read	2051*	-N-	00
Drum 2 Write	2053*	-N-	00
Drum 2 Erase	0052	0001	00
Drum 3 Read	2351	-N-	00
Drum 3 Write	2351*	-N-	00
Drum 3 Erase	0352	0001	00
Card Reader	0162	-N-	00

TAPE		TAPE CONTROL WORD		
Rewind	0251	7N	0002	00
Test	0252	7N	-N-	-N-
Write	0253	7N	-N-	00
Write EOF	0254	7N	0002	00
Rewind and Lockout	0255	7N	0002	00
Erase	0256	7N	-N-	00
Write N Records	0257	7N	-N-	-N-
Backsapce 1 Record and read N Records	0251	ON	-N-	-N-
Backspace N Records	0252	ON	0001	-N-
Backspace to EOF	0253	ON	0001	00
Read N Records	0254	ON	-N-	-N-
Advance N-1 Records and Read 1 Record	0255	ON	-N-	-N-
Advance N Records	0256	ON	0001	-N-
Advance to EOF	2057	ON	0001	00

*Priority bit is set for drum read and write operations.

TABLE IV. Command Descriptor Bit Setting (Cont'd)

	BITS 37-48	WORD COUNT	RECORD COUNT
Status Display Console Write	0410	0001	00
Status Display Console Read	0430	0001	00
Display Console 1	0524	0002	00
Display Console 2	0564	0002	00
Display Console 3	0624	0002	00
Display Console 4	0664	0002	00
Display Console 5	0724	0002	00
Display Console 6	0764	0002	00
Display Console 7	1024	0002	00
Display Console 8	1064	0002	00
Display Console 9	1124	0002	00
Display Console 10	1164	0002	00
Display Console 11	1224	0002	00
PRINTER			
Print and Space	0300	-N-	00
Print and Double Space	0301	-N-	00
Print and Advance to new page	0302	-N-	00
Simulator Group	0327	0001	00

Ex: TIO A C,COMND,RELI

COMND = 0012000100000162_8

This command directs the reading of 12_8 (10_{10}) words from the card reader into memory location beginning at location 10000_8.

IN PROCESS DESCRIPTORS

In Process Descriptors are generated by I/O modules engaged in I/O transmission. These descriptors are transmitted to the Descriptor List table in core memory where they may be examined by the program monitoring the I/O operation. In Process Descriptors will be transmitted to the location specified by the DBAR setting plus twice the I/O module number. There are three types of In Process Descriptors: (1) One which follows the Release Descriptor, (2) One which follows the Setup Descriptor, (3) One which follows the Command Descriptor.

The Release In Process Descriptor is transmitted only if there is a parity error and an I/O module associated with the designated BUS which was not busy. If there is a parity error, Bits 17 thru 19 of the In Process Descriptor are set to one.

The Setup In Process Descriptor is an image of the Setup Descriptor with the exception that Bits 17 thru 19 are set to indicate I/O Module status. The following bit settings indicate the referenced status:

BITS 17 18 19	
0 0 1	Setup Descriptor satisfactorily received.
1 1 1	Parity error from memory in Setup Descriptor.

If all the I/O modules of a particular BUS are busy, no In Process Descriptor will be transmitted.

The Command In Process Descriptor is an image of the Command Descriptor, except that the I/O module status bits 17 thru 19 have been set. The following bit settings indicate the referenced status.

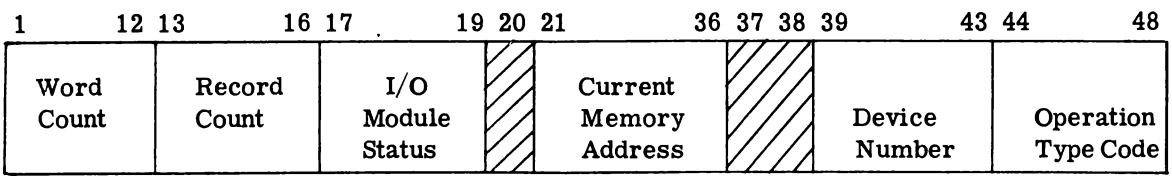
BITS 17 18 19	
0 0 0	Command Descriptor received satisfactorily, I/O operation has started.
0 0 1	I/O Module busy or not ready.
1 1 1	Parity error in Command Descriptor from memory.

RESULT DESCRIPTORS

Result Descriptors are generated by I/O modules and show the extent of completion of the I/O operation. A Result Descriptor is transmitted to the Descriptor List table in core memory where it may be examined by the program monitoring the I/O operation. Result Descriptors will be transmitted to the location specified by the DBAR setting plus twice the I/O module number plus one. Result Descriptors of a given I/O module will be contained in a location one greater than that of the In Process Descriptor of the same I/O module.

A Result Descriptor will be transmitted under the following three conditions: (1) Normal completion of an I/O operation; (2) Existence of an error status condition during an I/O operation; (3) Release of an I/O operation in progress by a Release Descriptor.

The following diagram represents a Result Descriptor. Bits 39-48 are identical to those of the Command Descriptor.



The following settings of Bits 17 thru 19 indicate the referenced status.

BITS 17, 18, 19	STATUS
010	Operation was terminated by sending a release descriptor to the IOCU.
011	Operation is complete, the word count went to zero.
100	No access to memory.
101	Power failure occurred.
110	Parity error from terminal device.
111	Parity error from memory.

Bits 20, 37 and 38 of the Result Descriptor represent the status of the terminal device. Table V indicates the status of the devices as referenced by appropriate bit settings.

Bits 20, 37 and 38 refer to condition that occurs at the terminal device.

TERMINAL DEVICE (BITS 20, 37-38)

	001	010	011	100	101	110	111
Flexowriter	end of record	---	---	no character typed	---	off line	parity error
Card Reader	---	end of file (input hopper empty	mal-function	---	---	data too slow	---
Magnetic Tape	end of record	end of file	abnormal condition or not ready	file protected	end of tape	data too slow	parity error
Message Processor	time elapsed	---	---	not available	---	---	parity error
Drum	---	illegal instruction	abnormal condition	display parity error	radar channel update error	data too slow	parity error
Data Display Console	---	---	---	---	---	---	---
Status Display Console	---	---	parity	---	---	---	---
Printer	---	parity error or malfunction	---	---	---	---	---
Simulator Group	---	---	---	malfunction	---	---	---

TABLE V. Terminal Device Status Bit Settings

CHAPTER 8

SUBROUTINE CODING TECHNIQUES FOR THE AN/GSA-51A

INTRODUCTION

A subroutine is a separate set of instructions which executes an arithmetic or logical operation. Each time this operation is to be performed, the main program branches to the subroutine. After the subroutine has completed its function, program control is returned to the main program. Whenever an operation which requires more than a few steps is to be performed repeatedly in a program, a subroutine should be used to eliminate duplication of coding.

Two types of subroutines are system subroutines and PCR subroutines.

SYSTEM SUBROUTINES makes use of two special subroutine instructions. One of these instructions causes a branch to the subroutine and automatically stores all control information necessary for correct return to the main program. The other instruction causes a return to the main program.

PRC SUBROUTINES do not use the two subroutine instructions. Branching to the subroutine is accomplished simply by an unconditional transfer to the subroutine location. To insure that the control will return to the main program at the correct place, the contents of the PCR are stored into temporary storage just before the transfer is made. Then, after the subroutine has completed its function, control is returned to the main program by placing the contents of the temporary storage back into the PCR.

THE FOUR GENERAL TYPES OF SUBROUTINES

(1) Non-tabular PCR Subroutines.

(2) Table Processing PCR Subroutines.

(3) Simple SRJ Subroutines.

(4) Normal SRJ Subroutines.

These four types of subroutines are distinguished by (a) the manner in which the subroutine is entered and returned from, and (b) the location of the data which is input to the subroutine. In reference to (a), the programmer may enter a subroutine by storing the contents of the PCR into thin film just prior to executing an unconditional jump to the subroutine, or he may use the special Subroutine Jump (SRJ) instruction to branch to the subroutine. If he stores the PCR into thin film, then he is entering a PCR subroutine; if he uses the Subroutine Jump instruction, then he is entering an SRJ subroutine. With regard to (b) above, the input to the subroutine (whether SRJ or PCR) may all be stored in thin film (the stack, the index or limit registers, or any spares, etc.); or it may not (one reason being that it is so extensive that loading it all into thin film is impossible). In the first case, where all the inputs are in thin film, the subroutine is classed either as a Non-Tabular PCR subroutine, or as a Simple SRJ subroutine. In the latter case, where the inputs cannot all be stored in thin film, the subroutine is classed as either a Table Processing PCR subroutine, or as a Normal SRJ subroutine. Normally, when data is too numerous to be held simultaneously in thin film, it is organized into a table. It is possible that one table or several tables of data may be input to a

subroutine; in such cases, the subroutine will have to be either a table processing PCR subroutine, or a Normal SRJ subroutine. As a general rule, whenever a table or tables are input to a subroutine what is directly given to the subroutine is not the table or tables themselves, but merely their CHARACTERISTICS: their length, their location, the size of their entries, the relative location of a given item within each of their entries, and so on. These table characteristics allow the subroutine itself to fetch the table entries, and the user is spared the labor of successively loading table entries into thin film.

There is another general category into which GSA-51 subroutines fall. GSA-51 subroutines are either "floating" or "non-floating" subroutines. A "floating" subroutine is one in which all the symbolic tags appearing in the subroutine have been replaced by addresses relative to the START OF THE SUBROUTINE (not the start of the program).

Such subroutines have SET BAR, and SET BPR cards appearing before them, and these cards are so set that they establish all subsequent symbolic addresses as relative to the start of the subroutine. Floating subroutines can be stored anywhere in main memory, and are, in effect, independent of the user's BAR and BPR settings. ALL SRJ subroutines (whether Simple or Normal) are floating subroutines. PCR subroutines of whatever type may be floating or non-floating depending on the wishes of the writer of the subroutine.

All four types of subroutines, including floating subroutines, will be discussed at length in the following sections.*

*NOTE: PCR and SRJ subroutines are sometimes called "Internal", and "External" subroutines, respectively. This alternate terminology is explained in the sections dealing with the two types of subroutines. It will not be employed in this document. The reasons are given further in the body of the text.

SUBROUTINE CALLING SEQUENCES

The code by which the main program enters the subroutine is termed "the calling sequence.' This calling sequence may and often does include data as to how the subroutine is to operate in this instance of its use. For example, the subroutine may be a read or write flexo-subroutine. The user, in his calling sequence would have to specify somehow whether a read or a write was required. The manner in which he would so indicate his desire would be specified by the form of the subroutine calling sequence, which is rigidly fixed by the designer of the subroutine. The form, then, of the calling sequence is a fixed scheme showing the sequence of instructions and of data registers whereby the subroutine is entered and the information necessary to its operation is correctly located. What precise pieces of information are required by a subroutine in its operation depends on what type of subroutine it is, and the degree of complexity and sophistication it exhibits. The minimal amount of such information would include (1) the location to which the subroutine should return at the end of its operation, and (2) either the location of the data upon which it is to operate or those data themselves. As calling sequences vary so greatly from subroutine to subroutine, their form will be illustrated rather than defined (no definition seems universally applicable) in this document.

PCR SUBROUTINES

The PCR subroutine is the basic subroutine of the GSA-51. It, however, has a wide applicability, and can be adapted to the use of almost any conceivable subroutine. It is also called an "internal" subroutine because it is used mainly for short routines written by the programmer for use within his program. We will not be bound by this terminology and will

employ PCR subroutines more extensively. As its name implies, the PCR subroutine exploits the Program Control Register (PCR), such that the subroutine is capable of returning to the correct location in the main program after operation of the subroutine. This is done, effectively, by the user storing the contents of the PCR into the stack or some other thin film register, and then branching to the subroutine. The subroutine, upon completion of processing, restores to the PCR its old value such that the effective address of the NEXT instruction, AFTER the store into thin film, is the old value plus 1. Much in the way of qualification must be made to this general picture, but this is essentially how a PCR subroutine is entered and returned from. Let us look at a sample of the code by which such a subroutine -- called SUBX -- is ENTERED.

```
*   STF   PCR,N
    UCT   SUBX
```

The syllable structure of these two instructions is important. They each take two syllables, and together they exhaust one 48-bit word. The star before the STF instruction guarantees that the entire pair of instructions will be stored by the assembler in ONE WORD. This is important. The value of the PCR which is stored into the top of the stack (or other thin film register) must be one less than the absolute address of the next instruction to be executed in the main program. This is necessary because of the manner in which the PCR and PSR (Program Store Register) function. The PCR always contains the absolute address of the last word which was used to fill the PSR, and, upon the exhaustion of the current contents of the PSR, the PCR is automatically incremented by 1, and the resulting address is used to fetch the word whose contents will re-fill the PSR. (Upon execution of a branch instruction, the PCR is re-set rather than incremented by 1; it should be clear that by deliberately re-setting the PCR other than by a branch instruction, the new setting is incremented by 1 and this sum is the effective address of the next instruction.) The PCR will not be incremented until each syllable of the PSR has been de-coded and if possible, executed. If an instruction occupies two words in core, then after the first word in the PSR has been de-coded, the PCR will be incremented by 1 and whatever address the PCR may then contain will be assumed to be the address of the remaining syllables of that instruction. So, if the PCR is deliberately re-set by program action, care must be taken to make sure that the new setting plus 1 equals the start of a new instruction which is left justified in the word.

Left-justifying a store-PCR into the stack before a UCT to the subroutine guarantees that the next instruction in the main program starts in the word whose address is equal to the stored PCR value plus 1. Relying on this guarantee, the subroutine returns to the main program by using the following code:

```
SUBX
     -
     -
     -
*    LTF N,PCR
     NOP
     NOP
```

The old value of the PCR is restored, two NOP's are executed, and then the PCR is incremented by 1. The next instruction is located at 1 plus the absolute address of the user's STF PCR,N instruction. This next instruction is left-justified in the word, and, hence, no syllables are left missing.

It may be necessary for the subroutine to use the top of the stack during operation of the subroutine. In that case, the old setting of the subroutine (called old-PCR from here on) may be stored in some limit or index register, and the stack thus freed for processing purposes. The return-code might then be as follows:

```
SUBX  LTF  N,X1
        -
        -
        -
*       STF X1,N
        LTF N,PCR
```

It is important that the load-PCR instruction be the last operational instruction in the subroutine; any instruction following it may not be correctly executed unless it is completely contained in the word which holds the last syllable of the load-PCR instruction.

In the user's calling sequence the UCT SUBX instruction must occupy the same word as the STF PCR,N instruction. If not, then the subroutine will cause a fill of the UCT SUBX instruction, and the subroutine will be caught in a loop. For example, the following code would cause a PCR-loop.

```
TAG     TRS  1ØA,N
        STF  PCR,N
        UCT  SUBX
*       TRS  N,1ØA
```

This code causes the absolute address of TAG to be stored into the top of the stack prior to branching to SUBX. But the UCT SUBX instruction is located at TAG+1, and this is the location to which the subroutine will return, causing a second and invalid execution of the UCT instruction. At this point, the top of the stack would contain the new value for core register 1ØA, and it would not be the right value for the old-PCR. The subroutine would then re-compute its output, and the re-stored PCR would be incorrect. To avoid the difficulty always make sure that the store-PCR and UCT instructions occupy the same word. The following code correctly enters and returns from a PCR-subroutine.

```
        *       STF  PCR,N
                UCT  SUBX
ENTER             -            (next instruction to be executed upon return to main
                                program.)
                  .
                . . .

                SUBX  LTF  N,X1
                        -
RETURN                  -
                        -
                *       STF X1,N
                        LTF N,PCR
```

NON-TABULAR PCT SUBROUTINES

The most elementary form of the PCR-subroutine is one in which all the inputs are stored in thin film (that is, the stack or the TFC) prior to entering the subroutine. Subroutines are sometimes referred to as logical or mathematical functions whose inputs are the "arguments" of the functions, and whose outputs are its "values". This reference assumes that subroutines of whatever sort simply perform a single complex but discernable operation, and that the results (output) of the routine are a function of the data (input) supplied to it. Whether this is precisely so is not as important as the fact that it appears to define the essential characteristic of a simple PCR or SRJ subroutine, one in which all the input (arguments) are stored in thin film prior to branching to the subroutine. Because of the limited capacity of thin film, and the inconvenience of loading it with the "arguments" the inputs to such a subroutine are few in number, and usually involve only one 48-bit register. Examples of such subroutines are numerous: square-root routines, a binary-to-decimal conversion, decimal-to-binary conversions, raising a number to a power, and other mathematical computations. What follows is a binary-to-octal conversion subroutine. It converts a 48-bit register to two full registers (the top of the stack and TFC) of octal (6-bit hollerith) digits. The calling sequence is illustrated by a loading of the TFC with the 48-bit register to be converted.

CALLING SEQUENCE

```
     LTF NUMB,M  TFC
*    STF  PCR,N
     UCT SUB8
      -  (RETURN)
```

SUBROUTINE

```
SUB8    STF     X4,SUB81        Save old contents of X4 and L4
        STF     L4,SUB82
        CLA     N
        LTF     H,X4            Save the old PCR in X4
        LTF     D(+16),L4

SUB8A   FRCD    H,3,H
        FRS     H,3,H
        XLC     +1,X4 LS L4,SUB8A
        LTF     SUB81,X4        Restore the contents of X4 and L4
        LTF     SUB82,L4
*       SSF
        LTF     H,PCR           Restore old-PCR
        SSS
SUB81   OCT     ø               Temporary storage registers for saving
SUB82   OCT     ø               information used by main program.
```

There are several things to be noted about this subroutine. First, it is always courteous and sometimes, a necessity, to save the user's setting of any index or limit registers which the subroutine employs. These, of course, must be restored before leaving the subroutine. This is the subroutine writer's responsibility; the user should not have to bother about saving any limit or index registers which are important to him; the subroutine should do it for him. Secondly, the subroutine input (argument) is stored by the user into the TFC. The subroutine assumes it is there but clears the top of the stack itself, and then, prior to returning to the

main program, leaves the converted 16-digits in the top of the stack and in the TFC. In re-setting the PCR to its old value, the asterisk (*) is not placed before the load instruction but instead, before the SSF instruction; this allows the SSS instruction to be executed before a fill of the PSR is required, then with the PCR restored to its old setting the next instruction to be executed is the one following the UCT SUB8 instruction in the calling sequence. Instead of keeping the old-PCR in the stack and rotating the stack back and forth, the old-PCR could have been temporarily loaded into an index register, say X6, and then stored back into the stack, and from the stack into the PCR. But this would have meant that another index register (or limit register, whichever one might use) would have had to be unloaded and restored as was X4. But this is no inconvenience, since a multiple-thin film store and load could as easily have been accomplished in the subroutine as the original store and load X4 alone. In fact, most subroutines automatically store and load multiple thin film registers. The additional registers are then available, and no harm has been done and no additional time has been taken. Hence, the conversion subroutine could have been written as follows:

```
SUB8    STF   M X4,SUB81          Save contents of X4,X5,X6
        STF   M L4,SUB82          Save contents of L4,L5,L6
        STF   N,X6                X6 = old-PCR
        CLA   H
        LTF   H,X4
        LTF   D(+16),L4
SUB8A   FRCD  H,3,H
        FRS   H,3,H
        XLC   +1,X4 LS L4,SUB8A
        STF   X6,N                Restore old PCR
*       LTF   SUB81,M X4          Restore X4,X5,X6
        LTF   SUB82,M L4          Restore L4,L5,L6
        LTF   N,PCR
SUB81   OCT   Ø
SUB82   OCT   Ø
```

Note that L4, L5 and L6 are saved but only L4 is used, yet execution time for the subroutine is not increased; and note that X6 is stored back in to the stack BEFORE the old contents of X4, X5, and X6 are restored; this is necessary, because the PCR value in X6 would be clobbered by the restored value of X6, and therefore, it must be preserved in the stack. And finally, note that the asterisk is placed before the first restore index and limit instruction; this is because the two instructions contain a total of six syllables, filling one word and the first two syllables of the next, and allowing the last two syllables of the word to hold both syllables of the LTF N,PCR instruction. In this way no incomplete or incorrect instruction is executed, and the main program is safely reached.

TABLE PROCESSING PCR SUBROUTINES

The subroutines so far reviewed have only a small number of inputs or "arguments" and these have been easy to store into thin film. But a subroutine which must process a large set of data cannot receive all of its arguments in thin film. There is not enough room, and the loading of thin film would become a bother. Some other means must be employed to input data to the subroutine.

Rather than input data to the subroutine, the location and organization of the data might be input to the subroutine. Since data is usually arranged in a table, this method normally involves the input to the subroutine of table characteristics, such as the core location of the table, whether the entries in the table are of fixed or variable length and the length of the table expressed either in word-count or entry number. Other characteristics of the table may either be assumed by the subroutine or explicitly provided by the user. In any case, what is directly input to the subroutine is table characteristics and not the data in the table. The table characteristics are much fewer in number than the actual data, and are easy to input to the subroutine; in this way, tables of great size can be processed by subroutines.

Almost any table subroutine of real use becomes rather involved, and the complexities quickly obscure rather than clarify the actual subroutine aspects of the code. A fairly simple but still involved example of a table subroutine is one which deletes zeros from a table of values. The entries in the table are one word numeric values, and the purpose of the subroutine is to delete zeros and repack the table. To better understand the subroutine aspects of the following code, let us first examine a short program which deletes and repacks one table called TAB. This is NOT a subroutine because it operates on one and only one particular table called TAB, and NOT on ANY table located ANYWHERE in core which has one-word entries.

```
START   LTF   NENT,L4
        CLA   N
        LTF   H,X4
        XLC   +Ø,X4 EQ L4,ZZX
ZA      CEQ   H,TAB+X4,ZZ
        XLC   +1,X4 LS L4,ZA
        UCT   ZZX
ZZ      STF   X4,N
        LTF   N,X5
ZZA     XLC   +1,X5 EQ L4,ZZZ
        CEQ   H,TAB+X5,ZZA
        TRS   TAB+X5,TAB+X4
        XLC   +1,X4 LS L4,ZZA
ZZZ     STF   X4,NENT
ZZX     HLT   77
```

To convert this code into a subroutine, many changes would be required, but the principle one would be substituting a variable for the constant TAB, whenever it occurs in the code. A variable in machine code is represented by an index register, the CONTENTS of which varies. A constant is represented by a tag, such as TAB. Hence, one must substitute in the above routine an index register for TAB. This requires that the index register contains the address which is normally stored in an address syllable. TAB functions simply as a symbol for a relative address, and that address is normally stored in the address syllables of the instructions in which TAB appears. Now, if some index register, say X6, were set equal to TAB, and all uses of the symbol "TAB" were replaced by "Ø+X6", then the effect as far as instruction execution was concerned would be the same. The Ø in "Ø+X6", of course, is the value (all zeros) which is stored by the assembler into the appropriate address syllable so that it's effect is nil. Under the present substitution scheme, the address of table TAB is the contents of X6 plus the BAR. The BAR has not changed so we can ignore it, and concentrate on X6. X6 is equal to TAB (the relative address). Hence, Ø+X6+BAR is equal to TAB+BAR, and this is the address that we want.

The only difficulty, now, is to correctly load X6. How is the subroutine to obtain the correct value? The answer is that the user must supply the information in the calling sequence. One possible calling sequence is as follows:

```
    TRS   NENT,H      The user stores the length of the table in the
*   STF   PCR,N       stack.
    UCT   SRZ
.   ADR   TABØ        The relative address of the table is stored in
    -                 the word following the UCT to the subroutine.
```

The "ADR TAB" card causes the assembler to load the relative address of TAB into the word following the UCT to the subroutine. It is clear that the starting location of ANY table could be loaded into the word following the branch instruction. Whichever table tag is stored in that register will be processed by the subroutine; so complete generally in this respect is achieved. The table to be processed can be of any length; the number of entries in the table is stored into the stack by the user prior to jumping to the subroutine. As the subroutine can be entered from any point within the user's program, the subroutine never knows exactly where the register containing the table address is located. But it does know the location of the "UCT SRZ" instruction; it is the old-PCR, which is stored in the top of the stack. It knows, further, that the ADR word is the ond immediately following the UCT instruction. Hence, it knows that the table address for this particular calling sequence is located at old-PCR PLUS 1. Old-PCR+1 gives the ABSOLUTE address of the register containing the relative address of the table to be processed. To fetch the table address, the absolute address must be used; but if this is done, then the BAR setting must somehow be nullified. This could be done by saving the BAR somewhere, then clearing it, and then using the absolute address, but experience has shown that the single best means of nullifying the BAR when dealing with absolute addresses is to load an index register with -BAR (the 2's complement of the BAR) and then indexing the absolute address with that index register; this effectively nullifies the BAR, yet leaves it intact for later use when dealing with relative addresses once again.

Pulling these techniques together, we may modify our earlier code and produce the following subroutine:

```
SRZ     STF     M X4,SRZØ1
        STF     M X8,SRZØ2
        STF     M L4,SRZØ3
        LTF     H,X1Ø              X1Ø = old PCR
        STF     BAR,H              NOTE:  Top of stack is now
        LCM     H,H                       negative value.
        BSU     H,0(1),H           Load X9 with the 2's complement
        LTF     N,X9                 (module - 2^16) of the BAR
        LTF     H,L4               Set L4 = NENT of table
        LTF     1+X1Ø+X9,X6        Load relative address of table
        CLA     N                    into X6. 1+old PCR+(-BAR)+BAR=
        LTF     H,X4                 absolute address of word following
        XLC     +Ø,X4 EQ L4,SRZZZ    UCT SRZ.
SRZA    CEQ     H,Ø+X6+X4,SRZZ     Ø+X6= relative address of the table.
        XLC     +1,X4 LS L4, SRZA
        UCT     SRZZZ
```

```
SRZZ    STF    X4,N
        LTF    N,X5
SRZZA   XLC    +1,X5 EQ L4,SRZZZ
        CEQ    H,Ø+X6+X5,SRZZA
        TRS    Ø+X6+X5,Ø+X6+X4
        XLC    +1,X4 LS L4,SRZZA
SRZZ    STF    X4,N          Store new NENT of table into stack.
        XLC    +1,X1Ø NO LØ,Ø Increment old-PCR by 1 so that
        STF    X1Ø,N            return is at old-PCR+2.
        LTF    SRZØ1,M X4
*       LTF    SRZØ2,M X8
        LTF    SRZØ3,M L4
        LTF    N,PCR
SRZØ1   OCT    Ø
SRZØ2   OCT    Ø
SRZØ3   OCT    Ø
```

X1Ø which is equal to the old PCR is incremented by 1 so that the effective return address is one greater than the word containing the relative address of the table. Remember that a fill of the PSR occurs immediately after the PCR is loaded, and this new setting of the PCR is immediately incremented by 1; hence, the PCR is loaded with the absolute address of the ADR word in the calling sequence but the automatic PCR incrementation, prior to the next fill of the PSR, effectively causes the word following the ADR word to be fetched from memory. This latter word, of course, contains the next instruction of the main program, which is what is wanted. Remember, too, that logically complementing a positive number gives a negative-value, the magnitude of which is incremented by subtracting (a positive one). While this subroutine works well enough, it does contain an inefficiency which could be eliminated. X6 is used to contain a constant - the relative address of the table - which is never modified by the subroutine. X4 is used to step through the table. There is no reason why X6 could not have been used for that purpose, thereby saving X4 for other uses or at least eliminating the extra repetitive fetch of the X4 index-syllables. To use X6 to step through the table, the limit register L4 could simply have been set to the initial value of X6 plus the length of the table. In place of the "LTF H,L4" instruction, we could have put the following:

```
BAD     H,1+X1Ø+X9,N
LTF     H,L4
```

after which all use of X4 could have been dropped; and where X4 appears in XLC and STF instructions X6 could be substituted and X5 would function by itself. Immediately after instruction SRZZ, the relative address in the table would have to be subtracted from the top of the stack in hold mode. This would give the new table length. This modification is included in the next subroutine.

The above zero-delete subroutine operates on one-word-entries. But tables of a more complex kind can be processed by subroutines of this sort. Multi-word entry tables utilizing a one-word item can be processed by the following subroutine. The routine requires that the user supply the word-length of an entry (which must be of fixed length) and the location within the entry of the one-word item for which zero is to be checked. If that one-word item is zero, then the entire entry is deleted from the table, and the table, repacked. The calling sequence and the subroutine are as follows:

CALLING SEQUENCE

```
      TRS    NENT,H              NENT = Number of entries
*     STF    PCR,N
      UCT    SRØ
      DEC    +L                  L = Words in entry.
      DEC    +I                  I = Which item in entry (Ø-N) on which
      ADR    TABØ                      to make zero check.
       -     RETURN
```

SUBROUTINE

```
SRØ   STF    M X4,SRØØ1
      STF    M X8,SRØØ2
      STF    M L4,SRØØ3
      STF    M TFC,SRØØ4
      LTF    H,X1Ø               X1Ø = old-PCR.
      STF    BAR,H
      LCM    H,H
      BSU    H,0(1),H
      LTF    N,X9                X9 = -BAR
      CLA    N
      CEQ    N,H,SRØZZ           Does NENT equal zero?
      CLS    2+X1Ø+X9,1+X1Ø+X9,SRØ1A      Is item register location EQ or
                                          GR to the number of wds/entry.

      UCT    SRØZZ


SRØ1A LTF    3+X1Ø+X9,X6         X6 = Tbl relative adr.
      LTF     2+X1Ø+X9,X8        X8 = item loc. within ent. to check for Ø.
      BMU    H,1+X1Ø+X9,N
      STF    M TFC,H        TOS = NENT  NENT x NWDSEN = TBL LENGTH
      STF    X6,N           Rel. addr. of tbl+tbl len. = addr. of last reg.
      BAD    N,H,N          in tbl = L4.
      LTF    N,L4
      LTF    1+X1Ø+X9,X4         X4 = entry-length.
      XLC    +Ø,X4 EQ LØ,SRØZZ   Check for Ø-length entry error.
      CLA    H
SRØA  CEQ    H,Ø+X6+X8,SRØZ      Check item in entry = Ø.
      XLC    +Ø+X4,X6 LS L4,SRØA       Increment X6 by
      SSF                              entry length and see if X6 =
      UCT    SRØZZ                     last Ø-check word in table.
SRØZ  STF    X6,N
      LTF    N,X5
SRØZA XLC    +Ø+X4,X5 EQ L4,SRØZD
SRØZB CEQ    H,Ø+X5+X8,SRØZA
      XLC    +Ø+X4,X5 NO LØ,Ø    Set L5 = 1st word of next entry to
      STF    X5,N                stop transfers from good entry to
      LTF    N,L5                open entry.
      XLC    -Ø+X4,X5 NØ LØ,Ø
```

```
SRØZC   TRS   Ø+X5,Ø+X6          X8 = -Ø-check location and hence,
        XLC   +1,X6 NO LØ,Ø       initially restores X5 and X6 =
        XLC   +1,X5 LS L5,SRØZC   1st word in entry.
        XLC   +Ø,X5 LS L4,SRØZB   To allow ENTRY MOVE.
SRØZD   STF   X4,N                Compute entry-length X6-start of
        ARC   H,16,H              table and scale it B16 (Binary pt.
        STF   X6,N                after Bit-17).
        BSU   H,3+X1Ø+X9,H        Computer new entry number = new
        BDV   N,H,N               word length/entry length.
        ARS   H,16,H
SRØZZ   XLC   +3,X1Ø NO LØ,Ø
        STF   X1Ø,N
        LTF   SRØØ1,M X4
        LTF   SRØØ2,M X8
*       LTF   SRØØ3,M L4
        LTF   SRØØ4,M TFC
        LTF   N,PCR               Store old-PCR+3 into PCR.
SRØØ1   OCT   Ø
SRØØ2   OCT   Ø
SRØØ3   OCT   Ø
SRØØ4   OCT   Ø
```

Virtually all of the PCR subroutines one might write for the GSA-51 would employ one or more of the techniques illustrated by the above routine. Certain subroutines could exit at different points from the routine, and others might have different error returns built into their calling sequences so that if some error (such as arithmetic overflow, for example) had occurred, during execution of the subroutine, return would not be made to the normal point after the calling sequence but to some specific error location. Other variations might be incorporated into the code of a subroutine; the extent, and complexity, and sophistication of subroutines, in fact, seem to have no limit. However, we are concerned with those considerations when writing subroutines which are unique to the GSA-51. These include the use of thin film, especially the stack, for input data; the fetching of information from a subroutine's calling sequence; the manipulation of data both outside and inside a subroutine, and one method for entering and returning from a subroutine.

"FLOATING" PCR SUBROUTINES

One critical matter in regard to PCR-subroutines has so far been omitted. That has to do with the settings of the BAR and BPR. So far, the settings have been identical for both the subroutine and the main program. But if the main program is long and the subroutines are stored at the end of the program, then the relative addresses used by the subroutine may be too great to be served by the BAR and BPR settings of the main program. The maximum difference that can be allowed between an unindexed absolute address and the BAR is 2^{11}-1 (2047). If any portion of the subroutine is located past address 2047 (decimal) relative to the start of the program, then the subroutine will not operate correctly with the BAR or BPR set equal to the start of the program; the same is true if relative to the main program's last SET BAR or SET BPR card the subroutine addresses exceed 2047.

The only feasible way to overcome this limitation is to "float" the subroutine, that is, make all subroutine instructions contain addresses relative to the start of the subroutine (not the main program). This drastically reduces the magnitude of the subroutine's relative addresses in subroutine instructions and there is then no danger of trying to use, in instructions, relative addresses which are greater than 2047. But floating the subroutine requires

that the subroutine have its own BAR and BPR settings while at the same time utilizing the main program's BAR setting to fetch data from areas outside the subroutine. This involves saving the main program's BAR/BPR settings and re-setting the BAR/BPR during the operation of the subroutine. How this is done and what is involved can best be understood by a pair of examples. First, we will modify our earlier binary-to-octal subroutine from a non-floating to a floating format. The calling sequence is essentiallly the same except that the user cannot directly branch to the floating subroutine (unless it happens to have a relative address less than 4096, in which case, he can make a direct branch). The maximum branch address that can be stored in a branch syllable is 4095; if the subroutine is located at some point beyond address 4095 (relative to the start of the program) then the user cannot branch to it while using his current BPR setting; rather than re-set the BPR, the user might load an index register with the relative address of the subroutine, and use it to index a zero branch syllable. If so, then his calling sequence to the floating binary-to-octal subroutine could look as follows:

```
        LTF     NUMB,M TFC
        LTF     IFSR8,X1        IFSR8 is some register in core
*       STF     PCR,N           which contains the relative address
        UCT     Ø+X1            of FSR8, the conversion subroutine.
         -      RETURN
```

The following conversion subroutine looks as follows:

```
        SET     BAR,FRS8
        SET     BPR,FSR8
FSR8    STF     M BPR,N         Store the user's BAR/BPR into the stack
        STF     PCR,N           Store the absolute address of the sub-
                                routine.
        LTF     H,BAR           Set BAR/BPR equal to the start of
                                the subroutine.
        LTF     N,BPR
        STF     M X4,FSR81
        STF     M L4,FSR82
        TRS     N,FSR83         SAVE user's BAR/BPR.
        LTF     N,X6            Load user's PCR into X6.
        CLA     H
        LTF     H,X4
FSR8A   FRCD    H,3,H
        FRS     H,3,H
        XLC     +1,X4 LS L4,FSR8A
        STF     X6,N            Store old-PCR back into stack
        TRS     FSR83,N         Store old-BAR/BPR into stack
        LTF     FSR82,M L4
        LTF     FSR81,M X4
*       LTF     N,M BPR         Restore old-BAR/BPR
        LTF     N,PCR           Restore old-PCR
FSR81   OCT     Ø
FSR82   OCT     Ø
FSR83   OCT     Ø
        SET     BAR,Ø
        SET     BPR,Ø
```

The SET BAR,FSR8 and SET BPR,FSR8 cards instruct the assembler to store into all the following instructions addresses relative to FSR8, the start of the subroutine. Immediately upon entering the floating subroutine, the user's BAR/BPR are saved, and the absolute location of the subroutine (found by storing the PCR at the start of the subroutine) is loaded into the BAR/BPR. This loading of the BAR/BPR corrects all the following relative addresses by incorporating into the BAR and BPR a value equal to the relative address of FSR8 which when added to the subroutine's relative addresses, increases them by an amount equal to the relative address of FSR8. So the effect of reducing the subroutine's relative address by a value equal to relative-FSR8 is offset by increasing the BAR/BPR by an equal amount. For example, the tag "FSR81" in the subroutine instruction "STF M X4,FSR81" has been replaced, during assembly, by the octal equivalent of FSR81 - FSR8. When this instruction is executed, the BAR has been set equal to the absolute location of FSR8 (the start of the subroutine). If we assume that the entire program which contains this subroutine has been loaded at absolute address 1000_8, then the absolute location of FSR8 is (the relative address) FSR8+1000_8. The BAR, then, is equal to FSR8+1000_8. "FSR81" in the STF instruction is equal to FSR81 - FSR8. The absolute location of FSR81 is equal to FSR81+1000_8. Hence, FSR81 - FSR8 plus the BAR must be equal to FSR81+1000_8 if the instruction is to be correctly executed. This we can see is the case from the following expression:

FSR81+1000_8	=	(FSR81 - FSR8)	+	(FSR8+1000_8) .
absolute location	=	(address in subroutine)		(setting of BAR) .

The relative address FSR81 plus 1000_8 is equal to the relative address FSR81 minus the relative address FSR8 PLUS the relative address FSR8 and 1000_8. The negative and positive addresses FSR8 cancel each other out, and there is left but FSR81 and 1000_8, which is what we wanted. The equation is valid for the other addresses within the subroutine, and it guarantees that the subroutine accesses the correct core locations.

In the conversion subroutine, no use is made of the main program's setting of the BAR or BPR. These are simply saved and restored prior to leaving the subroutine. This was made possible by the fact that all of the input to the subroutine was stored in thin film. But if the input consists of a large set of data located outside the subroutine and not in thin film, then some means of addressing it must be provided for the subroutine. This can be done by storing, in the user's calling sequence, the relative address of a table containing the input data. This was the procedure which was used in calling the zero-delete subroutine. But the relative address so provided was always relative to the user's BAR setting - this did not cause any inconvenience or difficulty to the earlier subroutine since it could use the identical BAR setting, but in a floating subroutine, this is not the case. With a floating subroutine any relative addresses supplied by the user to the subroutine must be modified and made relative to the SUBROUTINE'S BAR setting. This can be done easily enough by first making the table's relative address an absolute address and then decrementing the absolute address by a value equal to the subroutine's BAR setting. The result will usually be a negative number, since the data (table) is often located before the subroutine in the complete program. This negative number, of course, is in 2's complement form, and effectively functions to reduce the BAR setting so as to yield the absolute address. The modifications required to make the one-word entry, zero-delete and repacking subroutine a floating subroutine are not difficult and the result is as follows:

CALLING SEQUENCE

```
      LTF    IFSØ,X1
      TRS    NENT,H
*     STF    PCR,N
      UCT    Ø+X1
      ADR    TABØ
       -     RETURN
```

IFSØ contains the relative address of the subroutine.

TABØ is the relative address (relative to the user's BAR) of the table.

SUBROUTINE

```
      SET    BAR,FSØ
      SET    BPR,FSØ
FSØ   STF    M BPR,N
      STF    PCR,N
      LTF    H,BAR
      LTF    H,BPR
      STF    M X4,FSØØ1
      STF    M X8,FSØØ2
      STF    M L4,FSØØ3
      LCM    H,H
      BSU    H,0(1),H
      LTF    N,X9
      TRS    H,FSØØ4
      FRS    H,16,H
      LAN    H,0(177777),H
      LTF    N,X8
      LTF    N,X1Ø
      LTF    H,L4
      LTF    1+X1Ø+X9,X6
      XLC    +Ø+X8,X6 NO LØ,Ø
      XLC    +Ø+X9,X6 NO LØ,Ø
      CLA    N
      LTF    H,X4
      XLC    +Ø,X4 EQ L4,FSØZZ
FSØA  CEQ    H,Ø+X6+X4,FSØZ
      XLC    +1,X4 LS L4,FSØA
      UCT    FSØZZ
FSØZ  STF    X4,N
      LTF    N,X5
FSØZA XLC    +1,X5 EQ L4,FSØZZ
      CEQ    H,Ø+X6+X5,FSØZA
      TRS    Ø+X6+X5,Ø+X6+X4
      XLC    +1,X4 LS L4,FSØZA
FSØZZ STF    X4,N
      XLC    +1,X1Ø NO LØ,Ø
```

Load BAR/BPR with absolute FSØ

Load 2's complement of subroutine's BAR into X9. X9 = -BAR

Save user's BAR/BPR

Load user's BAR into X8

Load old-PCR into X1Ø

Load NENT of table into L4.

Load address of table relative to user's BAR into X6.

Increment X7 by user's BAR giving absolute addresses.

Decrement X6 by subroutine's BAR giving address relative to subroutine's BAR. Note: X9 = -BAR.

X6 is start of table relative to the BAR.

```
        STF     X1Ø,N           Load old-PCR into stack
        TRS     FSØØ4,N         Load user's BAR/BPR back into stack
        LTF     FSØØ1,M X4
        LTF     FSØØ2,M X8
        LTF     FSØØ3,M L4
*       LTF     N,M BPR
        LTF     N,PCR
FSØØ1   OCT     Ø
FSØØ2   OCT     Ø
FSØØ3   OCT     Ø
FSØØ4   OCT     Ø
        SET     BAR,Ø
        SET     BPR,Ø
```

Similar, but more extensive steps might be taken to modify the multi-word entry delete subroutine so as to make it a floating PCR-subroutine, but these steps are somewhat tedious and they add nothing to an understanding of what is involved in a table-manipulation floating subroutine. Hence, their consideration will be eliminated.

To recapitulate, briefly: PCR-subroutines involve (1) the use and manipulation of the PCR to enter and return from subroutines, (2) the use of absolute addresses with the old-PCR as a base when fetching information from the user's calling sequence, and the consequent use of the 2's complement of the subroutine's BAR in some index register while fetching such information, and finally, (3) the adjustment of addresses relative to the user's BAR to that of addresses relative to the subroutine's BAR, always, assuming, of course, that the BAR/BPR had been previously re-set to the subroutine's starting location.

SRJ SUBROUTINES

SRJ subroutines utilize the SRJ (Subroutine Jump) and SSR (Subroutine Return) instructions. They are also called "external" subroutines in contrast to PCR subroutines which are called "internal" subroutines. This latter terminology reflects the use made of these two sorts of subroutines. PCR subroutines are used mainly for short routines which are internal to the user's program, while SRJ subroutines are usually more elaborate and designed for use by many programs, or by the control program of a system. SRJ subroutines are often not written by the user but are called off a subroutine tape, and incorporated into the user's program. They are "external" to the user's program only in the sense that the user, himself, did not design or code the subroutine, and the subroutine may be employed at different times by any number of different programs. But the distinction between "internal" and "external" subroutines is no more precise than the corresponding one of PCR and SRJ subroutines, and, in fact, may be ambiguous in its implications. Hence, in this document, it will be dropped in favor of the PCR/SRJ designations.

The operational distinction between PCR and SRJ subroutines is that many of the functions performed by program action in a PCR subroutine are performed automatically by the hardware in a SRJ subroutine. Hence, much labor is saved on the part of the programmer when he uses a SRJ subroutine. Because of the automatic hardware functions performed under the SRJ subroutines, much more efficient utilization and control of a BODY of subroutines can be achieved. The PCR subroutine involves much tedious storing and saving of critical thin film registers (such as the PCR, and the various settings of the BAR and BPR); this is reduced in the SRJ subroutine.

The automatic hardware action taken by a SRJ subroutine upon execution of an SRJ instruction includes:

1. Storing the present contents of the BAR,BPR, and PCR into the SSR (thin film addresses Ø5Ø,Ø51, and Ø52, respectively). This is the user's setting of the BAR/BPR and the absolute location of the user's SRJ instruction.

2. The SAR (Subroutine Address Register) is added to the first operand of the SRJ instruction. This first operand is the increment syllable, and contains the value which together with the contents of the SAR yields the location of the address of the start of the subroutine. The increment syllable plus the index plus the SAR equals the location of the absolute address. It is the user's responsibility to store the absolute address of the subroutine into the appropriate location.

3. The present setting of the BAR is incremented by the second operand of the SRJ instruction. This second operand is the BAR increment syllable, which may be indexed. The contents of this syllable (as augmented or decreased by indexing) is added to the current setting of the BAR (prior to branching to the subroutine). Usually what is stored in this second operand is the relative address (relative to the current setting of the BAR) of the table containing the input data or arguments to the subroutine, so that reference WITHIN the subroutine to data stored in the user's data area is made relative to the start of the user's data area (or the table which contains the subroutine's arguments). This greatly facilitates data fetches by the subroutine from outside the subroutine.

4. The absolute starting address of the subroutine is loaded into the BPR and the PCR (not the BAR, of course) upon execution of the SRJ instruction. This absolute address is fetched from the location specified by the user as the first operand of his SRJ instruction. Along with the other automatic hardware functions performed upon execution of the SRJ instruction, one other is performed: the algebraic difference between the new (subroutine) BPR and BAR settings is loaded into X15. This is done automatically, the user has no option in this regard. X15, then, is set equal to new BPR - new BAR. All SRJ subroutines are "floating" subroutines, that is, the symbolic addresses appearing in the subroutine's instructions are all relative to the start of the subroutine (not the start of the main program). Therefore, to fetch constants and to reference temporary storage areas within the subroutine, the BAR effectively must be set equal to the starting address of the subroutine. The new BPR is equal to the absolute start of the subroutine. X15 is equal to the new BPR - new BAR. Hence, indexing a symbolic tag within the subroutine which references a subroutine storage area or constant with X15 effectively increments that tag (or rather its octal equivalent) by the contents of the new BPR. Thus:

$$TAG+X15+BAR = TAG+(BPR-BAR)+BAR = TAG+BPR$$

The BPR is equal to the absolute starting address of the subroutine, so that TAG is being incremented by the absolute starting address of the subroutine, which is what is required.* Of course, the new BAR setting is presumably correct for fetching data stored OUTSIDE the subroutine, and for this purpose X15 is not used.

*As all SRJ subroutines are floating subroutines, all symbolic tags appearing within such subroutines are decremented by the relative address of the start of the subroutine. To compensate for this, the BPR and, effectively, by the use of X15, the BAR are both set equal to the absolute address of the subroutine. A symbolic tag within a floating subroutine is effectively TAG-subroutine start.

TAG - subroutine start plus BPR (or BAR+X15) is equal to
TAG - subroutine start plus subroutine start plus program starting location, which in turn is equal to the absolute location of TAG. See Page 169 for a discussion of floating PCR subroutines.

The format for the SRJ instruction is as follows:

SRJ Ja,Ji

Ja = the address, relative to the current setting of the SAR, of the register which contains the absolute address of the SRJ subroutine. It may be a tag, or a DECIMAL number, and it may be indexed.

Ji = This is the value by which the current BAR setting is to be incremented prior to jumping to the subroutine. It may be a tag or a DECIMAL number, and it may be indexed.

The SRJ instruction is used in conjunction with the SSR (Subroutine Return) instruction. The SRJ effects a branch to the subroutine, and the SSR effects a return from the subroutine to the location immediately after the SRJ instruction which caused the jump to the subroutine. Hence, the instruction following the SRJ instruction should be left-justified in the word as the SRR instruction causes a fill of the PSR (Program Storage Register).

Upon execution of the SRR instruction, the BAR,BPR and PCR are re-set from the contents of the SSR (Subroutine Storage Register). Further, X15 is set equal to the algebraic difference between the restored BPR and the BAR. This allows for the facilitation of successive returns from subroutines which branch to other subroutines during their operation. To branch from one SRJ subroutine to another SRJ subroutine during operation of the first SRJ subroutine, one need only save the current contents of the SSR - so that these settings of the BAR, BPR, and PCR can be restored upon exiting from the parent subroutines - and then branch by means of a SRJ instruction to the next SRJ subroutine. Upon return to the parent subroutine, the difference between the restored settings of the BPR and the BAR is loaded into X15, and the parent subroutine need not recompute the difference.

The SRJ instruction is particularly useful when the programmer wishes to deal with a large body of subroutines. In that case, the SAR (Subroutine Address Register) could be set equal to the absolute starting address of a table of subroutine addresses, and then the appropriate subroutine could be called by its corresponding (decimal) number: subroutine-1, subroutine-2, etc. Further, if one wished, one could index his way through the table of subroutine addresses - he would do this when changing relative addresses to absolute addresses in the table - and reference individual subroutines by means of the index register. Something of the same could be said of the BAR-increment feature of the SRJ instruction. This increment can be the starting address of some table or of the data area of the main program. A table of table-starting addresses (relative to some appropriate setting of the BAR), could be used to successively load an index register, which could augment a BAR-increment syllable (otherwise set to zero), and in this way distinct tables could be processed by the subroutine called in this fashion. Much flexibility and power is incorporated into the SRJ instruction and the associated type of subroutine. These can be utilized as the programmer or subroutine designer wills. Only certain of the many possible applications of the SRJ subroutine will be illustrated. Assuming, then, that we shall be dealing with a body of illustrative SRJ subroutines, let us suppose that the SAR has been loaded with the absolute starting address of a table which contains the relative addresses of these subroutines, and that they are stored in the table in the order in which they are numbered. Hence, SRTAB, the table of subroutine addresses could look as follows:

```
SRTAB   ADR   SRTAB     Relative address of the SR table itself.
        ADR   SRJØ1     Binary-to-octal conversion subroutine.
        ADR   SRJØ2     Edit leading zeros change to blanks.
        ADR   SRJØ3     Conversion and editing subroutine.
        ADR   SRJØ4     Zero-delete table subroutine.
```

The four illustrative subroutines are called SRJØ1-Ø2-Ø3-Ø4, respectively. They perform the following functions: SRJØ1 converts a 48-bit word stored in the TFC from 16 octal digits into 2 words of 16 6-bit hollerith digits stored in the top of the stack and the TFC. SRJØ2 replaces leading zeros with blanks in a 48-bit word of 8 6-bit hollerith digits stored in the TFC. SRJØ3 combines SRJØ1 and Ø2 and allows the user an option as to whether to edit leading zeros or not. SRJØ4 deletes zero values from a one-word entry table and repacks the table. Assuming then, that some large program wishes to utilize these four subroutines (and any others which might be theoretically stored in SRTAB), the following code would establish the correct values in SRTAB and the SAR.

```
START   STF   PCR,N
        LTF   H,BAR
        LTF   H,BPR
        LTF   NSRTB,L1          NSRTB = Number of words in SRTAB
        CLA   N
        LTF   N,X1
        BAD   H,SRTAB+X1,SRTAB+X1
        XLC   +1,X1 LS L1,START+3
        LTF   SRTAB,SAR
```

MAIN PROGRAM

We will illustrate the use of each SRJ subroutine by a sample calling sequence for each, and the actual subroutine code - except that of the edit subroutine, which will be assumed. A few comments on more extended uses of SRJ subroutines will conclude the discussion.

SIMPLE SRJ SUBROUTINES

The following is an example of a simple SRJ subroutine. A simple subroutine is defined as one in which all of the input (arguments) to the subroutine are stored in thin film prior to entering the subroutine. In the case of the simple subroutine, the BAR is not incremented to equal the user's data area, there being none outside of the subroutine or thin film. The BAR can instead be set to the start of the subroutine itself thereby allowing the subroutine to dispense with the use of X15 in fetching its own constants and storage areas. But so setting the BAR is the user's responsibility. If he does do this, X15 should be equal to ZERO, and the subroutine may incorporate a check on this point.

CALLING SEQUENCE (Binary-to-octal conversion)

```
      LTF   NUMB,M TFC      Store number to be converted
      SRJ   1,SRJØ1         BAR is incremented by Relative Address of the
*     -     RETURN          conversion subroutine. SAR+1 = location of
                            subroutine address.
```

SUBROUTINE

```
         SET    BAR,SRJØ1
         SET    BPR,SRJØ1
SRJØ1    XLC    +Ø,X15 EQ LØ,SRJA1        X15 = (BPR-BAR)=Ø
         STF    BPR,N
         LTF    N,BAR
SRJA1    STF    M X4,SRJ11
         STF    M L4,SRJ21
         CLA    H
         LTF    H,X4
         LTF    D(+16),L4
SRJB1    FRCD   H,3,H
         FRS    H,3,H
         XLC    +1,X4 LS L4,SRJB1
         LTF    SRJ11,M X4
         LTF    SRJ21,M L4
SRJZ1    SRR                      USER's BAR/BPR and PCR restored and re-
                                  turn is to SRJ instruction +1.
```

Assuming that a leading zero editing subroutine has been coded in simple SRJ subroutine form, then the following is a binary-to-octal conversion AND editing subroutine. The user of this subroutine indicates that he does not want editing to be done on the converted number by storing an octal 1 in the top of the stack prior to branching to the subroutine. If the user does want leading zeros deleted and replaced by hollerith blanks, then he stores a non-1 in the top of the stack.

CALLING SEQUENCE (Conversion and editing subroutine)

```
         CLA    N                  (Or: TRS 0(1),N, if no editing is desired.)
         LTF    NUMB,M TFC
         SRJ    3,SRJØ3            BAR is incremented by octal equivalent of SRJØ3
*         -     RETURN             SAR+3 = location of subroutine address in SRTAB.
```

SUBROUTINE

```
         SET    BAR,SRJØ3
         SET    BPR,SRJØ3
SRJØ3    XLC    -Ø,X15 EQ LØ,SRJA3    X15 (BPR - BAR) should equal zero.
         STF    BPR,N                 If not set BAR = BPR
         LTF    N,BAR
SRJA3    STF    M X4,SRJ13
         STF    M L4,SRJ23
         CLA    N
         LTF    H,X4
         LTF    D(+16),L4
SRJB3    FRCD   H,3,H
         FRS    H,3,H
         XLC    +1,X4 LS L4,SRJB3
         SSF    +1,X4 LS L4,SRJB3
         CEQ    H,0(1),SRJX3          If 1, do not perform edit.
         SSS
         CLA    N
```

```
        CEQ     N,H,SRJC3       If 1st 8-octal digits zero, then jump
        STF     M TFC,SRJ33     SAVE last 8 digits/edit first 8
        LTF     H,M TFC
        STF     M SSR,SRJ43     Save current contents of SSR.
        SRJ     2,Ø             Branch to editing subroutine - SRJØ2
                                It will correct its own BAR setting.
*       LTF     SRJ33,M TFC     Restore last 8 digits to TFC/1st 8 in stack
SRJD3   LTF     SRJ43,M SSR     Restore return values to SSR
SRJE3   LTF     SRJ13,M X4
        LTF     SRJ23,M L4
        SRR                     Return to main program
SRJC3   STF     M SSR,SRJ43     Save contents of SSR
        SRJ     2,Ø             SRJØ2 will correct the current BAR setting
*       LTF     H,M TFC         Place edited last 8 digits in TFC.
        TRS     H(      ),H     Place blanks for 1st 8 digits in stack.
        UCT     SRJD3
SRJX3   SSS
        UCT     SRJE3
SRJ13   OCT     Ø               Store user contents of X4-5-6.
SRJ23   OCT     Ø               Store user contents of L4-5-6
SRJ33   OCT     Ø               Holds last 8 octal digits, if necessary
SRJ43   OCT     Ø               Holds this subroutine's SSR contents during
                                execution of editing subroutine

        SET     BAR,SRJØ2
        SET     BPR,SRJØ2
SRJØ2    -                      Location of leading zeros editing subroutine -
         .                      stored here in program AFTER SRJØ3, the
         .                      conversion and edit subroutine.
         .
        SRR
        SET     BAR,Ø
        SET     BPR,Ø
```

The point must be stressed that in the above subroutine (subroutine SRJØ3) the use of the tag "SRJØ2" to increment the subroutine BAR setting before branching to SRJØ2 would be valid only if subroutine SRJØ2 has a larger relative address than does subroutine SRJØ3, that is, only if SRJØ2 were located after SRJØ3. This is necessary because the use of tag "SRJØ2" within SRJØ2 would be such that SRJØ3 would be subtracted from "SRJØ2" wherever it appeared in some instruction of SRJØ3's. If SRJØ2 were located before SRJØ3, and, therefore, SRJØ2 were less than SRJØ3, then an assembler error would be generated with zeros stored in the BAR-increment syllable when a negative number (SRJØ2 - SRJØ3), would have to be stored in some address syllable in SRJØ3. One avoids this problem by using Ø in the BAR-increment and having SRJØ2 always adjust its BAR as necessary. But the assembler error so generated would not harm the operation of the object program since zeros would be stored in the appropriate syllables. If one knows that the called subroutine is stored after the calling subroutine, then there is no problem.

NORMAL SRJ SUBROUTINES

All of the previous SRJ subroutines have been simple subroutines; subroutines, all of whose input is stored in thin film. Normal SRJ subroutines are the opposite of simple subroutines in respect to the location of their input: it is not all in thin film, but rather is located

somewhere in the user's main program and is outside of the subroutine. In the case of such normal subroutines, the user must provide the subroutine with the location and organization of the input data. He does so in his calling sequence code. The location within the calling sequence of the input data information is within the control of the subroutine writer, but it is the user who is responsible for properly storing the right information in the correct locations of his calling sequence code.

A normal subroutine makes use of the BAR-increment feature of the SRJ instruction. The user usually increments the BAR by a value equal to the address relative to the BAR of a table containing the subroutine's input data. This allows table processing by the subroutine, and it eliminates the use of an index register to hold the starting location of the table. The information regarding the critical characteristics of the table (table-length, entry-size, location of key-item(s)) can be stored by the user in thin film or in special registers which are convenient to him. If the subroutine is designed so that it has other than the normal exit or return to the main program, the stored value of the PCR in the Subroutine Storage Register (SSR) can be appropriately modified before executing the SSR instruction. This is, if one wished to return to old-PCR plus 2, he would execute these instruction,

"STF Ø52,N",BAD H,0(1),H", "LTF N,Ø52",

which would increment the stored old-PCR in the 3rd 16 bit byte (Ø52) of the SSR; then upon execution of the SSR instruction, the old setting plus 1 of the PCR would be restored to the PCR, and the effective address of the next instruction would be the old-PCR plus 2.

The following is an example of a normal SRJ subroutine. The subroutine deletes zero-entries from a one-word entry table. The structure of the table is simple. What is needed by the subroutine is the length of the table, and the starting location of the table. In his calling sequence code, the user increments the current BAR setting by a value equal to the relative address (relative to the current BAR), and hence the starting location of the table is given by the BAR. The length of the table is stored by the user in the top of the stack.

CALLING SEQUENCE (Zero-delete subroutine)

```
        TRS     NTAB,H              NTAB contains the length of the table.
        SRJ     4,TAB               BAR is incremented by TAB, the
*       -       RETURN              relative address of the table to be
                                    processed. SAR+4 = location of absolute
                                    address of SRJØ4.
```

SUBROUTINE

```
        SET     BAR,SRJØ4
        SET     BPR,SRJØ4
SRJØ4   STF     M X4,SRJ14+X15      Use of X15 corrects BAR so that it
        STF     M L4,SRJ24+X15      effectively equals absolute SRJØ4
        LTF     H,L4                (to which the BPR is set).
        CLA     N
        LTF     H,X4
        XLC     -Ø,X4 EQ L4,SRJZ4   As BPR is correctly set branch
                                    syllables are not indexed
SRJA4   CEQ     H,Ø+X4,SRJB4        BAR = absolute TAB
        XLC     +1,X4, LS L4,SRJA4  so Ø+BAR = TAB
```

```
        UCT     SRJZ4
SRJB4   STF     X4,N
        LTF     N,X5
SRJC4   XLC     +1,X5 EQ L4,SRJZ4
        CEQ     H,Ø+X5,SRJC4
        TRS     Ø+X5,Ø+X4
        XLC     +1,X4 LS L4,SRJC4
SRJZ4   STF     X4,N                    Store new table length in the top
        LTF     SRJ14+X15,M X4          of the stack.
        LTF     SRJ24+X15,M L4
        SSR
SRJ14   OCT     Ø
SRJ24   OCT     Ø
        SET     BAR,Ø
        SET     BPR,Ø
```

If the normal subroutine was capable of processing tables of more complex organization than TAB, then the additional information regarding table characteristics could be stored in additional stack operands, or packed together in one operand. If there is data in addition to the table outside the subroutine which must be fetched, then either the additional data themselves may be stored in thin film if there is room, or if the addition is numerous and organized within a table, then that table's characteristics, especially its location relative to the location of the first table, must be given in thin film to the subroutine. Quite often, if more than one table or if a large set of data only some of which is stored in tables is to be processed by a subroutine, then all of these tables and/or data will be grouped together under a command SET BAR card, so that the subroutine user need only set the BAR equal to the start of that table and data area, and give, in thin film, the addresses relative to that BAR setting of the various tables and data to be processed by the subroutine. The following scheme and sample calling sequence for some complex subroutine, called SRJØM, illustrates the point.

USER'S DATA AREA GROUPED UNDER A COMMON SET BAR CARD

```
        SET     BAR,DATA
DATA    ADRA    TAB1
NTAB1   DEC     +N1, where N1 = length of table expressed in entries.
        OCT     IL, where I = key-item location, and L = size of entry.
          .
          .
          .
ITABM   ADRA    TABM
NTABM   DEC     +NM
        OCT     ImLm
TAB1    OCT     Ø
        DIT     L*N1-1(L*N1 = number of words in table).
          .
          .
          .
TABM    OCT     Ø
        DIT     Lm*NM-1
```

SAMPLE CALLING SEQUENCE

```
*     LTF     ITABI,X1
      LTF     NTABI,L1
      TRS     NTABI+1,N
      LTF     ITABJ,X2
      LTF     NTABJ,L2
      TRS     NTABJ+1,N
      TRS     INFO,N
      SRJ     M,DATA
       -      RETURN
```

Set X1 = TABI's address rel. to DATA
Set L1 = TABI's entry length.
Load stack with TABI's key/item/entry information.
Do likewise for TABJ.

Transmit some constant to the stack
Branch to subroutine SRJØM. DATA is the address relative to the start of the program of the start of the user's table and data area. Since the X1 and X2 contain the addresses of TABI and TABJ relative to DATA, the Ø+X1+ increased-BAR = location of TABI, and similarily for Ø+X2, and TABJ.

CHAPTER 9

INTERRUPT SYSTEM

INTRODUCTION

How often have you sat down intent to study, only to be disturbed by interruption after interruption? It happens to most of us too many times. The intentions of a typical student are pictured in Figure 9-1; prepare to study, do serious study, then go to bed. These desires proved too lofty. Soon after beginning his work, the student was interrupted by the phone. After returning to his studies, he was interrupted by a television show which proved irresistable. Again he returned to his studies, and again the student was interrupted, this time by a knock on the door. No sooner had his visitor left when the student recalled that it was happy hour at the club. He never returned.

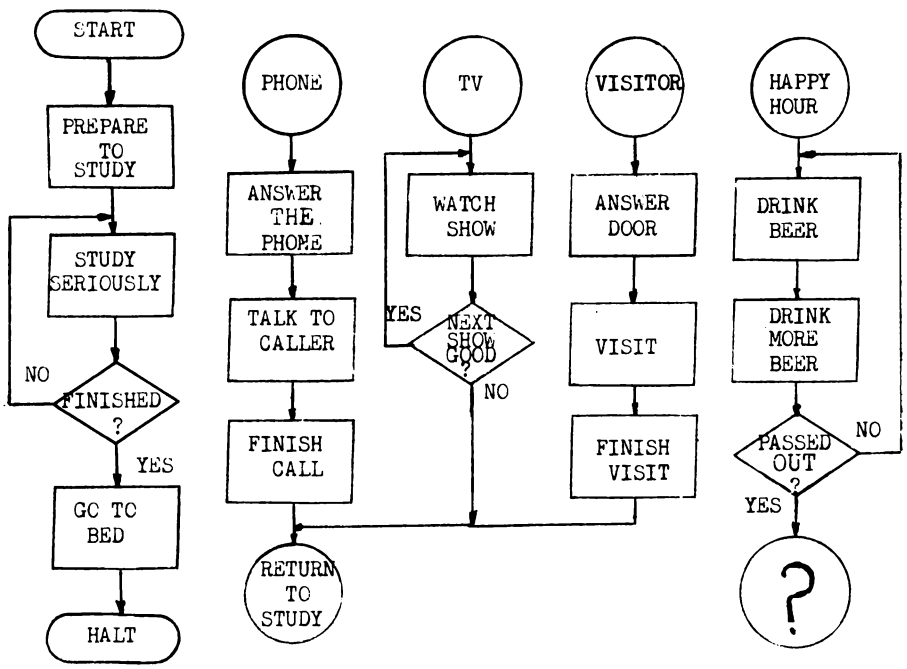


Figure 9-1. The Intentions and Interruptions of a Typical Student

Anyone who tries to follow a schedule, will find themselves interrupted from time to time. Unfortunately interruptions are difficult to ignore. They must be responded to, in some manner, before the original schedule can be resumed. Some interruptions, like a call to happy hour, would make a return to the schedule worthless.

Such is the way of life, and such is the way of the BUIC III computer. Whereas a man is subjected to many types of interruptions, the computer is subjected to, at most, twelve. Some of these result from a program action, others from an external request, and still others from equipment malfunction. Two of the computer interrupts are responded to entirely by the computer hardware and are often termed "pseudo interrupts". With the computer in the normal mode of operation, the remaining ten interrupts must be handled partially by program action.

An interrupt, thru the use of a common hardware routine, causes the computer operations to be transferred to a location (the interrupt table) which is defined in the thin film Interrupt Address Register (IAR). After the hardware routine branches control, there must be a programmed routine ready to respond to the interrupt. Otherwise the results, after the computer branches control, will be unpredictable. The response to the interrupt is the programmers choice. There might be a message printed out or there might be no response other than returning to the interrupted program.

Thus there must be routines in core to respond to EVERY interrupt EACH time a program is operated in the normal mode. This means ten interrupt processing routines. Since these routines can get lengthy and difficult to write, it is nearly impossible to tailor your interrupt responses around each individual program. It is, therefore, convenient and time-saving to use a master CONTROL PROGRAM to process each interrupt. All the large systems use extensive control programs. The BUIC III Air Defense System has COP and the BUIC III utility system has CUE. Both control programs print out complete diagnostics including interrupt type and settings of the BAR, BPR and PCR. Control programs which can read programs into memory, perform I/O, and handle interrupts can save the programmer time and effort.

The BUIC III computer was designed to facilitate the use of such supervisory control programs. The BUIC III computer is designed to operate in either of two modes: CONTROL mode or NORMAL mode. Most programs should be operated in the normal mode. In this mode all twelve interrupt conditions are honored and can be a valuable tool in debugging a program. However, in the normal mode certain instructions are illegal and will cause an interrupt if tried. ALL INTERRUPTS CAUSE the computer TO SHIFT into the CONTROL MODE. In the control mode, all instructions are legal. Thus a program operating in this mode has the added capability of performing I/O, setting certain control registers, and responding to interrupts. The "pseudo interrupts" are handled the same in either mode. However, of the other ten interrupt conditions, three will cause the computer to halt and seven will not be recognized in the control mode.

DESCRIPTION OF SPECIAL INTERRUPT CIRCUITRY

Each computer in the AN/GYK-10 contains special interrupt circuitry for implementing the interrupt capability of the system. This circuitry consists of the following registers:

INTERRUPT AND MASK REGISTER

To provide visual indications of the interrupt conditions, each computer contains a 12-bit Interrupt (I) register and a 21-bit mask register (P and Q registers). Ten of the 12 interrupt conditions are indicated by a bit being set in the I register. The highest priority interrupts, primary power failure and RTC update, are not assigned I register bits. These interrupts are recognized and processed automatically. The two least significant bits of the I register (bits 11 and 12) are spares.

The mask register is used to permit four of the interrupt conditions to be either recognized or ignored by the computer. These interrupts, which are maskable interrupts, are external request, I/O termination, RTC overflow, and arithmetic overflow. If a maskable interrupt is to be recognized by the computer, the bit in the mask register that corresponds to the I register bit for signifying the interrupt must be set.

MEMORY BOUNDS REGISTERS

Each computer contains two memory bounds registers, one for defining the upper limit of the memory write area and one for defining the lower limit of the memory write area. An interrupt condition is generated when these bounds are violated.

INTERRUPT STORAGE REGISTERS

When an interrupt occurs, special interrupt storage registers contained in the thin-film memory provide storage for computer control information relating to the program in progress so that it can be restarted at a later time. These registers consist of the Interrupt Dump Register (IDR), the Interrupt Storage Register (ISR), and the Interrupt Program Register (IPR). These registers are loaded during the Interrupt Jump (IRJ) routine, the automatic hardware routine for transferring computer control to an interrupt service routine. The program control data stored in these registers are returned to the operating registers of the computer by the execution of an Interrupt Return (IRR) instruction, which can only be given in the control mode.

DESCRIPTION OF INTERRUPT CONDITIONS

The following write-ups of the 12 interrupts, listed in the order of priority, contain in the description, the response to the interrupt in both modes, the bit set in the I register and the maskability of the interrupt. A composite of the information given can be found in Table 9-3.

PRIMARY POWER FAILURE

The primary power failure interrupt occurs whenever the primary ac input voltage to the system is detected out of tolerance. Special storage circuits maintain the dc supply voltages at normal levels for 500 microseconds after detection of a primary power failure to ALLOW SUFFICIENT TIME FOR COMPLETION OF THE INSTRUCTION CURRENTLY BEING EXECUTED AND TO STORE ALL NECESSARY INFORMATION REQUIRED FOR RESTARTING THE PROGRAM. The primary power failure interrupt signal bypasses the interrupt register and initiates the automatic power failure dump routine (see Figure 9-2). This interrupt is RECOGNIZED IN BOTH THE NORMAL AND CONTROL MODES. When the interrupt is detected in the normal mode, the processing of this interrupt does not require a transfer to the control mode. In the power failure dump routine, the contents of certain control flip-flops and the I register are stored in the Power Failure Dump Registers (PDR) which consist of thin film registers 064 and 065. The contents of the control flip-flops are stored in bits 1 thru 15 of thin film register 064, and bits 1 thru 12 of the I register are stored in bits 1 thru 12, respectively, of thin-film register 065. The contents of each bit of the PDR are listed in Table 9-1. This information is the same as that stored in the IDR during an IRJ operation. After the power failure dump is completed, the computer is halted.

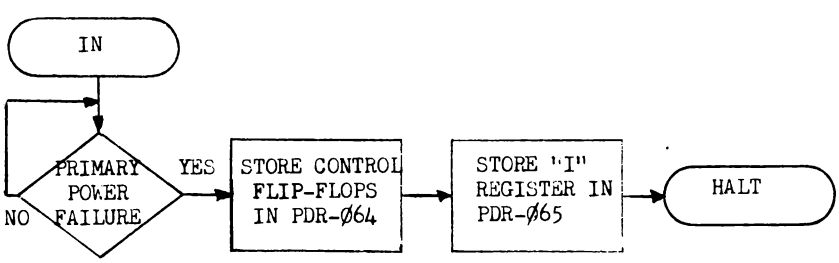


Figure 9-2. Power Failure Dump Routine (Hardware Routine)

TABLE 9-1. Contents of Power Failure Dump Register (PDR-064) and Interrupt Dump Register (IDR)

REGISTER BITS	DESCRIPTION
1, 2, 3	Bits represent states of PS1, PS2, and PS3 flip-flops. These bits indicate the address of the next PSR syllable. This syllable is an operator syllable, since the transfer to control mode can occur only at the end of an instruction. The PSR1 syllables are numbered, from the most significant end, "3," "2," "1," and "0" and the PSR2 syllables are numbered "7," "6," "5," and "4".
4	Bit represents state of RPF flip-flop. This bit is a 1 if a repeated instruction was interrupted.
5	Bit represents state of FRP flip-flop. This bit is a 1 if a repeated instruction was interrupted before execution of the first iteration.
6, 7	Bits represent state of PF1 and PF2 flip-flops. These bits contain 1 for each PSR that still contained information after execution of the last instruction before the interrupt was processed (bit 6 for PSR1 and bit 7 for PSR2). If the last syllable of a PSR was used as the last syllable of the instruction before interrupt, this PSR is empty; if the last syllable was not used, this PSR is filled. If overlap has occurred, the other PSR is filled; otherwise it is empty. When bits 6 and 7 are restored to flip-flops PF1 and PF2 and both of the bits are 1's, one of the flip-flops will be reset, since overlap has been lost.
8	Bit represents state of POV flip-flop.
9	Bit represents state of PUN flip-flop.
10	Bit represents state of PNN flip-flop.
11, 12	Bits contain contents of stack address counter (SA1 and SA2) to indicate which location was at the top of the stack. SA counts of 0, 1, 2, and 3 designate stack locations 1, 2, 3, and 4, respectively.
13	Bit represents state of INT flip-flop. A 1 indicates that computer was operating in control mode when primary power failure was recognized.
14	Bit represents state of IPF flip-flop. A 1 indicates that interrupt condition is a primary power failure.
15	Bit represents state of RSF flip-flop. A 1 indicates reverse operation of the stack.
16	Spare.

A power failure return operation of the hardware will proceed when power is ready (CABINET READY indicator on) and the START push button on the control panel is pressed if the FUNCTION switch is in position OPERATE, CONDITIONAL HALT, SINGLE INSTRUCTION, SINGLE PHASE or SINGLE PULSE (see Figure 9-3). The power failure return routine is also performed when power is first turned on in the equipment and the processing of an object program is initially started. The power failure return operation restores the contents of the control flip-flops and the I register to their pre-power failure status. If the operation actually is a return from a power failure, bit I1 in the interrupt register is set to indicate that a restart after primary power failure interrupt condition exists. Therefore, as soon as normal computer operation is restored, a restart after primary power failure interrupt occurs and the interrupt jump (IRJ) routine is initiated to transfer computer operation to the appropriate interrupt service routine.

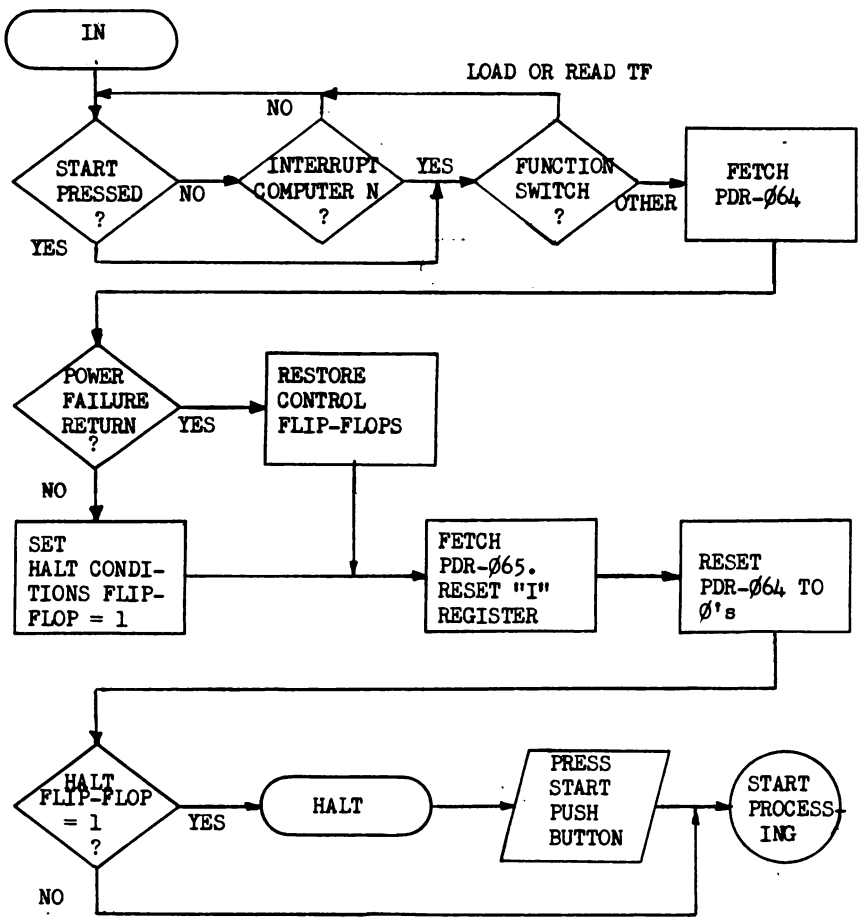


Figure 9-3. Power Failure Return Routine (Hardware Operation)

REAL-TIME CLOCK (RTC) UPDATE

The Real-Time Clock (RTC) update interrupt, which is the second highest priority interrupt, occurs once every 10 milliseconds for the purpose of increasing the contents of the Real-Time Clock (RTC) register in the computer thin-film memory by 1. THE RTC UPDATE INTERRUPT SIGNAL ALSO BYPASSES THE INTERRUPT REGISTER and the interrupt is handled automatically. This interrupt is also RECOGNIZED IN EITHER THE NORMAL OR CONTROL MODES. When detected in the normal mode, no transfer to the control mode occurs for processing this interrupt. However, if an overflow occurs as a result of increasing the count of the RTC register, and if the corresponding mask bit is set, bit 15 of the interrupt register is set. A real-time clock (RTC) overflow interrupt is then detected and processed. The sequence of operations performed in the RTC update routine is illustrated in Figure 9-4.

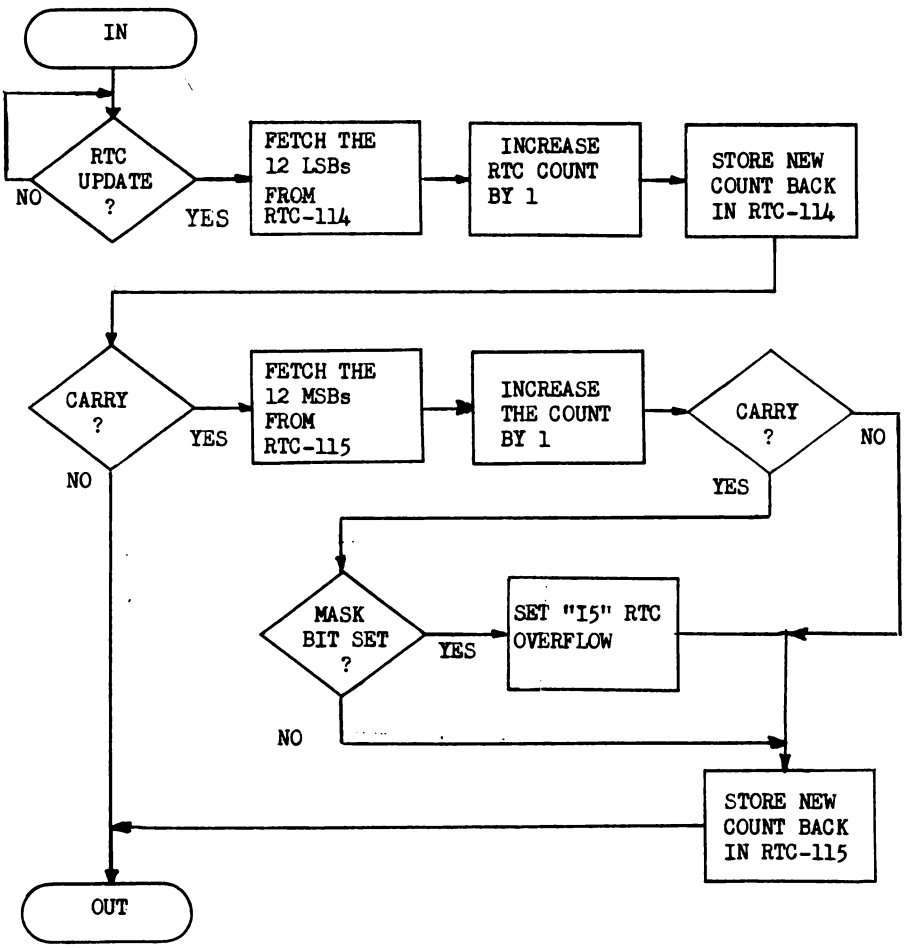


Figure 9-4. Real Time Clock (RTC) Update Routine (Hardware Operation)

RESTART AFTER PRIMARY POWER FAILURE

When the computer is turned on and bit I1 of the I register has been set, a restart after primary power failure interrupt is indicated. THIS INTERRUPT CANNOT BE MASKED AND IS RECOGNIZED ONLY IF THE COMPUTER IS IN THE NORMAL MODE (that is, if operation was in the normal mode when the power failure occurred). In this case, the computer will transfer to the control mode to process the interrupt. If the computer was operating in the control mode when the primary power failure occurred, the computer will continue to operate in the control mode after restarting. HOWEVER, WHEN THE COMPUTER RETURNS TO THE NORMAL MODE, THE RESTART AFTER PRIMARY FAILURE INTERRUPT WILL BE RECOGNIZED.

EXTERNAL REQUEST

An external request interrupt occurs as a result of an external request generated by a terminal device when information (program and data) is to be entered into the system. Sixteen terminal device request lines are available to each computer. The external request interrupt is RECOGNIZED ONLY IN THE NORMAL MODE AND IS MASKABLE. A terminal device can generate an external request interrupt in a computer only if the mask register bit corresponding to the line to which the device is connected is set. If the mask bit is set, bit I2 in the interrupt register is set when an external request interrupt is generated. To determine which terminal device generated the request, a Store External Request (SER) instruction is used during the service routine. THE SER INSTRUCTION CAUSES THE INPUT LEVELS on the 16 external request lines TO BE PLACED IN BITS 33 THRU 48 OF THE MEMORY LOCATION SPECIFIED IN THE INSTRUCTION. Refer to Table 9-2.

TABLE 9-2. Mask Register Bit Assignments and Corresponding Memory Word Bits in LSR and SER Instructions

MASK (P and Q) REGISTER BITS	INTERRUPT BIT MASKED	FUNCTION	LSR MEMORY WORD BITS	SER MEMORY WORD BITS
P1	12 (ER1)	"Flexowriter" ext req	21	33
P2	12 (ER2)	Simulator group ext req	22	34
P3	12 (ER3)	Status display console ext req	23	35
P4	12 (ER4)	Spare	24	36
P5	12 (ER5)	Spare	25	37
P6	12 (ER6)	Spare	26	38
P7 thru P16	12 (ER7) thru 12 (ER16)	Message processor ext req	27 thru 36	39 thru 48
Q3	13	I/O termination for bus A	39	--
Q5	15	RTC overflow	41	--
Q9	19	Arithmetic overflow	45	--
Q12	13	I/O termination for bus B	48	--

I/O TERMINATION

At the termination of an I/O operation, a result descriptor is returned to the descriptor list in memory. A signal is then generated by the I/O module and sent to all computers to indicate the I/O termination and the return of the result descriptor. This signal causes interrupt register bit 13 to be set if the corresponding mask register bit is set. THE I/O TERMINATION INTERRUPT CAN ONLY BE RECOGNIZED IN THE NORMAL MODE. TWO MASK BITS are associated with this interrupt. Mask register bit Q3 masks the I/O termination signal from all I/O modules on bus A, and mask register bit Q12 masks the I/O termination signal from all I/O modules on bus B. Both mask bits control the setting of bit 13 in the interrupt register.

INTERRUPT COMPUTER N

An interrupt computer N condition is normally generated by one computer and sent to another computer. It can be used to start a halted computer or to direct an operating computer to a special processing activity that is to be initiated. This interrupt condition, which is indicated by the setting of bit 14 of the interrupt register, CANNOT BE MASKED AND INVOLVES ONLY THE INTERRUPTED COMPUTER. Bit 14 in the interrupt register is SET IN EITHER THE CONTROL MODE OR THE NORMAL MODE; the interrupt can be recognized by an operating computer only in the normal mode. However, the computer N interrupt can be used to start a computer which is halted in either the normal mode or the control mode. The interrupt computer N signal is generated by a computer operating in the control mode by executing an LSR A1, INTER instruction, which sets interrupt register bit 14 of the computer specified in the A1 syllable of the instruction. The specified computer will recognize that an interrupt condition exists and, if halted will begin operation in the normal mode. The operation of the interrupted computer is automatically transferred to the interrupt table to service the interrupt. It should be noted that the computer specified in the A1 syllable of the LSR instruction can be the computer executing the LSR instruction.

REAL-TIME CLOCK (RTC) OVERFLOW

The Real-Time Clock (RTC) overflow interrupt IS A MASKABLE INTERRUPT which can be used to indicate a specific program time duration of initiate real-time operations. An RTC overflow interrupt will occur after an RTC update routine in which the capacity of the real-time clock has been exceeded. If mask register bit Q5 is set, an RTC overflow interrupt will be signaled when the overflow occurs. The RTC overflow interrupt condition is ONLY RECOGNIZED by the computer when it is operating in the NORMAL MODE. The RTC register may be preset to a predetermined count to establish any required time interval between RTC overflow interrupts.

WRITE OUT OF BOUNDS

The write out of bounds interrupt condition is used to prevent the writing of data into memory areas not assigned to the program or into memory areas within the program in which constants are stored. The write out of bounds interrupt cannot be masked and is RECOGNIZED ONLY IN THE NORMAL MODE. The interrupt occurs when an attempt is made to write into memory areas outside the bounds specified by the lower and upper limit registers (Y and X) or if an attempt is made to use the LTF instruction to load the Interrupt Address Register (IAR) in the normal mode. When this interrupt occurs, the store operation that would normally complete the execution of the instruction that is currently being executed is not performed, so that THE CONTENTS OF THE SPECIFIED MEMORY LOCATION (thin-film or core) ARE NOT DESTROYED.

ILLEGAL INSTRUCTION

An illegal instruction during a normal mode operation IS THE USE OF A CONTROL MODE INSTRUCTION OR A NONEXISTENT OPERATION CODE. Instructions which are not permitted in normal mode operation are LSR, TIO, and IRR. The nonexistent operation codes, in octal, are as follows: 07, 13, 17, 23, 27, 33, 37, 53, 57, 73, and 77. When an illegal instruction is decoded in the normal mode, an interrupt occurs, and a transfer to a control mode routine is performed. IN THE CONTROL MODE, this interrupt OCCURS ONLY WHEN A NONEXISTENT OPERATION CODE IS USED AND CAUSES THE COMPUTER TO HALT. An illegal instruction interrupt causes bit 17 of the interrupt register to be set. This interrupt CANNOT BE MASKED.

PARITY ERROR

A parity error interrupt, which is indicated by the setting of bit 18 in the interrupt register, CANNOT BE MASKED and occurs whenever a parity error exists in a word read from memory. Internal parity is checked every time a data word or program word is read from memory. On each instance of memory write, the parity bit is appended to the word if necessary to make the word contain an odd number of 1's. If a parity error occurs DURING CONTROL MODE OPERATION, THE COMPUTER WILL HALT. If the error condition occurs during NORMAL MODE OPERATION, THE INSTRUCTION IS COMPLETED while using the bad data; however, the interrupt is sensed and processed prior to beginning the next instruction. The I/O module has a SEPARATE parity-checking circuit which is NOT CONNECTED to the interrupt system. Thus, a parity error in any transfer between the core memory and an I/O module will not cause a parity error interrupt to occur in the computer.

NO ACCESS TO MEMORY

The no-access to memory interrupt condition also CANNOT BE MASKED and is detected by bit 18 of the interrupt register, and in the parity error interrupt condition. This interrupt condition is signaled whenever the core memory request flip-flop (REF) in the computer is set (indicating that an attempt is being made to gain access to core memory) and the computer cannot gain access to core within two RTC updates (10 to 20 milliseconds) to complete the execution of the instruction. If the no-access to memory interrupt condition occurs DURING CONTROL MODE OPERATION, THE COMPUTER HALTS; in normal mode operation, this interrupt condition is processed after the instruction involved has been completed WITHOUT THE DESIRED MEMORY WORD.

INDIRECT ADDRESS LOOP

The indirect address loop interrupt SHARES BIT 18 of the interrupt register with the parity error and no-access to memory interrupts and THUS also CANNOT BE MASKED. This interrupt condition is signaled whenever the Indirect Address Flip-Flop (IAF) is set and the computer continuously addresses core memory indirectly for two consecutive RTC update periods (10 to 20 milliseconds) and does not proceed with the execution of the instruction. Since interrupt register bit 18 is also used to detect this interrupt condition, the SAME RULES for normal and control mode operations apply.

ARITHMETIC OVERFLOW

The arithmetic overflow interrupt condition IS MASKABLE and is detected when the program overflow (POV) flip-flop is set during NORMAL MODE operation. This interrupt

condition is indicated by the setting of bit 19 of the interrupt register. A MASK BIT corresponding to the arithmetic overflow MUST also BE SET TO RECOGNIZE THIS INTERRUPT condition. During CONTROL MODE operation, this interrupt is BYPASSED. When the POV flip-flop is set, it remains set until the branch on condition (BRC) instruction is used to reset it or until interrupt register bit 19 is set to signal the overflow condition. The following conditions cause the POV flip-flop to be set:

1. Fixed-point arithmetic overflow resulting from addition, subtraction, or division.

2. Overflow resulting from a round operation in the transmit modified (TRM) instruction.

3. Exponent overflow resulting from a floating-point arithmetic operation.

4. Quotient overflow resulting from use of the floating-divide (FDV) instruction with non-normalized operands.

NORMAL MODE HALT OR SNAG BIT

The normal mode halt or snag bit interrupt is indicated by the setting of bit I10 of the interrupt register. The halt (HLT) instruction, used in the NORMAL MODE, causes the normal mode halt interrupt condition and consequent transfer of computer operation to the control mode. When given in the CONTROL MODE, THE HLT INSTRUCTION HALTS the computer. The snag bit interrupt, which uses the same interrupt register bit as the normal mode halt interrupt, occurs only during indirect addressing. During indirect addressing, if the 18th least significant bit of any level indirect address after the first contains a one, the interrupt register bit I10 is set and the interrupt is recognized upon completion of the instruction involved. The snag bit interrupt condition capability can be employed to implement a program lockout for maintaining control of the execution of certain program areas.

DESCRIPTION OF AUTOMATIC INTERRUPT PROCESSING

The processing of interrupt conditions in the BUIC III computer begins upon their detection. Following detection, one of the following three automatic hardware responses will result:

1. Power failure dump routine, after which the computer is halted.

2. Real-time clock (RTC) update routine, after which the processing of the program in progress is continued.

3. Interrupt jump (IRJ) routine, which transfers computer operation in the control mode to an entry in an interrupt transfer table.

DETECTION OF INTERRUPT CONDITIONS

Prior to executing each instruction, a test is made for the presence of an interrupt condition which may have occurred during the execution of the previous instruction. At that time it is determined if one of the two highest priority interrupts have occurred or if an interrupt which causes a bit in the I register to be set has occurred. This determination

TABLE 9-3. BUIC III Interrupt Conditions

Priority	Interrupt Condition	I Register Bit No.	Mask Register Bits	Mode in Which I Register Bit Set	Mode in Which Recognized	Action upon Recognition
1	Primary power failure	NA	NA	NA	Normal or control	Power failure dump and return routine
2	Real-time clock(RTC) update	NA	NA	NA	Normal or control	Real-time clock (RTC) update routine
3	Restart after primary power failure	I1	0	Normal or control	Normal only	Interrupt jump routine
4	External request	I2	P1 thru P16	Normal only	Normal only	Interrupt jump routine
5	I/O termination	I3	Q3 and Q12	Normal or control	Normal only	Interrupt jump routine
6	Interrupt computer N	I4	0	Normal or control	Normal only (except when halted; then normal or control)	Interrupt jump routine
7	Real-time clock (RTC) overflow	I5	Q53	Normal or control	Normal only	Interrupt jump routine
8	Write out of bounds	I6	0	Normal only	Normal only	Interrupt jump routine

TABLE 9-3. BUIC III Interrupt Conditions (Cont'd)

Priority	Interrupt Condition	I Register Bit No.	Mask Register Bits	Mode in Which I Register Bit Set	Mode in Which Recognized	Action upon Recognition
9	Illegal instruction	I7	0	Normal or control	Normal or control	Normal mode: interrupt jump routine control mode: halt
10	Internal parity error, no access to memory, or indirect address loop	I8	0	Normal or control	Normal or control	Normal mode: interrupt jump routine control mode: halt
11	Arithmetic overflow	I9	Q9	Normal only	Normal only	Interrupt jump routine
12	Normal mode halt or snag bit	I10	0	Normal for halt.	Halt: normal or control	Normal mode: Interrupt jump routine Control mode halt
				Normal or control snag bit	Snag bit: normal only	Snag bit: interrupt jump routine
	Spare	I11	Q11			
	Spare	I12				

is made automatically. (See Figure 9-5). If the interrupt is one of the highest priority interrupts, the power failure dump routine or the RTC update routine is performed as required. If the interrupt conditions is indicated by the setting of an I register bit and the computer is operating in the normal mode, the interrupt jump (IRJ) routine is performed to detect which interrupt condition has occurred, to transfer computer operation to the control mode, and to effect a transfer to an appropriate service routine. All three of these routines are performed automatically by the computer hardware.

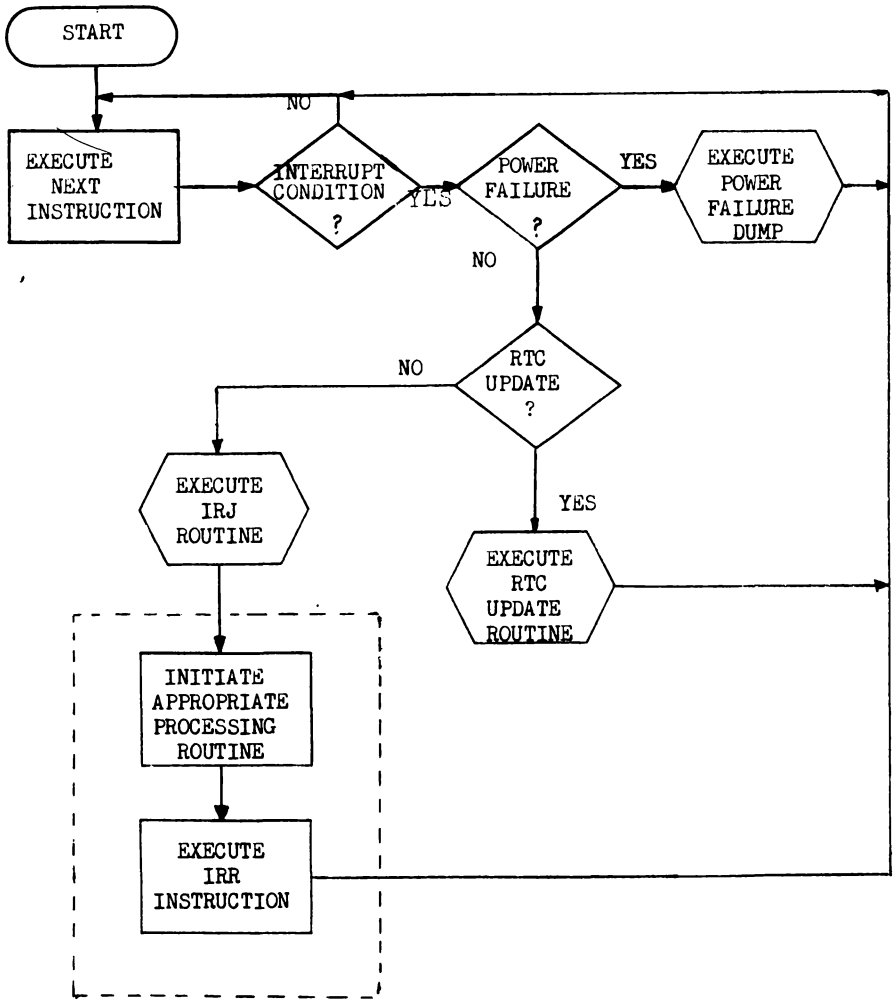


Figure 9-5. General Interrupt Processing Flow Diagram

INTERRUPT JUMP (IRJ) ROUTINE

All interrupt conditions except the primary power failure and RTC update interrupts are processed by routines which are addressed through the interrupt jump (IRJ) operation of the hardware. The IRJ routine is initiated when an interrupt other than the primary power failure interrupt or the real-time clock (RTC) update interrupt is detected in THE NORMAL MODE of computer operation. A flow diagram of the functions performed in the IRJ routine is presented in Figure 9-6.

The first function accomplished by the IRJ routine is the store of all control data relating to the program in progress. This control information consists of the contents of certain control flip-flops and the contents of the Program Storage Register (PSR), the Base Address Register (BAR), the Base Program Register (BPR), and the Program Count Register (PCR). The contents of the control flip-flops are stored in the Interrupt Dump Register (IDR)

as indicated in Table 9-1. The contents of the PSR are stored in the least significant section of the Interrupt Storage Register (ISR-040), the contents of the BPR are stored in the next higher order section (ISR-041), and the contents of the PCR are stored in the most significant section of the ISR (ISR-042).

The aforementioned control data must be stored because the processing of the interrupt condition requires the use of the computer control flip-flops and registers. After the interrupt condition has been processed, this control data is restored by the execution of an IRR instruction so that the program can be resumed AT THE POINT AT WHICH IT WAS INTERRUPTED. After the control information has been stored, the contents of the Interrupt Address Register (IAR) are placed in the BAR and the BPR. The address of a specific entry in an interrupt table (which may contain transfer instructions for branching the computer operation

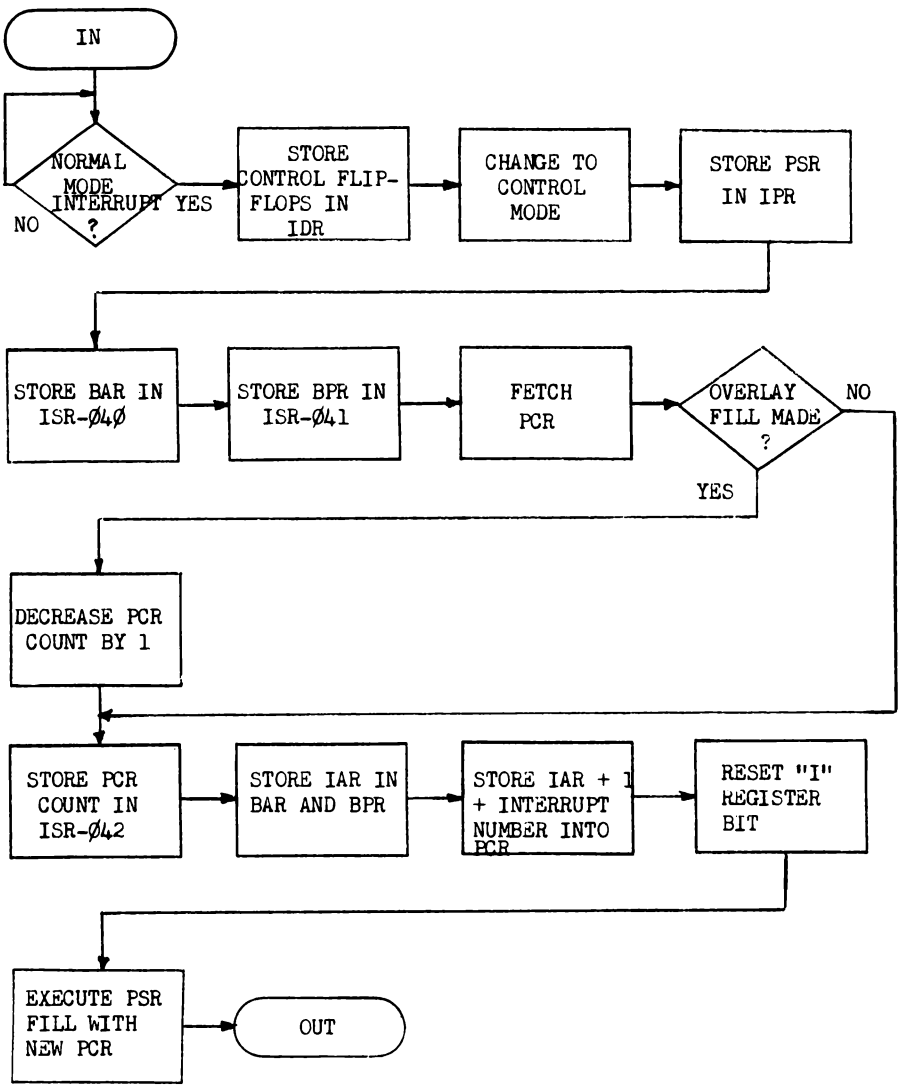


Figure 9-6. Interrupt Jump (IRJ) Routine, Flow Diagram

to appropriate service routines) is then computed by adding the encoded output of the interrupt register (the interrupt number) to the contents of the IAR. This address is inserted into the program count register (PCR). The IRJ routine is now completed, and the computer will fill the PSR using the new contents of the PCR. Thus, the IRJ routine transfers computer operation in the control mode to a fixed location equal to the IAR PLUS THE INTERRUPT NUMBER PLUS 1.

The entire interrupt response operation thus far has been performed automatically by the logic circuitry of the computer (that is, no specific program has been in control of the computer). The subsequent program response to the interrupt condition is initiated as a result of the addresss inserted into the BPR and PCR by the IRJ routine. When the fill occurs, the entry from the interrupt table is loaded into the PSR and is executed. It should be noted that computer operation will remain in the control mode until an interrupt return (IRR) instruction is executed.

DESCRIPTION OF CONTROL MODE OPERATION

As mentioned previously, one function of the interrupt system is to transfer computer operation from the normal mode to the control mode so that instructions and computer responses which can be implemented only in the control mode may be used.

INSTRUCTIONS PERFORMED IN CONTROL MODE

A computer operating in the control mode is capable of executing any of the 53 basic types of instructions available, including three instructions (TIO, LSR, IRR) which are not available in the normal mode. In addition, two instructions (HLT and LTF) have features which can be used in the control mode but which are not available in the normal mode.

TRANSMIT TO INPUT-OUTPUT (TIO). The transmit to input-output instruction is used to set up, initiate, and control input-output operations. The TIO instruction is used to send the three types of I/O descriptors to the I/O modules.

LOAD SPECIAL REGISTER (LSR). The LSR instruction has three variations that enable the loading of the interrupt mask register, the loading of the memory bounds register, or the interrupting of another computer in the system.

INTERRUPT RETURN (IRR). The interrupt return instruction is used to change computer operation from the control mode to the normal mode. During the execution of this instruction, the contents of the registers that are used to store program control information (ISR, IPR, and IDR) are restored to the appropriate program control registers in the computer. Thus, at the completion of the IRR instruction, computer control is returned to the program which was being executed when the interrupt occurred. The first instruction that is executed when control is returned to the original program is the instruction immediately following the one that was being executed when the interrupt occurred; this instruction is specified by the restored contents of the program storage register (PSR1 or PSR2), the syllable counter (PS1, PS2, and PS3), the program fill flip-flops (PF1 and PF2), and the program count register (PCR).

HLT. The halt instruction, when given in the control mode, is interpreted differently by the computer than when given in the normal mode. In the normal mode, the halt instruction causes an interrupt condition, and the computer is placed in the control mode. In the control mode, the computer halts upon decoding a halt instruction. The computer can then be restarted by pressing the START push button on the computer control panel.

LTF. The LTF instruction, when given in the control mode of operation, can be used to load any of the thin-film registers in the computer. However, in the NORMAL MODE, THE INTERRUPT ADDRESS REGISTER (IAR) CANNOT BE LOADED by means of this instruction. The contents of the IAR are protected in the normal mode so that they will not be inadvertently altered. An attempt to load this register in the normal mode will result in a write out of bounds interrupt condition.

COMPUTER RESPONSE TO INTERRUPTS IN CONTROL MODE

The responses to interrupt conditions differ in the normal and control modes. All interrupt conditions are recognized in the normal mode. However, with the exception of the primary power failure and RTC update interrupts, no interrupts are recognized in the control mode. The control mode is thus an interrupt-protected mode of operation. However, if an illegal instruction, parity error, no-access to memory, or indirect address loop interrupt occurs in the control mode, the appropriate I register bit is set, and the computer is halted. If an HLT instruction is given in the control mode, the computer is also halted, but the I register bit is not set. These interrupts are thus recognized in the control mode, but no transfer of computer operation to service routines is performed upon their detection as in the normal mode. The write out of bounds interrupt condition is not recognized in the control mode; therefore, all areas of the core memory may be addressed. The arithmetic overflow interrupt condition and the external request are also completely ignored in the control mode.

All other interrupt conditions occurring in the control mode cause the appropriate I register bit to be set. The interrupt will not be recognized, however, until the computer operation is returned to the normal mode. Computer responses to interrupt conditions are listed in Table 9-3 in the following columns: "Mode in Which I Register Bit Set," "Mode in Which Recognized", and "Action upon Recognition".

PROGRAMMING REQUIREMENTS FOR INTERRUPT RESPONSE

In addition to writing appropriate service routines for handling the interrupt conditions, the programmer must include in the main program sufficient coding for performing the following tasks to prepare the interrupt system for use.

ESTABLISHING THE INTERRUPT TRANSFER TABLE

Establish an interrupt transfer table in the program and load the starting address of this table into the IAR.

STRUCTURE OF TABLE. The interrupt transfer table consists of a list of instructions thru which computer control is transferred to a processing routine for response interrupt condition. Normally, a transfer instruction is provided for all the interrupt conditions except the primary power failure interrupt, the RTC update interrupt, and any maskable interrupt for which a mask bit has not been set. The transfer instructions in the table are listed in the order of the priority of the interrupts as indicated in Table 9-3. The selection of the appropriate transfer instruction from the table is performed by the interrupt jump (IRJ) routine.

A fill occurs following the IRJ routine. The address of the selected entry in the interrupt transfer table is equal to the address in the IAR plus the interrupt number plus 1. Therefore, THE ADDRESS OF THE FIRST ENTRY IN THE INTERRUPT TABLE IS GREATER BY 2 THAN THE BASE ADDRESS PLACED IN THE IAR. The format of the interrupt table is

illustrated in Figure 9-7. Each entry in the table consists of one memory word, which can contain any instruction (transfer or otherwise) for initiating the program response for the interrupt condition.

LOADING THE INTERRUPT ADDRESS REGISTER (IAR). The interrupt address register (IAR) can only be loaded in the control mode by the execution of an LTF instruction which specifies the IAR. The least significant 16 bits of the contents of the memory location from which the IAR is loaded must contain the absolute address of the start of the interrupt table.

INTERRUPT REGISTER BIT NO.	MEMORY ADDRESS (OCTAL)	ENTRY IN INTERRUPT TABLE
--	IAR+0000	Unused
--	IAR+0001	Unused
I1	IAR+0002	Restart after Power Failure
I2	IAR+0003	External Request
I3	IAR+0004	I/O Termination
I4	IAR+0005	Interrupt Computer N
I5	IAR+0006	RTC Overflow
I6	IAR+0007	Write Out of Bounds
I7	IAR+0010	Illegal Instruction
I8	IAR+0011	Parity Error, No Access to Memory, or Indirect Address Loop
I9	IAR+0012	Arithmetic Overflow
I10	IAR+0013	Normal Mode Halt or Snag Bit

Figure 9-7. Format of Interrupt Table

RESPONSE TO MASKABLE INTERRUPTS

Computer response to maskable interrupts is controlled by the setting of the mask (P and Q) registers. The mask registers can only be loaded by means of an LSR instruction, which must be executed in the control mode. The assembly language symbolic instruction for loading the mask register is LSR A_1, MASK.

The contents of the memory location specified by A_1 in the LSR instruction must contain a 1 in the appropriate bit position if the corresponding mask register bit is to be set and must contain a 0 if the mask is not to be set. The correspondence between the bits in the A_1 memory

word, the mask (P and Q) register bits, and the interrupt register bits is shown in Table 9-2. This table also contains similar information regarding the A_1 memory word for the SER instruction, which is used for storing the contents of the external request lines. The bits corresponding to the 16 external request lines are placed in bit locations 21 thru 36 of the memory word. These bits are placed in bits P1 thru P16 by the LSR A_1, MASK instruction. The bits for the remaining three interrupts are also indicated in Table 9-2 and are also placed in the indicated Q register bits by the execution of the LSR instruction. Two mask bits, Q3 and Q12, are used with the I/O termination interrupts. Bits Q3 provides a mask for a bus A I/O termination interrupt, and bit Q12 provides the mask for the bus B I/O termination interrupt. The corresponding mask bit must be set for the associated interrupt bit to be recognized.

SETTING OF MEMORY WRITE BOUNDS

The upper and lower boundaries of the memory data write area used by a computer program must be established if the program is to be run in the normal mode. The bounds are used to prevent an inadvertent write operation into a memory area used by another program or into a portion of the object program's memory area which must not be destroyed.

The upper limit of the data write area is established by the setting of the X register, and the lower limit is established by the setting of the Y register. Each of these registers contain eight bits. During every write operation, the four most significant bits of the 16-bit absolute memory address (the memory module number) are contained in the L register. The next four bits (of 16) are the four most significant bits of an address IN THE MEMORY MODULE. These bits are contained in M register bits 1 thru 4. A write out of bounds interrupt occurs whenever the value of the eight L and M register bits is greater than the value of the X register (upper limit exceeded) or less than the value of the Y register (lower limit exceeded).

The lower memory boundary is established by placing into the Y register the eight most significant bits of the absolute address of the STARTING LOCATION of the data write area. The upper memory boundary is established by loading into the X register the eight most significant bits of the absolute address of the LAST LOCATION of the data write area. In many cases, the aforementioned actions will result in the same setting for both the X and Y registers since only the eight most significant bits of the address are used. Since the eight least significant bits in the M register are not compared, THE UPPER LIMIT MAY BE EXCEEDED BY AS MUCH AS 255 MEMORY LOCATIONS BEFORE A WRITE OUT OF BOUNDS CONDITION IS INDICATED. The lower memory boundary is violated whenever any address less than the setting of the Y register is used. However, if the start of the data write area (the Y register setting) cannot be located at an address which is a multiple of 256, some portion of the program preceding the data write area will be within the write bounds. The number of locations in these buffer areas can be reduced if the data write area is started at an address which is equal to a multiple of 256 or is ended at a multiple of 256 (with the upper limit set to the multiple of 256 minus 1). In general, if the data write area is 256 words or less, the optimum memory bounds setting is with the X and Y registers both set to the starting address of the data write area (at a multiple of 256 if possible). The size of the unprotected or overprotected buffer areas is directly proportional to (1) the relationship of the starting address of the data write area to a multiple of 256 and (2) to the size of the data write area.

The loading of the limit registers is accomplished by performing the LSR A_1, BOUND instruction. The eight least significant bits of the memory location specified by A_1, in the LSR instruction are loaded into the lower limit (Y) register, and the eight next higher-order

bits are loaded into the upper limit (X) register. Hence, the data word must be constructed by the programmer so that the eight most significant bits of the lower boundary are contained in the eight least significant bits of A_1 and the eight most significant bits of the upper boundary are contained in the eight next higher-order bits.

PERFORMING INTERRUPT SERVICE ROUTINES IN NORMAL AND CONTROL MODES

The service routine for processing interrupts may be performed in either the control mode or the normal mode. Access to the service routines is obtained thru the interrupt transfer table. As stated previously, no interrupts signaled by the setting of the I register bits are recognized in the control mode; however, some conditions cause the computer to halt in the control mode. If the interrupt service routine is performed in the control mode, the contents of the ISR, IPR, and IDR cannot be destroyed, since no other interrupt jump (IRJ) routine can occur.

If some portion of the service routine is to be performed in the normal mode, however, the contents of the ISR, IDR, and IPR must first be saved in some locations in the core memory. The return to the normal mode is then made simply by loading the PCR portion of the ISR (ISR-042) with the address minus 1 of the desired return point, clearing the IDR, and then giving an IRR instruction. If a second interrupt now occurs during the normal mode processing of the first interrupt, the original contents of the interrupt storage registers are not lost when the IRJ routine causes new information to be placed in these registers. At the end of the second service routine, the computer must be returned to the control mode by use of a HLT instruction, and the program control data stored in the core memory locations are then placed back in the interrupt storage registers. An Interrupt Return (IRR) instruction is then executed so that the control information is loaded back into the appropriate registers and flip-flops, and the processing of the object program is restored at the point at which the first interrupt occurred.

The programmer may require that the first interrupt routine be completed before the service routine for the second interrupt condition is performed. This may be accomplished by the use of appropriate instructions in the service routines associated with the interrupts. The second service routine could contain a test for determining whether the program interrupted was an interrupt service routine (for example, by testing the contents of a flag word set at the beginning of the first service routine). If the test indicates that an interrupt service routine was interrupted, the PCR count at which the second service routine is to begin will be stored in the core memory, and an IRR instruction will be performed to return to the normal mode at the point at which the first service routine was interrupted. At the completion of the first service routine, the PCR count will be fetched from memory and placed in the PCR and thus transfer computer operation to begin the service routine for the second interrupt condition. After the second routine is completed, the ISR, IDR, and IPR contents stored in the core memory when the first service routine was transferred to the normal mode are returned to their respective registers. The computer is then placed in the control mode, in which an IRR instruction is given to return to the point of the original interrupt in the main program. This scheme can be carried out for as many interrupt conditions as necessary.

INDEX OF INSTRUCTIONS

Instructions are indexed alphabetically according to their mnemonic code. Each code designation is followed by the full name of the instruction and its octal code.

SAVE A LIFE

If you observe an accident involving electrical shock,
DON'T JUST STAND THERE - DO SOMETHING!

RESCUE OF SHOCK VICTIM

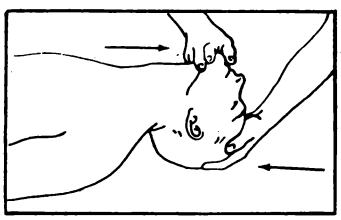
The victim of electrical shock is dependent upon you to give him prompt first aid. Observe these precautions:

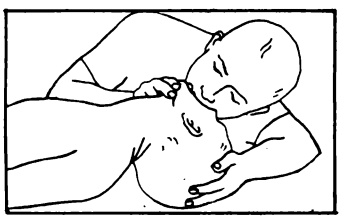
1. Shut off the high voltage.
2. If the high voltage cannot be turned off without delay, free the victim from the live conductor. REMEMBER:
   a. Protect yourself with dry insulating material.
   b. Use a dry board, your belt, dry clothing, or other non-conducting material to free the victim. When possible PUSH - DO NOT PULL the victim free of the high voltage source.
   c. DO NOT touch the victim with your bare hands until the high voltage circuit is broken.

FIRST AID

The two most likely results of electrical shock are: bodily injury from falling, and cessation of breathing. While doctors and pulmotors are being sent for, DO THESE THINGS:

1. Control bleeding by use of pressure or a tourniquet.
2. Begin IMMEDIATELY to use artificial respiration if the victim is not breathing or is breathing poorly:
   a. Turn the victim on his back.
   b. Clean the mouth, nose, and throat. (If they appear clean, start artificial respiration immediately. If foreign matter is present, wipe it away quickly with a cloth or your fingers).





   c. Place the victim's head in the "sword-swallowing" position. (Place the head as far back as possible so that the front of the neck is stretched).
   d. Hold the lower jaw up. (Insert your thumb between the victim's teeth at the midline - pull the lower jaw forcefully outward so that the lower teeth are further forward than the upper teeth. Hold the jaw in this position as long as the victim is unconscious).
   e. Close the victim's nose. (Compress the nose between your thumb and forefinger).
   f. Blow air into the victim's lungs. (Take a deep breath and cover the victim's open mouth with your open mouth, making the contact air-tight. Blow until the chest rises. If the chest does not rise when you blow, improve the position of the victim's air passageway, and blow more forcefully. Blow <u>forcefully</u> into <u>adults</u>, and <u>gently</u> into <u>children</u>.
   g. Let air out of the victim's lungs. (After the chest rises, quickly separate lip contact with the victim allowing him to exhale).
   h. Repeat steps f. and g. at the rate of 12 to 20 times per minute. Continue rhythmically without interruption until the victim starts breathing or is pronounced dead. (A smooth rhythm is desirable, but split-second timing is not essential).

DON'T JUST STAND THERE - DO SOMETHING!