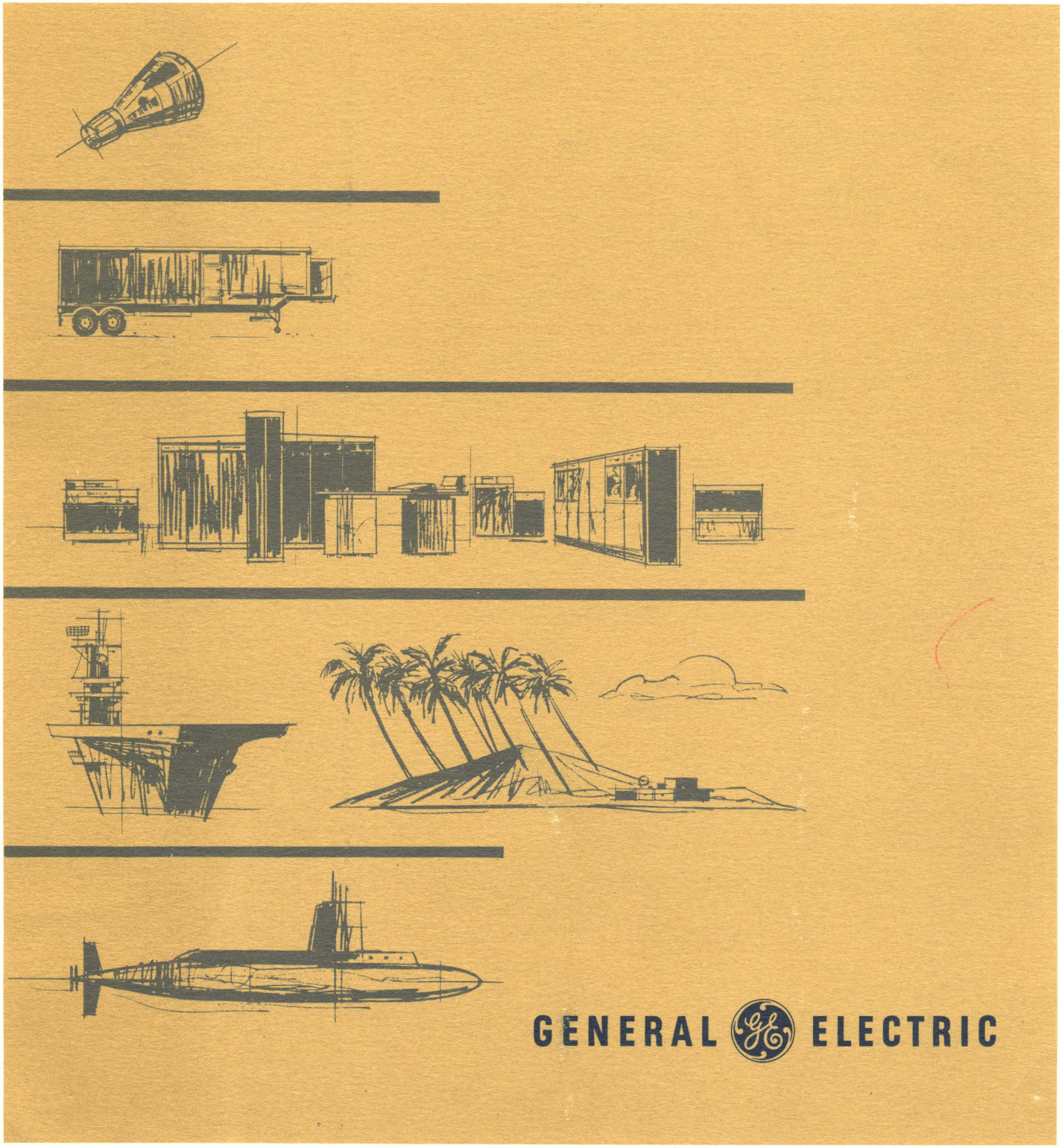
Military M-605 Programming Reference Manual



GENERAL ELECTRIC

Military Computer M-605 Programming Reference Manual

GENERAL ELECTRIC

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

I. M-605 SYSTEM DESCRIPTION

A. GENERAL DESCRIPTION

The M-605 Computer is a militarized digital computing system that is designed for medium-scale real-time applications. It is one member of General Electric's high performance Compatibles/600 family, which also includes the GE-635 and GE-625 for large scale business, scientific and real-time applications; the M-625 for large scale aerospace and defense applications; and the micro-miniaturized M-605 for airborne and spaceborne applications.

The GE-635, GE-625, and M-625 Computers are exact functional equivalents that differ only in speed and construction. The M-605 and A-605 Computers are identical to the GE-625 and GE-635 Computer systems in concept and organization. Their features and instruction repertoire are a compatible subset of those in the GE-635 Computer, and are directed to real-time applications not requiring the features of the larger systems.

B. COMPUTER COMPONENTS

The M-605 Computer System consists of three (3) major modules, the Memory, the Processor, and the real-time input-output controller (RT-IOC). These modules can be arranged in a variety of configurations, using multiple memory modules to provide the required storage, multiple processor modules to provide the necessary computation capability, and, if necessary, multiple RT-IOC modules for the complement of real-time and peripheral equipments required in an installation. System expansion is accomplished by the addition of modules and connecting cables. Various options are available for each of the major modules.

The processor module is that portion of the system which performs the function of executing the various programs stored in the memory module and processing execute interrupts acknowledged by the memory modules. The M-605 processor uses a 36-bit, single address instruction and a 36-bit operand. Each processor can be connected to, and can communicate with as many as four memory modules. Therefore, each processor can directly address as many as 262,144 words of magnetic core storage. All of the memory modules appear to the processor as a single memory with contiguous addresses. All M-605 processor modules contain a basic set of all fixed point, single-precision, real-time, character handling and special instructions. The floating point and double-precision instructions are handled by macro-operations or an optional hardware package.

The memory module is the heart of the computer system through which all communications and control functions are routed, whether between processor and external devices or between several processors. The M-605 Computer System uses an asynchronous, coincident-current magnetic core memory available in a 1 or 2-microsecond cycle time. Each memory module is normally available with from 16,384 to 65,536 36-bit words of storage, but can be provided with more memory capacity if so required for a specific application.

The RT-IOC module is the input-output terminal for the M-605 Computer System. Standard and non-standard peripherals and real-time devices are interfaced through the RT-IOC. Each RT-IOC module can contain up to 30 channels. These may be expanded using channel multiplexers to

serve as many as 64 low data-rate devices for each external channel. The RT-IOC channel provides an efficient interface for the device since the channel is customized to the device.

C. SYSTEM CHARACTERISTICS

1. Dual Mode Processor

Two classes of programs are executed by the M-605 Computer system: those that provide system control, and those that are directly related to the application. Control programs are executed in the master mode, whereas all applications programs are executed in the slave mode. In a multiprogramming environment, this essential and distinct delineation of operating modes assures that each program in the system will not alter or affect the others.

2. Dynamic Program Relocation

Each object program in memory is stored with addresses relative to zero. The absolute effective address is determined for each instruction as it is executed. Therefore, programs can be moved within memory, or they can be temporarily interrupted, placed in secondary storage, and returned to any available block of memory locations without the need for software relocation.

3. Processor- Oriented Memory Protection

Multiple programs occupying the same memory modules at the same time must be protected from each other. Each processor module in an M-605 Computer System has provisions for automatically limiting itself to any predetermined memory area when it is in its slave mode.

4. Input-Output-Oriented Memory Protection

Input-output activities that are requested by one object program must not be permitted to disturb any other object program unintentionally. Consequently, the memory has provision to protect blocks of memory for data transfers through the RT-IOC to insure that pre-assigned data-transfer area limits are not violated. This is controlled by the Real-Time Input-Output Supervisor within GECOS/605.

5. Execute Interrupt Orientation

In the modern multi-programming computer system it is necessary to free both the hardware and software from any specific timing requirements as well as from the responsibility of checking other components of the system for either completion of tasks or requests for service. Therefore, in the M-605 System, devices that have completed assigned tasks or that require service will generate execute interrupts to the current flow of instructions. These interrupts may be generated by processors as well as by input-output devices.

6. Interval Timer

Over-all systems control is facilitated by a timer register that is provided in each processor module. The timer is used to prevent any single program from monopolizing the processor or from running longer than the maximum time specified by the user. The timer is also used as a countdown clock to provide time-of-day and component utilization data.

7. Fault Traps

Because of the continuous access needs in a real-time programming application, the M-605 Computer provides for continuous on-line operation. Any operation that could cause the system to "hang up" results in a fault trap to the master mode supervisor program, GECOS/605, so that immediate remedial action can be initiated.

D. SOFTWARE SYSTEM

The primary objective of the M-605 software system is to provide support of the M-605 hardware in the performance of real-time missions and to provide the capability to use the M-605 concurrently for scientific data processing. The standard M-605 software is user-compatible with equivalent software for the GE-635 Computer. The following standard software is available for the M-605:

GECOS/605	Operating Supervisor
GMAP	MACRO Assembly Program
FORTRAN IV	Compiler
COBOL 61 Extended	Compiler +
JOVIAL J3*	Compiler +
GELOAD/605	Loader
Utility-Library Routines	
Diagnostic Package	
Support Software	

The M-605 Computer and its software system is managed by the General Comprehensive Operating Supervisor (GECOS/605). GECOS/605 permits the concurrent processing of a real-time program and any completely unrelated non real-time program. Both types of programs can occupy the magnetic core memory at the same time and time-share the use of the processor. This multi-programming environment is completely automatic under the control of GECOS/605. The real-time program is guaranteed highest priority in the system and is always given control of the processor when it requires it. The non-real-time program is only given processor time during slack periods in the real-time program. A sequential monitor version of GECOS/605 is also available for real-time only or job shop only applications.

GECOS/605 performs the following basic functions:

On-line media conversion

Allocation of memory and peripherals to each program

Dispatching of programs on a time-shared processor basis

Input-output supervision of all real-time and peripheral devices

Processing of multiplexed execute interrupts from the RT-IOC

Standard fault processing

Queuing of input-output requests

Job sequencing

The user program interface to GECOS/605 is identical with the GECOS interface on the GE-625, GE-635, and M-625 Computers. However, because different peripheral devices are used with the M-605 Computer, the status returned may vary in some cases from status returned by a similar peripheral device used with the GE-625 and GE-635 Computers.

+Presently available only with optional floating point hardware.
*Including I/O capabilities of J3X.

ii. PROGRAMMING ENVIRONMENT

A. PROGRAMMING CHARACTERISTICS

1. Number System

The binary number system is used in the M-605 Computer. All negative numbers are expressed in two's complement form. The full range of numbers possible in the computer considering the 36-bit word length is shown in binary and decimal form below:

Binary	Decimal
011111111111111111111111111111111111	$+2^{35}-1$
011111111111111111111111111111111110	$+2^{35}-2$
.	
.	
.	
000000000000000000000000000000000010	+2
000000000000000000000000000000000001	+1
000000000000000000000000000000000000	0
111111111111111111111111111111111111	-1
111111111111111111111111111111111110	-2
.	
.	
.	
100000000000000000000000000000000001	$-2^{35}-1$
100000000000000000000000000000000000	-2^{35}

Under this system of notation, all positive numbers are represented by their binary equivalent and all negative numbers by the two's complement of the positive value. The leftmost or most significant bit (bit position 0) gives the sign of the number with a "0" indicating positive numbers and a "1" indicating negative numbers. It should be noted that as positive numbers increase positively, the "1" bits propagate to the left. i.e., become more significant, while as negative numbers increase negatively, the "0" bits propagate to the more significant bit positions.

The two's complement of a number is formed by taking the one's complement of the number (by changing all "0" bits to "1" and all "1" bits to "0") and adding "1", or by the following method:

a) All low order (least significant, i.e., rightmost) bits are left unchanged up to and including the first low order "1".

b) All bits of higher order than the lowest order "1" are changed substituting "0" for "1" bits and "1" bits for "0" bits.

Example:

```
0···000000000001000110    =    +70
1···111111111110111010    =    -70
```

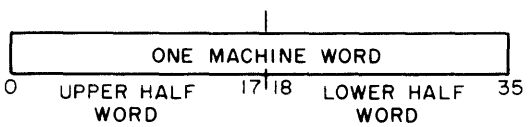
2. Representation of Information

The processor is fundamentally organized to deal with 36-bit groupings of information. Special features are also included for ease in manipulating 6-bit or 9-bit characters, 18-bit half words, and 72-bit double precision words.

The numbering of bit positions, character positions, words, etc., increases in the direction of conventional reading and writing: from the most- to the least-significant digit of a number, and from left to right in conventional alphanumeric text.

Graphical presentations in this manual show registers and data with position numbers increasing from left to right.

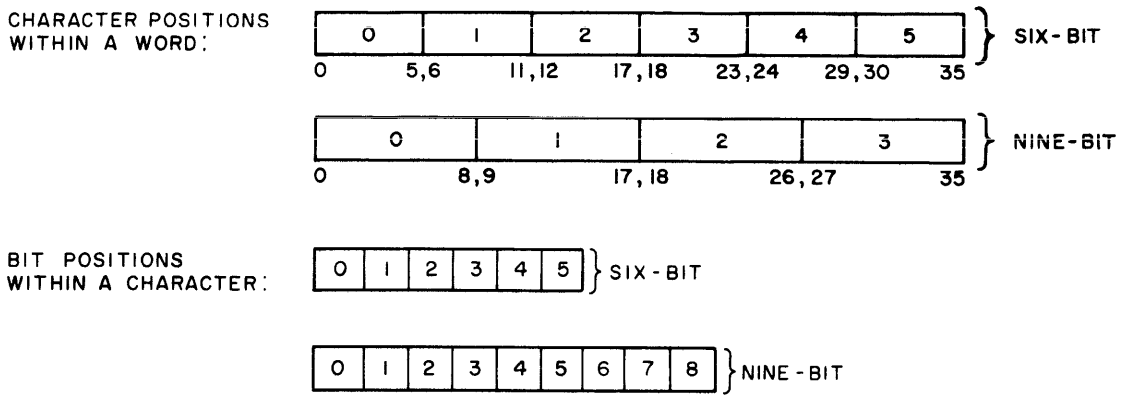
The machine word consists of 36 bits arranged as follows:



Data transfers between the processor and memory are word orientated: 36 bits are transferred at a time. When words are transferred to a magnetic core storage unit, this unit adds a parity bit to each 36-bit word before storing it. When words are requested from a magnetic core storage unit, this unit verifies the parity bit read from the core and removes it from the word transferred prior to sending each word to the processor.

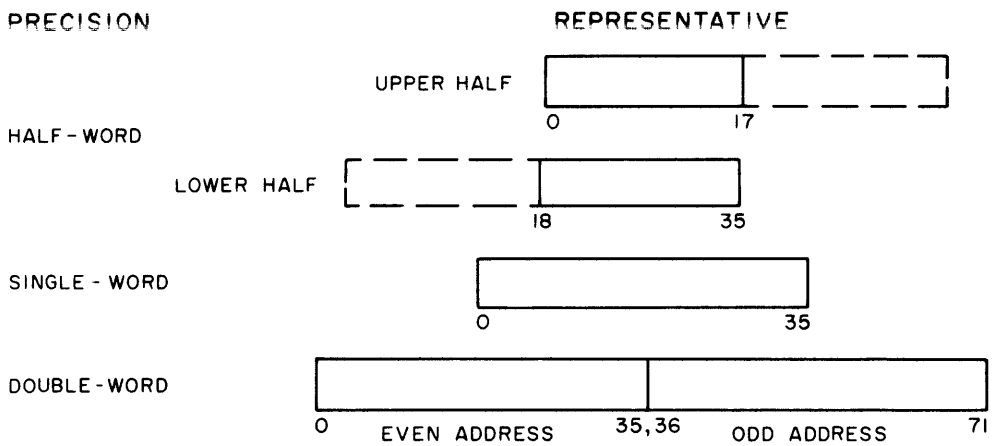
a. ALPHANUMERIC DATA

Alphanumeric data are represented by six-bit or nine-bit characters. A machine word contains either six or four characters:



b. BINARY FIXED-POINT NUMBERS

The instruction set comprises instructions for binary fixed-point arithmetic with half-word and single-word precision. Double-word precision is handled by macro-instructions or by an optional hardware package.



Instructions can be divided into two groups according to the way in which the operand is interpreted: the "logic" group and the "algebraic" group.

For the "logic" group, operands and results are regarded as unsigned, positive binary numbers. In the case of addition and subtraction, the occurrence of any overflow is reflected by the carry out of the most-significant (leftmost) bit position:

- Addition -- If the carry out of the leftmost bit position equals 1, then the result is above the range.

- Subtraction -- If the carry out of the leftmost bit position equals 0, then the result is below the range.

For the "algebraic" group, operands and results are regarded as signed, binary numbers, the leftmost bit being used as a sign bit, (a 0 being plus and 1 minus). When the sign is positive all the bits represent the absolute value of the number; and when the sign is negative, they represent the 2's complement of the absolute value of the number.

In the case of addition and subtraction the occurrence of an overflow is reflected by the carries into and out of the leftmost bit position (the sign position). If the carry into the leftmost bit position does not equal the carry out of that position then overflow has occurred. If overflow has been detected and if the sign bit equals 0, the resultant is below range; if with overflow, the sign bit equals 1, the resultant is above range. (See Paragraphs 4c and 4d, Carry and Overflow Indicators, below.)

An explicit statement about the assumed location of the binary point is necessary only for multiplication and division; for addition, subtraction, and comparison it is sufficient to assume that the binary points are "lined up".

In the M-605 processor, multiplication and division are implemented in two forms for 2's complement numbers: integer and fractional.

In integer arithmetic, the location of the binary point is assumed to the right of the least-significant bit position, that is, depending on the precision, to the right of bit position 35 or 71. The general representation of a fixed-point integer is then:

$$-a_n 2^n + a_{n-1} 2^{n-1} + a_{n-2} 2^{n-2} + \ldots + a_1 2^1 + a_0 2^0$$

where a_n is the sign bit.

In fractional arithmetic, the location of the binary point is assumed to the left of bit position 1. The general representation of a fixed-point fraction is then:

$$-a_0 2^0 + a_1 2^{-1} + a_2 2^{-2} + \ldots + a_{n-1} 2^{-(n-1)} + a_n 2^{-n}$$

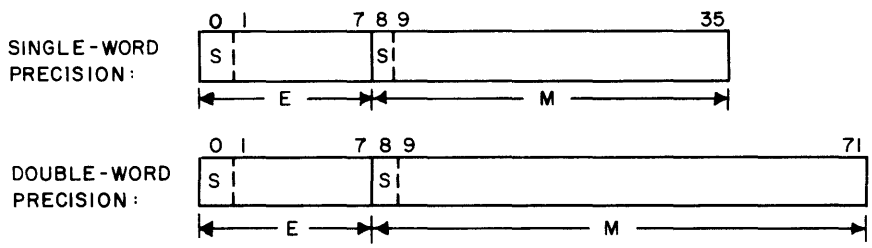
The number ranges for the various cases of precision, interpretation, and arithmetic are listed below:

INTER-PRETATION	ARITHMETIC	PRECISION		
		HALF-WORD (X_n, $Y_{0..17}$)	SINGLE-WORD (A,Q,Y)	DOUBLE-WORD (AQ, Y-PAIR)
ALGEBRAIC	INTEGRAL	$-2^{17} \leq N \leq (2^{17}-1)$	$-2^{35} \leq N \leq (2^{35}-1)$	$-2^{71} \leq N \leq (2^{71}-1)$
	FRACTIONAL	$-1 \leq N \leq (1-2^{-17})$	$-1 \leq N \leq (1-2^{-35})$	$-1 \leq N \leq (1-2^{-71})$
LOGIC	INTEGRAL	$0 \leq N \leq (2^{18}-1)$	$0 \leq N \leq (2^{36}-1)$	$0 \leq N \leq (2^{72}-1)$
	FRACTIONAL	$0 \leq N \leq (1-2^{-18})$	$0 \leq N \leq (1-2^{-36})$	$0 \leq N \leq (1-2^{-72})$

c. BINARY FLOATING-POINT NUMBERS

Instructions for binary floating-point arithmetic with numbers of single-word and double-word precision are handled by macro-instructions or by an optional hardware package. The upper 8 bits represent the integral exponent E and the lower 28 or 64 bits represent the fractional mantissa M. The notation for a floating-point number Z is:

$$Z_{(2)} \quad M_{(2)} \times 2^{E_{(2)}}.$$



WHERE S = SIGN BIT

Before doing floating-point additions or subtractions, the processor aligns the number which has the smaller positive exponent. To maintain accuracy, the lowest permissible exponent of -128 together with the mantissa equal to 0.00....0 has been defined as the machine representation of the number zero (which has no unique floating-point representation). Whenever a floating-point operation yields a resultant untruncated machine mantissa equal to zero (71 bits plus sign because of extended precision), the exponent is automatically set to -128.

The general representation of the exponent for single and double precision is:

$$-e_7 2^7 + e_6 2^6 + \ldots + e_1 2^1 + e_0 2^0$$

where e_7 is the sign.

The general representations of single- and double-precision mantissas are:

$$\text{Single Precision: } -m_0 2^0 + m_1 2^{-1} + m_2 2^{-2} + \ldots + m_{26} 2^{-26} + m_{27} 2^{-27}$$

and

$$\text{Double Precision: } -m_0 2^0 + m_1 2^{-1} + m_2 2^{-2} + \ldots + m_{62} 2^{-62} + m_{63} 2^{-63}$$

where m_0 is the sign in both cases.

For normalized floating-point numbers, the binary point is placed at the left of the most-significant bit of the mantissa (to the right of the sign bit). Numbers are normalized by shifting the mantissa (and correspondingly adjusting the exponent) until no leading zeros are present in the mantissa for positive numbers, or until no leading ones are present in the mantissa for negative numbers. Zeros fill in the vacated bit positions. With the exception of the number zero (represented as 0×2^{-128}), all normalized floating-point numbers will contain a binary 1 in the most-significant bit position for positive numbers and a binary 0 in the most-significant bit position for negative numbers. Some examples are:

Unnormalized positive number	$(0 \mid_{S} 0001101) \times 2^7$
Same number normalized	$(0 \mid_{S} 1101000) \times 2^4$
Unnormalized negative number	$(1 \mid_{S} 11010111) \times 2^{-4}$
Same number normalized	$(1 \mid_{S} 01011100) \times 2^{-6}$

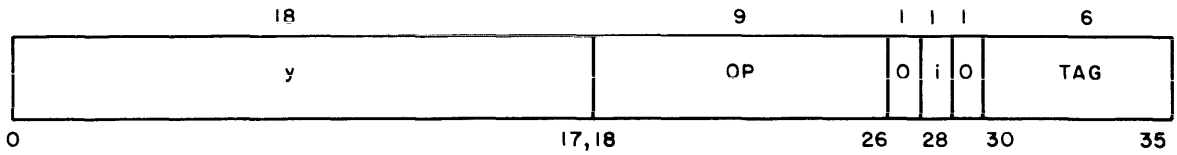
The number ranges resulting from the various cases of precision, normalization, and sign are listed in the table following:

	SIGN	SINGLE PRECISION	DOUBLE PRECISION
NORMALIZED	POSITIVE	$2^{-129} \leq N \leq (1-2^{-27})2^{127}$	$2^{-129} \leq N \leq (1-2^{-63})2^{127}$
	NEGATIVE	$-(1+2^{-26})2^{-129} \geq N < -2^{127}$	$-(1+2^{-62})2^{-129} \geq N \geq -2^{127}$
UNNORMALIZED	POSITIVE	$2^{-155} \leq N \leq (1-2^{-27})2^{127}$	$2^{-191} \leq N \leq (1-2^{-63})2^{127}$
	NEGATIVE	$-2^{-155} \geq N \geq -2^{127}$	$-2^{-191} \geq N \geq -2^{127}$

NOTE: THE FLOATING-POINT NUMBER ZERO IS NOT INCLUDED IN THE TABLE

d. INSTRUCTIONS

Machine instructions have the following general format:



where

y = address field: specifies the address of the memory location, whose contents is to be used as the operand for this instruction; also used to specify the number of shifts in shifting instructions.

OP = Operation Code: specifies the code of the machine instruction to be executed.

i = interrupt inhibit: when set to "1", prevents the interruption of the program after this instruction by an execute interrupt.

0 = not used: must be zero.

TAG = specifies the address modification to be performed

Certain instructions in the repertoire, e.g., the repeat instructions, use a different format (see individual instruction descriptions).

3. Program Addressable Registers

The registers of a computer are used for temporary storage of data in the processor. Most instructions deal with the loading or storing of information to and from the machine registers or the arithmetic or logical combination of this information. The registers of the M-605 processor which are accessible by machine instruction are shown below:

Register	Mnemonic	Length
Accumulator	AR	36 bits
Quotient	QR	36 bits
Combined Accumulator-Quotient	AQ	72 bits
Index (0-7)	$X_{n_{(n=0,\ldots,7)}}$	18 bits each
Exponent	E	8 bits
Base Address	BAR	18 bits
Indicator	IR	18 bits
Timer	TR	24 bits
Instruction Counter	IC	18 bits

The accumulator, quotient and combined accumulator quotient registers are the basic registers for holding data. These registers are used as follows:

- In fixed point operations as operand registers
- In floating point operations as mantissa registers
- In address modification as index registers.

These halves then are called AU (namely A_{0-17}), AL (namely A_{18-35}), QU (namely Q_{0-17}), and QL (namely Q_{18-35}) where U means upper and L means lower.

The eight index registers are used as follows:

- In fixed-point operations as operand registers for half precision
- In address modification as index registers.

The exponent register supplements the AQ-register in floating-point operations, serving as the register which holds the 8-bit exponent.

The base address register is used in address translation and memory protection. It stores the base address (absolute address of the object program being executed) and the number of 1024-word blocks assigned to that program.

The indicator register is a generic term for all the program-accessible indicators within the processor. The name is used where the set of indicators appears as a register, that is, as source or destination of data.

The timer register is decremented by one each 1/64 milliseconds (15.625 microseconds) and a timer runout fault trap occurs whenever its contents reach zero. If timer runout occurs in master mode, the trap does not occur until the processor returns to slave mode; but decrementation continues beyond zero.

The instruction counter holds the address of the next instruction to be executed.

4. Indicators

The indicators give the programmer information about the present state of the processor and the program it is executing. The indicators are set automatically by the processor and, in general, indicate the results after the execution of the present instruction. The indicators can be regarded as individual bit positions in an 18-bit half-word indicator register (IR). An indicator is set to the ON or OFF state by certain events in the processor, or by certain instructions. The ON state corresponds to a binary 1-in the respective bit position of the IR; the OFF state corresponds to a 0.

The description of each machine instruction includes a statement about those indicators that may be affected by the instruction and the condition under which a setting of the indicators to a specific state occurs. If the conditions stated are not satisfied, the status of this indicator remains unchanged.

The instruction set includes certain instructions which transfer data between the lower half of a storage location and the indicator register directly. The following table lists the indicators that have been implemented, their relation to the bit positions of the lower half of a memory location, and the instructions directly affecting indicators.

Implementation	Bit Position	Indicator	Indicator Instructions
Assigned	18	Zero	1. Load Indicators (LDI)
	19	Negative	2. Store Indicators (STI)
	20	Carry	3. Store Instruction Counter Plus 1 and Indicators (STC1)
	21	Overflow	
	22	Exponent Overflow	4. Return (RET)
	23	Exponent Underflow	
	24	Overflow Mask	
	25	Tally Runout	
	26	Parity Error	
	27	Parity Mask	
	28	Master Mode	
Unassigned	29		
	30		
	31		
	32	Must be	
	33	Zero	
	34		
	35		

a. ZERO INDICATOR

The zero indicator is used to test for zero or non-zero operands or resultants. It is affected by instructions that change the contents of a processor register (A, Q, AQ, Xn, BAR, IR, TR) or adder, and by the comparison instructions. The indicator is set ON when the new contents of the affected register or adder output contains all binary 0's; otherwise the indicator is set OFF.

The zero indicator is tested by the Transfer on Zero (TZE) and the Transfer on Not Zero (TNZ) instructions.

b. NEGATIVE INDICATOR

The negative indicator is used to test for negative or positive operands or resultants. It is affected by instructions that change the contents of a processor register (A, Q, AQ, Xn, BAR, IR, TR) or adder, and by comparison instructions. The indicator is set ON when the new contents of bit position 0 of this register or adder output is a binary 1; otherwise it is set OFF.

The negative indicator is tested by the Transfer of Minus (TMI) and Transfer on Plus (TPL) instructions.

c. CARRY INDICATOR

The carry indicator is used to determine if an operation has generated a carry out of the two most significant bits (bit positions 0 and 1). This is not an arithmetic overflow. The carry

indicator is affected by left shifts, additions, subtractions, and comparisons. The indicator is set ON when a carry is generated out of bit position 0; otherwise it is set OFF.

In single precision arithmetic operations, a carry out of bit position zero is normally ignored by the programmer since it does not affect the operation. In multi-precision arithmetic, the carry out of each lower portion of the resultant must be recognized and added to the next higher portion of the operand. The addition of two negative numbers is an example of the generation of a carry:

single precision:

```
                1111....11100     -4
                1111....11101     -3
                ----------------  --
carry           1111....11001     -7
```

double precision:

```
                upper operand     lower operand
                1111...1111       1111....11100     -4
                1111...1111       1111....11101     -3
                ----------------  ----------------  --
carry           1111...1110       1111....11001
(not used)                  1     carry - (used on upper operand)
                ----------------
                1111...1111                         -7
```

The Transfer on Carry (TRC) and the Transfer on No Carry (TNC) instructions test the state of the carry indicator. The Add with Carry (AWCA, AWCQ) and the Subtract with Carry (SWCA, SWCQ) instructions facilitate the handling of multi-precision arithmetic.

d. OVERFLOW INDICATOR

The overflow indicator is used to determine if the resultant of an operation has exceeded the capacity of the computer. It is affected by the arithmetic instructions, but not by compare instructions and Add Logical (ADL(R)) or Subtract Logical (SBL(R)) instructions. When the indicator is set, it is not automatically reset until it is specifically reset by the program.

The overflow indicator is set if there is a carry out of either the most significant bit (bit position 0) or the next most significant bit (bit position 1) but not both.

Example:

```
0111....11111        +2^35-1
0000....00001        +1
-------------------  ------
1000....00000        -2^35
carry
```

On arithmetic shifts to the left, an overflow is produced whenever the number involved is changed in sign during the shift.

The Transfer on Overflow (TOV) instruction tests the status of the overflow indicator and sets it OFF. The Load Indicator (LDI) and Return (RET) instructions destroy the contents of the overflow indicator since they reset it a specified position.

e. EXPONENT OVERFLOW INDICATOR

The exponent overflow indicator is affected by arithmetic operations with floating-point numbers or with the exponent register (E). The indicator is set ON when the exponent of the result is larger than +127 which is the upper limit of the exponent range. Since it is not automatically set to OFF otherwise, the exponent overflow indicator reports any exponent overflow that has happened since it was last set OFF by certain instructions (LDI, RET, and Transfer on Exponent Overflow (TEO)).

f. EXPONENT UNDERFLOW INDICATOR

The exponent underflow indicator is affected by arithmetic operations with floating-point numbers, or with the exponent register (E). The indicator is set ON when the exponent of the result is smaller than -128 which is the lower limit of the exponent range. Since it is not automatically set to OFF otherwise, the exponent underflow indicator reports any exponent underflow that has happened since it was last set OFF by certain instructions (LDI, RET, and Transfer on Exponent Underflow (TEU)).

g. OVERFLOW MASK INDICATOR

When the overflow mask indicator is ON, then the setting ON of the overflow indicator, exponent overflow indicator, or exponent underflow indicator does not cause an overflow fault trap to occur. When the overflow mask indicator is OFF, such a trap will occur. The overflow mask indicator can be set ON or OFF only by the instructions LDI and RET. Clearing of the overflow mask indicator to the unmask state does not generate a fault from a previously set overflow indicator, exponent overflow indicator, or exponent underflow indicator. The status of the overflow mask indicator does not affect the setting, testing or storing of these indicators.

h. TALLY RUNOUT INDICATOR

The tally runout indicator is affected by the Indirect Then Tally (IT) address modification type (all designators except Indirect and Fault) and by the Repeat, Repeat Double, and Repeat Link instructions (RPT, RPD, and RPL). The termination of a Repeat instruction because a specified termination condition is met sets the tally runout indicator to OFF. The termination of a Repeat instruction because the tally count reaches 0 (and for RPL because of a 0 link address) sets the tally runout indicator to ON; the same is true for tally equal to 0 in some of the IT address modifications. The tally runout indicator is tested by means of the Transfer On Tally Runout Indicator OFF (TTF) instruction.

i. PARITY ERROR INDICATOR

The parity error indicator is set to ON when a parity error is detected during the access of words from memory. It may be set to OFF by the LDI or RET instruction.

j. PARITY MASK INDICATOR

When the parity mask indicator is ON, the setting of the parity error indicator does not cause a parity error fault trap to occur. When the parity mask indicator is OFF, such a trap will

occur. The parity mask indicator can be set to ON or OFF only by the instructions LDI and RET. Clearing of the parity mask indicator to the unmasked state does not generate a fault from a previously set parity error indicator. The status of the parity mask indicator does not affect the setting, testing, or storing of the parity error indicator.

k. MASTER MODE INDICATOR

The master mode indicator can be changed only by an instruction. For a description of how the indicator can be changed, refer to the description of the response to execute interrupts on page II-30 and to the following instruction descriptions:

Instruction
Master Mode Entry (MME)
Return (RET)
Derail (DRL)
Transfer and Set Slave (TSS)

When the master mode indicator is ON, the processor is in the master mode; however, the converse is not necessarily true. (See the MME and DRL descriptions.)

5. Instruction Classifications

Most of the instructions available on the M-605 Computer are familiar to experienced programmers of large-scale computers. However, additional instructions have been provided to give the M-605 programmer extended capability for character handling, decision making, and advanced programming techniques involving list processing. A large portion of the instruction repertoire is devoted to real-time applications.

The instructions are grouped into the following classifications and sub-classifications:

- Data Movement
  - Load
  - Store
  - Shift
- Fixed-Point Arithmetic
  - Addition
  - Subtraction
  - Multiplication
  - Division
  - Negation
- Boolean Operations
  - AND
  - OR
  - EXCLUSIVE OR
- Comparison
  - Compare
  - Comparative AND
  - Comparative NOT AND

- Floating-Point
  - Load
  - Store
  - Addition
  - Subtraction
  - Multiplication
  - Division
  - Negation and Normalization
  - Comparison
- Transfer of Control
  - Transfer
  - Conditional Transfer
- Miscellaneous Operations
- Master Mode Operations
  - Master Mode
  - Master Mode and Control Processor

The double precision and floating point instructions in the above groups may be handled by macro-instructions or by an optional hardware package. The results of the execution of these instructions, however, are completely compatible with the results of the hardware instructions on the GE-635.

The following paragraphs briefly describe the uses and salient features of the major instruction types. For a complete description of each instruction see Section III.

a. DATA MOVEMENT

Besides the ability to load and store all processor registers, the Effective Address to (Register) instructions permit inter-register transfer. A zero address with Register modification replaces the contents of the register specified by the instruction with the contents of the register specified by address and modification.

The Store Zero (STZ) instruction permits the clearing of a memory location. This may be executed in a repeat mode. The Store Instruction Counter plus 1 (STC1) instruction stores both the instruction counter plus the indicators. This is complemented by the Return (RET) instruction which restores these indicators as it transfers.

Character handling and manipulation is facilitated by indirect-and-tally address modification and by instructions for directly loading and storing selected character positions of the accumulator or quotient register. The A and Q registers can be shifted individually or as one unit. The shift commands include right or left shift arithmetic, right shift logical, and left shift rotate.

b. FIXED-POINT ARITHMETIC

Fractional and integer instructions for both multiplication and division afford the programmer freedom from scaling the results of these operations. Normally, integer divide or multiply operations take place in the Q register and fractional divide or multiply operations take place in the A register. This convention permits easy programming of fixed-point arithmetic operations.

Arithmetic operations which add directly to a memory location and, which place the result of a subtraction directly in memory are included. An Add One to Store (AOS) instruction facilitates distribution and analysis and switch word settings.

c. BOOLEAN OPERATIONS

The logical operations AND, OR, and EXCLUSIVE OR can be performed by both the arithmetic and the index registers. The result may be placed in either the register or directly in memory.

d. COMPARISON

Compare operations do not alter the contents of storage or the specified register but merely set or clear indicators as the result dictates.

The fixed-point Compare instructions are shown below:

Instruction	Principal Functions
Compare Magnitude	Compare absolute values
Compare with Register	1. Compare algebraic values 2. Compare characters
Comparative AND with Register	Test for zeros in word fields
Comparative NOT AND with Register	Test for ones in word fields
Compare Masked	Search for identical, selectable fields
Compare with Limits	Search for a word whose value is within given limits

e. FLOATING-POINT ARITHMETIC

Although all models of the M-605 Computer do not have hardware for floating point arithmetic, all have hardware and instructions to facilitate and speed up floating point macro-instructions. The GMAP assembler will recognize the floating point instructions of the GE-635 and place macros in the assembled program. Sub-routines are used with some of the macro-instructions to minimize the length of the required macro.

Floating-point operations can be performed on both single- and double-precision data words; complete sets of data movement, arithmetic, and control instructions are provided for use in both types of operations. Unless specified otherwise by the programmer, the mantissas of all floating-point operation resultants are automatically normalized by the hardware. In performing addition and subtraction, addends and subtrahends are automatically aligned by the circuit components of the processor. Operations on floating-point numbers are performed by means of the A register or the 72-bit A-Q register to hold the mantissa, and a separate 8-bit exponent register.

The floating-point instruction repertoire includes two divide instructions that are especially convenient: Floating Divide Inverted (FDI) and Double-Precision Floating Divide Inverted (DFDI). These instructions cause the contents of the memory location to be divided by the contents of the A Register or the combined A-Q register - the reciprocal of other divide instructions in the repertoire. Therefore, regardless of whether the contents of the A Register must be a dividend or a divisor, the programmer can always perform a division without recourse to wasteful data movement operations. Floating Negate, Normalize, and Single- and Double-Precision Compare instructions are also included in the repertoire.

f. TRANSFER OF CONTROL

Transfer instructions are included which transfer only when the indicator condition specified is met. The Transfer and Set Index Register (TSXn) instructions are a set of eight separate and unique instructions which save the contents of the instruction counter in the specified index register. Because there is a unique instruction code for each of these instructions, address modification by another register is possible for transfer destination calculation.

g. SPECIAL OPERATIONS

Several special instructions are provided for expanding programmer options and reducing coding work through utilization of hardware features.

Three repeat instructions in the repertoire provide unusual programming advantages: Repeat (RPT), Repeat Double (RPD), and Repeat Link (RPL). The Repeat and Repeat Double instructions permit execution of the next one or two instructions a selected number of times, according to program requirements; they are especially useful for operating upon sequential lists in memory. For example, if Repeat is used with any of several compare instructions to search a list, termination of the repeats will occur when a "hit" is made.

The Repeat Link instruction is similar in its execution to the Repeat and Repeat Double instructions; it facilitates the processing of threaded lists scattered throughout memory.

The Binary-to-Binary Coded Decimal (BCD) instruction performs one step in an algorithm for the conversion of a binary number to its BCD equivalent. The instruction can be executed in the Repeat mode.

The Gray Code-to-Binary (GTB) instruction converts a 36-bit number from Gray code to its binary equivalent (in one execution of the instruction). This instruction is particularly useful when physical measurements are read directly into the computer.

h. INPUT-OUTPUT INITIATION

The Connect instruction is the only instruction in the M-605 instruction repertoire that initiates input-output action. The processor, having set up the input-output control words in the system memory, issues a Connect instruction to the input-output controller, which then assumes input-output responsibility.

6. Address Modification

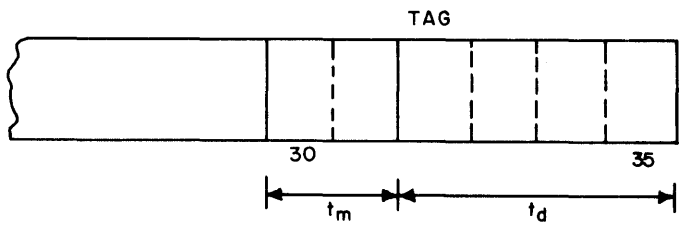
The address specified by the address field of an instruction is translated into an "effective address" before it is submitted to the memory as the operand address. An effective address is the final address produced by the address modification process; it is the address used for obtaining an operand or for storing a result. If no address modification is specified by the instruction, the address specified by the instruction address field is the effective address.

It should be noted that the effective address described may not be the absolute address of the operand in memory: it is the relative address of the operand within a program. The absolute address is formed automatically by the processor, however, and is not usually of concern to the programmer. (See Section II B 1 - Master/Slave Mode of Operation.)

The address specified in the address field of the instruction may be modified in a specified manner to form an effective address. The manner in which this address modification takes place is specified by the tag field of the instruction. The address may be modified by adding the contents of a register to the address, by using the address to access a memory word (indirect word) whose contents specify the effective address, or various combinations of the above.

The first case mentioned above is called register modification. The second case is called indirect modification. Indirect modification is a technique whereby the effective address is found in a memory location specified by the address field of the instruction word. Register and indirect modification types may be combined into one or indirect modification may be extended such that the effective address is only found after several levels of indirecting takes place.

The instruction tag field consists of two parts, the modifier (t_m) and the designator (t_d):



Where

t_m specifies one of the four possible modification types.

t_d specifies further the action for each modification type.

The four basic methods of address modification in the M-605 computer are:

Mnemonic	Modifier
R	Register
RI	Register Then Indirect
IR	Indirect Then Register
IT	Indirect Then Tally

There are a number of variations of each of the four. These variations are designated by the tag designator (t_d) field. In Register, Register Then Indirect and Indirect Then Register modification, t_d is the register designator which generally specifies the register to be used in the address modification. In Indirect Then Tally modification, t_d is the tally designator and specifies the tallying in detail.

The following table gives a general characterization of each of the four modification types.

t_m	Binary	Modification Type
R	00	Register Indexing according to t_d as register designator and termination of the address modification procedure.
RI	01	Register then Indirect Indexing according to t_d as register designator, then substitution and continuation of the modification procedure as directed by the Tag field of this indirect word.
IR	11	Indirect then Register Saving of t_d as final register designator, then substitution and continuation of the modification procedure as directed by the Tag field of this indirect word.
IT	10	Indirect then Tally Substitution, then use of this indirect word according to t_d as tally designator.

a. REGISTER DESIGNATOR

Each of the three modification types R, RI, IR includes an indexing step which is further specified by the register designator t_d. In most cases, t_d specifies a register which is added to the address field of the instruction. However, t_d may also specify that the address field of the instruction is to be used directly as operand and not as address of an operand (DU, DL), or that nothing takes place at all (N). Nevertheless, t_d is called "register designator" in these cases.

REGISTER DESIGNATOR		ACTION	
SYMBOLIC	BINARY		
N	0000	Y	REPLACES Y
X0	1000		
X1	1001		
•	•		
•	•	Y + C(Xn)	REPLACES Y
•	•		
X7	1111		
AU	0001	Y + C(A) $_{0...17}$	REPLACES Y
AL	0101	Y + C(A) $_{18...35}$	REPLACES Y
QU	0010	Y + C(Q) $_{0...17}$	REPLACES Y
QL	0110	Y + C(A) $_{18...35}$	REPLACES Y
IC	0100	Y + C(IC)	REPLACES Y
DU	0011	Y,00...0	IS THE OPERAND
DL	0111	00...0,Y	IS THE OPERAND

b. REGISTER MODIFICATION (R)

The effective address Y is formed by: (1) adding the contents of a specified register to the address field of the instruction word or (2) using the address field directly as the effective address - no modification.

When a register is used for modification, the contents of the register remain unchanged.

The specific type of Register address modification desired is specified symbolically by the programmer. Given below are the registers which may be used for address modification.

MNEMONIC SUBSTITUTION LIST

Mnemonic	Register	Effective Address
(R)=X0	XR_0	$Y=y+C(XR_0)_{0-17}$
X1	XR_1	$Y=y+C(XR_1)_{0-17}$
X2	XR_2	$Y=y+C(XR_2)_{0-17}$
X3	XR_3	$Y=y+C(XR_3)_{0-17}$
X4	XR_4	$Y=y+C(XR_4)_{0-17}$

Mnemonic	Register	Effective Address
X5	XR_5	$Y=y+C(XR_5)_{0-17}$
X6	XR_6	$Y=y+C(XR_6)_{0-17}$
X7	XR_7	$Y=y+C(XR_7)_{0-17}$
AU	AR_{0-17}	$Y=y+C(AR)_{0-17}$
AL	AR_{18-35}	$Y=y+C(AR)_{18-35}$
QU	QR_{0-17}	$Y=y+C(QR)_{0-17}$
QL	QR_{18-35}	$Y=y+C(QR)_{18-35}$
IC	IC_{0-17}	$Y=y+C(IC)_{0-17}$
DU	IR_{0-17}	$C(Y)_{0-17} = y$
DL	IR_{0-17}	$C(Y)_{18-35} = y$
N	None	$Y = y$

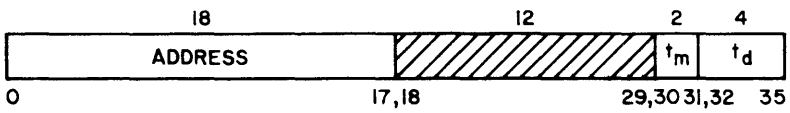
A special kind of address modification is provided. The use of the instruction address field as the operand is referred to as "Direct" address modification, of which there are two types:

- Direct Upper
- Direct Lower

With Direct Upper, the address field of the instruction serves as bits 0-17 of the operand and 0's are used as bits 18-35 of the operand. With Direct Lower modification, the address field of the instruction serves as bits 18-35 of the operand and 0's are used as bits 0-17 of the operand.

c. REGISTER THEN INDIRECT (R)I

The effective address is found by first performing the specified Register modification on the address field of the instruction to obtain an indirect word from the address so formed. The format of the indirect word is interpreted to be:



Next, the address modification specified by the indirect word is carried out. Thus, if the indirect word specifies RI, IR, or IT modification, the indirect sequence is continued. When an indirect word is found that specifies R modification, the R modification is carried out using the register specified by the tag of this indirect word and the address field of that final indirect word to form the effective address, Y.

If indirect modification, not preceded by Register modification is desired, it is accomplished by specifying the "no-modification" Register variation, (R) = N.

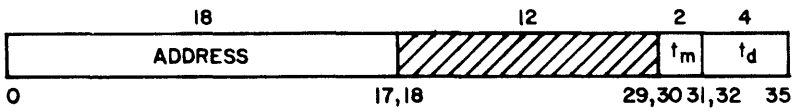
The mnemonic substitutions for (R) are listed under the Register modification description. All can be used except for DU or DL which cannot be substituted for the (R) of the (R)I modification.

The effective address, Y, is equal to $C(Y+C(R))_{0-17}$ for a reference to an indirect word that specifies no modification.

d. INDIRECT THEN REGISTER I(R)

The effective address is found by first obtaining an indirect word from the memory location specified by the address field, y, of the instruction.

The format of the indirect word is interpreted to be:



Secondly, the address modification specified by the indirect word is carried out. If that modification is RI, the indirect sequence is continued until an indirect word is found that specifies R or IT modification. If the indirect word specifies R modification, the register R specified by the instruction is substituted for the R of the indirect word, producing an effective address which is the address field of the indirect word as modified by the R of the instruction word. If the indirect word specifies IT modification, it is converted to an R modification which is performed as above.

If any indirect word in the sequence specifies IR, the R of that indirect word supersedes the R of either the instruction word or any preceding indirect word in the final R modification.

If an indirect modification without Register modification is desired, the "no-modification" variation of Register modification should be specified in the instruction.

The mnemonic substitutions for (R) are listed under the Register modification description. All can be used except for DU or DL which cannot be substituted for the (R) of the I(R) modification.

The effective address, Y, is equal to $C(Y)_{0-17} + C(R)$ for a single indirect reference.

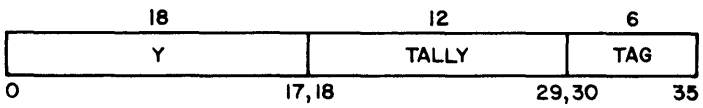
e. INDIRECT THEN TALLY I(T)

The effective address is the address field of the indirect word obtained from the location specified by the address field of the instruction or a preceding indirect word, whichever one specified the IT modification. There are ten variations of the IT modification. The variation desired is specified symbolically by the programmer by substituting the mnemonic from the substitution list for (t_d).

The following table gives the possible tally designators under IT type modification.

TALLY DESIGNATOR		NAME
SYMBOLIC	BINARY	
I	1001	INDIRECT ONLY
ID	1110	INCREMENT ADDRESS, DECREMENT TALLY
DI	1100	DECREMENT ADDRESS, INCREMENT TALLY
IDC	1111	INCREMENT ADDRESS ,DECREMENT TALLY, AND CONTINUE
DIC	1101	DECREMENT ADDRESS, INCREMENT TALLY, AND CONTINUE
AD	1011	ADD DELTA (TO ADDRESS FIELD)
SD	0100	SUBTRACT DELTA (FROM ADDRESS FIELD)
CI	1000	CHARACTER FROM INDIRECT
SC	1010	SEQUENCE CHARACTER
F	0000	FAULT

The format of the indirect word is:



Where

y = address field

Tally = tally field

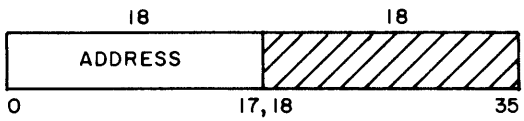
Tag = tag field

A description of the use of the tally and tag fields of the indirect word is found under the description of each type of IT modification.

f. INDIRECT ONLY (I)

The effective address is the address field of the indirect word obtained from the memory location specified by the address field of the instruction or indirect word whichever one specified the indirect modification.

The format of the indirect word is interpreted as:

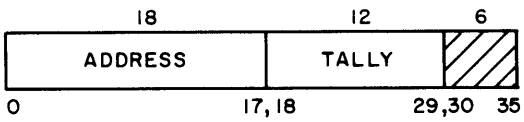


The tally and tag fields of the indirect word are not used. This instruction may be used in conjunction with the ID or DI modifier when it is desired to reference the indirect word without incrementing or decrementing either the address or tally portion of the indirect word.

g. INCREMENT ADDRESS, DECREMENT TALLY (ID)

The effective address is the address field of the indirect word obtained from the location specified by the address field of the instruction or preceding indirect word, whichever one specified the ID modification.

The indirect word is interpreted as:



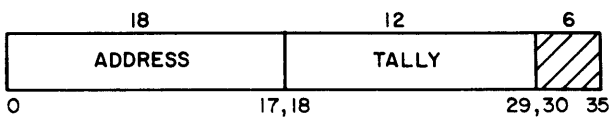
Each time such a reference is made to the indirect word, the address field of the indirect word is incremented by one and the tally portion of the indirect word is decremented by one. The incrementing and decrementing is done after the effective address is provided for the instruction operation. The tag field is not used.

When the tally reaches 0, the tally runout indicator is set.

h. DECREMENT ADDRESS, INCREMENT TALLY (DI)

The effective address is the address field-1 of the indirect word obtained from the location specified by the address field of the instruction or preceding indirect word, whichever one specified the DI modification.

The indirect word is interpreted as:



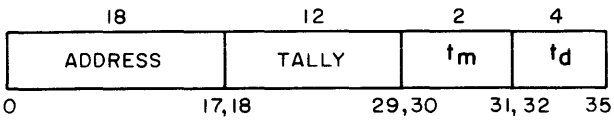
Each time a reference is made to the indirect word, the address field of the indirect word is decremented by 1 and the tally portion is incremented by 1. The incrementing and decrementing is done prior to providing the effective address for the instruction operation. The tag field is not used.

When the tally reaches 0, the tally runout indicator is set.

i. INCREMENT ADDRESS, DECREMENT TALLY, AND CONTINUE (IDC)

IDC modification is the same as ID modification except the tag field of the indirect word may specify a continuation of the indirect chain.

The indirect word is interpreted as:

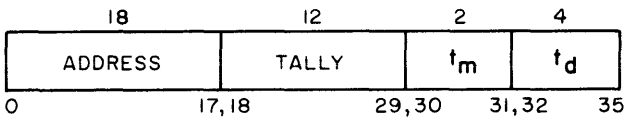


The tag field may specify any form of IT or IR modification; but if R or RI is used, the register designator must specify N (none).

j. DECREMENT ADDRESS, INCREMENT TALLY, AND CONTINUE (DIC)

DIC modification is the same as DI modification except the tag field of the indirect word may specify a continuation of the indirect chain.

The indirect word is interpreted as:

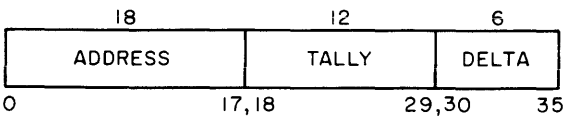


The tag field may specify any form of IT or IR modification: but if R or RI is used, the register designator must specify N (none). The incrementing and decrementing is done prior to obtaining the contents of the address from memory.

k. ADD DELTA (AD)

The effective address is the address field of the indirect word specified by the address field of the instruction or the preceding indirect word, whichever one specified the ID modification.

The indirect word is interpreted as:



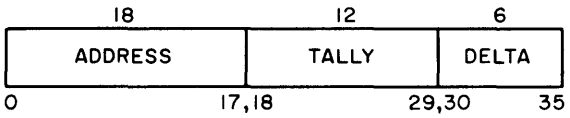
Each time a reference is made to the indirect word, the address field of the indirect word is increased by delta and the tally is decremented by one. The addition of delta and the decrementing is done after the contents of the address is provided for the instruction operation.

When the tally reaches 0, the tally runout indicator is set.

1. SUBTRACT DELTA (SD)

The effective address is the address field minus the tag field of the indirect word specified by the address field of the instruction or the preceding indirect word, whichever one specified the SD modification.

The indirect word is interpreted as:

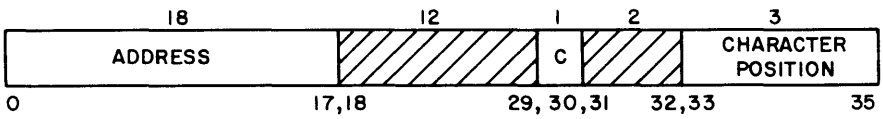


Each time a reference is made to the indirect word, the address of the indirect word is decreased by delta and the tally is incremented by one. The subtraction of delta and the incrementing is done prior to obtaining the contents of the address from memory.

m. CHARACTER FROM INDIRECT (CI)

The effective address is the address field of the indirect word obtained from the location specified by the address field of the instruction or preceding word, whichever one specified the CI modification.

The indirect word is interpreted as:



The character size to be used is specified by bit 30(C) of the indirect word:

C	Character Size
0	6-bit
1	9-bit

The character position field is used to specify the character involved in the operation. The character position field specifies characters in accordance with the following:

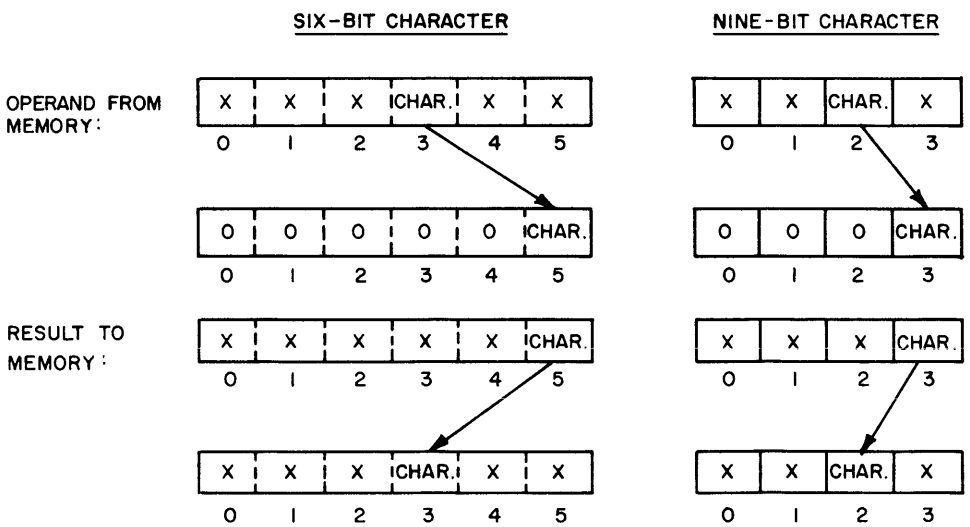
Character Position	Character Position Field	Character Handling 6-Bits	Character Handling 9-Bits
0	000	$C(Y)_{0-5}$	$C(Y)_{0-8}$
1	001	$C(Y)_{6-11}$	$C(Y)_{9-17}$
2	010	$C(Y)_{12-17}$	$C(Y)_{18-26}$
3	010	$C(Y)_{18-23}$	$C(Y)_{27-35}$
4	100	$C(Y)_{24-29}$	---
5	101	$C(Y)_{30-35}$	---

This form of IT modification is intended for use only with those instructions which involve the A or Q Registers.

For six-bit character operations in which the operand is taken from memory, the effective operand from memory is presented as a single word with the specified character justified to character position 5: positions 0-4 are presented as zero. For operations in which the resultant is placed in memory, character 5 of the resultant replaces the specified character in memory location Y: the remaining characters in memory location Y are not changed.

For nine-bit character operations in which the operand is taken from memory, the effective operand from memory is presented as a single word with the specified character justified to character position 3: positions 0-2 are presented as zero. For operations in which the resultant is placed in memory, character 3 of the resultant replaces the specified character in memory location Y: the remaining characters in memory location Y are not changed.

Example:

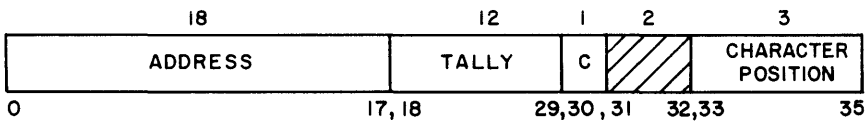


This modifier is similar to the Sequence Character modifier except that no incrementing or decrementing of the address, tally, or character position is performed. This instruction can be used in conjunction with the SC indirect word when it is desired to reference the indirect word and use the character position, without disturbing the indirect word.

n. SEQUENCE CHARACTER (SC)

For the SC modifier the effective address is the address field of the indirect word obtained from the location specified by the address field of the instruction or preceding indirect word, whichever one specified the (SC) modification.

The indirect word is interpreted as:



The type of character handling to be used is specified by Bit 30(C) of the indirect word. If C=1, 9 bit character handling is specified or if C=0, 6 bit character handling is specified.

The character position field is used to specify the character to be involved in the operation. The character position field is interpreted the same as for CI modification.

This form of IT modification is intended for use only with those instructions which involve the A or Q register. For operations in which the resultant is placed in the A or Q register, the effective operand from storage is 36-bits in length with the specified character justified to bits 30-35 (6 bit character) or 27-35 (9 bit character). Bits 0-29 (6 bit character) and 0-26 (9 bit character) are set to zero. For operations in which the resultant is placed in storage, the justified character from bits 30-35 (6 bit character) or 27-35 (9 bit character) of the resultant is placed in the character position specified by the indirect word. The remaining bits in the specified storage location are unchanged.

The tally is used to count the number of times a reference is made to the indirect word. Each time a reference is made to the indirect word by an SC modification, the tally is decremented by one; and the character position is incremented by one to specify the next character position. When the character position 5 (6 bit character handling) or 3 (9 bit character handling) is incremented, it is changed to position "0", and the address field is incremented by one. All incrementing and decrementing is done after the effective address has been provided for the instruction execution.

Characters are operated on in sequence from left to right. The Tally runout indicator is set when the Tally reaches "0".

o. FAULT (F)

The use of this address modification will cause a fault trap to occur. The tally and tag fields of the indirect word are not used. For examples of the coding and applications of these address modifications see Section IV.

B. OPERATIONAL CHARACTERISTICS

1. Master/Slave Modes of Operation

To permit separation of control programs and object programs with corresponding protection of control programs from undebugged object programs, two modes of operation, Master and Slave, are provided in the processor. Control programs will run in the Master Mode, and object programs will run in the Slave Mode. Programs running in Master Mode have access to the entire memory, may initiate peripheral and internal control functions, and do not have base address relocation applied. Programs running in Slave Mode have access to a limited portion of the memory, cannot generate peripheral control functions and have the base address register added to all relative memory addresses of the object program.

Master Mode operation is the state in which the processor:

- Presents an "unrelocated" address to the memory
- Has an unbounded access to memory
- Causes the memory to be in the unprotected state when accessed by the processor

  This permits access to protected areas of memory (protected by the File protect register -- when provided), setting of execute interrupt cells, generation of peripheral commands, alteration of the file protect register (when installed) and channel and execute interrupt masks.

- Permits setting the timer and base address register by the appropriate instructions.

The processor is in the Master Mode when any of the following exists:

- The Master Mode Indicator is in the master condition
- An execute interrupt is recognized
- A fault is recognized

Slave Mode operation is the state in which the processor:

- Presents a relocated address to the memory, as specified by the base address register.
- Restricts the effective address formed to the bounds specified by the boundary register (lower half of the base address register).
- Causes the memory to be in the "protected" state when accessed by the processor.

  a. This prohibits access to protected areas of memory (controlled by the file protect register).

  b. This prohibits generation of peripheral commands, alteration of the file protect register, interrupt masks, or setting of execute interrupt cells, even if the processor is designated the control processor by the memory module.

- Prohibits setting of the timer, and base address register.

The processor is in the Slave Mode when the Master Mode indicator is in the slave condition or when the Transfer and Set Slave (TSS) instruction is being executed.

The processor base address register contains a base address in bit positions 0-7 for the purpose of address translation. The translation takes place only in the Slave Mode of operation. It consists of adding this base address to bit positions 0-7 of the program address.

In the Master Mode no address translation takes place. Any program address to be used in a memory access request while the processor is in the Master Mode is used directly as an actual address and submitted to the memory without any translation.

Address translation is actually based on nine bits, namely the base address register positions 0-8 and the bit positions 0-8 of the program address; this permits address relocation by multiples of 512 words. In order to maintain compatibility with the GE-635, bit positions 8 and 17 of the base address register contain 0's and cannot be altered by the Load Base Address Register (LBAR) instruction. Thus, address relocation is performed in multiples of 1024.

Any object program address to be used in a memory access request while the processor is in the Slave Mode is checked, just prior to the fetch, for being within the address range allocated by the Comprehensive Operating Supervisor (GECOS/605) to the program for this execution. This address range protection is commonly referred to as memory protection.

For the purpose of memory protection, the 18-bit processor base address register is loaded by GECOS with an address range in bit positions 9-16. The portion of the base address register is called the bounds register. The check takes place only in the Slave Mode. It consists of subtracting bit positions 0-7 of the program address from this address range. When the result is zero or negative, then the program address is out of range; and a Memory Fault Trap occurs. (Refer to Section II B 3.)

More specifically, the checking is actually based on nine bits, namely the base address register positions 9-17 and the bit positions 0-8 of the program address. Memory protection is performed in multiples of 1024 words.

In the Master Mode no checking takes place: thus, any memory location (in those memory modules that are connected to this processor) can be accessed.

2. Program Execute Interrupts

Data transfer between the M-605 memory module and external devices is normally completely asynchronous with processor operation or program execution. The program execute interrupt facility of the M-605 is the means by which these external devices can interrupt the program being executed by the processor and thereby notify it that an external event has occurred.

Located in each memory module is a program interrupt facility that consists of up to 32 unique interrupt cells. Although any of the eight devices - either processor or RT-IOC modules - that may be connected to the memory module can set any of the cells, only specific cells will generally be assigned to a given device. Associated with the 32 cells is a 32 bit execute interrupt mask register that is read or set by program control and which can be used to change the wired priority of the 32 cells. A binary 1 or 0 in a given bit position of the mask register will respectively permit or inhibit the acknowledgement of interrupt requests made by one of the devices connected to the memory. If a device requests a program interrupt by setting one of the cells and if the cell is unmasked, the interrupt will be acknowledged. If several demands are made simultaneously, the highest in priority will be serviced first.

Whenever an unmasked interrupt cell has been set, the memory presents an "interrupt present" flag to the processor designated as its control processor. As soon as the processor has completed the current instruction and assuming that interruption has not been inhibited, the processor will interrupt its program sequence and request of the memory the number of the cell causing the interrupt. Using this number as a part of an address, the processor executes the pair of instructions corresponding to the 32 interrupt cells. These instructions can transfer program control to the entry point of the desired routine which can safe store the processor's instruction counter and registers to permit a later return to the interrupted program.

The execute mask register is used to change the priority. Once a program is initiated, the mask register is set to permit interruption of the program only by events of a higher priority than the one that initiated the current program.

Although interrupts commonly cause a transfer of control to the operating system, the transfer can be direct to a specific program that is to respond to the interrupt without the intervention of an executive program to determine the priority of the interrupt. This is significant in real-time applications to minimize the computer response time.

The 64 core locations associated with the 32 interrupt cells are located in the block of memory starting with absolute location 0. If a processor has a control relationship with more than one memory module, each block of 64 locations, one block per memory module, is contiguous.

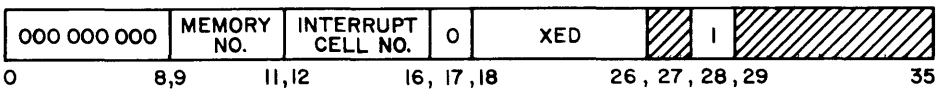
A program may inhibit interruption by placing a binary 1 in bit position 28 of an instruction. When specified, interruption is inhibited until the execution of an instruction that does not inhibit interruption, or until a lockup fault occurs (see Section II B 3).

The processor carries out the execute interrupt procedure as soon as an instruction is being executed that:

- Did not have its interrupt inhibit bit (bit position 28) set to 1
- Did not cause an actual transfer of control (A transfer of control is effected if the instruction is an unconditional transfer, or a conditional transfer with the condition satisfied.)
- Was not an Execute or Execute Double (XEC or XED) instruction (Note than an XEC or XED instruction and the one or two instructions carried out under its control are regarded as a single instruction execution.)

The step by step execute interrupt procedure is as follows:

- Enter the Master Mode (the Master Mode Indicator is not affected.)
- Return the transfer interrupt number command code to the memory (system) controller that sent the interrupt request present signal.
- Receive a five-bit interrupt code on the data lines from the memory module (bit positions 12-16), specifying the number of the highest priority nonmasked interrupt cell that was set to ON when the transfer interrupt number command code was recognized at the system controller.
- Carry out an Execute Double (XED) instruction with an effective address (Y) as shown below, bits 0-17:



The memory number is determined by the position of the address reassignment switches associated with the system controller causing the execute interrupt. The switches are three-position toggles having the positions 0, 1, and EITHER. A switch in the EITHER position is interpreted as a 0 in preparing the address for the instruction.

The cell number is determined by the highest priority unmasked interrupt cell (in the system controller) causing the execute interrupt.

- Return to the mode specified by the Master Mode Indicator (see below) and continue with the instruction from the memory location specified by the Instruction Counter.

Each of the two instructions from the memory location Y-pair may affect the Master Mode Indicator as follows:

- If this instruction results in an actual transfer of control and is not the Transfer and Set Slave instruction (TSS), then ON (that is, Master Mode).
- If this instruction is either the Return instruction (RET) with bit 28 equal to 0 or the TSS instruction, then OFF (that is, Slave Mode).

The first of the two instructions from the memory location Y must not alter the contents of the location of the second instruction, and must not be an XED instruction. If the first of the two instructions alters the contents of the instruction counter, then this transfer of control is effective immediately; and the second of the two instructions is not executed.

3. Faults

The M-605 processor also responds to interrupts caused by internal events. This class of interrupt is called a "fault" although not all are true faults with the computer itself but rather are used to request a specific action from the processor. The connect fault, for example, is used by a control processor to initiate the action of a non-control processor. There are four general categories of faults:

- Instruction generated
- Program generated
- Hardware generated
- Manually generated

a. INSTRUCTION GENERATED FAULTS

The Instruction generated faults are:

- Master Mode Entry (MME)

  The instruction Master Mode Entry has been executed. This is a normal request to the supervisor, GECOS/605.

- Derail (DRL)

  The instruction Derail has been executed. This is normally used in maintenance procedures.

- Fault Tag

  The address modifier I(T) where T=F has been recognized. The indirect cycle will not be made upon recognition of F, nor will the operation be completed; a fault trap will be entered.

- Connect (CON)

  The processor has received a Connect from a control processor via a system controller.

- Illegal OP Code (ZOP)

  An operation code of all zeros has been executed.

b. PROGRAM GENERATED FAULTS

Program generated faults are defined as:

- The Arithmetic Faults

  a. Overflow (FOFL) -- An arithmetic overflow, exponent overflow, or exponent underflow has been generated. The generation of this fault is inhibited when the overflow mask is in the mask state. Subsequent clearing of the overflow mask to the unmasked state will not generate this fault from previously set indicators. The overflow mask state does not affect the setting, testing, or storing of indicators.

b. Divide Check (FDIV) -- A divide check fault occurs when the actual division cannot be carried out for one of the reasons specified with each divide instruction.

- The Elapsed Time Interval Faults

a. Timer Runout (TROF) -- This fault is generated when the timer count reaches zero. If the processor is in Master Mode, recognition of this fault will be delayed until the processor returns to the Slave Mode; this delay does not inhibit the counting in the timer register.

b. Lockup (LUF) -- The processor is in a program lockup which inhibits recognizing an execute interrupt or interrupt type fault for greater than 16 milliseconds.† Examples of this condition are the coding TRA * or the continuous use of inhibit bit.

c. Operation Not Completed (FONC) -- This fault is generated due to one of the following:

1. No memory attached to the processor for the address.

2. Operation not completed. (See Hardware Generated Faults)

- The Memory Faults

a. Command (FCMD) -- This fault is interpreted as an illegal request by the processor for action of the system controller. These illegal requests are:

1. The processor is in the Slave Mode, and issues a CIOC, RMCM, RMFP, SMCM, SMFP, or SMIC. The CIOC, SMCM, SMFP, and SMIC commands will not be executed. (Refer to Section III for descriptions and references concerning these instruction mnemonics.)

2. When the processor has issued a connect to a channel that is masked off (by program or switch).

b. Memory (FMEM) -- This fault is generated when:

1. No physical memory existed for the address.

2. An address (in Slave Mode) is outside the program boundary or inside file protected memory.

3. The memory did not respond to a request within several milliseconds.

c. HARDWARE-GENERATED FAULTS

The hardware-generated faults are defined as:

- Operation Not Completed (FONC) -- This fault is generated due to one of the following:

a. The processor has not generated a memory operation within 1 to 2 milliseconds and is not executing the Delay Until Interrupt Signal (DIS) instruction.

b. The system controller closed out a double-precision or read-alter-rewrite cycle.

c. See Operation Not Completed under Program Generated Faults (above).

†The time interval can be changed for individual site requirements.

- Parity (FPAR) -- This fault is generated when a parity error exists in a word which is read from a core location:

  a. Instruction word fetch -- if the odd instruction contains a parity error, the instruction counter retains the location of the even instruction.

  b. Indirect word fetch -- if a parity error exists in an indirect and tally word in which the word is normally altered and replaced, the contents of that memory location are destroyed.

  c. Operand fetch -- when a single-precision operand, C(Y) is requested, the contents of the memory pair located at Y, Y+1 where Y is even, or Y-1, Y, where Y is odd are read from memory by the system controller. The system controller will not report a parity error if it occurs in C(Y+1) or C(Y-1), but will restore the $\overline{C(Y+1)}$, C(Y-1) with a parity bit equal to 1.

  If a parity error occurs on any instruction for which the C(Y) are taken from a core location (this includes "to storage" instructions, ASA, ANSA, etc.), the processor operation is completed with the faulty operand before entering the fault routine.

  The generation of this fault is inhibited when the parity mask indicator is in the mask state. Subsequent clearing of the parity mask to the unmasked state will not generate this fault from a previously set parity error indicator. The parity mask does not effect the setting, testing, or storing of the parity indicator.

d. MANUALLY GENERATED FAULTS

Manually generated faults are.

- Execute (EXF)

  a. The EXECUTE PUSHBUTTON on the processor maintenance panel has been activated.

  b. The external frequency of a pulse generator has been substituted for the EXECUTE pushbutton.

  The above two are dependent on other switch positions on the processor control panel.

- The Power Turn On/Off Faults

  a. Startup (SUF) -- A power turn-on has occurred.

  b. Shutdown (SDF) -- Power will be turned off in approximately 1 millisecond.

  The 16 faults are organized into five groups to establish priority for the recognition of a specific fault when faults occur in more than one group. Group I has highest priority.

  Only one fault within a priority group is allowed to be active at any one time. In the event that two or more faults occur concurrently, only the fault which occurs first through normal program sequence is permitted.

Faults in Groups I and II cause the operations in the processor to abort unconditionally.

Faults in Groups III and IV cause the operations in the processor to abort conditionally upon the completion of the operation presently being executed.

Faults in Group V are recognized under the same conditions that program interrupts are recognized. Faults in Group V have priority over program interrupts and are also subject to being inhibited from recognition by use of the inhibit bit in the instruction word.

Upon recognition of a fault, the contents of the Instruction Counter (IC) are as shown in the Table of Faults below.

Fault No.	Fault Name	Group (Priority)	IC Contents
1100	Startup	I	N+0, 1, or 2
1111	Execute	I	N+0, 1, or 2
1011	Operation Not Completed	II	N+0, 1, or 2
0111	Lockup	II	N+0, 1, or 2
1110	Divide Check	III	N (note 4)
1101	Overflow	III	N
1001	Parity	IV	N (note 2)
0101	Command	IV	N+1
0001	Memory	IV	N+1 (note 4)
0010	Master Mode Entry	IV	N (note 4)
0110	Derail	IV	N (note 4)
0011	Fault Tag	IV	N (note 4)
1010	Illegal Op Code	IV	N
1000	Connect	V	N
0100	Timer Runout	V	N
0000	Shut Down	V	N

Notes:

1. N = Last operation completed
2. If parity occurred on operand fetch, operation N+1 was completed with faulty data.

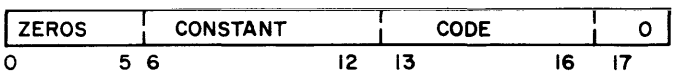
   If parity occurred on instruction fetch, operation N+1 was not completed.

   If parity occurred on IT, IT was not completed.
3. Number of IND cycles, and ITs performed is unknown.
4. These operations are considered complete when the fault is recognized.

Each of the sixteen types of faults and other events have a fault trap assigned.

The fault trap procedure is similar to the program execute interrupt procedure except that the effective address is defined differently. The fault trap procedure consists of the following steps:

- Automatically enter the Master Mode (the Master Mode Indicator is not affected).
- Carry out an Execute Double (XED) instruction with an effective address (Y) as defined for bits 0-17 of a machine word as follows:



Constant: Set up by the fault switches in the processor (also see the description of the instructions Master Mode Entry (MME) and Derail (DRL)

Code: The four-bit fault trap code which identifies the respective fault trap (See Table above)

- Return to the mode specified by the Master Mode indicator, and continue with the instruction from the memory location specified by the instruction counter. Unless the executed instructions under the XED caused a transfer of control.

Each of the two instructions from the memory location Y-pair may affect the Master Mode Indicator as follows: If this instruction results in an actual transfer of control and is not the Transfer and Set Slave instruction (TSS), the ON: if this instruction is either the Return instruction (RET) with bit 28 equal to 0 or the TSS instruction, then OFF.

The first of the two instructions from the memory location Y must not alter the contents of the location of the second instruction, and must not be an Execute Double instruction (XED). If the first of the two instructions alters the contents of the Instruction Counter, then this transfer of control is effective immediately; and the second of the two instructions is not executed.

4. Memory Cycles

The M-605 memory is capable of three basic types of memory cycles: (1) read-restore, (2) real-alter-rewrite, (3) clear-write. The type of cycle required for a particular memory operation is specified by the processor or the external device, whichever is involved in the operation.

The first type of memory cycle, read-restore, is normally used to obtain a memory word. The contents of the specified memory location are transferred from the magnetic core storage unit to a register in the memory (system) controller. Immediately, both the write-back to storage and the data transfer to the requesting device is started. By the time the original contents of the memory location has been restored, the communicating device has received (and usually used) the information. The memory permits both single- and double-precision read-restore cycles.

The second type of cycle, clear-write, is most commonly used when it is desired to place a word in storage. This type of cycle is started as before by reading the memory location; but the contents of the location are inhibited from entering the memory register in the system controller. Shortly after the start of the memory cycle, the given word that is to be entered

into storage is placed in the memory register. During the rewrite part of the cycle, the contents of the memory register are placed into storage. Thus, the contents of the specified location are replaced with the given value. The memory permits both single- and double-precision clear-write cycles.

The third type, read-alter-rewrite, is used for those processor instructions where the resultant of an arithmetic operation is placed in storage (such as Add Stored to A-ASA) and the indirect then tally address modifications. For the Read-Alter-Rewrite memory cycle the contents of the requested memory location is transferred to the system controller as in the Read-Restore cycle. The rewrite part of the cycle is delayed, however, until the communicating device e.g. processor or RT-IOC, processes the word just obtained and returns the altered value to the system controller for subsequent restorage. For example, in the instruction Add Stored to A, the contents of the specified memory location are transferred to the processor, added to the contents of the A Register, and the resulting sum returned to the memory for storage in the location from which the addend was obtained. Thus, an extra store instruction is not necessary.

In addition to single- and double-precision cycles, the memory also contains zone control to permit the reading of six-bit or nine-bit characters.

5. Instruction Execution Timing

The instruction execution times listed in Appendix J are based on fetching of instructions in pairs from memory. Unlike the GE-635, however, the M-605 does not perform overlap between the operation execution and the address modification and fetching of the operand of the next instruction. The execution of the even numbered instruction is completed before the address modification of the odd numbered instruction is started. The transfer of control instructions include the time to procure another instruction pair. If the transfer does not take place in a conditional transfer instruction, the execution time will be lower than indicated.

The instruction execution times of shift and floating-point operations are average times based on a five-shift step. A single shift step may effect a shift by one, four, or sixteen positions. Thus a shift of 22 positions will be executed in a four-shift step consisting of one 16-position, one 4-position, and two 1-position shifts. Each shift step takes approximately 0.24 micro-seconds.

III. INSTRUCTION REPERTOIRE

A. GENERAL REMARKS AND FORMAT

For the description of the machine instructions that follow it is assumed that the reader is familiar with the general structure of the processor, the representation of information, the data formats, and the method of address modifications, as presented in the preceding sections of this manual.

The M-605 instruction set described in this Section is arranged by functional class in two categories: Section III B describes the M-605 standard hardware implemented instructions; Section IIIC describes those instructions which are implemented by optional hardware. In those cases where the optional Floating-point hardware is not implemented, those instructions are software implemented by use of a Macro-operation. In some cases where the length of a Macro is prohibitive, a Macro-Subroutine combination is used, in which case the Macro serves as a linkage to the subroutine. The appendices to this manual listing the instruction set by both functional class and in alphabetical order afford convenient page references to all instructions in this section.

A fixed format is used for the description of each machine instruction, this is summarized in the comments following.

Mnemonic	Name of Instruction	Op Code (Octal)

Summary: (The change in the status of the system effected by the execution of the instruction is described in a short and generally symbolic form. If reference is made here to the status of an indicator, then it is the status of this indicator before the operation is executed.)

Modifications: (Those designators are listed explicitly that cannot be used with this instruction either because they are not permitted with this instruction or because their effect cannot be predicted from the general address modification procedure.)

Indicators Affected: (Only those indicators are listed whose status can be changed by the execution of this instruction. In most cases, a condition for setting ON as well as one for setting OFF is stated. Unless explicitly stated otherwise, the conditions refer to the contents of registers, etc., as existing after the execution of the instruction's operation.)

Notes: (This part of the description exists only in those cases where the SUMMARY is not sufficient for an understanding of the operation.)

Abbreviations and Symbols.

The following abbreviations and symbols are used for the description of the machine operations.

Registers:

A = A Register (36 bits)
Q = Q Register (36 bits)
AQ = Combined A-Q Register (72 bits)
Xn = Index Register n (n = 0, 1, . . . , 7) (18 bits)
E = Exponent Register (8 bits)
EA = Combined Exponent -A Register (8 + 36 bits)
EAQ = Combined Exponent-A-Q Register (8 + 72 bits)
BAR = Base Address Register (18 bits)
IC = Instruction Counter (18 bits)
IR = Indicator Register (18 bits, 11 of which are used at this time)
TR = Timer Register (24 bits)
Z = Temporary Pseudo-result of a non-store comparative operation.

Effective Address and Memory Locations:

Y = The effective address (18 bits) of the respective instruction.

Register Positions and Contents:

("R" standing for any of the registers listed above as well as for a memory location or a pair of memory locations.)

R_i = the ith position of R
$R_{i...j}$ = the positions i through j of R
C(R) = the contents of the full register R
$C(R)_i$ = the contents of the ith position of R
$C(R)_{i...j}$ = the contents of the positions i through j of R

When the description of an instruction states a change only for a part of a register or memory location, then it is always understood that the part of the register or memory location which is not mentioned remains unchanged.

Other Symbols:

$\Rightarrow$ = replaces

$\begin{matrix}\cdot\cdot \\ \cdot\cdot\end{matrix}$ = compare with

AND = the Boolean connective AND (symbol $\wedge$)

OR = the Boolean connective OR (symbol $\vee$)

$\not\equiv$ = the Boolean connective NON-EQUIVALENCE (or EXCLUSIVE OR)

DATA MOVEMENT - LOAD

LDA — Load A — 235_8

C(Y) ⇒ C(A)

SUMMARY: The contents of Y replace the contents of the A Register.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	If C(A)	= 0,	then ON;	otherwise OFF
Negative	If $C(A)_0$	= 1,	then ON;	otherwise OFF

LDQ — Load Q — 236_8

C(Y) ⇒ C(Q)

SUMMARY: The contents of Y replace the contents of the Q Register.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	If C(Q)	= 0,	then ON;	otherwise OFF
Negative	If $C(Q)_0$	= 1,	then ON;	otherwise OFF

LDXn — Load Xn — $22n_8$

$C(Y)_{0...17}$ ⇒ C(Xn)

SUMMARY: The contents of Y, bit positions 0 through 17, replace the contents of the Index Register specified by n.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	If C(Xn)	= 0,	then ON;	otherwise OFF
Negative	If $C(Xn)_0$	= 1,	then ON;	otherwise OFF

LDLXn — Load Xn in Lower — $72n_8$

$C(Y)_{18...35} \Rightarrow C(Xn)$

SUMMARY: The contents of Y, bits 18 through 35, replace the contents of the Index Register specified by n.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	If C(Xn) = 0, then ON; otherwise OFF
Negative	If $C(Xn)_0$ = 1, then ON; otherwise OFF

LDI — Load Indicator Register — 634_8

$C(Y)_{18...35} \Rightarrow C(IR)$

SUMMARY: The contents of Y, bit positions 18 through 35, replace the contents of the Indicator Register.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

All except Master Mode	If corresponding bit in C(Y) is ONE, then ON; otherwise OFF

NOTES:

1. The relation between bit positions of C(Y) and the indicators is as follows:

Bit Position	Indicators
18	Zero
19	Negative
20	Carry
21	Overflow
22	Exponent Overflow
23	Exponent Underflow
24	Overflow Mask
25	Tally Runout
26	Parity Error
27	Parity Mask
28	Master Mode
29	Not used at this time
30	
31	
32	
33	
34	
35	

2. The Tally Runout Indicator will reflect $C(Y)_{25}$ regardless of what address modification is performed on the LDI instruction (for Tally Operations).

LREG — Load Registers — 073_8

C(Y, Y + 1, ... Y + 7) ⇒ C(X0, X1, . . . X7, A, Q, E, TR)

SUMMARY: The contents of Y through Y + 7 replace the contents of the Index, A, Q, exponent, and Timer Registers.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

None

NOTES: 1.

$C(Y)_{0-17}$ ⇒ C(X0)		$C(Y+3)_{0-17}$ ⇒ C(X6)	
$C(Y)_{18-35}$ ⇒ C(X1)		$C(Y+3)_{18-35}$ ⇒ C(X7)	
$C(Y+1)_{0-17}$ ⇒ C(X2)		$C(Y+4)_{0-35}$ ⇒ C(A)	
$C(Y+1)_{18-35}$ ⇒ C(X3)		$C(Y+5)_{0-35}$ ⇒ C(Q)	
$C(Y+2)_{0-17}$ ⇒ C(X4)		$C(Y+6)_{0-7}$ ⇒ C(E)	
$C(Y+2)_{18-35}$ ⇒ C(X5)		* $C(Y+7)_{0-24}$ ⇒ C(TR)	

*2. The contents of the Timer Register are not changed when the processor is in Slave Mode.

LCA — Load Complement A — 335_8

- C(Y) ⇒ C(A)

SUMMARY: The two's complement of the contents of Y replace the contents of the A Register.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	If C(A) = 0, then ON; otherwise OFF
Negative	If $C(A)_0$ = 1, then ON; otherwise OFF
Overflow	If range of A is exceeded, then ON

LCQ — Load Complement Q — 336_8

$-C(Y) \Rightarrow C(Q)$

SUMMARY: The two's complement of the contents of Y replace the contents of the Q Register.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	If $C(Q) = 0$, then ON; otherwise OFF
Negative	If $C(Q)_0 = 1$, then ON; otherwise OFF
Overflow	If range of Q is exceeded, then ON

LCXn — Load Complement Xn — $32n_8$

$-C(Y)_{0...17} \Rightarrow C(Xn)$

SUMMARY: The two's complement of the contents of Y, bit positions 0 through 17, replace the contents of the Index Register specified by n.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	If $C(Xn) = 0$, then ON; otherwise OFF
Negative	If $C(Xn)_0 = 1$, then ON; otherwise OFF
Overflow	If range of Xn is exceeded, then ON

EAA — Effective Address to A — 635_8

$Y \Rightarrow C(A)_{0...17}$; $00...0 \Rightarrow C(A)_{18...35}$

SUMMARY: The address field of the instruction replaces the contents of bits 0 through 17 of the A register. Bit positions 18 through 35 of the A Register are set to zero.

MODIFICATIONS: All except DU, DL

INDICATORS AFFECTED:

Zero	If $C(A) = 0$,	then ON;	otherwise OFF
Negative	If $C(A)_0 = 1$,	then ON;	otherwise OFF

NOTE: This instruction, and the instructions EAQ and EAXn, facilitate interregister data movements. The data source is specified by the address modification, and the data destination by the operation of the instruction.

EAQ — Effective Address to Q — 636_8

$Y \Rightarrow C(Q)_{0...17}$; $00...0 \Rightarrow C(Q)_{18...35}$

SUMMARY: The address field of the instruction replaces the contents of bits 0 through 17 of the Q Register. Bit positions 18 through 35 of the Q Register are set to zero.

MODIFICATIONS: All except DU, DL

INDICATORS AFFECTED:

Zero	If $C(Q) = 0$,	then ON;	otherwise OFF
Negative	If $C(Q)_0 = 1$,	then ON;	otherwise OFF

NOTE: This instruction, and the instructions EAA and EAXn, facilitate interregister data movements. The data source is specified by the address modification, and the data destination by the operation of the instruction.

EAXn — Effective Address to Xn — $62n_8$

Y ⇒ C(Xn)

SUMMARY: The address field of the instruction replaces the contents of the Index Register specified by n.

MODIFICATIONS: All except DU, DL

INDICATORS AFFECTED:

Zero	If C(Xn) = 0,	then ON;	otherwise OFF
Negative	If $C(Xn)_0$ = 1,	then ON;	otherwise OFF

NOTE: This instruction, and the instructions EAA and EAQ facilitate interregister data movements. The data source is specified by the address modification, and the data destination by the operation of the instruction.

STA — Store A — 755_8

C(A) ⇒ C(Y)

SUMMARY: The contents of the A Register replace the contents of Y.

MODIFICATIONS: All except DU, DL

INDICATORS AFFECTED: None

STQ — Store Q — 756_8

C(Q) ⇒ C(Y)

SUMMARY: The contents of the Q Register replace the contents of Y.

MODIFICATIONS: All except DU, DL

INDICATORS AFFECTED: None

STXn — Store Xn — $74n_8$

$C(Xn) \Rightarrow C(Y)_{0...17}$

SUMMARY: The contents of the Index Register specified by n replace the contents of Y, bits 0 through 17.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED: None

STLXn — Store Xn in Lower — $44n_8$

$C(Xn) \Rightarrow C(Y)_{18...35}$

SUMMARY: The contents of the Index Register specified by n replace the contents of Y, bits 18 through 35.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED: None

SREG — Store Registers — 753_8

C(X0, X1, X2,....X7, A, Q, E, TR) ⇒ C(Y, Y+1,....Y+7)

SUMMARY: The contents of the Index, A, Q, Exponent, and Timer Registers replace the contents of Y through Y+7. Bits 8 through 35 of Y+6 and 24 through 35 of Y+7 are set to zero.

MODIFICATION: All except DU, DL, CI, SC

INDICATORS AFFECTED: None

NOTE:

C(X0) ⇒ $C(Y)_{0-17}$		C(X6) ⇒ $C(Y+3)_{0-17}$	
C(X1) ⇒ $C(Y)_{18-35}$		C(X7) ⇒ $C(Y+3)_{18-35}$	
C(X2) ⇒ $C(Y+1)_{0-17}$		C(A) ⇒ $C(Y+4)_{0-35}$	
C(X3) ⇒ $C(Y+1)_{18-35}$		C(Q) ⇒ $C(Y+5)_{0-35}$	
C(X4) ⇒ $C(Y+2)_{0-17}$		C(E) ⇒ $C(Y+6)_{0-7}$; 00..0	⇒ $C(Y+6)_{8-35}$
C(X5) ⇒ $C(Y+2)_{18-35}$		C(TR) ⇒ $C(Y+7)_{0-23}$; 00...0	⇒ $C(Y+7)_{24-35}$

STCA — Store Characters of A (Six Bit) — 751_8

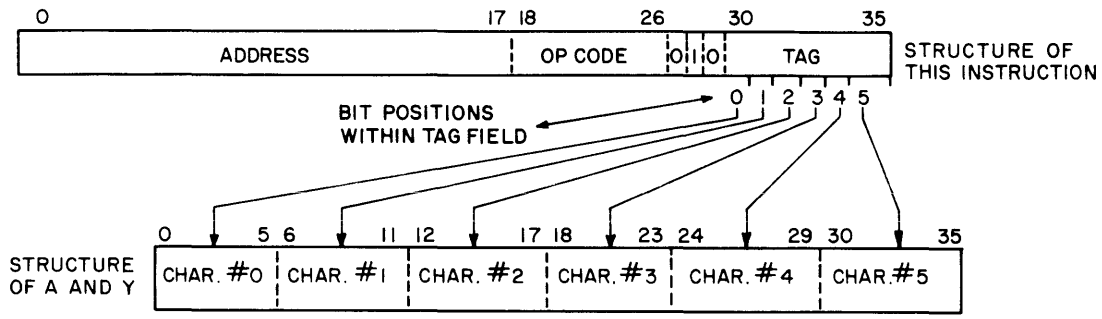
Characters of C(A) ⇒ corresponding characters of C(Y).

SUMMARY: The contents of the selected 6 bit characters of the A Register replace the contents of the corresponding 6 bit characters of Y. Character positions are specified in the instruction tag field.

MODIFICATIONS: No modification can take place

INDICATORS AFFECTED: None

NOTE: Binary ones in the tag field of this instruction specify the character positions of A and Y that are affected by this instruction. The control relation is shown in the diagram below.



STCQ Store Characters of Q (Six Bit) 752_8

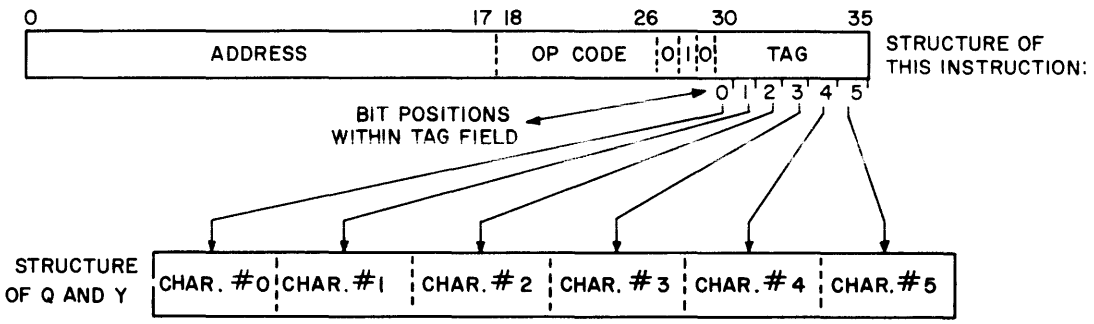
Characters of C(Q) ⇒ corresponding characters of C(Y).

SUMMARY: The contents of the selected 6 bit characters of the Q Register replace the contents of the corresponding 6 bit characters of Y. Character positions are specified in the instruction tag field.

MODIFICATIONS: No modification can take place

INDICATORS AFFECTED: None

NOTE: Binary ones in the tag field of this instruction specify the character positions of Q and Y that are affected by this instruction. The control relation is shown in the diagram below.



STBA Store Characters of A (Nine Bit) 551_8

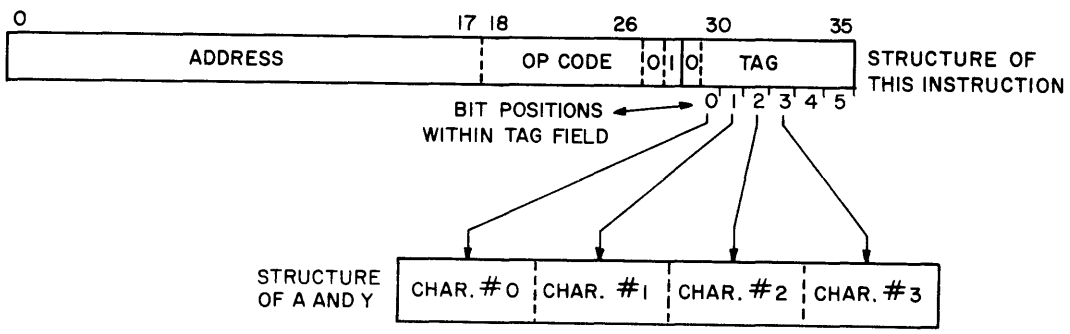
Characters of C(A) ⇒ corresponding characters of C(Y).

SUMMARY: The contents of the specified 9 bit characters of the A Register replace the contents of the corresponding 9 bit characters of Y. Character positions are specified in the instruction tag field.

MODIFICATIONS: No modification can take place

INDICATORS AFFECTED: None

NOTE: Binary ones in the tag field of this instruction specify the character positions of A and Y that are affected by this instruction. The control relation is shown in the diagram below.



STBQ Store Characters of Q (Nine Bit) 552_8

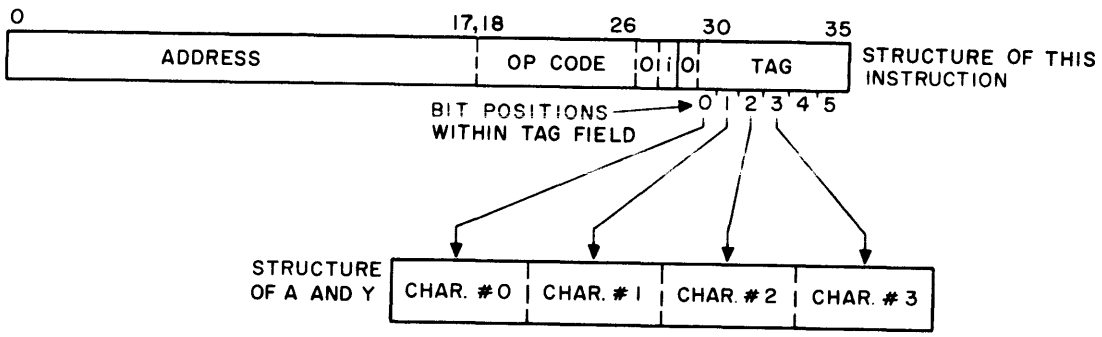
Characters of C(Q) ⇒ corresponding characters of C(Y).

SUMMARY: The contents of the specified 9 bit characters of the Q Register replace the contents of the corresponding 9 bit characters of Y. Character positions are specified in the instruction tag field.

MODIFICATIONS: No modification can take place

INDICATORS AFFECTED: None

NOTE: Binary ones in the tag field of this instruction specify the character positions of A and Y that are affected by this instruction. The control relation is shown in the diagram below.



STI — Store Indicator Register — 754_8

$C(IR) \Rightarrow C(Y)_{18...35}$

SUMMARY: The contents of the Indicator Register replace the contents of Y, bit positions 18 through 35.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED: None

NOTE: 1. The relation between bit positions of C(Y) and the indicators is as follows:

Bit Position	Indicators
18	Zero
19	Negative
20	Carry
21	Overflow
22	Exponent Overflow
23	Exponent Underflow
24	Overflow Mask
25	Tally Runout
26	Parity Error
27	Parity Mask
28	Master Mode
29	Not used at this time; these indicators appear always as if being set OFF
30	
31	
32	
33	
34	
35	

2. The ON state corresponds to a ONE bit, the OFF state to a ZERO bit.

3. The $C(Y)_{25}$ will contain the state of the Tally Runout Indicator prior to address modification of the STI instruction (for Tally operations).

STT — Store Timer Register — 454_8

$C(TR) \Rightarrow C(Y)_{0...23}$
$00...0 \Rightarrow C(Y)_{24...35}$

SUMMARY: The contents of the Timer Register replace the contents of Y, bit positions 0 through 23. Bit positions 24 through 35 are set to zero.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED: None

SBAR — Store Base Address Register — 550_8

$C(BR) \Rightarrow C(Y)_{0...17}$

SUMMARY: The contents of the Base Address Register replace the contents of Y, bit positions 0 through 17.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED: None

STZ — Store Zero — 450_8

$00...0 \Rightarrow C(Y)$

SUMMARY: The contents of Y are replaced with zeros.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED: None

STC1 — Store Instruction Counter Plus One — 554_8

$C(IC) + 0...01 \Rightarrow C(Y)_{0...17}$

$C(IR) \Rightarrow C(Y)_{18...35}$

(Note the difference between STC1 and STC2!)

SUMMARY: The contents of the Instruction Counter plus one replace the contents of Y, bits positions 0 through 17. The contents of the Indicator Register replace the contents of Y, bit positions 18 through 35.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED: None

NOTES:

1. The relation between bit positions of C(Y) and the indicators is as follows:

Bit Position	Indicators
18	Zero
19	Negative
20	Carry
21	Overflow
22	Exponent Overflow
23	Exponent Underflow
24	Overflow Mask
25	Tally Runout
26	Parity Error
27	Parity Mask
28	Master Mode
29	Not used at this time; these indicators appear always as if being set OFF
30	
31	
32	
33	
34	
35	

2. The ON state corresponds to a ONE bit, the OFF state to a ZERO bit.

3. The $C(Y)_{25}$ will contain the state of the Tally Runout Indicator prior to address modification of the STC1 instruction (for Tally operations).

STC2 Store Instruction Counter Plus Two 750_8

$C(IC) + 0...010 \Rightarrow C(Y)_{0...17}$ (Note the difference between STC1 and STC2!)

SUMMARY: The contents of the Instruction Counter plus two replace the contents of Y, bit positions 0 through 17.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED: None

ARS — A Right Shift — 731_8

Shift right C(A) by $Y_{11...17}$ positions; fill vacated positions with $C(A)_0$

SUMMARY: The contents of the A Register are shifted right the number of positions specified in bit positions 11 through 17 of the instruction address field. Positions vacated by the shift are filled with the contents of the A Register, bit 0 (sign bit).

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(A) = 0,	then ON;	otherwise OFF
Negative	If $C(A)_0$ = 1,	then ON;	otherwise OFF

QRS — Q Right Shift — 732_8

Shift right C(Q) by $Y_{11...17}$ positions; fill vacated positions with $C(Q)_0$

SUMMARY: The contents of the Q Register are shifted right the number of positions specified in bit positions 11 through 17 of the instruction address field. Positions vacated by the shift are filled with the contents of the Q Register, bit 0 (sign bit).

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(Q) = 0,	then ON;	otherwise OFF
Negative	If $C(Q)_0$ = 1,	then ON;	otherwise OFF

LRS — Long Right Shift — 733_8

Shift right C(AQ) by $Y_{11...17}$ positions; fill vacated positions with $C(AQ)_0$

SUMMARY: The contents of the combined A and Q Registers are shifted right the number of positions specified in bit positions 11 through 17 of the instruction address field. The vacated positions are filled with the contents of the A Register, bit 0 (sign bit).

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If	C(AQ) = 0,	then ON;	otherwise OFF
Negative	If	$C(AQ)_0$ = 1,	then ON;	otherwise OFF

ALS — A Left Shift — 735_8

Shift left C(A) by $Y_{11...17}$ positions, fill vacated positions with zeros

SUMMARY: The contents of the A Register are shifted left the number of positions specified in bit positions 11 through 17 of the instruction address field. Positions vacated are filled with zeros.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If	C(A) = 0,	then ON;	otherwise OFF
Negative	If	$C(A)_0$ = 1,	then ON;	otherwise OFF
Carry	If	$C(A)_0$	ever changes during the shift,	then ON; otherwise OFF

QLS — Q Left Shift — 736_8

Shift left C(Q) by $Y_{11...17}$ positions; fill vacated positions with zeros

SUMMARY: The contents of the Q Register are shifted left the number of positions specified in bit positions 11 through 17 of the instruction address field. Positions vacated are filled with zeros.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(Q) = 0, then ON; otherwise OFF
Negative	If $C(Q)_0$ = 1, then ON; otherwise OFF
Carry	If $C(Q)_0$ ever changes during the shift, then ON; otherwise OFF

LLS — Long Left Shift — 737_8

Shift left C(AQ) by $Y_{11...17}$ positions; fill vacated positions with zeros

SUMMARY: The contents of the combined A and Q Registers are shifted left the number of positions specified in bit positions 11 through 17 of the instruction address field. Positions vacated are filled with zeros.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF
Carry	If $C(AQ)_0$ ever changes during the shift, then ON; otherwise OFF

ARL — A Right Logic — 771_8

Shift right C(A) by $Y_{11...17}$ positions; fill vacated positions with zeros

SUMMARY: The contents of the A Register are shifted right the number of positions specified in bit positions 11 through 17 of the instruction address field. Positions vacated are filled with zeros.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(A) = 0, then ON; otherwise OFF
Negative	If $C(A)_0$ = 1, then ON; otherwise OFF

QRL — Q Right Logic — 772_8

Shift right C(Q) by $Y_{11...17}$ positions; fill vacated positions with zeros

SUMMARY: The contents of the Q Register are shifted right the number of positions specified in bit positions 11 through 17 of the instruction address field. Positions vacated are filled with zeros.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(Q) = 0, then ON; otherwise OFF
Negative	If $C(Q)_0$ = 1, then ON; otherwise OFF

LRL — Long Right Logic — 773_8

Shift right C(AQ) by $Y_{11...17}$ positions; fill vacated positions with zeros

SUMMARY: The contents of the combined A and Q Registers are shifted right the number of positions specified in bit positions 11 through 17 of the instruction address field. Positions vacated are filled with zeros.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF

ALR — A Left Rotate — 775_8

Rotate C(A) by left $Y_{11...17}$ positions; enter each bit leaving position 0 into position 35

SUMMARY: The contents of the A Register are rotated, bit position 0 into bit position 35, the number of positions specified in bit positions 11 through 17 of the instruction address field.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(A) = 0, then ON; otherwise OFF
Negative	If $C(A)_0$ = 1, then ON; otherwise OFF

QLR — Q Left Rotate — 776_8

Rotate C(Q) left by $Y_{11...17}$ positions; enter each bit leaving position 0 into position 35

SUMMARY: The contents of the Q Register are rotated, bit 0 into bit 35, the number of positions specified in bit positions 11 through 17 of the instructions address field.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(Q) = 0, then ON; otherwise OFF
Negative	If $C(Q)_0$ = 1, then ON; otherwise OFF

LLR — Long Left Rotate — 777_8

Rotate C(AQ) left by $Y_{11...17}$ positions; enter each bit leaving position 0 into position 71

SUMMARY: The contents of the combined A and Q Registers are rotated, bit 0 into bit 71, the number of positions specified in bit positions 11 through 17 of the instruction address field.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF

ADA — Add to A — 075_8

$C(A) + C(Y) \Rightarrow C(A)$

SUMMARY: The contents of Y are added to the contents of the A Register and the result replaces the contents of the A Register.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	If $C(A) = 0$, then ON; otherwise OFF
Negative	If $C(A)_0 = 1$, then ON; otherwise OFF
Overflow	If range of A is exceeded, then ON
Carry	If a carry out of A_0 is generated, then ON; otherwise OFF

ADQ — Add to Q — 076_8

$C(Q) + C(Y) \Rightarrow C(Q)$

SUMMARY: The contents of Y are added to the contents of the Q Register and the result replaces the contents of the Q Register

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	If $C(Q) = 0$, then ON; otherwise OFF
Negative	If $C(Q)_0 = 1$, then ON; otherwise OFF
Overflow	If range of Q is exceeded, then ON
Carry	If a carry out of Q_0 is generated, then ON; otherwise OFF

ADXn — Add to Xn — $06n_8$

$C(Xn) + C(Y)_{0...17} \Rightarrow C(Xn)$

SUMMARY: The contents of Y, bit positions 0 through 17, are added to the contents of the Index Register specified by n and the result replaces the contents of that Index Register.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	If $C(Xn) = 0$, then ON; otherwise OFF
Negative	If $C(Xn)_0 = 1$, then ON; otherwise OFF
Overflow	If range of Xn is exceeded; then ON
Carry	If a carry out of Xn_0 is generated, then ON; otherwise OFF

ASA — Add Stored to A — 055_8

$C(A) + C(Y) \Rightarrow C(Y)$

SUMMARY: The contents of Y are added to the contents of the A Register and the result replaces the contents of Y.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If $C(Y) = 0$, then ON; otherwise OFF
Negative	If $C(Y)_0 = 1$, then ON; otherwise OFF
Overflow	If range of Y is exceeded, then ON
Carry	If a carry out of Y_0 is generated, then ON; otherwise OFF

ASQ — Add Stored to Q — 056_8

$C(Q) + C(Y) \Rightarrow C(Y)$

SUMMARY: The contents of Y are added to the contents of the Q Register and the result replaces the contents of Y.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If $C(Y) = 0$, then ON; otherwise OFF
Negative	If $C(Y)_0 = 1$, then ON; otherwise OFF
Overflow	If range of Y is exceeded, then ON
Carry	If a carry out of Y_0 is generated, then ON; otherwise OFF

ASXn — Add Stored to Xn — $04n_8$

$C(Xn) + C(Y)_{0...17} \Rightarrow C(Y)_{0...17}$

SUMMARY: The contents of Y, bit positions 0 through 17, are added to the contents of the Index Register specified by n and the result replaces the contents of Y, bit positions 0 through 17.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If $C(Y)_{0...17} = 0$, then ON; otherwise OFF
Negative	If $C(Y)_0 = 1$, then ON; otherwise OFF
Overflow	If range of $Y_{0...17}$ exceeded, then ON
Carry	If a carry out of Y_0 is generated, then ON; otherwise OFF

ADLA — Add Logic to A — 035_8

C(A) + C(Y) ⇒ C(A)

SUMMARY: The contents of Y are added to the contents of the A Register and the result replaces the contents of the A Register.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	If C(A) = 0, then ON; otherwise OFF
Negative	If $C(A)_0$ = 1, then ON; otherwise OFF
Carry	If a carry out of A_0 is generated then ON, otherwise OFF

NOTE: This instruction is identical to the ADA instruction, except the Overflow Indicator is not affected by this instruction.

ADLQ — Add Logic to Q — 036_8

C(Q) + C(Y) ⇒ C(Q)

SUMMARY: The contents of Y are added to the contents of the Q Register and the result replaces the contents of the Q Register.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	If C(Q) = 0, then ON; otherwise OFF
Negative	If $C(Q)_0$ = 1, then ON; otherwise OFF
Carry	If a carry out of Q_0 is generated then ON; otherwise OFF

NOTE: This instruction is identical to the ADQ instruction, except the Overflow Indicator is not affected by this instruction.

ADLXn — Add Logic to Xn — $02n_8$

$C(Xn) + C(Y)_{0...17} \Rightarrow C(Xn)$

SUMMARY: The contents of Y, bit positions 0 through 17 are added to the contents of the Index Register specified by n and the result replaces the contents of that Index Register.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	If $C(Xn) = 0$, then ON; otherwise OFF
Negative	If $C(Xn)_0 = 1$, then ON; otherwise OFF
Carry	If a carry out of Xn_0 is generated, then ON; otherwise OFF

NOTE: This instruction is identical to the ADXn instruction, except the Overflow Indicator is not affected by this instruction.

AWCA — Add with Carry to A — 071_8

Carry Indicator OFF: $C(A) + C(Y) \Rightarrow C(A)$

Carry Indicator ON: $C(A) + C(Y) + 0...01 \Rightarrow C(A)$

SUMMARY: The contents of Y are added to the contents of the A Register and the result replaces the contents of the A Register. If the Carry Indicator is ON before the addition takes place, a 1 is added to the result.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	If $C(A) = 0$, then ON; otherwise OFF
Negative	If $C(A)_0 = 1$, then ON; otherwise OFF
Overflow	If range of A is exceeded, then ON
Carry	If a carry out of A_0 is generated, then ON; otherwise OFF

AWCQ — Add with Carry to Q — 072_8

Carry Indicator OFF: C(Q) + C(Y) ⇒ C(Q)

Carry Indicator ON: C(Q) + C(Y) + 0...01 ⇒ C(Q)

SUMMARY: The contents of Y are added to the contents of the Q Register and the result replaces the contents of the Q Register. If the Carry Indicator is ON before the addition takes place, 1 is added to the result.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	If C(Q) = 0, then ON; otherwise OFF
Negative	If $C(Q)_0$ = 1, then ON; otherwise OFF
Overflow	If range of Q is exceeded, then ON
Carry	If carry out of Q_0 is generated, then ON; otherwise OFF

ADL — Add Low to AQ — 033_8

C(AQ) + C(Y), right adjusted, ⇒ C(AQ)

SUMMARY: The sign bit of the contents of Y (Y_0) is extended 36 bits. The resultant 72 bit number is added to the contents of the combined A and Q Registers and the results replace the contents of the A and Q Registers.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF
Overflow	If range of AQ is exceeded, then ON
Carry	If a carry out of AQ_0 is generated, then ON; otherwise OFF

AOS — Add One to Storage — 054_8

C(Y) + 0...01 ⇒ C(Y)

SUMMARY: The contents of Y are incremented by 1.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(Y) = 0, then ON; otherwise OFF
Negative	If $C(Y)_0$ = 1, then ON; otherwise OFF
Overflow	If range of Y is exceeded, then ON
Carry	If a carry out of Y_0 is generated, then ON; otherwise OFF

SBA — Subtract from A — 175_8

C(A) - C(Y) ⇒ C(A)

SUMMARY: The contents of Y are subtracted from the contents of the A Register and the result replaces the contents of the A Register

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	If C(A) = 0, then ON; otherwise OFF
Negative	If $C(A)_0$ = 1, then ON; otherwise OFF
Overflow	If range of A is exceeded, then ON
Carry	If a carry out of A_0 is generated, then ON; otherwise OFF

SBQ — Subtract from Q — 176_8

C(Q) - C(Y) ⇒ C(Q)

SUMMARY: The contents of Y are subtracted from the contents of the Q Register and the result replaces the contents of the Q Register.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	If C(Q) = 0, then ON; otherwise OFF
Negative	If $C(Q)_0$ = 1, then ON; otherwise OFF
Overflow	If range of Q is exceeded, then ON
Carry	If a carry out of Q_0 is generated, then ON; otherwise OFF

SBXn — Subtract from Xn — $16n_8$

$C(Xn) - C(Y)_{0...17} \Rightarrow C(Xn)$

SUMMARY: The contents of Y, bit positions 0 through 17, are subtracted from the contents of the Index Register specified by n and the result replaces the contents of the Index Register.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	If $C(Xn) = 0$, then ON; otherwise OFF
Negative	If $C(Xn)_0 = 1$, then ON; otherwise OFF
Overflow	If range of Xn is exceeded, then ON
Carry	If a carry out of Xn_0 is generated, then ON; otherwise OFF

SSA — Subtract Stored from A — 155_8

$C(A) - C(Y) \Rightarrow C(Y)$

SUMMARY: The contents of Y are subtracted from the contents of the A Register and the result replaces the contents of Y.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If $C(Y) = 0$, then ON; otherwise OFF
Negative	If $C(Y)_0 = 1$, then ON; otherwise OFF
Overflow	If range of Y is exceeded, then ON
Carry	If a carry out of Y_0 is generated, then ON; otherwise OFF

SSQ — Subtract Stored from Q — 156_8

C(Q) - C(Y) ⇒ C(Y)

SUMMARY: The contents of Y are subtracted from the contents of the Q Register and the result replaces the contents of Y.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(Y) = 0, then ON; otherwise OFF
Negative	If $C(Y)_0$ = 1, then ON; otherwise OFF
Overflow	If range of Y is exceeded, then ON
Carry	If a carry out of Y_0 is generated, then ON; otherwise OFF

SSXn — Subtract Stored from Xn — $14n_8$

C(Xn) - $C(Y)_{0...17}$ ⇒ $C(Y)_{0...17}$

SUMMARY: The contents of Y, bits positions 0 through 17, are subtracted from the contents of the Index Register specified by n and the result replaces bits 0 through 17 of the contents of Y.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If $C(Y)_{0...17}$ = 0, then ON, otherwise OFF
Negative	If $C(Y)_0$ = 1, then ON, otherwise OFF
Overflow	If range of $Y_{0...17}$ is exceeded, then ON
Carry	If a carry out of Y_0 is generated, then ON; otherwise OFF

SBLA — Subtract Logic from A — 135_8

C(A) - C(Y) ⇒ C(A)

SUMMARY: The contents of Y are subtracted from the contents of the A Register and the result replaces the contents of the A Register.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	If C(A) = 0, then ON; otherwise OFF
Negative	If $C(A)_0$ = 1, then ON; otherwise OFF
Carry	If a carry out of A_0 is generated, then ON; otherwise OFF

NOTE: This instruction is identical to the SBA instruction, except the Overflow Indicator is not affected by this instruction.

SBLQ — Subtract Logic from Q — 136_8

C(Q) - C(Y) ⇒ C(Q)

SUMMARY: The contents of Y are subtracted from the contents of the Q Register and the result replaces the contents of the Q Register.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	If C(Q) = 0, then ON; otherwise OFF
Negative	If $C(Q)_0$ = 1, then ON; otherwise OFF
Carry	If a carry out of Q_0 is generated, then ON; otherwise OFF

NOTE: This instruction is identical to the SBQ instruction except the Overflow Indicator is not affected by this instruction.

SBLXn — Subtract Logic from Xn — $12n_8$

$C(Xn) - C(Y)_{0...17} \Rightarrow C(Xn)$

SUMMARY: The contents of Y, bit positions 0 through 17, are subtracted from the contents of the Index Register specified by n and the result replaces the contents of that Index Register.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	If C(Xn) = 0, then ON; otherwise OFF
Negative	If $C(Xn)_0$ = 1, then ON; otherwise OFF
Carry	If a carry out of Xn_0 is generated, then ON; otherwise OFF

NOTE: This instruction is identical to the SBXn instruction except the Overflow Indicator is not affected by this instruction.

SWCA — Subtract with Carry from A — 171_8

Carry Indicator ON: $C(A) - C(Y) \Rightarrow C(A)$

Carry Indicator OFF: $C(A) - C(Y) - 0...01 \Rightarrow C(A)$

SUMMARY: The contents of Y are subtracted from the contents of the A Register and the result replaces the contents of the A Register. If the Carry Indicator is OFF before the subtraction takes place, 1 is subtracted from the result.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	If C(A) = 0, then ON; otherwise OFF
Negative	If $C(A)_0$ = 1, then ON; otherwise OFF
Overflow	If range of A is exceeded, then ON
Carry	If a carry out of A_0 is generated, then ON; otherwise OFF

NOTE: This instruction is used for multiple-word precision arithmetic. The SUMMARY can also be worded as follows in order to show the intended use:

Carry Indicator ON: C(A) + 1's complement of C(Y) + 0...01 $\Rightarrow$ C(A)

Carry Indicator OFF: C(A) + 1's complement of C(Y) $\Rightarrow$ C(A)

(The +1 which is added in the first case represents the carry from the next lower part of the multiple-length subtraction.)

SWCQ — Subtract with Carry from Q — 172_8

Carry Indicator ON: C(Q) - C(Y) ⇒ C(Q)

Carry Indicator OFF: C(Q) - C(Y) - 0...01 ⇒ C(Q)

SUMMARY: The contents of Y are subtracted from the contents of the Q Register and the result replaces the contents of the Q Register. If the Carry Indicator is OFF before the subtraction takes place, 1 is subtracted from the result.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	If C(Q) = 0, then ON; otherwise OFF
Negative	If $C(Q)_0$ = 1, then ON; otherwise OFF
Overflow	If range of Q is exceeded, then ON
Carry	If carry out of Q_0 is generated, then ON; otherwise OFF

NOTE: This instruction is used for multiple-word precision arithmetic. The SUMMARY can also be worded as follows in order to show the intended use:

Carry Indicator ON: C(Q) + 1's complement of C(Y) + 0...01 ⇒ C(Q)

Carry Indicator OFF: C(Q) + 1's complement of C(Y) ⇒ C(Q)

(The +1 which is added in the first case represents the carry from the next lower part of the multiple-length subtraction).

MPY Multiply Integer 402_8

$C(Q) \times C(Y) \Rightarrow C(AQ)$, right-adjusted

SUMMARY: The contents of the Q Register are multiplied by the contents of Y and the results, right-adjusted, replaces the contents of the combined A and Q Registers.

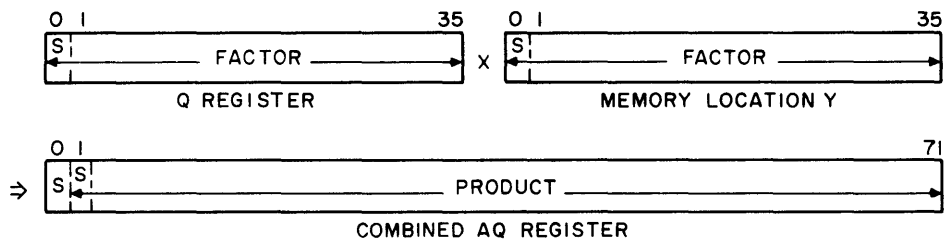
MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF

NOTES:

1. Two 36-bit integer factors (including sign) are multiplied to form a 71-bit integer product (including sign), which is stored in AQ, right-adjusted. Bit position AQ_0 is filled with an "extended sign bit".



2. In the case of $(-2^{35}) \times (-2^{35}) = +2^{70}$, the position AQ_1 is used to represent this product without causing an overflow.

MPF Multiply Fraction 401_8

C(A) × C(Y) ⇒ C(AQ), left-adjusted

SUMMARY: The contents of the A Register are multiplied by the contents of Y and the result, left-adjusted, replaces the contents of the combined A and Q Registers.

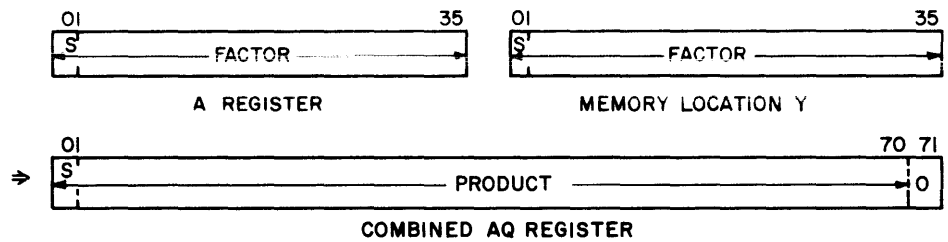
MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF
Overflow	If range of AQ is exceeded, then ON

NOTES:

1. Two 36-bit fractional factors (including sign) are multiplied to form a 71-bit fractional product (including sign), which is stored in AQ, left-adjusted. Bit position AQ_{71} is filled with a zero bit.



2. An overflow can occur only in the case (-1) × (-1).

DIV — Divide Integer — 506_8

C(Q) ÷ C(Y); integer quotient ⇒ C(Q)
fractional remainder ⇒ C(A)

SUMMARY: The contents of the Q Register are divided by the contents of Y. The results replace the contents of the A and Q Registers with the integer quotient in the Q Register and the fractional remainder in the A Register.

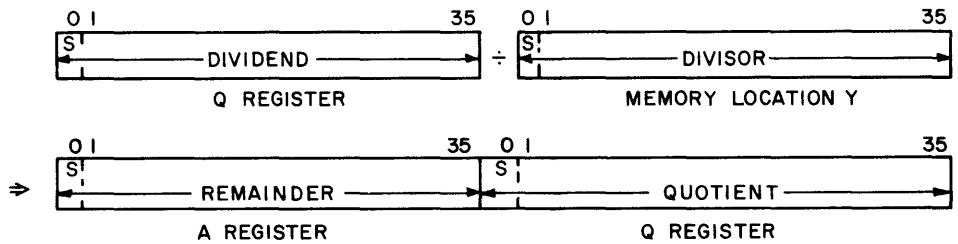
MODIFICATIONS: All

INDICATORS AFFECTED:

	If division takes place:	If no division takes place:
Zero	If C(Q) = 0, then ON; otherwise OFF	If divisor = 0, then ON; otherwise OFF
Negative	If $C(Q)_0$ = 1, then ON; otherwise OFF	If dividend < 0, then ON; otherwise OFF

NOTES:

1. A 36 bit integer dividend (including sign) is divided by a 36 bit integer divisor (including sign) to form a 36-bit integer quotient (including sign) and a 36 bit fractional remainder (including sign). The remainder sign is equal to the dividend sign unless the remainder is zero.



2. If dividend = -2^{35} and divisor = -1 or if divisor = 0, then the division itself does not take place.

   Instead, a Divide-Check Fault Trap occurs; the divisor C(Y) remains unchanged. C(Q) contains the dividend magnitude in absolute, and the Negative Indicator reflects the dividend sign.

DVF Divide Fraction 507_8

C(AQ) ÷ C(Y); fractional quotient ⇒ C(A)
remainder ⇒ C(Q)

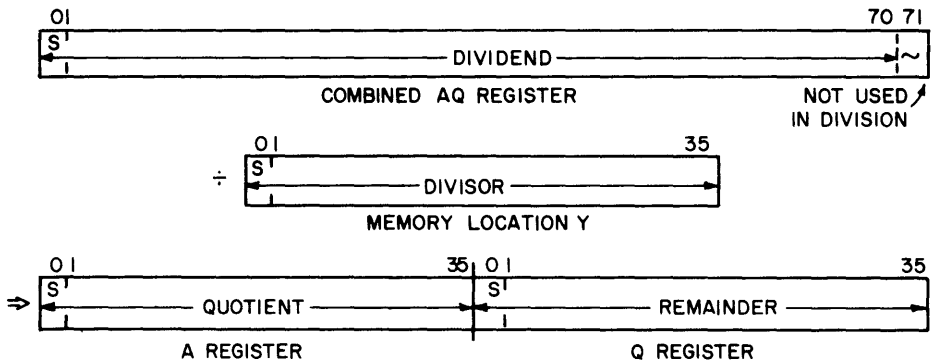
SUMMARY: The contents of the combined A and Q Registers are divided by the contents of Y. The results replace the contents of the A and Q Register, with the fractional quotient in the A Register and the remainder in the Q Register.

MODIFICATIONS: All

INDICATORS AFFECTED:

	If division takes place:	If no division takes place:
Zero	If C(A) = 0, then ON; otherwise OFF	If divisor = 0, then ON; otherwise OFF
Negative	If $C(A)_0$ = 1, then ON; otherwise OFF	If dividend < 0, then ON; otherwise OFF

NOTES: 1. A 71-bit fractional dividend (including sign) is divided by a 36-bit fractional divisor (including sign) to form a 36-bit fractional quotient (including sign) and a 36-bit remainder (including sign), bit position 35 of the remainder corresponding to bit position 70 of the dividend. The remainder sign is equal to the dividend sign unless the remainder is zero.



2. If |dividend| ≥ |divisor| or if divisor = 0, then the division itself does not take place.

   Instead, a Divide-Check Fault Trap occurs; the divisor C(Y) remains unchanged, C(AQ) contains the dividend magnitude in absolute, and the Negative Indicator reflects the dividend sign.

NEG — Negate A — 531_8

$-C(A) \Rightarrow C(A)$

SUMMARY: The contents of the A Register are negated by forming the two's complement and the result replaces the contents of the A Register.

MODIFICATIONS: Are without any effect on the operation

INDICATORS AFFECTED:

Zero	If C(A) = 0, then ON; otherwise OFF
Negative	If $C(A)_0$ = 1, then ON; otherwise OFF
Overflow	If range of A is exceeded, then ON

NEGL — Negate Long — 533_8

$-C(AQ) \Rightarrow C(AQ)$

SUMMARY: The contents of the combined A and Q Registers are negated by forming the two's complement and the result replaces the contents of the A and Q Registers.

MODIFICATIONS: Are without any effect on the operation

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF
Overflow	If range of AQ is exceeded, then ON

ANA — AND to A — 375_8

$C(A)_i$ AND $C(Y)_i \Rightarrow C(A)_i$ for all i = 0, 1, . . . , 35

SUMMARY: The logical AND of each bit of the contents of the A Register and the corresponding bit of the contents of Y replace the contents of the A Register.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	If C(A) = 0, then ON; otherwise OFF
Negative	If $C(A)_0$ = 1, then ON; otherwise OFF

ANQ — AND to Q — 376_8

$C(Q)_i$ AND $C(Y)_i \Rightarrow C(Q)_i$ for all i = 0, 1, . . . , 35

SUMMARY: The logical AND of each bit of the contents of the Q Register and the corresponding bit of the contents of Y replaces the contents of the Q Register.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	If C(Q) = 0, then ON; otherwise OFF
Negative	If $C(Q)_0$ = 1, then ON; otherwise OFF

ANXn — AND to Xn (n = 0, 1, . . . , 7) — $36n_8$

$C(Xn)_i$ AND $C(Y)_i \Rightarrow C(Xn)_i$ for all i = 0, 1, . . . , 17

SUMMARY: The logical AND of each bit of the contents of the Index Register specified by n and the corresponding bit of the contents of Y replace the contents of that Index Register.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	If C(Xn) = 0, then ON; otherwise OFF
Negative	If $C(Xn)_0$ = 1, then ON; otherwise OFF

ANSA — AND to Storage A — 355_8

$C(A)_i$ AND $C(Y)_i \Rightarrow C(Y)_i$ for all i = 0, 1, . . . , 35

SUMMARY: The logical AND of each bit of the contents of the A Register and the corresponding bit of the contents of Y replace the contents of Y.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(Y) = 0, then ON; otherwise OFF
Negative	If $C(Y)_0$ = 1, then ON; otherwise OFF

ANSQ — AND to Storage Q — 356_8

$C(Q)_i$ AND $C(Y)_i \Rightarrow C(Y)_i$ for all i = 0, 1, ... , 35

SUMMARY: The logical AND of each bit of the contents of the Q Register and the corresponding bit of the contents of Y replace the contents of Y.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(Y) = 0, then ON; otherwise OFF
Negative	If $C(Y)_0$ = 1, then ON; otherwise OFF

ANSXn — AND to Storage Xn — $34n_8$

$C(Xn)_i$ AND $C(Y)_i \Rightarrow C(Y)_i$ for all i = 0, 1, . . . , 17

SUMMARY: The logical AND of each bit of the contents of the Index Register specified by n and the corresponding bit of the contents of Y replace the contents of Y.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If $C(Y)_{0...17}$ = 0, then ON; otherwise OFF
Negative	If $C(Y)_0$ = 1, then ON; otherwise OFF

ORA — OR to A — 275_8

$C(A)_i$ OR $C(Y)_i \Rightarrow C(A)_i$ for all i = 0, 1, . . . , 35

SUMMARY: The logical OR of each bit of the contents of the A Register and the corresponding bit of the contents of Y replaces the contents of the A Register.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	If C(A) = 0, then ON; otherwise OFF
Negative	If $C(A)_0$ = 1, then ON; otherwise OFF

ORQ — OR to Q — 276_8

$C(Q)_i$ OR $C(Y)_i \Rightarrow C(Q)_i$ for all i = 0, 1, . . . , 35

SUMMARY: The logical OR of each bit of the contents of the Q Register and the corresponding bit of the contents of Y replaces the contents of the Q Register.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	If C(Q) = 0, then ON; otherwise OFF
Negative	If $C(Q)_0$ = 1, then ON; otherwise OFF

ORXn — OR to Xn — $26n_8$

$C(Xn)_i$ OR $C(Y)_i \Rightarrow C(Xn)_i$ for all i = 0, 1, . . . , 17

SUMMARY: The logical OR of each bit of the contents of the Index Register specified by n and the corresponding bit of the contents of Y replaces the contents of that Index Register.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	If C(Xn) = 0, then ON; otherwise OFF
Negative	If $C(Xn)_0$ = 1, then ON; otherwise OFF

ORSA — OR to Storage A — 255_8

$C(A)_i$ OR $C(Y)_i \Rightarrow C(Y)_i$ for all i = 0, 1, . . . , 35

SUMMARY: The logical OR of each bit of the contents of the A Register and the corresponding bit of the contents of Y replaces the contents of Y.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero — If C(Y) = 0, then ON; otherwise OFF

Negative — If $C(Y)_0$ = 1, then ON; otherwise OFF

ORSQ — OR to Storage Q — 256_8

$C(Q)_i$ OR $C(Y)_i \Rightarrow C(Y)_i$ for all i = 0, 1, . . . , 35

SUMMARY: The logical OR of each bit of the contents of the Q Register and the corresponding bit of the contents of Y replaces the contents of Y.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero — If C(Y) = 0, then ON; otherwise OFF

Negative — If $C(Y)_0$ = 1, then ON; otherwise OFF

ORSXn — OR to Storage Xn — $24n_8$

$C(Xn)_i$ OR $C(Y)_i \Rightarrow C(Y)_i$ for all i = 0, 1, . . . , 17

SUMMARY: The logical OR of each bit of the contents of the Index Register specified by n and the corresponding bit of the contents of Y replaces the contents of Y.

MODIFICATIONS: For all except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero — If $C(Y)_{0...17}$ = 0, then ON; otherwise OFF

Negative — If $C(Y)_0$ = 1, then ON; otherwise OFF

ERA — EXCLUSIVE OR to A — 675_8

$C(A)_i \not\equiv C(Y)_i \Rightarrow C(A)_i$ for i = 0, 1, . . . , 35

SUMMARY: The logical EXCLUSIVE OR of each bit of the contents of the A Register and the corresponding bit of the contents of Y replaces the contents of the A Register.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	If C(A) = 0, then ON; otherwise OFF
Negative	If $C(A)_0$ = 1, then ON; otherwise OFF

ERQ — EXCLUSIVE OR to Q — 676_8

$C(Q)_i \not\equiv C(Y)_i \Rightarrow C(Q)_i$ for i = 0, 1, . . . , 35

SUMMARY: The logical EXCLUSIVE OR of each bit of the contents of the Q Register and the corresponding bit of the contents of Y replaces the contents of the Q Register.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	If C(Q) = 0, then ON; otherwise OFF
Negative	If $C(Q)_0$ = 1, then ON; otherwise OFF

ERXn — EXCLUSIVE OR to Xn — $66n_8$

$C(Xn)_i \not\equiv C(Y)_i \Rightarrow C(Xn)_i$ for i = 0, 1, . . . , 17

SUMMARY: The logical EXCLUSIVE OR of each bit of the contents of the Index Register specified by n and the contents of Y replaces the contents of that Index Register.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	If C(Xn) = 0, then ON; otherwise OFF
Negative	If $C(Xn)_0$ = 1, then ON; otherwise OFF

ERSA — EXCLUSIVE OR to Storage A — 655_8

$C(A)_i \not\equiv C(Y)_i \Rightarrow C(Y)_i$ for i = 0, 1, . . . , 35

SUMMARY: The logical EXCLUSIVE OR of each bit of the contents of the A Register and the corresponding bit of the contents of Y replaces the contents of Y.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(Y) = 0, then ON; otherwise OFF
Negative	If $C(Y)_0$ = 1, then ON; otherwise OFF

ERSQ — EXCLUSIVE OR to Storage Q — 656_8

$C(Q)_i \not\equiv C(Y)_i \Rightarrow C(Y)_i$ for i = 0, 1, . . . , 35

SUMMARY: The logical EXCLUSIVE OR of each bit of the contents of the Q Register and the corresponding bit of the contents of Y replaces the contents of Y.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(Y) = 0, then ON; otherwise OFF
Negative	If $C(Y)_0$ = 1, then ON; otherwise OFF

ERSXn — EXCLUSIVE OR to Storage Xn — (n = 0, 1, . . . , 7) — $64n_8$

$C(Xn)_i \not\equiv C(Y)_i \Rightarrow C(Y)_i$ for i = 0, 1, . . . , 17

SUMMARY: The logical EXCLUSIVE OR of each bit of the contents of the Index Register specified by n and the corresponding bit of the contents of Y replaces the contents of Y.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If $C(Y)_{0 \ldots 17}$ = 0, then ON; otherwise OFF
Negative	If $C(Y)_0$ = 1, then ON; otherwise OFF

CMPA — Compare with A — 115_8

Comparison C(A) :: C(Y)

SUMMARY: The contents of the A Register are compared with the contents of Y.

MODIFICATION: All

INDICATORS AFFECTED:

Zero	Negative	Carry	Algebraic Comparison Relation	Sign
0	0	0	C(A) > C(Y)	$C(A)_0 = 0,\ C(Y)_0 = 1$
0	0	1	C(A) > C(Y)	$C(A)_0 = C(Y)_0$
1	0	1	C(A) = C(Y)	
0	1	0	C(A) < C(Y)	
0	1	1	C(A) < C(Y)	$C(A)_0 = 1,\ C(Y)_0 = 0$

Zero	Carry	Logic Comparison Relation
0	0	C(A) < C(Y)
1	1	C(A) = C(Y)
0	1	C(A) > C(Y)

CMPQ — Compare with Q — 116_8

Comparison C(Q) :: C(Y)

SUMMARY: The contents of the Q Register are compared with the contents of Y.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	Negative	Carry	Algebraic Comparison Relation	Sign
0	0	0	C(Q) > C(Y)	$C(Q)_0 = 0,\ C(Y)_0 = 1$
0	0	1	C(Q) > C(Y)	$C(Q)_0 = C(Y)_0$
1	0	1	C(Q) = C(Y)	
0	1	0	C(Q) < C(Y)	
0	1	1	C(Q) < C(Y)	$C(Q)_0 = 1,\ C(Y)_0 = 0$

Zero	Carry	Logic Comparison Relation
0	0	C(Q) < C(Y)
1	1	C(Q) = C(Y)
0	1	C(Q) > C(Y)

CMPXn — Compare with Xn — $10n_8$

Comparison C(Xn) :: $C(Y)_{0...17}$

SUMMARY: The contents of the Index Register specified by n are compared with the contents of Y, bits 0 through 17.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	Negative	Carry	Algebraic Comparison Relation	Sign
0	0	0	$C(Xn) > C(Y)_{0...17}$	$C(Xn)_0 = 0$, $C(Y)_0 = 1$
0	0	1	$C(Xn) > C(Y)_{0...17}$	$C(Xn)_0 = C(Y)_0$
1	0	1	$C(Xn) = C(Y)_{0...17}$	
0	1	0	$C(Xn) < C(Y)_{0...17}$	
0	1	1	$C(Xn) < C(Y)_{0...17}$	$C(Xn)_0 = 1$, $C(Y)_0 = 0$

Zero	Carry	Logic Comparison Relation
0	0	$C(Xn) < C(Y)_{0...17}$
1	1	$C(Xn) = C(Y)_{0...17}$
0	1	$C(Xn) > C(Y)_{0...17}$

CWL — Compare with Limits — 111_8

Algebraic comparison of C(Y) with the closed interval $[\,C(A);\ C(Q)]$

SUMMARY: The contents of Y are compared with the contents of the A Register and the Q Register to determine if the value of the contents of Y falls between an upper and lower limit set into the A Register and the Q Register, respectively.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero — If C(Y) is contained in the closed interval $[\,C(A);\ C(Q)]$, i.e.,
either $C(A) \leqq C(Y) \leqq C(Q)$
or $C(A) \geqq C(Y) \geqq C(Q)$,
then ON; otherwise OFF

Negative	Carry	Relation between C(Q) and C(Y)	Signs of C(Q) and C(Y)
0	0	$C(Q) > C(Y)$	$C(Q)_0 = 0,\ C(Y)_0 = 1$
0	1	$C(Q) \geqq C(Y)$	$C(Q)_0 = C(Y)_0$
1	0	$C(Q) < C(Y)$	$C(Q)_0 = C(Y)_0$
1	1	$C(Q) < C(Y)$	$C(Q)_0 = 1,\ C(Y)_0 = 0$

CMG Compare Magnitude 405_8

Algebraic comparison $|C(A)| :: |C(Y)|$

SUMMARY: The absolute value of the contents of the A Register is compared with the absolute value of the contents of Y.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	Negative	Relation
0	0	$\|C(A)\| > \|C(Y)\|$
1	0	$\|C(A)\| = \|C(Y)\|$
0	1	$\|C(A)\| < \|C(Y)\|$

SZN — Set Zero and Negative Indicators from Memory — 234_8

Test the Number C(Y)

SUMMARY: The zero and negative indicators are set to reflect the contents of Y.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	Negative	Relation
0	0	Number C(Y) >0
1	0	Number C(Y) = 0
0	1	Number C(Y) <0

CMK — Compare Masked — 211_8

$$Z_i = \overline{C(Q)}_i \text{ AND } [\, C(A)_i \not\equiv C(Y)_i \,] \text{ for all } i = 0,\ 1, \ldots, 35$$

SUMMARY: Selected bits of the A Register are compared with the corresponding bits of the contents of Y. The bits to be compared are the corresponding bits in the Q Register which are not masked by a 1.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero — If Z = 0, then ON; otherwise OFF

Negative — If $Z_0 = 1$, then ON; otherwise OFF

NOTE: This instruction compares those corresponding bit positions of A and Y for identity that are not masked by a 1 in the corresponding bit position of Q.

The Zero Indicator is set ON, if the comparison is successful for all bit positions; i.e. if for all i = 0, 1, ... 35 there is

either $C(A)_i \not\equiv C(Y)_i$ (identical) or $C(Q)_i = 1$ (masked)

Otherwise it is set OFF.

The Negative Indicator is set ON, if the comparison is unsuccessful for bit position 0, i.e. if

$C(A)_0 \not\equiv C(Y)_0$ (nonidentical) as well as $C(Q)_0 = 0$ (nonmasked)

Otherwise it is set OFF.

CANA — Comparative AND with A — 315_8

$Z_i = C(A)_i$ AND $C(Y)_i$ for all i = 0, 1, . . . , 35

SUMMARY: The logical AND of each bit of the contents of the A Register and the corresponding bit of the contents of Y is used to set appropriate indicators and the contents of Y and the A Register are not changed.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero — If Z = 0, then ON; otherwise OFF

Negative — If Z_0 = 1, then ON; otherwise OFF

CANQ — Comparative AND with Q — 316_8

$Z_i = C(Q)_i$ AND $C(Y)_i$ for all i = 0, 1, . . . , 35

SUMMARY: The logical AND of each bit of the contents of the Q Register and the corresponding bit of the contents of Y is used to set indicators and the contents of Y and the Q Register are not changed.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero — If Z = 0, then ON; otherwise OFF

Negative — If Z_0 = 1, then ON; otherwise OFF

CANXn — Comparative AND with Xn (n = 0, 1. . . , 7) — $30n_8$

$Z_i = C(Xn)_i$ AND $C(Y)_i$ for all i = 0, 1, . . . 17

SUMMARY: The logical AND of each bit of the contents of the Index Register specified by n and the corresponding bit of the contents of Y is used to set appropriate indicators and the contents of Y and the Index Register are not changed.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero — If Z = 0, then ON; otherwise OFF

Negative — If Z_0 = 1, then ON; otherwise OFF

CNAA — Comparative NOT AND with A — 215_8

$Z_i = C(A)_i$ AND $\overline{C(Y)_i}$ for all i = 0, 1, . . . , 35

SUMMARY: The logical AND of the contents of the A Register and the complement of the contents of Y is used to set appropriate indicators.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	If Z = 0, then ON; otherwise OFF
Negative	If Z_0 = 1, then ON; otherwise OFF

CNAQ — Comparative NOT AND with Q — 216_8

$Z_i = C(Q)_i$ AND $\overline{C(Y)_i}$ for all i = 0, 1, . . . , 35

SUMMARY: The logical AND of the contents of the Q Register and the complement of the contents of Y is used to set appropriate indicators.

MODIFICATIONS: All

INDICATORS AFFECTED:

Zero	If Z = 0, then ON; otherwise OFF
Negative	If Z_0 = 1, then ON; otherwise OFF

CNAXn — Comparative NOT AND with Xn — $20n_8$

$Z_i = C(Xn)_i$ AND $\overline{C(Y)_i}$ for all i = 0, 1, . . . 17

SUMMARY: The logical AND of the contents of the Index Register specified by n and the complement of the contents of Y is used to set appropriate indicators.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	If Z = 0, then ON; otherwise OFF
Negative	If Z_0 = 1, then ON; otherwise OFF

LDE — Load Exponent Register — 411_8

$C(Y)_{0...7} \Rightarrow C(E)$

SUMMARY: The contents of Y, bit positions 0 through 7, replace the contents of the Exponent Register.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	Set OFF
Negative	Set OFF

STE — Store Exponent Register — 456_8

$C(E) \Rightarrow C(Y)_{0...7}$; $00..0 \Rightarrow C(Y)_{8...17}$

SUMMARY: The contents of the Exponent Register replace the contents of Y, bits 0 through 7. Bits 8 through 17 of the contents of Y are filled with zeros.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED: None

ADE — Add to Exponent Register — 415_8

$C(E) + C(Y)_{0...7} \Rightarrow C(E)$

SUMMARY: The contents of Y, bits 0 through 7, are added to the contents of the Exponent Register and the result replaces the contents of the Exponent Register.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	Set OFF
Negative	Set OFF
Exp. Overflow	If exponent above +127, then ON
Exp. Underflow	If exponent below -128, then ON

FNO — Floating Normalize — 573_8

C(EAQ) normalized ⇒ C(EAQ)

SUMMARY: The contents of the combined Exponent, A, and Q Registers are normalized.

MODIFICATIONS: Are without any effect on the operation

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF
Exp. Overflow	If exponent above +127, then ON
Exp. Underflow	If exponent below -128, then ON
Overflow	Set OFF

NOTE: The instruction normalizes the number in EAQ. If the Overflow Indicator is ON, then the number in EAQ is normalized one place to the right; and then the sign bit $C(AQ)_0$ is inverted in order to reconstitute the actual sign. Furthermore, the Overflow Indicator is set OFF.

This instruction can be used to correct overflows that occurred with fixed-point numbers.

TRA — Transfer Unconditionally — 710_8

Y ⇒ C(IC)

SUMMARY: The address field of the instruction word replaces the contents of the Instruction Counter causing a transfer of control.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED: None

TSXn — Transfer and Set Xn — $70n_8$

C(IC) + 0...01 ⇒ C(Xn); Y ⇒ C(IC)

SUMMARY: The contents of the Instruction Counter, plus 1, replace the contents of the Index Register specified by n. The address field of the instruction word replaces the contents of the Instruction Counter causing a transfer of control.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED: None

TSS — Transfer and Set Slave — 715_8

Y ⇒ C(IC)

SUMMARY: The address field of the instruction word replaces the contents of the Instruction Counter and the processor enters the slave mode.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Master Mode	Set OFF

RET Return 630_8

$C(Y)_{0...17} \Rightarrow C(IC)$; $C(Y)_{18...35} \Rightarrow C(IR)$

SUMMARY: The contents of Y, bits 0 through 17, replace the contents of the Instruction Counter, and bits 18 through 35 replace the contents of the Indicator Register.

MODIFICATIONS: All except CI, SC, DU, DL

INDICATORS AFFECTED:

Master Mode	If corresponding bit in C(Y) is 1, then no change; otherwise OFF
All other indicators	If corresponding bit in C(Y) is 1, then ON; otherwise OFF

NOTES:

1. The relation between bit position of C(Y) and the indicators is as follows:

Bit Position	Indicator
18	Zero
19	Negative
20	Carry
21	Overflow
22	Exponent Overflow
23	Exponent Underflow
24	Overflow Mask
25	Tally Runout
26	Parity Error
27	Parity Mask
28	Master Mode
29	
30	
31	Not used
32	at this
33	time
34	
35	

2. A possible change of the status of the Master Mode Indicator takes place as the last part of the instruction execution.
3. The Tally Runout Indicator will reflect $C(Y)_{25}$ regardless of what address modification is performed on the RET instruction (for tally operations).

TZE — Transfer on Zero — 600_8

If Zero Indicator ON, then Y ⇒ C(IC)

SUMMARY: The address field of the instruction word replaces the contents of the Instruction Counter, if the Zero Indicator is ON.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED: None

TNZ — Transfer on Not Zero — 601_8

If Zero Indicator OFF, then Y ⇒ C(IC)

SUMMARY: The address field of the instruction word replaces the contents of the Instruction Counter IF the Zero Indicator is OFF.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED: None

TMI — Transfer on Minus — 604_8

If Negative Indicator ON, then Y ⇒ C(IC)

SUMMARY: The address field of the instruction word replaces the contents of the Instruction Counter if the Negative Indicator is ON.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED: None

TPL — Transfer on Plus — 605_8

If Negative Indicator OFF, then Y ⇒ C(IC)

SUMMARY: The address field of the instruction word replaces the contents of the Instruction Counter if the Negative Indicator is OFF.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED: None

TRC — Transfer on Carry — 603_8

If Carry Indicator ON, then Y ⇒ C(IC)

SUMMARY: The address field of the instruction word replaces the contents of the Instruction Counter if the Carry Indicator is ON.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED: None

TNC — Transfer on No Carry — 602_8

If Carry Indicator OFF, then Y ⇒ C(IC)

SUMMARY: The address field of the instruction word replaces the contents of the Instruction Counter if the Carry Indicator is OFF.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED: None

TOV — Transfer on Overflow — 617_8

If Overflow Indicator ON, then Y ⇒ C(IC)

SUMMARY: The address field of the instruction word replaces the contents of the Instruction Counter if the Overflow Indicator is ON.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Overflow Set OFF

TEO — Transfer on Exponent Overflow — 614_8

If Exponent Overflow Indicator ON, then Y ⇒ C(IC)

SUMMARY: The address field of the instruction word replaces the contents of the Instruction Counter if the Exponent Overflow Indicator is ON.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Exp. Overflow Set OFF

TEU — Transfer on Exponent Underflow — 615_8

If Exponent Underflow Indicator ON, then Y ⇒ C(IC)

SUMMARY: The address field of the instruction word replaces the contents of the Instruction Counter if the Exponent Underflow Indicator is ON.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Exp. Underflow Set OFF

TTF — Transfer on Tally Runout Indicator OFF — 607_8

If Tally Runout Indicator OFF, then Y ⇒ C(IC)

SUMMARY: The address field of the instruction word replaces the contents of the Instruction Counter if the Tally Runout Indicator is OFF.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED: None

NOP — No Operation — 011_8

SUMMARY: No operation takes place.

MODIFICATIONS: Generally the modification DU or DL is used (see the notes below)

INDICATORS AFFECTED: None

NOTES
1. If any modification other than DU or DL is used, the effective address will be used in a memory access request which could lead to memory faults.
2. The use of a modification ID, DI, IDC, DIC causes the respective changes in the address and the tally.

DIS — Delay Until Interrupt Signal — 616_8

SUMMARY: No operation takes place, and the processor does not continue with the next instruction, but waits for a program interrupt signal.

MODIFICATIONS: Are without any effect on the operation

INDICATORS AFFECTED: None

BCD Binary to Binary-Coded-Decimal 505_8

C(A) ÷ C(Y) ⇒ 4-bit quotient and remainder.
Shift C(Q) left 6 positions; 4-bit quotient ⇒ $C(Q)_{68...71}$
and remainder ⇒ C(A). Shift C(A) left 3 positions.

SUMMARY: The contents of the A Register are converted to their binary coded decimal equivalent and the result is formed in the Q Register. The conversion is made at the rate of one decimal digit per execution of the instruction; in the order of decreasing digit significance. A conversion constant stored in Y is used in the conversion and a new constant is used for each digit. With each instruction execution the contents of the Q Register are shifted left 4 positions and the converted 4 bit BCD digit is placed in the Q Register, bits 33-36.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero — If C(A) = 0, then ON

Negative — If before execution $C(A)_0 = 1$, then ON; otherwise OFF

NOTES:

1. This instruction carries out one step in an algorithm for the conversion of a number from the binary to the decimal system of notation, which requires the repeated short division of the binary number or last remainder by certain constants

$$C_i = 8^i \times 10^{N-i} \quad (\text{for } i = 1, 2, \ldots),$$

with N being defined by

$$10^{N-1} \leq |\text{number}| \leq 10^N - 1.$$

2. See example in Section 5.
3. See Appendix for Conversion constants.

GTB — Gray to Binary — 774_8

SUMMARY: C(A) converted from Gray Code to binary representation ⇒ C(A)
The contents of the A Register are converted from Gray Code to binary and replace the contents of the A Register.

MODIFICATIONS: Are without any effect on the operation

INDICATORS AFFECTED:

Zero	If C(A) = 0, then ON; otherwise OFF
Negative	If $C(A)_0$ = 1, then ON; otherwise OFF

NOTE: This conversion is defined by the following algorithm, when R_i and S_i denote the contents of bit positions i of the A-register before and after the conversion:

$$S_0 = R_0$$

$$S_i = (R_i \text{ AND } \overline{S_{i-1}}) \text{ OR } (\overline{R_i} \text{ AND } S_{i-1})$$
for $i = 1, 2, \ldots, 35$.

XEC — Execute — 716_8

SUMMARY: Obtain and execute the instruction stored at memory location Y.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED: None

The XEC instruction itself does not affect any indicator. However, the execution of the instruction from Y may affect indicators.

NOTE: After the execution of the instruction obtained from location Y, the next instruction to be executed is obtained from C(IC) + 1. This is the one stored in memory right after this XEC instruction, unless the contents of the Instruction Counter have been changed by the execution of the instruction obtained from memory location Y.

XED — Execute Double — 717_8

SUMMARY: Obtain and execute the two instructions stored at the memory location Y-pair.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED: None

The XED instruction itself does not affect any indicator. However, the execution of the two instructions from Y-pair may affect indicators.

NOTES:

1. The first instruction obtained from Y-pair MUST NOT alter the memory location from which the second instruction is obtained, and MUST NOT be another XED instruction.

2. If the first instruction obtained from Y-pair alters the contents of the Instruction Counter, then this transfer of control is effective immediately; and the second instruction of the pair is not executed.

3. After the execution of the two instructions obtained from Y-pair, the next instruction to be executed is obtained from C(IC) + 1. This is the instruction stored in memory right after this XED instruction unless the contents of the Instruction Counter have been changed by the execution of the two instructions obtained from the memory locations Y-pair.

MME — Master Mode Entry — 001_8

SUMMARY: Causes a fault which obtains and executes, in the Master Mode, the two instructions stored at the memory locations 4 + C and 5 + C (decimal).

MODIFICATIONS: Are without any effect on the operation.

INDICATORS AFFECTED: None

The MME instruction itself does not affect any indicator. However, the execution of the two instructions from 4 + C and 5 + C may affect indicators; particularly, each one in turn will affect the Master Mode Indicator as follows:

Master Mode
- If the instruction obtained actually results in a transfer of control and is not the TSS instruction, then ON
- If the instruction obtained is either the RET instruction with bit 28 = ZERO or the TSS instruction, then OFF

NOTES:

1. The value of the constant C is set up in the FAULT switches.
2. During the execution of this MME instruction and the two instructions obtained, the Processor is in the Master Mode, independent of the value of its Master Indicator. The Processor will stay in the Master Mode, if the Master Indicator is set ON after the execution of these three instructions.
3. The instruction from 4 + C MUST NOT alter the memory location 5 + C, and MUST NOT be an XED instruction.
4. If the instruction from 4 + C alters the contents of the Instruction Counter, then this transfer of control is effective immediately; and the instruction from 5 + C is not executed.
5. After the execution of the two instructions obtained from Y-pair, the next instruction to be executed is obtained from C(IC) + 1. This is the instruction stored in memory right after this MME instruction unless the contents of the Instruction Counter have been changed by the execution of the two instructions obtained from 4 + C and 5 + C.

DRL — Derail — 002_8

SUMMARY: Causes a fault which obtains and executes in the Master Mode the two instructions stored at the memory locations 12 + C and 13 + C (decimal).

MODIFICATIONS: Are without any effect on the operation

INDICATORS AFFECTED: None

The DRL instruction itself does not affect any indicator. However, the execution of the two instructions from 12 + C and 13 + C may affect indicators; particularly, each one in turn will affect the Master Mode Indicator as follows:

Master Mode
- If the instruction obtained actually results in a transfer of control and is not the TSS instruction, then ON
- If the instruction obtained is either the RET instruction with bit 28 = ZERO or the TSS instruction, then OFF

NOTES:

1. The value of the constant C is set up in the FAULT switches.
2. During the execution of this DRL instruction and the two instructions obtained, the processor is in the Master Mode, independent of the value of its Master Indicator. The processor will stay in the Master Mode, if the Master Indicator is ON after the execution of these three instructions.
3. The instruction from 12 + C MUST NOT alter the memory location 13 + C, and MUST NOT be an XED instruction.
4. If the instruction from 12 + C alters the contents of the Instruction Counter, then this transfer of control is effective immediately; and the instruction from 13 + C is not executed.
5. After the execution of the two instructions obtained from Y-pair, the next instruction to be executed is obtained from C(IC) + 1. This is the instruction stored in the memory right after this DRL instruction unless the contents of the Instruction Counter have been changed by the execution of the two instructions obtained from 12 + C and 13 + C.

RPT — Repeat — 520_8

SUMMARY: Execute the next instruction a specified number of times or until a specified terminate condition is met.

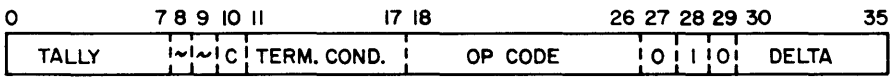
MODIFICATIONS: No modification can take place

INDICATORS AFFECTED:

Tally Runout	If termination because of Tally = 0, then ON If because terminate condition is met, then OFF
All other indicators	The RPT instruction itself does not affect any of the other indicators. However, the execution of the repeated instruction may affect indicators.

NOTES:

1. This RPT instruction has the following format:



2. If C = 1, then bits 0 - 17 of the RPT instruction $\Rightarrow$ X0.

3. In any case, the terminate condition and tally from X0 will control the repetition loop for the instruction following this RPT instruction; initial tally = 0 will be interpreted as 256.

4. The repetition loop consists of the following steps:
   a. Execute the repeated instruction,
   b. $C\ (X0)_{0\ldots7}-1 \Rightarrow C(X0)_{0\ldots7}$
   c. If Termination Condition met (see 7), then set Tally Runout Indicator OFF and terminate,
   d. If $C(X0)_{0\ldots7} = 0$, then set Tally Runout Indicator ON and terminate:
   e. Go to a.

5. All instructions can be used as repeated instructions except the following:

   All transfer of control instructions

   All miscellaneous instruction operations except BCD.

   All shift instructions, NEG, FNO

6. Address modification for the repeated instruction:

   The repeated instruction must be modified. Only the modifiers R and RI are permitted, and one of the designators X1, . . . , X7 must be specified.

The effective address Y (in the case of R) or the address Y of the indirect word to be referenced (in the case of RI) will be:

a. For the first execution of the repeated instruction
Y + C(R) ⇒ Y, Y + Delta ⇒ C(R)

b. For any successive execution
C(R) ⇒ Y, Y + Delta ⇒ C(R)

In the case of RI, only one indirect reference will be made per repeated execution. The Tag portion of the indirect word will not be interpreted as usual, but will be ignored; and instead the modifier R and the designator R = N will be applied.

7. The Terminate Conditions:

The possible terminate conditions are the same for all repeat instructions.

The bit configuration in bit positions 11 - 17 of the RPT instruction defines the terminate conditions for which the repetition loop will be terminated immediately. If more than one condition is specified, the repeat will terminate if any of the specified conditions are met.

Bit 17 = 1: any overflow terminates the repetition loop, and it is treated as usual; i.e., the respective Overflow Indicator is set ON, and if the Overflow Mask Indicator is OFF, then an Overflow Fault Trap occurs.

Bit 16 = 1: if Carry Indicator is OFF, terminate the repetition loop.

Bit 15 = 1: if Carry Indicator is ON, terminate the repetition loop.

Bit 14 = 1: if Negative Indicator is OFF, terminate the repetition loop.

Bit 13 = 1: if Negative Indicator is ON, terminate the repetition loop.

Bit 12 = 1: if Zero Indicator is OFF, terminate the repetition loop.

Bit 11 = 1: if Zero Indicator is ON, terminate the repetition loop.

A 0 in both positions for one indicator will cause this indicator to be ignored as a termination condition.

8. At the time of termination:

$X0_{0\ldots7}$ will contain the tally residue; i.e., the number of repeats remaining until a Tally Runout would have occurred, and also the terminate condition.

The Xn specified by the designator of the repeated instruction will contain the effective address of the next operand or indirect word that would have been secured (this is because of the overlap between an execution of the repeated instruction and the address modification for the next execution of the repeated instruction).

RPL — Repeat Link — 500_8

SUMMARY: Execute the next instruction a specified number of times, until a specified terminate condition is met, or until a Link Address Zero is found.

MODIFICATIONS: No modification can take place

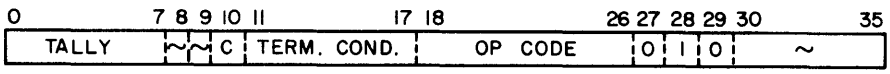
INDICATORS AFFECTED:

Tally Runout — If termination because of Tally = 0 or Link Address = 0, then ON. If because terminate condition is met, then OFF.

All other indicators — The RPL instruction itself does not affect any of the other indicators. However, the execution of the repeated instruction may affect indicators.

NOTES:

1. This RPL instruction has the following format:



2. If C = 1, then bits 0 - 17 of the RPL instruction $\Rightarrow$ X0.

3. In any case, the terminate condition and tally from X0 will control the repetition loop for the instruction following this RPL instruction; initial tally = 0 will be interpreted as 256.

4. The repetition loop consists of the following steps:

   a. Execute the repeated instruction

   b. $C(X0)_{0\ldots7}{}^{-1} \Rightarrow C(X0)_{0\ldots7}$

   c. If termination condition met (see 7), then set Tally Runout Indicator OFF and terminate

   d. If the tally $C(Xn)_{0\ldots7} = 0$ or the link address $C(Y)_{0\ldots17} = 0$, then set Tally Runout Indicator ON and terminate

   e. Go to a.

5. All instructions can be used as repeated instructions except the following:

   Instructions that could alter the link address $C(Y)_{0\ldots17}$

   EAA, EAQ, EAX, NEG, NEGL

   All miscellaneous operations instructions

   All shift instructions

   All transfer of control instructions.

6. Address modification for the repeated instruction:

   The repeated instruction must be modified. Only the modifier R is permitted, and one of the designators specifying X1...X7 must be used.

6. The effective address Y will be

For the first execution of the repeated instruction

$$Y + C(R) \Rightarrow Y, \quad Y \Rightarrow C(R)$$

For any successive execution of the repeated instruction

$$C(\, C(R)\,)_{0\ldots17} \Rightarrow Y, \quad Y \Rightarrow C(R)$$

The effective address Y is the address of the next list word. The lower half of this list word contains the operand to be used for this execution of the repeated instruction; the operand is

$$\underbrace{00\ldots0}_{18 \text{ times}}, \quad C(Y)_{18\ldots35} .$$

The upper half of the list word contains the Link Address, i.e., the address of the next successive list word, and thus the effective address for the next successive execution of the repeated instruction.

7. The Terminate Conditions:

The possible Terminate Conditions are the same for all repeat instructions.

The bit configuration in bit positions 11 - 17 of the RPL instruction defines the terminate conditions for which the repetition loop will be terminated immediately. If more than one condition is specified, the repeat will terminate if any of the specified conditions are met.

Bit 17 = 1 : any overflow terminates the repetition loop, and it is treated as usual; i.e., the respective Overflow Indicator is set ON, and if the Overflow Mask Indicator is OFF, an Overflow Fault Trap occurs.

Bit 16 = 1 : if Carry Indicator is OFF, terminate the repetition loop.

Bit 15 = 1 : if Carry Indicator is ON, terminate the repetition loop.

Bit 14 = 1 : if Negative Indicator is OFF, terminate the repetition loop.

Bit 13 = 1 : if Negative Indicator is ON, terminate the repetition loop.

Bit 12 = 1 : if Zero Indicator is OFF, terminate the repetition loop.

Bit 11 = 1 : if Zero Indicator is ON, terminate the repetition loop.

A 0 in both positions for one indicator will cause this indicator to be ignored as a termination condition.

8. At the time of termination:

   $X0_{0...7}$ will contain the tally residue, i.e., the numbers of repeats remaining until a tally runout would have occurred, and also the terminate condition.

   The Xn specified by the designator of this repeated instruction will contain the address of the list word that contains:

   In its lower half: the operand used in the last execution of the repeated instruction

   In its upper half: the address of the next list word.

   (This is because there is no overlap between an execution of the repeated instruction and the address modification for the next execution of the repeated instruction.)

RPD — Repeat Double — 560_8

SUMMARY: Execute the pair of instructions from the next location Y-pair a specified number of times or until a specified Terminate Condition is met.

MODIFICATIONS: No modification can take place

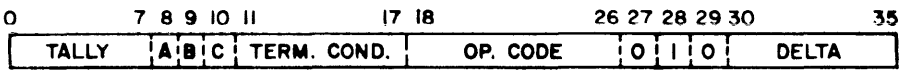
INDICATORS AFFECTED:

Tally Runout	If termination because of Tally = 0, then ON. If because Terminate Condition is met, then OFF.
All other indicators	The RPD instruction itself does not affect any of the other indicators. However, the execution of the repeated instructions may affect indicators.

NOTES:

1. The RPD instruction must be stored in an odd memory location except when accessed via the XEC instruction in which case the RPD instruction can be either even or odd.

2. This RPD instruction has the following format:



3. If C = 1, then bits 0 - 17 of the RPD instruction $\Rightarrow$ X0.

4. In any case, the Terminate Condition and Tally from X0 will control the repetition loop for the instruction following this RPD instruction; initial Tally = 0 will be interpreted as 256.

5. The repetition cycle consists of the following steps:

   a. Execute the pair of repeated instructions

   b. $C(X0)_{0...7}-1 = C(X0)_{0...7}$

   c. If Termination Condition met (see 8), then set Tally Runout Indicator OFF and terminate

   d. If $C(X0)_{0...7} = 0$, then set Tally Runout Indicator ON and terminate

   e. Go to a.

Note that if an Overflow Fault occurs on the even instruction, this precludes execution of the odd instruction.

6. All instructions can be used as repeated instructions except the following:
   a. Transfer of control instruction
   b. All miscellaneous operations instructions except BCD
   c. Macro operations
7. Address Modification for the pair of repeated instructions:

   Both of the two repeated instructions must be modified. Only the modifiers R and RI are permitted, and one of the designators X1,...,X7 for each of the two repeated instructions must be specified.

   The effective address Y (in the case of R) or the address Y of the indirect word to be referenced (in the case of RI) will be:

   a. For the first execution of each of the two repeated instructions

      $$Y + C(R) \Rightarrow Y, \quad Y + \text{Delta} \Rightarrow C(R)$$

   b. For any successive execution of

      The first of the two repeated instructions

      if A = 1, then $C(R) \Rightarrow Y$, $\text{Delta} + Y \Rightarrow C(R)$ or

      if A = 0, then $C(R) \Rightarrow Y$

      The second of the two repeated instructions

      if B = 1, then $C(R) \Rightarrow Y$, $\text{Delta} + Y \Rightarrow C(R)$ or

      if B = 0, then $C(R) \Rightarrow Y$

   (A and B are the contents of bit positions 8 and 9 of the RPD instruction)

   In the case of RI, only one indirect reference will be made per repeated execution. The Tag portion of the indirect word will not be interpreted as usual, but will be ignored; and instead the modifier R and the designator R = N will be applied.

8. The Terminate Conditions:

   The possible Terminate Conditions are the same for all repeat instructions.

   The bit configuration in bit positions 11 - 17 of the RPT instruction defines the Terminate Conditions for which the repetition loop will be terminated immediately. If more than one condition is specified, the repeat will terminate if any of the specified conditions are met.

   Bit 17 = 1 : any overflow terminates the repetition loop, and it is treated as usual; i.e., the respective Overflow Indicator is set ON, and if the Overflow Mask is OFF, then also an Overflow Fault Trap occurs. If the Overflow Fault Trap occurs on the even instruction, the odd instruction is not executed.

   Bit 16 = 1 : if Carry Indicator is OFF, terminate the repetition loop.

   Bit 15 = 1 : if Carry Indicator is ON, terminate the repetition loop.

Bit 14 = 1 : if Negative Indicator is OFF, terminate the repetition loop.

Bit 13 = 1 : if Negative Indicator is ON, terminate the repetition loop.

Bit 12 = 1 : if Zero Indicator is OFF, terminate the repetition loop.

Bit 11 = 1 : if Zero Indicator is ON, terminate the repetition loop.

9. At the time of termination:

$X0_{0...7}$ will contain the Tally Residue, i.e., the number of repeats remaining until a Tally Runout would have occurred, and also the Terminate Condition.

The Xn specified by the designator of each of the two repeated instructions will contain the effective address of the next operand or indirect word that would have been secured (special provisions have been made that this statement is true for both of the repeated instructions).

LBAR — Load Base Address Register — 230_8

$C(Y)_{0...17} \Rightarrow C(BR)$

SUMMARY: The contents of Y, bits 0 through 17 replace the contents of the Base Address Register.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	If C(BR) = 0, then ON; otherwise OFF
Negative	If $C(BR)_0$ = 1, then ON; otherwise OFF

NOTE: This instruction can be used in the Master Mode only. If its use is attempted in the Slave Mode, the instruction functions like the NOP instruction.

LDT — Load Timer Register — 637_8

$C(Y)_{0...23} \Rightarrow C(TR)$

SUMMARY: The contents of Y, bits 0 through 23, replace the contents of the Timer Register

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	If C(TR) = 0, then ON; otherwise OFF
Negative	If $C(TR)_0$ = 1, then ON; otherwise OFF

NOTE: This instruction can be used in the Master Mode only. If its use is attempted in the Slave Mode, the instruction functions like the NOP instruction.

SMIC Set Memory Controller Interrupt Cells 451_8

SUMMARY: C(A) is used to set selected Interrupt Cells ON in the system controller of the memory unit selected by Y_{0-2}

MODIFICATIONS: All except DU, DL, SC, and CI

INDICATORS AFFECTED: None

NOTES:

1. The effective address Y is used in selecting a memory module as with a normal memory access request. However, the selected module does not store the data received in a memory location, but uses it to set selected Interrupt Cells ON.

   For $i = 0, 1, \ldots, 15$ AND $C(A)_{35} = 0$:

   if $C(A)_i = 1$, then set Interrupt Cell i ON

   For $i = 0, 1, \ldots, 15$ AND $C(A)_{35} = 1$:

   if $C(A)_i = 1$, then set Interrupt Cell (16 + i) ON.

2. This instruction can be used in the Master Mode only. If the use of this instruction is attempted by a processor that is in the Slave Mode, a Command Fault Trap will occur.

RMCM Read Memory Controller Mask Register 233_8

SUMMARY: C (Memory Controller Interrupt Mask Register)
C (Memory Controller Access Mask Register)
of memory unit specified by Y_{0-2} } ⇒ C(AQ)

MODIFICATIONS: All except DU, DL, CI, SC

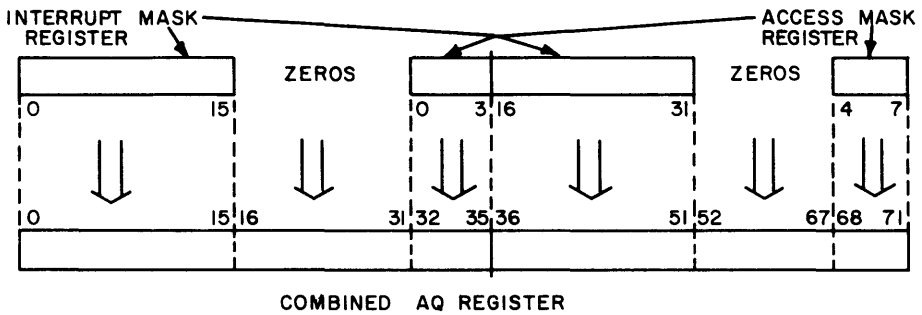
INDICATORS AFFECTED:

Zero — If C(AQ) = 0, then ON; otherwise OFF

Negative — If $C(AQ)_0$ = 1, then ON; otherwise OFF

NOTES:

1. The effective address Y is used in selecting a memory module as with a normal memory access request. However, the selected module does not transmit the contents of an addressed memory location, but the contents of its memory controller Interrupt Mask Register and memory controller Access Mask Register.



2. This instruction can be used in the Master Mode only. If the use of this instruction is attempted by a processor that is in the Slave Mode, a Command Fault Trap will occur.

RMFP — Read Memory File Protect Register — 633_8

SUMMARY: C (Memory File Protect Register) ⇒ C(AQ)
Of memory unit specified by Y_{0-2}

MODIFICATIONS: All except DU, DL, CI, SC

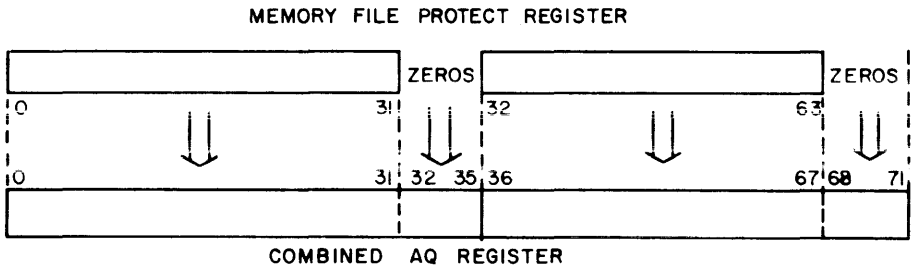
INDICATORS AFFECTED:

Zero — If C(AQ) = 0, then ON; otherwise OFF

Negative — If $C(AQ)_0$ = 1, then ON; otherwise OFF

NOTES:

1. The effective address Y is used in selecting a memory module as with a normal memory access request. However, the selected module does not transmit the contents of an addressed memory location, but the contents of its Memory File Protect Register.



2. This instruction can be used in the Master Mode only. If the use of this instruction is attempted by a processor that is in the Slave Mode, a Command Fault Trap will occur.

SMCM — Set Memory Controller Mask Register — 553_8

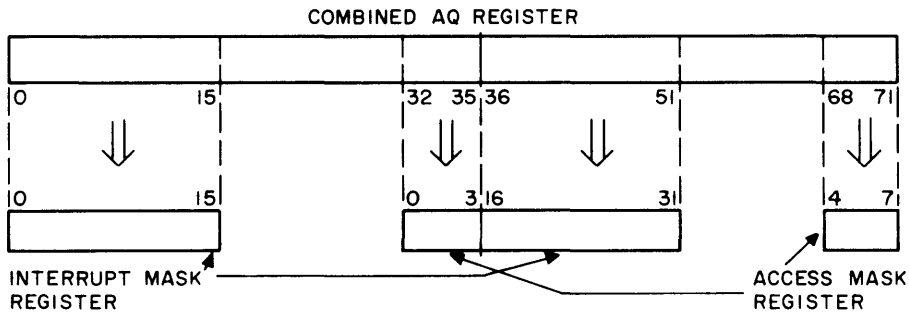
SUMMARY: C(AQ) ⇒ { C (Memory Controller Interrupt Mask Register); C (memory Controller Access Mask Register) } Of memory unit specified by Y_{0-2}

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED: None

NOTES:

1. The effective address Y is used in selecting a memory module as with a normal memory access request. However, the selected module does not store the data received in a memory location, but in its memory controller Interrupt Mask Register and memory controller Access Mask Register.



2. This instruction can be used in the Master Mode only. If the use of this instruction is attempted by a processor that is in the Slave Mode, a Command Fault Trap will occur.

SMFP — Set Memory File Protect Register — 453_8

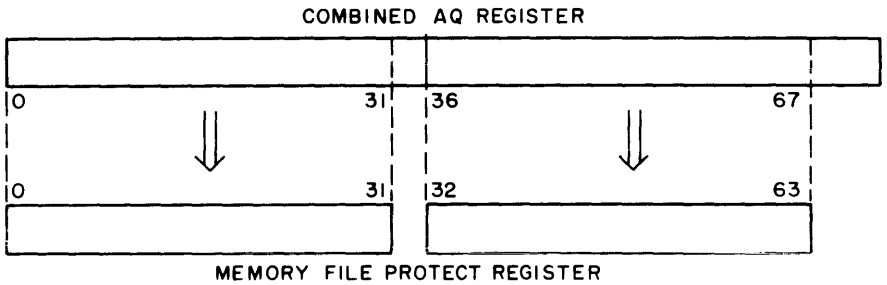
SUMMARY: C(AQ) → C (Memory File Protect Register) in the system controller of the memory unit specified by Y_{0-2}

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED: None

NOTES:

1. The effective address Y is used in selecting a memory module as with a normal memory access request. However, the selected module does not store the data received in a memory location, but in its memory File Protect Register.



2. This instruction can be used in the Master Mode only. If the use of this instruction is attempted by a processor that is in the Slave Mode, a Command Fault Trap will occur.

CIOC — Connect I/O Channel — 015_8

SUMMARY: C(Y) are transferred from the memory module via the channel that is specified by C(Y)

MODIFICATIONS: All except DU, DL, SC, and CI

INDICATORS AFFECTED: None

NOTES:

1. The effective address Y is used to access a memory location as usual. However, the memory module does not transmit the contents of this location to the processor that submitted the effective address; it uses $C(Y)_{33...35}$ to select one of its eight channels, sends a connect pulse to the unit on this channel, and then transmits C(Y) on the data lines to this unit.

2. This instruction can be used in the Master Mode only. If the use of this instruction is attempted by a processor that is in the Slave Mode, a Command Fault Trap will occur.

C. M-605 MACRO INSTRUCTIONS

The following instructions are handled by Macro operations or a Macro, subroutine combination if the optional floating point/double-precision hardware is not implemented.

DATA MOVEMENT - LOAD

LDAQ — Load AQ — 237_8

C(Y-pair) ⇒ C(AQ)

SUMMARY: The contents of the Y-pair of addresses replace the contents of the combined A and Q Registers with the contents of the even numbered location in the A Register.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF

LCAQ — Load Complement AQ — 337_8

- C(Y-pair) ⇒ C(AQ)

SUMMARY: The two's complement of contents of the Y-pair of addresses replaces the contents of the combined A and Q Registers. The contents of the even numbered location are in the A Register.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF
Overflow	If range of AQ is exceeded, then ON

STAQ — Store AQ — 757_8

C(AQ) ⇒ C(Y-pair)

SUMMARY: The contents of the combined A and Q Registers replace the contents of the Y-pair of addresses with the contents of A in the even numbered location.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED: None

ADAQ — Add to AQ — 077_8

C(AQ) + C(Y-pair) ⇒ C(AQ)

SUMMARY: The contents of the Y-pair of locations are added to the contents of the combined A and Q Registers and the result replaces the contents of the A and Q Registers. The contents of the even numbered location are added to the contents of the A Register.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF
Overflow	If range of AQ exceeded, then ON
Carry	If a carry out of AQ_0 is generated, then ON; otherwise OFF

ADLAQ — Add Logic to AQ — 037_8

C(AQ) + C(Y-pair) ⇒ C(AQ)

SUMMARY: The contents of the Y-pair of locations are added to the contents of the combined A and Q Registers and the result replaces the contents of the A and Q Registers. The contents of the even numbered location are added to the contents of the A Register.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF
Carry	If a carry out of AQ_0 is generated, then ON; otherwise OFF

NOTE: This instruction is identical to the ADAQ instruction, except the Overflow Indicator is not affected by this instruction.

SBAQ — Subtract from AQ — 177_8

C(AQ) - C(Y-pair) ⇒ C(AQ)

SUMMARY: The contents of the Y-pair of locations are subtracted from the contents of of the combined A and Q Registers and the result replaces the contents of the A and Q Registers. The contents of the even numbered location are subtracted from the contents of the A Register.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF
Overflow	If range of AQ exceeded, then ON
Carry	If carry out of AQ_0 is generated, then ON; otherwise OFF

SBLAQ — Subtract Logic from AQ — 137_8

C(AQ) - C(Y-pair) ⇒ C(AQ)

SUMMARY: The contents of the Y-pair of locations are subtracted from the contents of the combined A and Q Registers and the result replaces the contents of the A and Q Registers. The contents of the even numbered location are subtracted from the contents of the A Register.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF
Carry	If a carry out of AQ_0 is generated, then ON; otherwise OFF

NOTE: This instruction is identical to the SBAQ instruction, except the Overflow Indicator is not affected by this instruction.

ANAQ — AND to AQ — 377_8

$C(AQ)_i$ AND $C(Y\text{-pair})_i \Rightarrow C(AQ)_i$ for all $i = 0, 1, \ldots, 71$

SUMMARY: The logical AND of the contents of the combined A and Q Registers and the contents of the Y-pair of locations replaces the contents of the A and Q Registers. The contents of the even numbered location are ANDed with the contents of the A Register.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ)	= 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$	= 1, then ON; otherwise OFF

ORAQ — OR to AQ — 277_8

$C(AQ)_i$ OR $C(Y\text{-pair})_i \Rightarrow C(AQ)_i$ for all i = 0, 1, . . . , 71

SUMMARY: The logical OR of the contents of the combined A and Q Registers and the contents of the Y-pair of locations replaces the contents of the A and Q Registers. The contents of the even numbered location are ORed with the contents of the A Register.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ)	= 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$	= 1, then ON; otherwise OFF

ERAQ EXCLUSIVE OR to AQ 677_8

$C(AQ)_i \not\equiv C(Y\text{-pair})_i \Rightarrow C(AQ)_i$ for all i = 0, 1, . . . , 71

SUMMARY: The logical EXCLUSIVE OR of the contents of the combined A and Q Registers and the contents of the Y-pair of locations replaces the contents of the A and Q Registers. The contents of the even numbered locations are EXCLUSIVE ORed with the contents of the A Register.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF

CMPAQ — Compare with AQ — 117_8

Comparison C(AQ) :: C(Y-pair)

SUMMARY: The contents of the combined A and Q Registers are compared with the contents of the Y-pair of locations.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	Negative	Carry	Algebraic Comparison Relation	Sign
0	0	0	C(AQ) > C(Y-pair)	$C(AQ)_0 = 0, \quad C(Y\text{-pair})_0 = 1$
0	0	1	C(AQ) > C(Y-pair)	$C(AQ)_0 = C(Y\text{-pair})_0$
1	0	1	C(AQ) = C(Y-pair)	
0	1	0	C(AQ) < C(Y-pair)	
0	1	1	C(AQ) < C(Y-pair)	$C(AQ)_0 = 1, \quad C(Y\text{-pair})_0 = 0$

Zero	Carry	Logic Comparison Relation
0	0	C(AQ) < C(Y-pair)
1	1	C(AQ) = C(Y-pair)
0	1	C(AQ) > C(Y-pair)

CANAQ — Comparative AND with AQ — 317_8

$Z_i = C(AQ)_i$ AND $C(Y\text{-pair})_i$ for all i = 0, 1, . . . , 71

SUMMARY: The logical AND of the contents of the combined A and Q Registers and the contents of a Y-pair of locations are used to set appropriate indicators and the contents of the A and Q Register and the Y-pair are not changed.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If Z = 0, then ON; otherwise OFF
Negative	If Z_0 = 1, then ON; otherwise OFF

CNAAQ Comparative NOT AND with AQ 217_8

$Z_i = C(AQ)_i$ AND $\overline{C(Y\text{-pair})_i}$ for all i = 0, 1, . . . , 71

SUMMARY: The logical AND of the contents of the combined A and Q Registers and the complement of the contents of the Y-pair of locations are used to set appropriate indicators and the contents of the A and Q Register and the Y-pair are not changed.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If Z = 0, then ON; otherwise OFF
Negative	If Z_0 = 1, then ON; otherwise OFF

FLD — Floating Load — 431_8

C(Y) ⇒ C(EAQ)

SUMMARY: The contents of Y replace the contents of the Exponent, and A Registers. The Q Register is cleared.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF

DFLD — Double-Precision Floating Load — 433_8

C(Y-pair) ⇒ C(EAQ)

SUMMARY: The contents of a Y-pair replace the contents of the Exponent, A, and Q Registers.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF

FST — Floating Store — 455_8

C(EA) ⇒ C(Y)

SUMMARY: The contents of the Exponent Register replace the contents of Y, bits 0 through 7. The contents of the A Register, bits 0 through 27, replace the contents of Y, bits 8 through 35.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED: None

DFST — Double-Precision Floating Store — 457_8

C(EAQ) ⇒ C(Y-pair)

SUMMARY: The contents of the Exponent Register replace the contents of Y-pair, bits 0 through 7, and the contents of the combined A and Q Registers, bits 0 through 63, replace the contents of Y-pair, bits 8 through 71.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED: None

FAD — Floating Add — 475_8

C(EAQ) + C(Y) normalized ⇒ C(EAQ)

SUMMARY: The contents of Y are added to the contents of the Exponent, A, and Q Registers. The result is normalized and replaces the contents of the Exponent, A, and Q Registers.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF
Exp. Overflow	If Exponent above +127, then ON
Exp. Underflow	If Exponent below -128, then ON
Carry	If a carry out of AQ_0 is generated, then ON; otherwise OFF

UFA — Unnormalized Floating Add — 435_8

C(EAQ) + C(Y) not normalized ⇒ C(EAQ)

SUMMARY: The contents of Y are added to the contents of the Exponent, A, and Q Registers. The result replaces the contents of the Exponent, A, and Q Registers.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF
Exp. Overflow	If exponent above +127, then ON
Exp. Underflow	If exponent below -128, then ON
Carry	If a carry out of AQ_0 is generated, then ON; otherwise OFF

DFAD — Double-Precision Floating Add — 477_8

C(EAQ) + C(Y-pair) normalized ⇒ C(EAQ)

SUMMARY: The contents of a Y-pair are added to the contents of the Exponent, A, and Q Registers. The result is normalized and replaces the contents of the Exponent, A, and Q Registers.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF
Exp. Overflow	If exponent above +127, then ON
Exp. Underflow	If exponent below -128, then ON
Carry	If a carry out of AQ_0 is generated, then ON; otherwise OFF

DUFA — Double-Precision Unnormalized Floating Add — 437_8

C(EAQ) + C(Y-pair) not normalized ⇒ C(EAQ)

SUMMARY: The contents of a Y-pair are added to the contents of the Exponent, A, and Q Registers. The result replaces the contents of the Exponent, A, and Q Registers.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF
Exp. Overflow	If exponent above +127, then ON
Exp. Underflow	If exponent below -128, then ON
Carry	If a carry out of AQ_0 is generated, then ON; otherwise OFF

FSB — Floating Subtract — 575_8

C(EAQ) - C(Y) normalized ⇒ C(EAQ)

SUMMARY: The contents of Y are subtracted from the contents of the Exponent, A, and Q Registers. The result is normalized and replaces the contents of the Exponent, A, and Q Registers

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF
Exp. Overflow	If exponent above +127, then ON
Exp. Underflow	If exponent below -128, then ON
Carry	If a carry out of AQ_0 is generated, then ON; otherwise OFF

UFS — Unnormalized Floating Subtract — 535_8

C(EAQ) - C(Y) not normalized ⇒ C(EAQ)

SUMMARY: The contents of Y are subtracted from the contents of the Exponent, A, and Q Registers and the result replaces the contents of the Exponent, A, and Q Registers.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF
Exp. Overflow	If exponent above +127, then ON
Exp. Underflow	If exponent below -128, then ON
Carry	If a carry out of AQ_0 is generated, then ON; otherwise OFF

DFSB — Double-Precision Floating Subtract — 577_8

C(EAQ) - C(Y-pair) normalized ⇒ C(EAQ)

SUMMARY: The contents of a Y-pair are subtracted from the contents of the Exponent, A, and Q Register. The result is normalized and replaces the content of the Exponent, A, and Q Registers.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF
Exp. Overflow	If exponent above +127, then ON
Exp. Underflow	If exponent below -128, then ON
Carry	If a carry out of AQ_0 is generated, then ON; otherwise OFF

DUFS — Double-Precision unnormalized Floating Subtract — 537_8

C(EAQ) - C(Y-pair) not normalized ⇒ C(EAQ)

SUMMARY: The contents of a Y-pair are subtracted from the contents of the Exponent, A, and Q Register and the results replaces the contents of the Exponent, A and Q Registers.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF
Exp. Overflow	If exponent above +127, then ON
Exp. Underflow	If exponent below -128, then ON
Carry	If a carry out of AQ_0 is generated, then ON; otherwise OFF

FMP — Floating Multiply — 461_8

C(EAQ) × C(Y) normalized ⇒ C(EAQ)

SUMMARY: The contents of Y, bits 0 through 7, are added to the contents of the Exponent Register. The contents of the A and Q Register are multiplied by the contents of Y, bits 8 through 35. The result is normalized and replaces the contents of the Exponent, A, and Q Register.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$=1, then ON; otherwise OFF
Exp. Overflow	If exponent above +127, then ON
Exp. Underflow	If exponent below -128, then ON

UFM — Unnormalized Floating Multiply — 421_8

C(EAQ) × C(Y) not normalized ⇒ C(EAQ)

SUMMARY: The contents of Y, bits 0 through 7, are added to the contents of the Exponent Register. The contents of the A and Q Register are multiplied by the contents of Y, bits 8 through 35. The result replaces the contents of the Exponent, A, and Q Register.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF
Exp. Overflow	If exponent above +127, then ON
Exp. Underflow	If exponent below -128, then ON

NOTE: This multiplication is executed like the instruction FMP except the final normalization is performed only in the case of both factor mantissas being = - 1.00···0.

DFMP — Double-Precision Floating Multiply — 463_8

C(EAQ) × C(Y-pair) normalized ⇒ C(EAQ)

SUMMARY: The contents of a Y-pair, bits 0 through 7 are added to the contents of the Exponent Register. The contents of the A and Q Registers are multiplied by the contents of the Y-pair, bits 8 through 71. The result is normalized and replaces the contents of the Exponent, A, and Q Register.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF
Exp. Overflow	If exponent above +127, then ON
Exp. Underflow	If exponent below -128, then ON

DUFM — Double-Precision Unnormalized Floating Multiply — 423_8

C(EAQ) × C(Y-pair) not normalized ⇒ C(EAQ)

SUMMARY: The contents of the Y-pair, bits 0 through 7, are added to the contents of the Exponent Register. The contents of the A and Q Register are multiplied by the contents of the Y-pair, bits 8 through 71. The result replaces the contents of the Exponent, A, and Q Register.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF
Exp. Overflow	If exponent above +127, then ON
Exp. Underflow	If exponent below -128, then ON

NOTE: This multiplication is executed like the instruction DFMP, except the final normalization is performed only when both factor mantissas are = -1.00•••0 .

FDV Floating Divide 565_8

$C(EAQ) \div C(Y) \Rightarrow C(EA)$; $00\ldots0 \Rightarrow C(Q)$

SUMMARY: The contents of the A and Q Register are shifted right, and the contents of the Exponent Register are increased until the contents of the A and Q Register are less than the contents of Y, bits 8 through 35. The contents of Y, bits 0 through 7, are then subtracted from the contents of the Exponent Register. The contents of the A and Q Register are divided by the contents of Y, bits 8 through 35. The result replaces the contents of the Exponent and A Register. The Q Register is filled with zeros.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

	If division takes place:	If no division takes place:
Zero	If C(A) = 0, then ON; otherwise OFF	If divisor mantissa = 0, then ON; otherwise OFF
Negative	If $C(A)_0$ = 1, then ON; otherwise OFF	If dividend < 0, then ON; otherwise OFF
Exp. Overflow	If exponent above +127, then ON	
Exp. Underflow	If exponent below -128 then ON	

NOTES:

1. This division is executed as follows:

   The dividend mantissa C(AQ) is shifted right and the dividend exponent C(E) increased accordingly until

   $\left| C(AQ)_{0\ldots27} \right| < \left| C(Y)_{8\ldots35} \right|$;

   $C(E) - C(Y)_{0\ldots7} \Rightarrow C(E)$;

   $C(AQ) \div C(Y)_{8\ldots35} \Rightarrow C(A)$;

   $00\ldots0 \Rightarrow C(Q)$.

2. If mantissa of divisor = 0, then the division itself does not take place. Instead, a Divide-Check Fault Trap occurs; and all the registers remain unchanged.

FDI Floating Divide Inverted 525_8

$C(Y) \div C(EAQ) \Rightarrow C(EA)$; $00\ldots0 \Rightarrow C(Q)$

SUMMARY: The contents of Y, bits 8 through 35, are shifted right and the contents of Y, bits 0 through 7, are increased accordingly until the contents of Y, bits 8 through 35, are smaller than the contents of the A and Q Register, bits 0 through 27. The contents of the Exponent Register are then subtracted from the contents of Y, bits 0 through 7. The contents of Y, bits 8 through 35, are divided by the contents of the A and Q Register. The result replaces the contents of the Exponent and A Registers. The Q Register is filled with zeros.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

	If division takes place:	If no division takes place:
Zero	If C(A) = 0, then ON; otherwise OFF	If divisor mantissa = 0, then ON; otherwise OFF
Negative	If $C(A)_0$ = 1, then ON; otherwise OFF	If dividend < 0, then ON; otherwise OFF
Exp. Overflow	If exponent above +127, then ON	
Exp. Underflow	If exponent below -128, then ON	

NOTES:

1. This division is executed as follows:

   The dividend mantissa $C(Y)_{8\ldots35}$ is shifted right and the dividend exponent $C(Y)_{0\ldots7}$ increased accordingly until $\left|C(Y)_{8\ldots35}\right| < \left|C(AQ)_{0\ldots27}\right|$;

   $C(Y)_{0\ldots7}$ - C(E) $\Rightarrow$ C(E) ;

   $C(Y)_{8\ldots35} \div C(AQ) \Rightarrow C(A)$;

   $00\ldots0 \Rightarrow C(Q)$.

2. If mantissa of divisor = 0, then the division itself does not take place. Instead, a Divide-Check Fault Trap occurs; and all the registers remain unchanged.

DFDV Double-Precision Floating Divide 567_8

$C(EAQ) \div C(Y\text{-pair}) \Rightarrow C(EAQ)$

SUMMARY: The contents of the A and Q Registers are shifted right and the contents of the Exponent Register are increased accordingly until the contents of the A and Q Registers, bits 0 through 63, are smaller than the contents of the Y-pair, bits 8 through 71. The contents of the Y-pair, bits 0 through 7, are then subtracted from the contents of the Exponent Register. The contents of the A and Q Registers are divided by the contents of the Y-pair, bits 8 through 71. The result replaces the contents of the Exponent Register and bits 0 through 63 of the A and Q Registers. The Q Register, bits 64 through 71, is filled with zeros.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

	If division takes place:	If no division takes place:
Zero	If C(AQ) = 0, then ON; otherwise OFF	If divisor mantissa = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF	If dividend < 0, then ON; otherwise OFF
Exp. Overflow	If exponent above +127, then ON	
Exp. Underflow	If exponent below -128, then ON	

NOTES:

1. This division is executed as follows:

   The dividend mantissa C(AQ) is shifted right and the dividend exponent C(E) increased accordingly until $| C(AQ)_{0\ldots63} | < | C(Y\text{-pair})_{8\ldots71} |$;

   $C(E) - C(Y\text{-pair})_{0\ldots7} \Rightarrow C(E)$;

   $C(AQ) \div C(Y\text{-pair})_{8\ldots71} \Rightarrow C(AQ)_{0\ldots63}$;

   $00\ldots0 \Rightarrow C(AQ)_{64\ldots71}$.

2. If mantissa of divisor = 0, then the division itself does not take place. Instead, a Divide-Check Fault Trap occurs; and all the registers remain unchanged.

DFDI Double-Precision Floating Divide Inverted 527_8

C(Y-pair) ÷ C(EAQ) ⇒ C(EAQ)

SUMMARY: The contents of the Y-pair, bits 8 through 71, are shifted right and the contents of the Y-pair, bits 0 through 7 are increased accordingly until the the contents of the Y-pair, bits 8 through 71, are smaller than the contents of the A and Q Registers, bits 0 through 63. The contents of the Exponent Register are then subtracted from the contents of the Y-pair, bits 0 through 7. The contents of the Y-pair, bits 8 through 71, are divided by the contents of the A and Q Register. The result replaces the contents of the Exponent, A and Q Registers, bits 0 through 63. Q Register bits 64 through 71 are filled with zeros.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

	If division takes place:	If no division takes place:
Zero	If C(AQ) = 0, then ON; otherwise OFF	If divisor mantissa = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF	If dividend < 0, then ON; otherwise OFF
Exp. Overflow	If exponent above +127, then ON	
Exp. Underflow	If exponent below -128, then ON	

NOTES:

1. The dividend mantissa $C(Y\text{-pair})_{8\ldots71}$ is shifted right and the dividend exponent $C(Y\text{-pair})_{0\ldots7}$ increased accordingly until $|C(Y\text{-pair})_{8\ldots71}| < |C(AQ)_{0\ldots63}|$

   $C(Y\text{-pair})_{0\ldots7}$ - C(E) ⇒ C(E) ;

   $C(Y\text{-pair})_{8\ldots71}$ ÷ C(AQ) ⇒ $C(AQ)_{0\ldots63}$;

   00...0 ⇒ $C(AQ)_{64\ldots71}$.

2. If mantissa of divisor = 0, then the division itself does not take place. Instead, a Divide-Check Fault Trap occurs; and all the registers remain unchanged.

FNEG — Floating Negate — 513_8

- C(AQ) normalized ⇒ C(AQ)

SUMMARY: The two's complement of contents of the A and Q Registers are normalized. The result replaces the contents of the A and Q Registers.

MODIFICATIONS: Are without any effect on the operation

INDICATORS AFFECTED:

Zero	If C(AQ) = 0, then ON; otherwise OFF
Negative	If $C(AQ)_0$ = 1, then ON; otherwise OFF
Exp. Overflow	If exponent above +127, then ON
Exp. Underflow	If exponent below -128, then ON

NOTES:
1. Even if originally C(EAQ) were normalized, an exponent overflow can still occur, when originally C(AQ) = -1.00...0 and C(E) = +127.

FCMP Floating Compare 515_8

Algebraic comparison $C[(E)(AQ_{0\ldots27})] :: C(Y)$

SUMMARY: The contents of the Exponent Register are compared with the contents of Y, bits 0 through 7. The mantissa of the number with the lower exponent is shifted right as many places as the difference of the exponents. The contents of the A Register are then compared with the contents of Y, bits 8 through 35 and the appropriate indicators are set.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	Negative	Relation
0	0	$C[(E)(AQ_{0\ldots27})] > C(Y)$
1	0	$C[(E)(AQ_{0\ldots27})] = C(Y)$
0	1	$C[(E)(AQ_{0\ldots27})] < C(Y)$

FCMG — Floating Compare Magnitude — 425_8

Algebraic comparison $\left| C[(E)(AQ_{0\ldots27})] \right| :: \left| C(Y) \right|$

SUMMARY: The contents of the Exponent Register are compared with the contents of Y, bits 0 through 7. The mantissa of the number with the lower exponent is shifted right as many places as the difference of the exponents. The absolute value of the contents of the A Register is then compared with the absolute value of the contents of Y, bits 8 through 35 and the appropriate indicators are set.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	Negative	Relation
0	0	$\left\| C[(E)(AQ_{0\ldots27})] \right\| > \left\| C(Y) \right\|$
1	0	$\left\| C[(E)(AQ_{0\ldots27})] \right\| = \left\| C(Y) \right\|$
0	1	$\left\| C[(E)(AQ_{0\ldots27})] \right\| < \left\| C(Y) \right\|$

DFCMP — Double-Precision Floating Compare — 517_8

Algebraic comparison $C[(E)AQ_{0\ldots63})] :: C(Y\text{-pair})$

SUMMARY: The contents of the Exponent Register are compared with the contents of a Y-pair, bits 0 through 7. The mantissa of the number with the lower exponent is shifted right as many places as the difference of the exponents. The contents of the A and Q Registers are then compared with the contents of the Y-pair, bits 8 through 71 and the appropriate indicator is set.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	Negative	Relation
0	0	$C[(E)(AQ_{0\ldots63})] > C(Y\text{-pair})$
1	0	$C[(E)(AQ_{0\ldots63})] = C(Y\text{-pair})$
0	1	$C[(E)(AQ_{0\ldots63})] < C(Y\text{-pair})$

DFCMG — Double-Precision Floating Compare Magnitude — 427_8

Algebraic comparison $\left|C\left[(E)\ (AQ_{0\ldots63})\right]\right| :: \left|C(\text{Y-pair})\right|$

SUMMARY: The contents of the Exponent Register are compared with the contents of a Y-pair, bits 0 through 7. The mantissa of the number with the lower exponent is shifted right as many places as the difference of the exponents. The absolute value of the contents of the Y-pair, bits 8 through 71, is compared with the absolute values of the contents of the A and Q Registers, bits 0 through 63, and the appropriate indicator is set.

MODIFICATIONS: All except DU, DL, CI, SC

INDICATORS AFFECTED:

Zero	Negative	Relation
0	0	$\left\|C\left[(E)\ (AQ_{0\ldots63})\right]\right\| > \left\|C(\text{Y-pair})\right\|$
1	0	$\left\|C\left[(E)\ (AQ_{0\ldots63})\right]\right\| = \left\|C(\text{Y-pair})\right\|$
0	1	$\left\|C\left[(E)\ (AQ_{0\ldots63})\right]\right\| < \left\|C(\text{Y-pair})\right\|$

FSZN — Floating Set Zero and Negative Indicators from Memory — 430_8

SUMMARY: The zero and negative indicators are set to reflect the contents of Y.

MODIFICATIONS: All except CI, SC

INDICATORS AFFECTED:

Zero	Negative	Relation
0	0	Mantissa $C(Y)_{8\ldots35} > 0$
1	0	Mantissa $C(Y)_{8\ldots35} = 0$
0	1	Mantissa $C(Y)_{8\ldots35} < 0$

IV. SYMBOLIC MACRO ASSEMBLER--GMAP

A. GENERAL DESCRIPTION

The M-605 macro assembly program is a program which will translate symbolic machine language convenient for programmer use into binary machine instructions. The symbolic language is sufficiently like machine language to permit the programmer to utilize all the facilities of the computer which would be available to him if he were to code directly in machine language.

An assembler resembles a compiler in that it produces machine language programs. It differs from a compiler in that the symbolic language used with an assembler is closely related to the language used by the computer, while the source language used with a compiler resembles the technical language in which problems are stated by human beings.

Compilers have several advantages over assemblers. The language used with the compiler is easier to learn and is oriented toward the problem to be solved. The user of a compiler usually does not need an intimate knowledge of the inner workings of the computer. Programming is faster. Finally, the time required to obtain a finished, working program is greatly reduced since there is less chance for the programmer to make mistakes. The assembler compensates for its disadvantages by offering those programmers, who need a great degree of flexibility in writing their programs, that flexibility which is not currently found in compilers.

The M-605 Macro Assembler is provided to give the professional programmers some of the conveniences of a compiler and the flexibility of an assembler. The ability to design desired MACROS in order to provide convenient shorthand notations plus the use of all M-605 machine instructions, as well as a complete set of pseudo-operations, provides the programmer with a very powerful and flexible tool. The output options enable him to obtain binary text in relocatable as well as absolute formats.

The classic format of a variable field symbolic assembly program is used throughout the M-605 Macro Assembler. Typically, a symbolic instruction consists of four major divisions; location field, operation field, variable field, and comments field.

The location field normally contains a name by which other instructions may refer to the instruction named. The operation field contains the name of the machine operation or pseudo-operation. The variable field normally contains the location of the operand. The comments field exists solely for the convenience of the programmer and plays no part in the assembly process. An identification field is provided to give a means of identifying the location of a card within a deck.

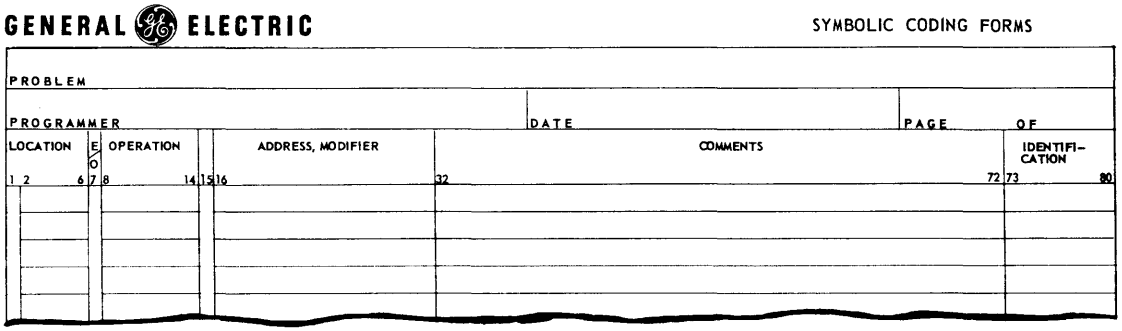
B. LANGUAGE CHARACTERISTICS

1. Language Format

Symbolic instructions are punched one per card, each card representing one line of the coding sheet (Figure IV-1). The following is a breakdown of the card columns normally used.

Columns	1 - 6	Location field
Column	7	Even/odd/eight subfield
Columns	8 - 13	Operation field
Columns	14 - 15	Blank
Columns	16 - Blank*	Variable field
Column	Blank - 72	Comments field (separated from variable field by at least one blank)
Columns	73 - 80	Identification field

* First blank column encountered within an expression will terminate the processing of the variable field.



Macro Assembler Coding Form

a. LOCATION FIELD

For machine instructions or MACROS this location may contain a symbol or may be left blank, if no reference is made to the instruction. (With certain pseudo-operations, this field has a special use and is described later in this publication.) Associated with the location field is a one-character field which allows the programmer to specify whether this generated machine word should fall in an even, odd or multiple of 8 memory location. If this is left blank, then the instruction will be located in the next available location. But, if there is an O in this field, the instruction will be located at the next available odd location; if an E, then at the next available even location; if an 8, then at the next available location which is a multiple of eight.

b. OPERATION FIELD

The operation field may contain from zero to six characters taken from the set 0-9, A-Z, and the period. The group of characters must be: (1) a legal M-605 operation, * (2) a Macro

Assembler pseudo-operation or a special MACRO call (CALL, SAVE, etc.) as described in this publication, (3) macro operation defined by programmer, (4) a GE-625/635 instruction which is not in the M-605 hardware implemented instruction repertoire, for which a macro will be substituted. The character group must begin in column eight (left-justified) and must be followed by at least one blank.

A blank field or the special code ARG will be interpreted as a zero operation. and the operation field will be all zeros in the assembly coding. Anything appearing in the operation field which is not in (1), (2), (3), or (4) above is in "illegal" operation and will result in an error flag in the assembly listing.

c. VARIABLE FIELD

The variable field contains one or more subfields that are separated by the programmer through the use of commas placed between subfields. The number and type of subfields vary depending upon the content of the operation field: (1) machine instruction, (2) Macro Assembler pseudo-operation. or (3) macro operation.

The subfields within the variable field of M-605 instructions consist of the address and the tag (modifier). The address may be any legitimate expression or a literal. This is the first subfield of the variable field and is separated from the tag by a comma. (See below for allowable tag mnemonics and their meanings.) Through address modification, as directed by the tag, a program address is defined. This program address is either (1) an instruction address used for fetching instructions, (2) a tentative address used for fetching an indirect word, or (3) an effective address used for obtaining an operand or storing a result.

The subfields used with pseudo-operations vary considerably; they are described individually in this publication under each pseudo-operation. Subfields used with macro operations are substitutable arguments which, in themselves, may be instructions, operand addresses, modifier tags, pseudo-operations, or other macro operations. All of these types of subfields are presented in the discussion on macro operations.

The first character of the variable field must begin by column 16. The end of the variable field is designated by the first blank character encountered in the variable field (except for the BCI instruction and in the use of Hollerith literals). If any subfield is null (no entry given when one is needed), it is interpreted to be zero.

---

* All indexing instructions (LDX. STX, ADX, etc.) may be used without the index register number appended. Thus,

```
LDX     1, 5, DU
```

is equivalent to

```
LDX1    5, DU
```

Also, the following is permissible:

```
LDX     B+A, Y, DU where B+A specifies the index register
```

d. COMMENTS FIELD

The comments field exists solely for the convenience of the programmer; it plays no part in the assembly process. Programmer comments follow the variable field and are separated from that field by at least one blank column.

e. IDENTIFICATION FIELD

This field is used or not used according to programmer option. Its intended use is for instruction identification and sequencing.

2. Symbols

A symbol is a string of from one to six non blank characters, at least one of which is non-numeric and the first of which is non-zero. The characters must be taken from the set made up of 0-9, A-Z and the period (.). Symbols can appear in the location and variable fields of the Assembler coding form. (Symbols are also known as location symbols and symbolic addresses.)

Symbols are defined by:

- Their appearance in the location field of an instruction, pseudo-operation, or MACRO.
- Their use as the name of a subprogram in a CALL pseudo-operation.
- Their appearance in the Symbol Reference (SYMREF) pseudo-operation.

Every symbol used in a program must be defined exactly once, except for those symbols which are initially defined and redefined by the SET pseudo-operation. An error will be indicated by the assembler if any svmbol is used but never defined, or if any symbol is defined more than once.

The following are examples of permissible symbols:

A	A1000	E1XP3	A.....
Z	FIRST	.XP3	B.707
B1	ALOG10	ADDTO	1234X
ERR	BEGIN	ERROR	3.141P

Symbols are classified into four types:

- Absolute -- A symbol which refers to a specific number.
- Common -- A symbol which refers to a location in common storage. These locations are defined by the use of the BLOCK pseudo-operation.
- Relocatable -- A symbol which appears in the location field of an instruction. Symbols that appear in the location field of symbol defining pseudo-operations are defined as the same type as the symbol in the variable field.
- SYMREF -- A symbol which appears in the variable field of a SYMREF pseudo-operation; it is considered to be defined external to the subprogram being assembled and is to be considered specially by the Loader.

3. Expressions

In writing symbolic instructions, the use of symbols only in the allowable subfields presents the programmer with too restrictive a language and, in effect, impairs efficient use of the hardware. Therefore, in the notation of subfields of machine instructions and in the variable fields of pseudo-operations in accordance with the rules set forth in each specific case, the capability to use expressions rather than just symbols is permitted. Before discussing expressions, it is necessary to describe the building blocks used to construct them. These building blocks are elements, terms, and operators.

a. ELEMENTS

The smallest component of a complete expression is an element. An element consists of a single symbol, an integer less than 2^{35}, or an asterisk.

An asterisk (*) may be used as an element in addition to being used as an operator. When it is used as an element, it refers to the location of the instruction in which it appears. For example, the instruction

```
A10     TRA     *+2
```

is equivalent to

```
A10     TRA     A10+2
```

and represents a transfer to the second location following the transfer instruction. There is no ambiguity between this usage of the asterisk as an element and its use as the operator for multiplication since the position of the asterisk always makes clear what is meant. Thus, **M means "the location of this instruction multiplied by the element M", and the ** means "the location of this instruction times the null element" and would be equal to zero. The notation *-* means "the location of this instruction minus the location of this instruction." (See description of the operators below.)

b. TERMS

A term is a string composed of elements and operators. It may consist of one element or, generally speaking, n elements separated by n - 1 operators of the type * and / where * indicates multiplication and / indicates division. If a term does not begin with an element or end with an element, then a null element will be assumed. It is not permissible to write two operators in succession or to write two elements in succession.

Examples of terms are:

M	MAN*T	7*Y
436/2	BETA/3	A*B*C/X*Y*Z
START	4*AB/ROOT	ONE*TWO/THREE

c. ALGEBRAIC EXPRESSIONS

An algebraic expression is a string composed of terms separated by the operators + (addition) - (subtraction). Therefore, an expression may consist of one term or, more generally speaking, n terms separated by n - 1 operators of the type + and -. It is permissible to write

two operators, plus and minus, in succession and the Assembler will assume a null element between the two operators. If no initial term or final term is stated, it will be assumed to be zero. An expression may begin with the operator plus or minus. Examples of permissible algebraic expressions are:

A	B+4		CX*DY+EX/FY-100
SINE	7		-EXP*FUNC/LOGX+XYZ/10-SINE
XYZ	+99	-X/Y	*+5*X (Note: the first asterisk refers to the instruction location)
A-3	-88+2	X*Y	-- (Note: equivalent to zero minus zero minus zero)

An algebraic expression is evaluated as follows: first, each symbolic element is replaced by its numerically-defined value; then, each term is computed from left-to-right in the order of its occurrence. In division, the integral part of the quotient is retained; the remainder is immediately discarded. For example, the value of the term 7/3 * 3 is 6. In the evaluation of an expression, division by zero is equivalent to division by one and is not regarded as an error. After the evaluation of terms, they are combined in a left-to-right order with the initial term of the expression assumed to be zero followed by the plus operator. If there is no final term, a null term will be used. At the completion of the expression evaluation, the Assembler reduces the result by modulo 2^n where n is the number of bits in the field being defined, 18 for address field evaluations and variable according to specified field size for the VFD pseudo-operation. Grouping by parentheses is not permitted, but this restriction may often be circumvented.

d. BOOLEAN EXPRESSIONS

A Boolean expression is defined similarly to an algebraic expression except that the operators *, /, +, or - are interpreted as Boolean operators. The meaning of these operators is defined below:

1. The expression that appears in the variable field of a BOOL pseudo-operation uses Boolean operators.
2. The expression that appears in the octal subfield of the variable field of a VFD pseudo-operation uses Boolean operators.

A Boolean expression is evaluated following the same procedure used for an algebraic expression except that the operators are interpreted as Boolean.

In a Boolean expression, the form operators +, -, *, and / have Boolean meanings, rather than their normal arithmetic meanings, as follows.

Operator	Meaning	Definition
+	OR, INCLUSIVE OR, union	0 + 0 = 0 0 + 1 = 1 1 + 0 = 1 1 + 1 = 1
-	EXCLUSIVE OR symmetric difference	0 - 0 = 0 0 - 1 = 1 1 - 0 = 1 1 - 1 = 0
*	AND, intersection	0 * 0 = 0 0 * 1 = 0 1 * 0 = 0 1 * 1 = 1
/	1's complement, complement, NOT	/0 = 1 /1 = 0

Although / is a unary operation involving only one term, by convention A/B is taken to mean A*/B; and the A is ignored. This is not regarded as an error by the Assembler. Thus, the table for / as a two-term operation is:

0/0 = 0	1/0 = 1
0/1 = 0	1/1 = 0

Other conventions are:

```
 A   A    A
-A = A- = A
*A = A* = 0
A/ = A/0 = A
```

For a discussion of relocatable and absolute expression evaluation see Section I.

4. Literals

A literal in a subfield is defined as being the data to be operated on rather than an expression which points to a location containing the data.

A programmer frequently must refer to a memory location containing a program constant. For example, if the constant 2 is to be added to the A Register, the number 2 must be somewhere in memory. Data generating pseudo-operations in the Macro Assembler enable the programmer to introduce data words and constants into his program; but often the introduction is more directly accomplished by the use of the literal that serves as the operand of a machine instruction. Thus, the literal is data itself.

The Assembler retains source program literals by means of a table called a literal pool. When a literal appears, the Assembler prepares a constant which is equivalent in value to the data in the literal subfield. This constant is then placed in the literal pool, providing an identical constant has not already been so entered. If the constant is placed in the literal pool,

it is assigned an address; and this address then replaces the data in the literal subfield, the constant being retained in the pool. If the constant is already in the literal pool, the address of the identical constant replaces the data in the literal subfield.

The Assembler processes five types of literals: decimal, octal, alphanumeric, instruction, and variable field. The appearance of an equal sign (=) in column 16 of the variable field instructs the Assembler that the subfield immediately following is a literal. The instruction and variable-field literal are placed in the literal pool; because they cannot be evaluated until pass two of the assembly, no attempt is made to check for duplicate entries into the pool. Literals on the CALL and TALLY pseudo-operations are restricted to decimal, octal, and alphanumeric where the character count is less than 13.

a. DECIMAL LITERALS

- Integers

  A decimal integer is a signed or unsigned string of digits. It is unique from the other decimal types by the absence of a decimal point, the letter B, the letter E, or the letter D.

- Single-Precision Floating-Point

  A floating-point subfield consists of two parts: the principle and the exponent.

  Principle part — is a signed or unsigned decimal number written with a decimal point. The decimal point is mandatory unless the exponent field is present. The decimal point may appear anywhere within the principle part. If absent, it is assumed to be at the right-hand end.

  Exponent part — if present, follows the principle part and consists of the letter E, followed by a signed or unsigned decimal integer. The floating-point number is distinguished by the presence of an E, or a decimal point, or both.

- Double-Precision Floating-Point

  The format of the double-precision floating-point number is identical to the normal single-precision format with two exceptions:

  1. There must always be an exponent
  2. The letter E must be replaced by the letter D

The Assembler will ensure that all double-precision numbers begin in even memory locations. Ambiguity of storage assignment as to even or odd will always cause the Assembler to force double-precision word pairs to even locations; it will then issue a warning in the printout listing. This feature is maintained for GE 625/635 compatibility.

- Fixed-Point

  A fixed-point quantity possesses the same characteristics as the floating-point — with one exception: it must have a third part present. This is the binary scale factor denoted by the letter B, followed by a signed or unsigned integer. The binary point is initially assumed at the left-hand end of the word between bit position 0 and 1. It is then adjusted by the binary scale factor, designated with plus implying a shift to the right and with minus, a shift to the left. Double-precision fixed-point follows the rules of double-precision floating-point with addition of the binary scale factor.

Examples of decimal literals are:

=-10	Integer
=26.44167E-1	Single-precision floating-point
=1.27743675385D0	Double-precision floating-point
=22.5B5	Fixed-point

b. OCTAL LITERALS

The octal literal consists of the character O followed by a signed or unsigned octal integer. The octal integer may be from one to twelve digits in length plus the sign. The Assembler will store it in a word, right-justified. The word will be stored in its real form and will not be complemented if there is the presence of a minus sign. The sign applies to bit 0 only.

Examples of octal literals are:

=O1257
=O-377777777742

c. ALPHANUMERIC LITERALS

The alphanumeric, or Hollerith, literal consists of the letters H or kH, where k is a character count followed by the data. If there is no count specified, a literal of exactly six 6-bit characters including blanks is assumed to follow the letter H. If a count exists, the k characters following the character H are to be used as the literal. If the value k is not a multiple of six, the last partial word will be left-justified and filled in with blanks. The value k can range from 1 through 63. (Imbedded blanks do not terminate scanning of the cards by the Assembler.)

Examples of alphanumeric literals are:

=HALPHA1
=HGONE
=4HGONE␣␣ (␣ represents a blank)
=7HTHE␣END

d. INSTRUCTION LITERALS

The instruction literal consists of the character = followed by the letter, M. This is followed in turn by an operation code, one blank, and a variable field. (The imbedded blank does not terminate scanning of the card in this instance.)

Examples of instruction literals are:

=MARG␣BETA
=MLDA␣5

Instructions containing instruction literals cannot make use of any of the forms of tag modifier, since any modifier encountered is assumed to be a part of the instruction literal.

e. VARIABLE FIELD LITERALS

The variable field literal begins with the letter V. Reference should be made to the description of the VFD pseudo-operation for the detailed description of using variable field data description. The subfields of a variable field literal may be one of three types: Algebraic, Boolean, and Alphanumeric.

Examples of variable field literals are:

```
=V10/895,5/37,H6/C,15/ALPHA
=V18/ALPHA,O12/235,6/0
```

Instructions containing variable field literals cannot make use of any of the forms of a tag modifier. See page IV-48.

f. LITERALS MODIFIED BY DU OR DL

When a literal is used with the modifier variations DU or DL, the value of the literal is not stored in the literal pool but is truncated to an 18-bit value, and is stored in the address field of the machine instruction. Normally, a literal represents a 36-bit number. For the DU or DL modifier variations, if the literal is a floating-point number or Hollerith, then bit 0-17 of the literal will be stored in the address field. In the case of all other literals, bits 18-35 of the literal will be stored in the address field.

Examples of literals modified by DU and DL are:

CODED LITERAL	RESULTANT ADDRESS FIELD (OCTAL)
=100,DL	000144
=-1.0,DU	001000
=320.,DU	022500
=0.,DU	400000
=O77,DU	000077
=2B25,DU	004000
=3H00A,DL	000021

5. Processor Instructions

Processor instructions written for the Assembler consist of a symbol (or blanks) in the location field, a 3- to 6-character alphanumeric code representing an M-605 operation in the operation field, and an operand address, (symbolic or numeric), plus a possible modifier tag in the variable field.

Standard machine mnemonics are entered left-justified in the operation field. These are any instruction mnemonic, as presented in the listings in the Appendices.

Several Assembler pseudo-operations are closely related to machine instructions. These are:

- OPSYN (operation synonym) - redefinition of a machine instruction by equating a new mnemonic to one already existing in the Assembler operation table.

- OPD (operation definition) - definition of a new machine instruction to the Assembler.
- MACRO (macro instruction definition) - define a mnemonic operation code to cause one or more standard operations to be generated by the Assembler.

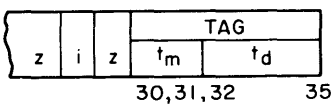
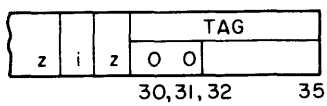
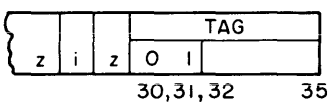
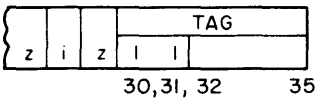
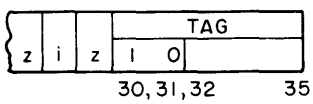
The operand address and modifier tag of most machine instructions comprise the subfield entries of the variable field. The address portion may be any legitimate expression, described earlier. The address is the first subfield in the variable field and begins in column 16. The modifier tag subfield is separated from the address subfield by a comma. Coding of the modifier tag subfield entries is described on the pages following.

6. Address Modification Features

a. Summary

The M-605 performs address modification in four basic ways: Register modification (R), Register Then Indirect modification (RI), Indirect Then Register modification (IR), and Indirect then Tally modification (IT). Each of these basic types has associated with it a number of variations in which selectable registers can be substituted for the R in R, RI, and IR and in which various tallying or other substitutions can be made for the T in IT. I always indicates indirect address modification and is represented by the asterisk * placed in the variable field of the Macro Assembler coding sheet as *R or R* when IR or RI is specified. To indicate IT modification, only the substitution for T appears in the coding sheet variable field; that is, the asterisk is not used.

In indirect addressing, the contents of the instruction address y are treated as another address, rather than as the operand of the instruction code. In the M-605, indirect address modification is handled automatically as a hardware function whenever called for by program instruction. This form of modification precedes direct address modification for IR and IT; for RI, it follows. When the I modification is called for by a program instruction, an indirect word is always obtained from memory. This indirect word may call for continued I modification, or it may specify the effective address Y to be used by the original instruction. Indirect addressing for RI, IR, and IT is performed by the processor whenever a binary 1 appears in either position of the t_m field (bit positions 30 and 31) of an instruction or an applicable indirect word. The four basic modifications types, their mnemonic substitutions as used in the variable field of the coding sheet. and the binary forms presented to the processor by the Assembler are as follows:

MODIFICATION TYPE	CODING SHEET MNEMONIC	BINARY FORMS
		
R	BETA,(R)	
RI	BETA,(R)*	
IR	BETA,*(R)	
IT	BETA,(T)	

The parentheses in (R) and (T) indicate that substitutions are made by the programmer for R and T; these are explained under the separate discussions of R, IR, RI, and IT modification. Binary equivalents of the substitution are used in the t_d subfield.

b. REGISTER (R) MODIFICATION

Simple R-type address modification is performed by the processor whenever the programmer codes an R-type variation (listed below) and causes the Assembler to place binary zeros in both positions of the modifier subfield t_m of the general instruction. Accordingly, one among 16 variations under R will be performed by the processor, depending upon bit configurations generated by the Assembler and placed in the designator subfield (t_d) of the general instruction. The 16 variations, their mnemonic substitutions used on the Assembler coding sheet, the t_d field binary forms presented to the processor, and the effective addresses Y generated by the processor are indicated in the following table.

A special kind of address modification variation is provided under R modification. The use of the instruction address field as the operand is referred to as direct operand address modification, of which there are two types; (1) Direct Upper and (2) Direct Lower, With the Direct Upper variation, the address field of the instruction serves as bit positions 0-17 of the operand and zeros serve as bit positions 18-35 of the operand. With the Direct Lower variation, the address field of the instruction serves as bit positions 18-35 of the operand and zeros serve as bit positions 0-17 of the operand.

MODIFICATION VARIATION	MNEMONIC SUBSTITUTION	BINARY FORM (t_d FIELD)	EFFECTIVE ADDRESS
(R) = X0	0	1000	$Y = y + C(X0)_{0-17}$
= X1	1	1001	$Y = y + C(X1)_{0-17}$
= X2	2	1010	$Y = y + C(X2)_{0-17}$
= X3	3	1011	$Y = y + C(X3)_{0-17}$
= X4	4	1100	$Y = y + C(X4)_{0-17}$
= X5	5	1101	$Y = y + C(X5)_{0-17}$
= X6	6	1110	$Y = y + C(X6)_{0-17}$
= X7	7	1111	$Y = y + C(X7)_{0-17}$
$= AR_{0-17}$	AU	0001	$Y = y + C(AR)_{0-17}$
$= AR_{18-35}$	AL	0101	$Y = y + C(AR)_{18-35}$
$= QR_{0-17}$	QU	0010	$Y = y + C(QR)_{0-17}$
$= QR_{18-35}$	QL	0110	$Y = y + C(QR)_{18-35}$
$= IC_{0-17}$	IC	0100	$Y = y + C(IC)_{0-17}$
$= IR_{0-17}$	DU	0011	$C(Y)_{0-17} = y$
$= IR_{0-17}$	DL	0111	$C(Y)_{18-35} = y$
= None	Blank or N	0000	Y = y
= Any symbolic index register	Any defined symbol*		

*Symbol must be defined as 0-7 by use of an applicable pseudo-operation. (See discussion of EQU and BOOL.)

The examples following show how R-type modification variations are entered in the variable field and their resultant control effects upon processor development of effective addresses.

			VARIABLE FIELD	COMMENTS	
	LOCATION	OPERATION	(ADDRESS, TAG)	MODIFICATION TYPE	EFFECTIVE ADDRESS
1.			B,0	(R)	Y = B + C(X0)
2.			C,AL	(R)	$Y = C + C(AR)_{18-35}$
3.			M,QU	(R)	$Y = M + C(QR)_{0-17}$
4.			-2,IC	(R)	Y = C(IC) - 2
5.			*,DU	(R)	$Operand_{0-17} = C(IC)$†
6.			1,7	(R)	Y = 1 + C(X7)
7.			2,DL	(R)	$Operand_{18-35} = 2$
8.			B	(R)	Y = B
9.			B,N	(R)	Y = B
10.			C,ALPHA	(R)	Y = C + C(X2)
	ALPHA	EQU	2		

†Note: When used in an indirect modification reference, $Operand_{0-17}$ = location of indirect word

c. REGISTER THEN INDIRECT (RI) MODIFICATION

Register Then Indirect address modification in the M-605 is a combination type in which both indexing (register modification) and indirect addressing are performed. For indexing modification under RI, the mnemonic substitutions for R are the same as those given under the discussion of Register (R) modification with the exception that DU or DL cannot be substituted for R. For indirect addressing (I), the processor treats the contents of the operand address associated with the original instruction or with an indirect word.

Under RI modification, the effective address Y is found by first performing the specified Register modification on the operand address of the instruction; the result of this R modification under RI obtains the address of an indirect word which is then retrieved.

After the indirect word has been accessed from memory and decoded, the processor carries out the address modification specified by this indirect word. If the indirect word specifies RI, IR, or IT modification (any type specifying indirection), the indirect sequence is continued. When an indirect word is found that specifies R modification, the processor performs R modification, using the register specified by the t_d field of this last encountered indirect word and the address field of the same word, to form the effective address Y.

It should be observed again that the variations DU and DL of Register modification (R) cannot be used with Register Then Indirect modification (RI).

If the programmer desires to reference an indirect word from the instruction itself without including Register modification, he specifies the "no modification" variation; under RI modification, this is indicated on the coding form by an asterisk alone placed in the variable field tag position.

The examples below illustrate the use of R combined with RI modification, including the use of (R) = N (no register modification). The asterisk (*) appearing in the modifier subfield is the Assembler symbol for I (Indirect). The address subfield, single-symbol expressions shown are not intended as realistic coding examples but rather to show the relation between operand addresses, indirect addressing, and register modification.

			VARIABLE FIELD	COMMENTS	
	LOCATION	OPERATION	(ADDRESS, TAG)	MODIFICATION TYPE	EFFECTIVE ADDRESS
1.		--	Z,AU*	(R)*	Y = B + C(XR1)
	$Z+C(AR)_{0-17}$	--	B,1	(R)	
2.		--	Z,*	(R)*	$Y = B + C(QR)_{0-17}$
	Z	--	B,QU	(R)	
3.		--	Z,*	(R)*	Y = M
	Z	--	B,5*	(R)*	
	B + C(X5)	--	C,3*	(R)*	
	C + C(X3)	--	M	(R)	

d. INDIRECT THEN REGISTER (IR) MODIFICATION

Indirect Then Register address modification is a combination type in which both indirect addressing and indexing (register modification) are performed. IR modification is not a simple inverse type of RI; several important differences exist.

Under IR modification, the processor first fetches an indirect word (obtained via I or IR) from the core storage location specified by the address field y of the machine instruction; and the C(R) of IR are safe-stored for use in making the final index modification to develop Y.

Next, the address modification, if any, specified by this first indirect word is carried out. If this modification is again IR, another indirect word is retrieved from storage immediately; and the new C(R) are safe-stored, replacing the previously safe-stored C(R). If an IR loop develops, the above process continues, each new R replacing the previously safe-stored R, until something other than IR is encountered in the indirect sequence — R, IT, or RI.

If the indirect sequence produces an RI indirect word, the R-type modification is performed immediately to form another address; but the I of this RI treats the contents of the address as an indirect word. The chain then continues with the R of the last IR still safe-stored, awaiting final use. At this point the new indirect word might specify IR-type modification, possibly renewing the IR loop noted above; or it might initiate an RI loop. In the latter case, when this loop is broken, the remaining modification types are R or IT.

When either R or IT is encountered, it is treated as type R where R is the last safe-stored C(R) of an IR modification. At this point the safe-stored C(R) are combined with the y of the indirect word that produced R or IT, and the effective address Y is developed.

If an indirect modification without Register modification is desired, the no-modification variation (N) of Register modification should be specified in the instruction. This normally will be entered on the coding sheet as *N in the modifier part of the variable field. (The entry * alone is equivalent to N* under RI modification and must be used in this way.) The mnemonic substitutions for (R) are listed under the Register modification description.

The examples below illustrate the use of IR-type modification, intermixed with R and RI types, under the several conditions noted above.

			VARIABLE FIELD	COMMENTS	
	LOCATION	OPERATION	(ADDRESS, TAG)	MODIFICATION TYPE	EFFECTIVE ADDRESS
1.		--	Z,*QL	*(R)	
	Z	--	M	(R)	$Y = M + C(QR)_{18-35}$
2.		--	Z,*3	*(R)	Y = C + C(X3)
	Z	--	B,5*	(R)*	
	B + C(X5)	--	C,IC	(R)	
3.		--	Z,*3	*(R)	$Y = M + C(QR)_{0-17}$
	Z	--	B,*5	*(R)	
	B	--	C,*QU	*(R)	
	C	--	M,7	(R)	
4.		--	Z,*DL	*(R)	$C(Y)_{18-35} = M$
	Z	--	B,3*	(R)*	
	B + C(X3)	--	M,QL	(R)	
5.		--	Z,*AL	*(R)	$Y = B + C(AR)_{18-35}$
	Z		B,AD	(T)	
6.		--	Z,*N	*(R)	Y = B
	Z	--	B,3	(R)	
7.		--	Z,*N	*(R)	Y = M + C(X5)
	Z	--	B,*5	*(R)	
	B	--	M,DU	(R)	
8.		--	Z,*	(R)*	Y = M + C(X5)
	Z	--	B,*5	*(R)	
	B	--	M,DU	(R)	
9.		--	Z,I	(T)	Y = B
	Z	--	B,*5	*(R)	(Note: I modification does not permit continuation of the indirect chain.)

e. INDIRECT THEN TALLY (IT) MODIFICATION

• Summary. Indirect Then Tally address modification in the M-605 is a combination type in which both indirect addressing and reference tallying are performed. In addition, automatic incrementing/decrementing of fields in the indirect word are done as hardware features, thus relieving the programmer of these responsibilities. The automatic tallying and other functions of the IT type modification greatly enhance the processing of tabular data in memory, provide the means for working upon character data, and allow termination on programmer-selectable numerical tally conditions. These features are explained in the nine subparagraphs to follow. (Refer to the special word formats TALLY, TALLYB, TALLYD, and TALLYC for Assembler coding of the indirect words used with IT.)

The ten variations under IT modification are summarized in the following table. It should be noted that the mnemonic substitution for IT on the Macro Assembler coding sheet is simply (T); the designator I for indirect addressing in IT is not represented. (Note that one of the substitutions for T is I.)

NAME OF THE VARIATION	CODING FORM SUBSTITUTION FOR I(T)	BINARY FORM (t_d FIELD)	EFFECT UPON THE INDIRECT WORD
Indirect	I	1001	None.
Increment address, Decrement tally	ID	1110	Add one to the address; subtract one from the tally.
Decrement address, Increment tally	DI	1100	Subtract one from the address; add one to the tally.
Sequence Character	SC	1010	Add one to the character position number; subtract one from the tally; add one to the address when the character count crosses a word boundary.
Character from Indirect	CI	1000	None.
Add Delta	AD	1011	Add an increment to the address; decrement the tally by one.
Subtract Delta	SD	0100	Subtract an increment from the address; increase the tally by one.
Fault	F	0000	None; the processor is forced to a fault trap starting at a predetermined, fixed location.
Increment address, Decrement tally, and Continue	IDC	1111	Same as ID variation except that further address modification can be performed.
Decrement address, Increment tally, and Continue	DIC	1101	Same as DI except that further address modification can be performed.

• Indirect (T) = I Variation. The Indirect (I) variation of IT modification is in effect a subset of the ID and DI variations described below in that all three — I, ID, and DI — make use of one indirect word in order to reference the operand. The I variation is functionally unique, however, in that the indirect word referenced by the program instruction remains unaltered — no incrementing/decrementing of the address field. Since the t_m and t_d subfields of the indirect word under I are not interrogated, this word will always terminate the indirect chain.

The following differences in the coding and effects of *, *N, and I should be observed:

1. RI modification is coded as R* for all cases, excluding R = N.

2. For R = N under RI, the modifier subfield can be written as N* or as * alone, according to programmer preference.

3. When N* or just * is coded, the Assembler generates a machine word with 20 in positions 30-35; 20 causes the processor to add 0 to the address y of the word containing the N* or * and then to access the indirect word at memory location y of the N* or * word.

4. IR modification is coded as *R for all cases, including R = N.

5. For R = N under IR, the modifier subfield must be written as *N.

6. When *N is coded, the Assembler generates 60 in positions 30-35 of the associated machine word; 60 causes the processor to (1) retrieve the indirect word at location y of the machine word, and (2) effectively safe-store zeros (for possible final index modification of the last indirect word — to develop the effective address Y).

7. IT modification is coded using only a variation designator (I, ID, DI, SC, CI, AD, SD, F, IDC, DIC); that is, the asterisk (*) is not written (for I). Thus, a written IT address modification appears as ALPHA, DI; BETA, AD; etc.

8. For the variation I under IT, the Assembler generates a machine word with 51 in bit positions 30-35; 51 causes the processor to perform one and only one indirect word retrieved from memory location y (of the word with I specified) to obtain the effective address Y.

• Increment Address, Decrement Tally (T) = ID Variation. The ID variation under IT modification provides the programmer with automatic (hardware) incrementing/decrementing of an indirect word that is best used for processing tabular operands (data located at consecutive memory addresses). The indirect word always terminates the indirect chain.

In the ID variation the effective address is the address field of the indirect word obtained via the tentative operand address of the instruction or preceding indirect word, whichever specified the ID variation. Each time such a reference is made to the indirect word, the address field of the indirect word is incremented by one; the tally portion of the indirect word is decremented by one. The incrementing and decrementing are done after the effective address is provided for the instruction operation. When the tally reaches zero, the tally runout indicator is set.

The example following shows the effect of ID.

LOCATION	OPERATION	VARIABLE FIELD ADDRESS, TAG	COMMENTS MODIFICATION TYPE	COMMENTS EFFECTIVE ADDRESS	REFERENCE
	--	Z,ID	(T)		
Z	--	B		B	1
				B + 1	2
				.	.
				.	.
Assuming an initial tally of j, the tally runout indicator is set on the jth reference				.	.
				B + n	n + 1
				.	.
				.	.
				.	.
				.	.

• Decrement Address, Increment Tally (T) = DI Variation. The DI variation under IT modification provides the programmer with automatic (hardware) incrementing/decrementing of an indirect word that is best used for processing tabular operands (data located at consecutive memory addresses). The indirect word always terminates the indirect chain.

In the DI variation the effective address is the address field minus one of the indirect word obtained via the tentative operand address of the instruction or preceding indirect word, whichever one specified the DI variation. Each time a reference is made to the indirect word, the address field of the indirect word is decremented by one; and the tally portion is incremented by one. The incrementing and decrementing is done prior to providing the effective address for the current instruction operation.

The effect of DI when writing programs is shown in the example following.

LOCATION	OPERATION	VARIABLE FIELD ADDRESS, TAG	COMMENTS MODIFICATION TYPE	COMMENTS EFFECTIVE ADDRESS	REFERENCE
	--	Z,DI	(T)		
Z	--	B		B - 1	1
				B - 2	2
				.	.
				.	.
				.	.
Assuming an initial tally of 4096-j the tally runout is set on the jth reference.				.	.
				B - n	n
				.	.
				.	.

• Sequence Character (T) = SC Variation. The Sequence Character (SC) variation is provided for programmed operations on 6-bit or 9-bit characters that are accessed sequentially in memory. Processor instructions that exclude character operations are so indicated in the individual instruction descriptions. For the SC variation, the effective operand address is the address field of the indirect word obtained via the tentative operand address of the instruction or

preceding indirect word that specified the SC variation. The character size is specified in the indirect word (see TALLY and TALLYB pseudo-operations).

Characters are operated on in sequence from left to right within the machine word. The character position field of the indirect word is used to specify the character to be involved in the operation and is intended for use only with those operations that involve the A- or Q-registers. The tally runout indicator is set when the tally field of the indirect word reaches 0.

The tally field of the indirect word is used to count the number of times a reference is made to a character. Each time an SC reference is made to the indirect word, the tally is decremented by one; and the character position is incremented by one to specify the next character position. When character position 5 is incremented, it is changed to position 0; and the address field of the indirect word is incremented by one. All incrementing and decrementing is done after the effective address has been provided for the correct instruction execution.

The effect of SC is shown in the following example.

			COMMENTS		
		VARIABLE FIELD	MODIFICATION	EFFECTIVE	
LOCATION	OPERATION	ADDRESS, TAG	TYPE	ADDRESS	REFERENCE
			Effective Address	Character Position	Reference
	--	Z, SC	(T)		
Z	--	B	B	0	1
			B	1	2
			.	.	.
			.	.	.
			.	.	.
An initial character position of 0 is assumed here.			B	5	6
Assuming an initial tally of j, the tally runout			B + 1	0	7
indicator is set on the jth reference.			.	.	.
			.	.	.
			.	.	.
			B + n	0	6n + 1
			.	.	.
			.	.	.

- Character From Indirect (T) = CI Variation. The Character from Indirect (CI) variation is provided for programmed operations on 6-bit or 9-bit characters in any situation where repeated reference to a single character in memory is required.

For this variation substitution, the effective address is the address field of the CI indirect word obtained via the tentative operand address of the instruction or preceding indirect word that specified the CI variation. The character position field of the indirect word is used to specify the character to be involved in the operation and is intended for use only with the operations that involve the A- or Q-register. The character size is specified in the indirect word (see TALLY and TALLYB pseudo-operations.)

This variation is similar to the SC variation except that no incrementing or decrementing of the address or character position is performed.

A CI example is:

LOCATION	OPERATION	VARIABLE FIELD ADDRESS, TAG	COMMENTS MODIFICATION TYPE	COMMENTS EFFECTIVE ADDRESS	REFERENCE
	- -	Z, CI	(T)	Y=B	
Z	- -	B			

- Add Delta (T) = AD Variation. The Add Delta (AD) variation is provided for programming situations where tabular data to be processed is stored at equally spaced locations, such as data words, each occupying two or more consecutive memory addresses. It functions in a manner similar to the ID variation, but the incrementing (delta) of the address field is selectable by the programmer.

Each time such a reference is made to the indirect word, the address field of the indirect word is increased by delta and the tally portion of the indirect word is decremented by one. The addition of delta and decrementing is done after the effective address is provided for the instruction operation.

The example following shows the effect of AD.

LOCATION	OPERATION	VARIABLE FIELD ADDRESS, TAG	COMMENTS MODIFICATION TYPE	COMMENTS EFFECTIVE ADDRESS	REFERENCE
	- -	Z, AD	(T)		
Z	- -	B	(R)	B	1
				B+δ	2
			.	B+2δ	3
			.	.	.
			.	.	.
			.	.	.
			.	B+nδ	n+1
			.	.	.
			.	.	.
				.	.

- Subtract Delta (T) = SD Variation. The Subtract Delta (SD) variation is useful in processing tabular data in a manner similar to the AD variation except that the table can easily be scanned from back to front using a programmer specified increment. The effective address from the indirect word is decreased by delta and the tally is increased by one each time the indirect word is used. This applies to the first reference to the indirect word, making the SD variation analogous to the DI variation.

• Fault (T) = F Variation. The fault variation enables the programmer to force program transfers to General Comprehensive Operating Supervisor routines or to his own corrective routines during the execution of an address modification sequence. (This will usually be an indication of some abnormal condition against which the programmer wishes to protect himself.)

• Increment Address, Decrement Tally and Continue (T) = IDC Variation. The IDC variation under IT modification functions in a manner similar to the ID variation except that, in addition to automatic incrementing/decrementing, it permits the programmer to continue the indirect chain in obtaining the instruction operand. Where the ID variation is useful for processing tabular data, the IDC variation permits processing of scattered data by a table of indirect pointers. More specifically, the ID portion of this variation gives the sequential stepping through a table; and the C portion (continuation) allows indirection through the tabular items. The tabular items may be data pointers, subroutine pointers or possibly a transfer vector.

The address and tally fields are used as described under the ID variation. The tag field uses the instruction address modification variations under the following restrictions: No variation is permitted which requires an indexing modification in the IDC cycle since the indexing adder is in use by the tally phase of the operation. Thus, permissible variations are any form of I(T) or I(R); but if (R)I or (R) is used, R must equal N.

The effect of IDC is indicated in the following example:

			COMMENTS		
LOCATION	OPERATION	VARIABLE FIELD ADDRESS, TAG	MODIFICATION TYPE	EFFECTIVE ADDRESS	REFERENCE
	- -	Z,IDC	(T)		
Z	- -	B	(R)	B	1
				B+1	2
				.	.
				.	.
Assuming an initial tally of j, the tally runout indicator is set on the jth reference.				.	.
				B+n	n+1
				.	.
				.	.
				.	.

• Decrement Address, Increment Tally, and Continue (T) = DIC Variation. The DIC variation under IT modification works in much the same way as the DI variation except that in addition to automatic decrementing/incrementing it allows the programmer to continue the indirect chain in obtaining an instruction operand. The continuation function of DIC operates in the same manner and under the same restrictions as IDC except that (1) it increments in the reverse direction, and (2) decrementing/incrementing is done prior to obtaining the effective address from the tally word. (Refer to the example under IDC; work from the bottom of the table to the top.) DIC is especially useful in processing last-in, first-out lists.

LOCATION	OPERATION	VARIABLE FIELD ADDRESS, TAG	COMMENTS MODIFICATION TYPE	COMMENTS EFFECTIVE ADDRESS	REFERENCE
	- -	Z, DIC	(T)		
Z	- -	B, *3	*(R)	C+C(X3)	1
B-1	- -	C, QU	(R)	A+C(X3)	2
B-2	- -	M, 5*	(R)*	$Q+C(AR)_{0-17}$	3
B-3	- -	D, *AU	*(R)	.	.
				.	.
.				.	.
M+C(XR5)	- -	A	(R)		
D	- -	Q	(R)		

Assuming an initial tally of 4096-j, the tally runout indicator is set on the jth reference.

C. PSEUDO-OPERATIONS

1. General

Pseudo-operations are so-called because of their similarity to machine operations in an object program. In general, however, machine operations are produced by computer instructions and perform some task, or part of a task, directly concerned with solving the problem at hand. Pseudo-operations work indirectly on the problem by performing machine conditioning functions, such as memory allocating, and by directing the Macro Assembler in the preparation of machine coding. A pseudo-operation affecting the Assembler may generate several, one, or no words in the object program. The Macro Assembler generative pseudo-operations are: OCT, DEC, BCI, DUP, CALL, SAVE, RETURN, and VFD.

All pseudo-operations for the Macro Assembler are grouped according to function and described as to composition and use. The pseudo-operation functional groups and their uses are:

FUNCTIONAL GROUP	PRINCIPAL USES
Control pseudo-operations	Selection of printout options for the assembly listing, direction of punchout of absolute/relocatable binary program decks, selection of format for the absolute binary deck.
Location counter pseudo-operations	Programmer control of single or multiple instruction counters.
Symbol defining pseudo-operations	Definition of Assembler source program symbols by means other than appearance in the location field of the coding form.
Data generating pseudo-operations	Production of binary data words for the assembly program.
Storage allocation pseudo-operations	Provision of programmer control for the use of memory.
Special pseudo-operations	Generation of zero operation code instructions, of binary words divided into two 18-bit fields, and of continued subfields for selected pseudo-operations.
MACRO pseudo-operations	Begin and end MACRO prototypes; Assembler generation of MACRO-argument symbols; and repeated substitution of arguments within MACRO prototypes.
Conditional pseudo-operations	Conditional assembly of variable numbers of input words based upon the subfield entries of these pseudo-operations.
Program linkage pseudo-operations	Generation of standard system subroutine calling sequences and return (exit) linkages.
Address, tally pseudo-operations	Control of automatic address, tally, and character incrementing/decrementing.

FUNCTIONAL GROUP	PRINCIPAL USES
Repeat mode coding formats	Control of the repeat mode of instruction execution (coding of RPT, RPD (macro-operation) and RPL instructions).

The above pseudo-operation functional groups, together with their pseudo-operations, are given as a complete listing with page references in Appendix D.

2. Control Pseudo-Operations

a. DETAIL ON/OFF (Detail Output Listing)

LOCATION	E/O	OPERATION		ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15	16	32
Blanks		DETAIL		ON	Normal mode
Blanks		DETAIL		OFF	
Blanks		DETAIL		SAVE, ON	
Blanks		DETAIL		SAVE, OFF	
Blanks		DETAIL		RESTORE	

Some pseudo-operations generate no binary words; however, several of them generate more than one. The generative pseudo-operations are: OCT, DEC, BCI, DUP, CALL, SAVE, RETURN, and VFD. The DETAIL pseudo-operation provides control over the amount of listing detail generated by the generative pseudo-operations.

The use of the DETAIL OFF pseudo-operation causes the assembly listing to be abbreviated by eliminating all but the first word generated by any of the above pseudo-operations. In the case of the DUP pseudo-operation, only the first iteration will be listed. The DETAIL ON pseudo-operation causes the Assembler to resume the listing which had been suspended by a DETAIL OFF pseudo-operation. The SAVE option in the variable field causes the present mode of the DETAIL pseudo-operation to be saved and then the mode specified by the second term in the variable field is set. The RESTORE option causes the saved status to be reset as the mode of DETAIL. If at the end of the listing the Assembler is in the DETAIL OFF mode, the literal pool will not be printed, but a notation will be made as to its origin.

b. EJECT (Restore Output Listing)

LOCATION	E/O	OPERATION	ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15 16	32
Blanks		EJECT		Column 16 must be blank

The EJECT pseudo-operation causes the Assembler to position the printer paper at the top of the next page, to print the title(s), and then print the next line of output on the second line below the title(s).

c. LIST ON/OFF (Control Output Listing)

LOCATION	E/O	OPERATION	ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15 16	32
Blanks		LIST	ON	Normal mode
Blanks		LIST	OFF	
Blanks		LIST	SAVE, ON	
Blanks		LIST	SAVE, OFF	
Blanks		LIST	RESTORE	

The use of LIST in the operation field with OFF in the variable field causes the normal listing to change as follows: the instruction LIST OFF will appear in the listing; thereafter, only instructions which are flagged in error will appear. If the assembly ends in the LIST OFF mode, only the error messages will appear.

The use of LIST in the operation field with ON in the variable field causes the normal listing, which was suspended by a LIST OFF pseudo-operation, to be resumed. The SAVE option in the variable field causes the present mode of the LIST pseudo-operation to be saved and then the mode specified by the second term in the variable field is set. The RESTORE option causes the saved status to be reset as the mode of LIST.

d. REM (Remarks)

LOCATION 1 2 6	E/O 7	OPERATION 8 14	15	ADDRESS, MODIFIER 16	COMMENTS 32
Blanks or remarks		REM			Remarks and comments in the variable field start at column 12 or later

The REM pseudo-operation causes the contents of this line of coding to be printed on the assembly listing (just as the comments appear on the coding sheet). However, for purposes of neatness, columns 8-10 are replaced by blanks before printing.

REM is provided for the convenience of the programmer; it has no other effect upon the assembly.

* (In Column One--Remarks)

LOCATION 1 2 6	E/O 7	OPERATION 8 14	15	ADDRESS, MODIFIER 16	COMMENTS 32
*					Remarks and comments in columns 2-80

A card containing an asterisk (*) in column 1 is taken as a remark card. The contents of columns 2-80 are printed on the assembly listing (just as they appear on the coding sheet); the asterisk has no other effect on the assembly program.

e. LBL (Label)

LOCATION 1 2 6	E/O 7	OPERATION 8 14	15	ADDRESS, MODIFIER 16	COMMENTS 32
Blanks		LBL			Blanks or up to 8 alphabetic and numeric characters in the variable field

LBL causes the Assembler to serialize the binary cards using columns 73-80, except when punching full binary cards by use of the FUL pseudo-operation. The LBL pseudo-operation allows the programmer to specify a left-justified alphabetic label for the identification field and begin serialization with some initial serial number other than zero.

The following conditions apply:

1. If the variable field is blank, the Assembler will discontinue serialization of the binary deck.

2. If the variable field is not blank, serialization will begin with the characters appearing in the variable field; the characters are left-justified and filled in with terminating zeros up to the position(s) used for the sequence number. Serialization is incremented until the rightmost nonnumeric character is encountered, at which time the sequence recycles to zero.

3. If no LBL pseudo-operation appears in the symbolic deck, the Assembler will begin serializing with 00000000.

f. PCC ON/OFF (Print Control Cards)

LOCATION	E/O	OPERATION		ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15	16	32
Blanks		PCC		ON	
Blanks		PCC		OFF	Normal mode
Blanks		PCC		SAVE, ON	
Blanks		PCC		SAVE, OFF	
Blanks		PCC		RESTORE	

The PCC pseudo-operation affects the listing of the following pseudo-operations:

DETAIL	LIST	TTL	PMC
EJECT	PCC	TTLS	PUNCH
LBL	REF	CRSM	IDRP
INE	IFE	IFG	IFL

PCC ON causes the affected pseudo-operations to be printed. PCC OFF causes the affected pseudo-operations to be suppressed; this is the normal mode at the beginning of the assembly. If the Assembler is already in a specified ON/OFF mode, then the pseudo-operation requesting the same ON/OFF mode is ignored. The SAVE option in the variable field causes the present mode of the PCC pseudo-operation to be saved and then the mode specified by the second term in the variable field is set. The RESTORE option causes the saved status to be reset as the mode of PCC.

g. REF ON/OFF (Reference)

LOCATION 1 2 6	E/O 7	OPERATION 8 14	15	ADDRESS, MODIFIER 16	COMMENTS 32
Blanks		REF		ON	Normal mode
Blanks		REF		OFF	
Blanks		REF		SAVE, ON	
Blanks		REF		SAVE, OFF	
Blanks		REF		RESTORE	

The REF pseudo-operation controls the Assembler in making entries in the symbol reference table.

REF ON causes the Assembler to begin making entries into the symbol reference table. REF OFF causes the Assembler to suppress making entries into the symbol reference table. The SAVE option in the variable field causes the present mode of the REF pseudo-operation to be saved and then the mode specified by the second term in the variable field is set. The RESTORE option causes the saved status to be reset as the mode of REF.

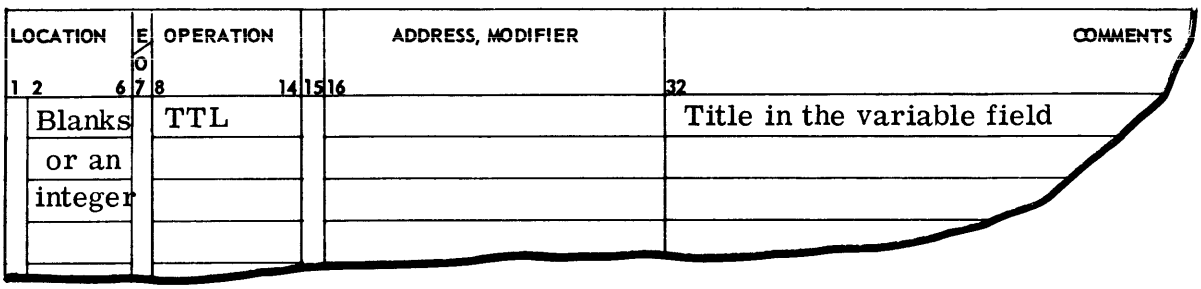
h. PMC ON/OFF (Print MACRO Expansion)

LOCATION 1 2 6	E/O 7	OPERATION 8 14	15	ADDRESS, MODIFIER 16	COMMENTS 32
Blanks		PMC		ON	
Blanks		PMC		OFF	Normal mode
Blanks		PMC		SAVE, ON	
Blanks		PMC		SAVE, OFF	
Blanks		PMC		RESTORE	

The PMC pseudo-operation causes the Assembler to list or suppress all instructions generated by a MACRO call.

PMC ON causes the Assembler to print all generated instructions. PMC OFF causes the Assembler to suppress all but the initial generated instruction. The SAVE option in the variable field causes the present mode of the PMC pseudo-operation to be saved and then the mode specified by the second term in the variable field is set. The RESTORE option causes the saved status to be reset as the mode of PMC.

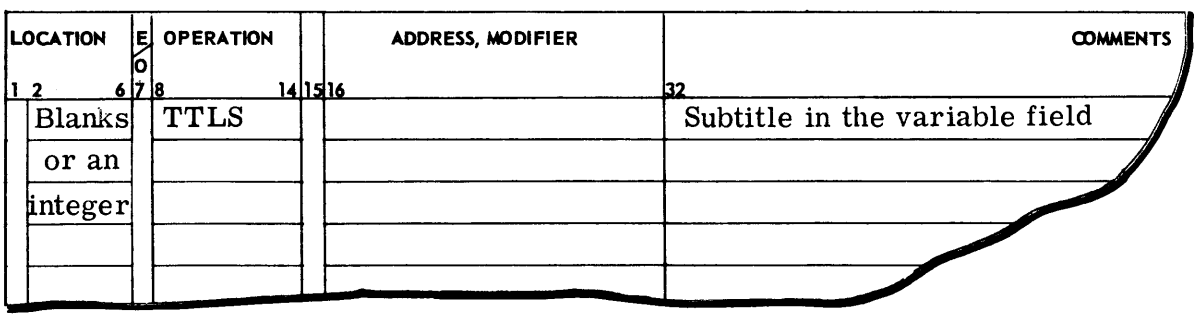
i. TTL (Title)



The TTL pseudo-operation causes the printing of a title at the top of each page of the assembly listing. In addition, when the assembler encounters a TTL card, it will cause the output listing to be restored to the top of the next page and the new title will be printed. The information punched in columns 16-72 is interpreted as the title.

Redefining the title by repeated TTL pseudo-operations may be used as often as the programmer desires. Deletion of the title may be accomplished by a TTL pseudo-operation with a blank variable field. If a decimal integer appears in the location field, the page count will be renumbered beginning with the specified integer.

j. TTLS (Subtitle)



The TTLS pseudo-operation is identical in function to the TTL pseudo-operation except that it causes subtitling to occur. When a TTLS pseudo-operation is encountered, the subtitle provided in columns 16-72 replaces the current subtitle; the output listing is restored to the top of the next page. The title and new subtitle are then printed. Only one level of subtitling may follow a title.

k. INHIB ON/OFF (Inhibit Interrupts)

LOCATION	E/O	OPERATION	ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15 16	32
Blanks		INHIB	ON	
Blanks		INHIB	OFF	Normal mode
Blanks		INHIB	SAVE, ON	
Blanks		INHIB	SAVE, OFF	
Blanks		INHIB	RESTORE	

The instruction INHIB ON causes the Assembler to set the program interrupt inhibit bit in bit position 28 of all machine instructions which follow the pseudo-operation. The setting of the program interrupt inhibit bit continues for the remainder of the assembly, unless the pseudo-operation INHIB OFF is encountered.

The INHIB OFF causes the Assembler to stop setting the program interrupt inhibit bit in each instruction, if used when the Assembler is in the INHIB ON mode. The SAVE option in the variable field causes the present mode of the INHIB pseudo-operation to be saved and then the mode specified by the second term in the variable field is set. The RESTORE option causes the saved status to be reset as the mode of INHIB.

l. ABS (Output Absolute Test)

LOCATION	E/O	OPERATION	ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15 16	32
Blanks		ABS		Column 16 must be blank

The ABS pseudo-operation causes the Assembler to output absolute binary text.

The normal mode of the Assembler is relocatable; however, if absolute text is required for a given assembly, the ABS pseudo-operation should appear in the deck before any instructions or data. It may be preceded only by listing pseudo-operations. It may, however, appear repeatedly in an assembly interspersed with the FUL pseudo-operation. It should be noted that the pseudo-operations affecting relocation are considered errors in an absolute assembly.

Those pseudo-operations that will be in error if used in an absolute assembly are:

BLOCK	SYMDEF
ERLK	SYMREF

(Refer to the descriptions of binary punched card formats in this chapter for details of the absolute binary text.)

m. FUL (Output Full Binary Text)

LOCATION	E/O	OPERATION		ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15	16	32
Blanks		FUL			Column 16 must be blank

The FUL pseudo-operation is used to specify absolute assembly and the FUL format for absolute binary text.

The FUL pseudo-operation has the same effect and restrictions on the Assembler as ABS, except for the format of the binary text output. The format of the text is of continuous information with no address identification; that is, the absolute binary cards are punched with program instructions in columns 1-78 (26 words). Such cards can be used in self-loading operations or other environments where control words are not required on the binary card.

n. TCD (Punch Transfer Card)

LOCATION	E/O	OPERATION		ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15	16	32
Blanks		TCD			An expression in the variable field
or a					
symbol					

In an absolute assembly, the binary transfer card, produced at the end of the deck as a result of the end card, directs the loading program to cease loading and turn control over to the program at the point specified by the transfer card. Sometimes it is desirable to cause a transfer card to be produced before encountering the end of the deck. This is the purpose of the TCD pseudo-operation. Thus, a binary transfer card is produced generating a transfer address equivalent to the value of the expression in the variable field.

TCD is an error in the relocatable mode.

o. PUNCH ON/OFF (Control Card Output)

LOCATION	E/O	OPERATION		ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15	16	32
Blanks		PUNCH		ON	Normal mode
Blanks		PUNCH		OFF	
Blanks		PUNCH		SAVE, ON	
Blanks		PUNCH		SAVE, OFF	
Blanks		PUNCH		RESTORE	

The normal mode of the Assembler is to punch binary cards. If PUNCH is used in the operation field with OFF in the variable field, the binary deck will not be punched, beginning at the point the Assembler encounters the pseudo-operation.

If PUNCH is used in the operation field with ON in the variable field, the punching of binary cards, which was suspended by the PUNCH OFF pseudo-operation, will be resumed. The SAVE option in the variable field causes the present mode of the PUNCH pseudo-operation to be saved and then the mode specified by the second term in the variable field is set. The RESTORE option causes the saved status to be reset as the mode of PUNCH.

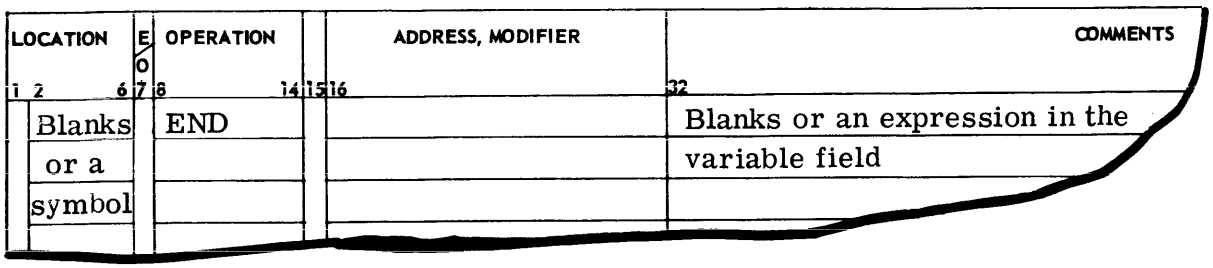
p. DCARD (Punch BCD Card)

LOCATION	E/O	OPERATION		ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15	16	32
Blanks		DCARD			Two subfields on the variable field

The first subfield contains a decimal integer N (limited only by the size of available memory), and the record subfield contains a single BCD character used as a decimal data identifier. The Assembler punches the next N cards after the DCARD instruction with the specified BCD identifier in column one of each of these N cards and with the BCD information taken from the corresponding source cards on a one-for-one basis.

There are no restrictions on the BCD information that can be placed in columns 2-72 of the source cards. (One of the significant uses of DCARD is to generate Operating Supervisor (GECOS/605) control cards.)

q. END (End of Assembly)



LOCATION	E/O	OPERATION	ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15 16	32
Blanks or a symbol		END		Blanks or an expression in the variable field

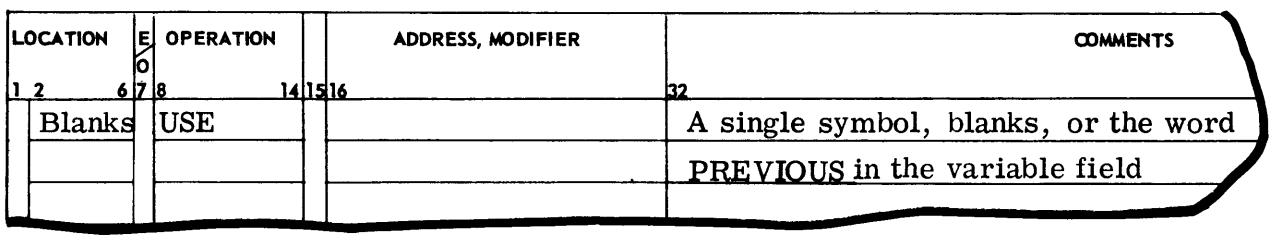
The END pseudo-operation signals the Assembler that it has reached the end of the symbolic input deck; it must be present as the last physical card encountered by the Assembler.

If a symbol appears in the location field, it is assigned the next available location.

In a relocatable assembly, the variable field must be blank; in an absolute assembly, the variable field may contain an expression. In relocatable decks, the starting location of the program will be an entry location and the location specified is given to the General Loader (GELOAD/605) by a special control card used with the GELOAD/605. Absolute programs require a binary transfer card which is generated by the END pseudo-operation. The transfer address is obtained from the expression in the variable field of the end card.

3. Location Counter Pseudo-Operations

a. USE (Use Multiple Location Counters)



LOCATION	E/O	OPERATION	ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15 16	32
Blanks		USE		A single symbol, blanks, or the word PREVIOUS in the variable field

The Assembler provides the ability to employ multiple location counters via the USE pseudo-operation. The location counters are established by the user and are usually originated with the location value of their first appearance in the program. However, their initial value may be specified by the BEGIN pseudo-operation.

The employment of this pseudo-operation causes the Assembler to place succeeding cards under control of the location counter represented by the symbol in the variable field. Any regular location counter in control at the appearance of USE is suspended at its current value and is preserved as the PREVIOUS counter.

If the word PREVIOUS appears in the variable field, the Assembler reactivates the regular

location counter which appeared just before the present one. The normal mode of the Assembler is under the blank location counter; that is, all instructions up to the first USE pseudo-operation are controlled by the blank location counter.

b. BEGIN (Origin of a Location Counter)

LOCATION	E/O	OPERATION		ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15	16	32
Blanks		BEGIN			Two subfields in the variable field

The BEGIN pseudo-operation is used to specify to the Assembler the origin of a given location counter if the location counter is to be other than the nominal (the blank counter).

The location counter symbol is specified in the first subfield and is given the value specified by the expression found in the second subfield. Any symbol appearing in the second subfield must have been previously defined and must appear under one location counter. The BEGIN pseudo-operation may appear anywhere in the deck.

If BEGIN is not used to give the nth location counter (under USE) an origin, its initial value is assigned as the first location not used by the (n-1)th location counter. The BEGIN pseudo-operation makes the location counter affected by it, independent of the order of location counter definition, i.e. if the origin of the Nth location counter is defined by BEGIN, the origin of the (N+1)th location counter is the first location not used by the (N-1)th counter, provided that neither is affected by BEGIN.

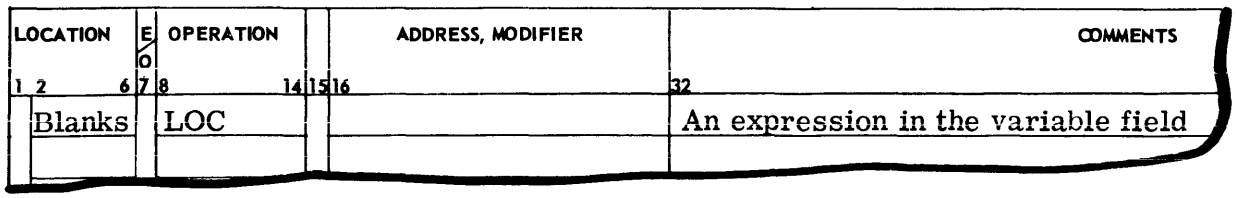
c. ORG (Origin Set by Programmer)

LOCATION	E/O	OPERATION		ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15	16	32
Blanks		ORG			An expression in the variable field
or a					
symbol					

The ORG pseudo-operation is used by the programmer to change the next value of a counter, normally assigned by the Assembler, to a desired value. If ORG is not used by the programmer, the counter is initially set to zero.

All symbols appearing in the variable field must have been previously defined. If a symbol appears in the location field, it is assigned the value of the variable field. If the result of the evaluation of a variable field expression is absolute, the instruction counter will be reset to the specified value relative to the current location counter. If an expression result is relocatable, the current location counter will be changed to the value given by the expression in the variable field.

d. LOC (Location of Output Text)

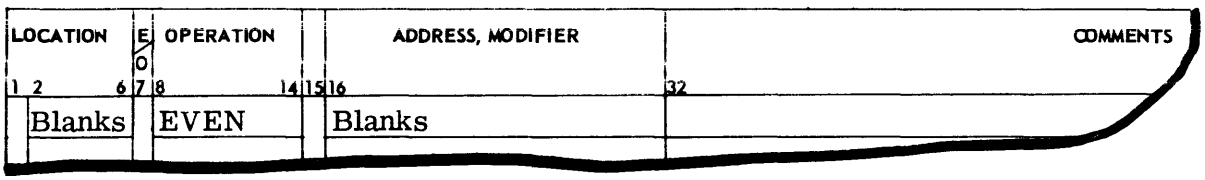


The LOC pseudo-operation functions identically to the ORG pseudo-operation, with one exception; it has no effect on the loading address when the Assembler is punching binary text. That is, the value of the location counter will be changed to that given by the variable field expression, but the loading will continue to be consecutive. This provides a means of assembling code in one area of memory while its execution will occur at some other area of memory.

All symbols appearing in the variable field of this pseudo-operation must have been previously defined.

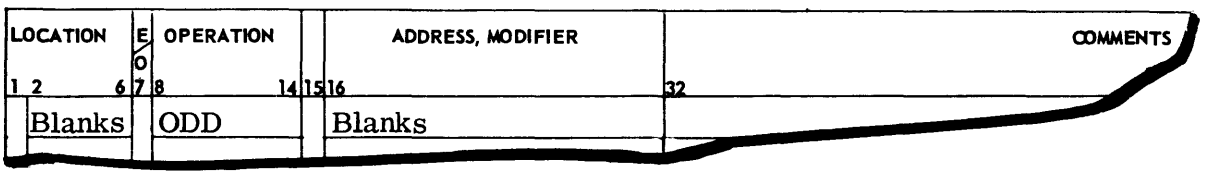
The sole purpose of this pseudo-operation is to allow program coding to be loaded in one section of memory and then to be subsequently moved to another section for execution.

e. EVEN



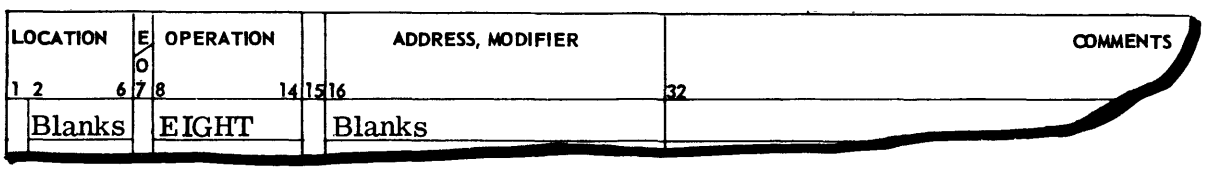
The EVEN pseudo-operation causes the machine instruction following the pseudo-operation to be located at the next even location. It is equivalent to an E in column 7 of that instruction.

f. ODD



The ODD pseudo-operation causes the machine instruction following the pseudo-operation to be located at the next odd location. It is equivalent to an O in column 7 of that instruction.

g. EIGHT



The EIGHT pseudo-operation causes the machine instruction following the pseudo-operation to be located at the next available location which is a multiple of eight. It is equivalent to an 8 in column 7 of that instruction.

4. Symbol Defining Pseudo-Operations

Increased facility in program writing frequently can be realized by the ability to define symbols to the Assembler by means other than their appearance in the location field of an instruction or by using a generative pseudo-operation. Such a symbol definition capability is used for (1) equating symbols, or (2) defining parameters used frequently by the program but which are subject to change. The symbol-defining pseudo-operations serve these and other purposes.

It should be noted that they do not generate any machine instructions or data but are available merely for the convenience of the programmer.

a. EQU (Equal To)

LOCATION	E/O	OPERATION		ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15 16		32
Symbol		EQU			An expression in the variable field

The purpose of the EQU pseudo-operation is to define the symbol in the location field to have the value of the expression appearing in the variable field. The symbol in the location field will assume the same mode as that of the expression in the variable field, that is, absolute or relocatable. (See Relocatable and Absolute Expressions.)

All symbols appearing in the variable field must have been previously defined and must fall under the same location counter, SYMDEF or SYMREF symbols cannot appear in the variable field.

If an asterisk (*) appears in the variable field denoting the current location counter value, it will be given the value of the next sequential location not yet assigned by the Assembler with respect to the unique location counter presently in effect.

b. FEQU (FORTRAN - Equal To)

LOCATION	E/O	OPERATION		ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15 16		32
Symbol		FEQU			An expression in the variable field

FEQU defines the symbol in the location field to have the value of the expression appearing in the variable field. FEQU is the same as EQU except that it does not require previous definition of the symbols appearing in the variable field.

Symbols defined by FEQU cannot be used on pseudo-operations affecting location counters, such as BSS, DUP, etc.

c. BOOL (Boolean)

LOCATION	E/O	OPERATION	ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15 16	32
Symbol		BOOL		A Boolean expression in the variable field

The BOOL pseudo-operation defines a constant of 18 bits and is similar to EQU except that the evaluation of the expression in the variable field is done assuming Boolean operators. By definition, all integral values are assumed in octal and are considered to be in error otherwise. The symbol in the location field will always be absolute, and the presence of any expression other than an absolute one in the variable field will be considered an error.

All symbols appearing in the variable field must have been previously defined.

d. SET (Symbol Redefinition)

LOCATION	E/O	OPERATION	ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15 16	32
Symbol		SET		An expression in the variable field

The SET pseudo-operation permits the redefinition of a symbol previously defined to the Assembler. This ability is useful in MACRO expansions where it may be undesirable to use created symbols (CRSM).

All symbols entered in the variable field must have been previously defined and must fall under the same location counter. SYMDEF or SYMREF symbols cannot be used in the variable field.

The symbol in the location field is given the value of the expression in the variable field. The SET pseudo-operation may not be used to define or redefine a relocatable symbol.

When a symbol occurring in the location field has been previously defined by a means other than a previous SET, the current SET pseudo-operation will be ignored and flagged as an error.

The last value assigned to a symbol by SET affects only subsequent in-line coding instructions using the redefined symbol.

e. MIN (Minimum)

LOCATION 1 2 6	E/O 7	OPERATION 8 14	15	ADDRESS, MODIFIER 16	COMMENTS 32
Symbol		MIN			A sequence of expressions, separated by commas, in the variable field -- all of the same type; that is, relocatable or absolute

The MIN pseudo-operation defines the symbol in the location field as having the minimum value among the various values of all relocatable or all absolute expressions contained in the variable field.

All symbols appearing in the variable field must have been previously defined and must fall under the same location counter. SYMDEF or SYMREF symbols cannot be used in the variable field.

f. MAX (Maximum)

The MAX pseudo-operation is coded in the same format as MIN above. It defines the symbol in the location field as having the maximum value of the various expressions contained in the variable field.

All symbols appearing in the variable field must have been previously defined and must fall under the same location counter. SYMDEF or SYMREF symbols cannot be used in the variable field.

g. HEAD (Heading)

LOCATION 1 2 6	E/O 7	OPERATION 8 14	15	ADDRESS, MODIFIER 16	COMMENTS 32
Blanks		HEAD			From 1 to 7 subfields in the variable field, each containing a single, nonspecial charact used as a heading character

In programming, it is sometimes desirable to combine two programs, or sections of the same program, that use the same symbols for different purposes. The HEAD pseudo-operation makes such a combination possible by prefixing each symbol of five or fewer characters with a heading character. This character must not be one of the special characters, that is, it must be one of the characters A-Z or 0-9. Using different heading characters, in different program sections later to be combined for assembly, removes any ambiguity as to the definition of a given symbol.

The effect of the HEAD pseudo-operation is to cause every symbol of five or less characters, appearing in either the location field or the variable field, to be prefixed by the current HEAD character. The current HEAD character applies to all symbols appearing after the current HEAD pseudo-operation and before the next HEAD or END pseudo-operation.

Deheading is accomplished by a zero or blanks in the variable field. To understand more thoroughly the operation of the heading function, it is necessary to know that the Assembler internally creates a six-character symbol by right-justifying the characters of the symbol and filling in leading zeros. Thus, if the Assembler is within a headed program section and encounters a symbol of five or fewer characters, it inserts the current HEAD character into the high-order, leftmost character position of the symbol. Each symbol, with its inserted HEAD character, then can be placed in the Assembler symbol table as unique entries and assigned their respective location values.

It is also possible to head a program section with more than one character. This is done by using the pseudo-operation HEAD in the operation field with from two to seven heading characters in the variable field, separated by commas. The effect of a multiple heading is to define each symbol of that section once for each heading character. Thus, for example, if the symbols SHEAR, SPEED, and PRESS are headed by

HEAD	X, Y, Z

nine unique symbols

XSHEAR	XSPEED	XPRESS
YSHEAR	YSPEED	YPRESS
ZSHEAR	ZSPEED	ZPRESS

are generated and placed in the Assembler symbol table. This allows regions by HEADX, HEADY or HEADZ to obtain identical values for the symbols SHEAR, SPEED, and PRESS.

Cross-referencing among differently headed sections may be accomplished by the use of six-character symbols or by the use of the dollar sign($). Six-character symbols are immune to HEAD; therefore, they provide a convenient method of cross-referencing among differently headed regions.

To allow the programmer more flexibility in cross-referencing, the Assembler language includes the use of the dollar sign ($) to denote references to an alien-headed region.

If the programmer wishes to reference a symbol of less than six characters in another program section, he merely prefixes the symbol by the HEAD character for that respective section, separating the HEAD character from the body of the symbol by a dollar sign ($).

To reference from a headed region into a region that is not headed, the programmer may use either the heading character zero (0) preceding the symbol or, if the symbol is the initial value of the variable field, then the appearance of the leading dollar sign will cause the zero heading to be attached to the symbol.

EXAMPLE OF HEAD PSEUDO-OPERATION

```
START     LDA     A          Initial instruction (no heading)
          ---
          ---
          TRA     B$SUM      Transfer to new headed section
A         BSS     1
          HEAD    B
SUM       LDA     $A
          ---
          ---                Section headed B
          TRA     0$START + 2
          END
```

The LDA $A could have been written as LDA 0$A, as they both mean the same.

h. SYMDEF (Symbol Definition)

LOCATION	E/O	OPERATION	ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15 16	32
Blanks		SYMDEF		Symbols separated by commas in the variable fields

The SYMDEF pseudo-operation is used to identify symbols which appear in the location field of a subroutine when these symbols are referred to from outside the subroutine (by SYMREF). Also, the programmer must provide a unique SYMDEF for use by the Loader to denote the main program entry point for the loading operations (non-FORTRAN). The symbols used in the variable field of a SYMDEF instruction will be called SYMDEF symbols. Multiple SYMDEF symbols cannot occur since the Assembler ignores the current definition if it finds the same symbol previously entered in the SYMDEF table.

The appearance of a symbol in the variable field of a SYMDEF instruction indicates that:

1. The symbol must appear in the location field of only one of the instructions within the subroutine in which SYMDEF occurs.

2. The Assembler will place each such SYMDEF symbol along with its relative address in the preface card at assembly time.

3. At load time, the Loader will form a table of SYMDEF symbols to be used for linkage with SYMREF symbols.

It is possible to classify SYMDEF symbols as primary and secondary. A secondary SYMDEF symbol is denoted by a minus sign in front of the symbol. The Loader will provide linkage for a secondary SYMDEF symbol only after linkage has been required to a primary SYMDEF within the same subprogram. The use of secondary SYMDEF symbols is intended for programmers who are specifically concerned with using the system subroutine library and generating routines for accessing the library. Secondary SYMDEF symbols are normally thought of as secondary

entries to subroutines contained within a subprogram library package that will be used as an entire package.

i. SYMREF (Symbol Reference)

LOCATION	E/O	OPERATION	ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15 16	32
Blanks		SYMREF		A sequence of symbols separated by commas entered in the variable field

The SYMREF pseudo-operation is used to denote symbols which are used in the variable field of a subroutine but are defined in a location field external to the subroutine. Symbols used in the variable field of a SYMREF instruction will be called SYMREF symbols.

When a symbol appears in the variable field of a SYMREF instruction, the following items apply:

1. The symbol should occur in the variable field of at least one instruction within the subroutine.

2. At assembly time the Assembler will enter the SYMREF symbol in the preface card of the assembled deck and place a special entry number (page IV-78, 79) in the variable fields of all instructions in the referenced subroutine which contain the symbol.

3. At load time the Loader will associate the SYMREF symbol with a corresponding SYMDEF symbol and place the appropriate address in all instructions that have been given the special entry entry number.

Symbols appearing in the variable field of a SYMREF instruction must not appear in the location field of any instruction within the subroutine in which SYMREF is used.

EXAMPLE OF SYMDEF AND SYMREF PSEUDO-OPERATIONS

Base Program or Subprogram				Referencing Subroutine	
	SYMDEF	ATAN, ATAN2		SYMREF	ATAN, ATAN2
ATAN2	STC2	INDIC		:	
ATANS	SAVE	0,1		:	
	SZN	INDIC		:	
	TZE	START	POLYX	FLD	X
	:			:	
ATAN	STZ	INDIC		TSX1	ATAN
	TRA	ATANS		:	
	:			TSX2	ATAN2
				:	

j. OPD (Operation Definition)

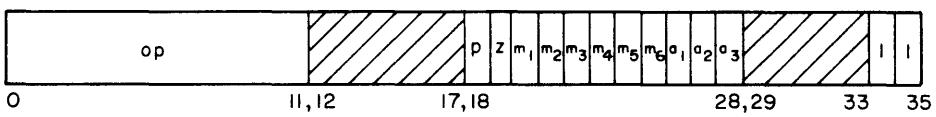
LOCATION	E/O	OPERATION		ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15	16	32
New		OPD			One or more subfields, separated by comma
opera-					in the variable field. The subfields define t
tion					bit configuration of the new operation code
code					

The OPD pseudo-operation may be used to define or redefine machine instructions to the Assembler. This allows programmers to add operation codes to the Assembler table of operation codes during the assembly process. This is extremely useful and powerful in defining new instructions or special bit configurations, unique in a particular program, to the Assembler.

The variable field subfields are bit-oriented and have the same general form as described under the VFD pseudo-operation. In addition, the variable field, considered in its entirety, requires the use of either of two specific 36-bit formats for defining the operation.

1. The normal instruction format
2. The input/output operation format

The normal instruction-defining format and subfields are shown below:



op--new operation code (bits 0-11)
p--p=1, machine operation
p=0, pseudo-operation
z--must be zero
m--modifier tag type (0=allowed; 1=not allowed)
m_1: register modification (R)
m_2: indirect addressing (*)
m_3: not used
m_4: Direct Upper (DU)
m_5: Direct Lower (DL)
m_6: Sequence Character (SC) and Character from Indirect (CI)
a--address field conditions (0 = not required; 1=required)
a_1: address required/not required

a_2: address required even

a_3: address required absolute

1--octal assembly listing format (x represents one octal digit)
00: xx xxxx xxxxxx
01: xxxxxxxxxxxx
10: xxxxxx xxxxxx
11: xxxxxx xxxx xx

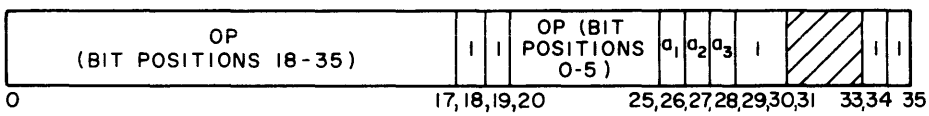
The assembly listing types 00, 01, 10, and 11 are used for input/output commands, data-generating pseudo-operations (OCT, DEC, BCI, etc.), special word-generating pseudo-operations (such as ZERO), and machine instructions.

To illustrate the use of OPD, assume one wished to define the extant machine instruction, Load A (LDA). Using the preceding format and the octal notation (as described under the VFD pseudo-operation), one could code OPD as

```
        LDA     OPD     O12/2350,6/,O2/2,6/,O3/4,5/,O2/3
or      LDA     OPD     O18/235000,O2/2,6/,O3/4,5/,O2/3
or      LDA     OPD     O36/235000401003
```

or in other forms, providing the bit positions of the instruction-defining format are individually specified to the Assembler.

The input/output operation-defining format and subfields are as follows:



op--new operation code for bit positions 18-35 and 0-5 (see Appendix E)
a--address field conditions (0=not required; 1=required)
a_1: address required/not required

a_2: address required even

a_3: address required absolute

i--type of input/output command (see Appendix E)
00: OP DA,CA KKDACAKKKKKK
01: OP NN,DA,CA KKDACAKKKKNN
10: OP CC,DA,CA KKDACAKKCCKK
11: OP A,C AAAAAAKKCCCC
l--see preceding normal instruction format

NOTE: Bit position 19 must be a binary 1 for input/output operations.

Input/output operation types 00, 01, and 10 are the formats for the commands; type 11 is the format for a Data Control Word (DCW).

As an example of the use of OPD to generate an input/output command (using the above format for the variable field and defining the bits according to the rules for VFD), assume one wanted to generate the command, Write Tape Binary. This could be written as

```
WTB         OPD         18/,O2/3,O6/15,10/0
```

or in various other bit-oriented forms.

k. OPSYN (Operation Synonym)

LOCATION 1 2 6	E/O 7	OPERATION 8 14	15	ADDRESS, MODIFIER 16	COMMENTS 32
A sym-		OPSYN			A mnemonic operation code in the
bol or					variable field
opera-					
tion					
code					

The OPSYN pseudo-operation is used for equating either a newly defined symbol or a presently defined operation to some operation code already in the operation table of the Assembler. The operation code may have been defined by a prior OPD or OPSYN pseudo-operation; in any case, it must be in the Assembler operation table.

5. Data Generating Pseudo-Operations

The Assembler language provides four pseudo-operations which can be used to generate data in the program at the time of assembly. These are BCI, OCT, DEC, and VFD. The first three, BCI, OCT, and DEC, are word-oriented while VFD is bit-oriented. There exists a fifth pseudo-operation, DUP, which in itself does not generate data, but through its repeat capability causes symbolic instruction and pseudo-operations to be iterated.

a. OCT (Octal)

LOCATION 1 2 6	E/O 7	OPERATION 8 14	15	ADDRESS, MODIFIER 16	COMMENTS 32
Symbol		OCT			One or more subfields appearing in the
or					variable field, each one containing a signed
blanks					or unsigned octal integer.

The OCT pseudo-operation is used to introduce data in octal integer notation into an assembled program. The OCT pseudo-operation causes the Assembler to generate n locations of OCT

data where the variable field contains n subfields (n-1 commas). Consecutive commas in the variable field cause the generation of a zero data word, as does a comma followed by a terminal blank. Up to 12 octal digits plus the leading sign may make up the octal number.

The OCT configuration is considered true and will not be complemented on negatively signed numbers. The sign applies only to bit 0. All assembly program numbers are right-justified, retaining the integer form.

EXAMPLE OF OCT PSEUDO-OPERATION

```
OCT  1,-4,7701,+3,,-77731,04
```

If the current location counter were set at 506, the above would be printed out as follows (less the column headings):

Location	Contents	Relocation	
000506	000000000001	000	OCT 1,-4,7701,+3,,-7731,04
000507	400000000004	000	
000510	000000007701	000	
000511	000000000003	000	
000512	000000000000	000	
000513	400000077731	000	
000514	000000000004	000	

b. DEC (Decimal)

LOCATION (1 2 — 6)	E/O (7)	OPERATION (8 — 14)	(15)	ADDRESS, MODIFIER (16)	COMMENTS (32)
Symbol		DEC			One or more subfields in the variable field,
or					each containing a decimal entry
blanks					

The Assembler language provides four types of decimal information which the programmer may specify for conversion to binary data to be assembled. The various types are uniquely defined by the syntax of the individual subfields of the DEC pseudo-operation. The basic types are single-precision, fixed-point numbers; single-precision, floating-point numbers; double-precision fixed-point numbers; and double-precision floating-point numbers. All fixed-point numbers are right-justified in the assembly binary words; floating-point numbers are left-justified to bit position eight with the binary point between positions 0 and 1 of the mantissa. (The rules for forming these numbers are described under Decimal Literals, see B4a.)

EXAMPLES OF SINGLE-PRECISION DEC PSEUDO-OPERATION

```
GAMMA    DEC    3,-1,6.,.2E1,1B27,1.2E1B32,-4
```

The above would print out the following data words (without column headings), assuming that GAMMA equals 1041.

Location	Contents	Relocation	
001041	000000000003	000	GAMMA DEC 3,-1,6.,.2E1, 1B27,1.2E1B32, -4
001042	77777777777	000	
001043	00660000000	000	
001044	00440000000	000	
001045	00000000400	000	
001046	00000000140	000	
001047	77777777774	000	

The presence of the decimal point and/or the E scale factor implies floating-point, while the added B (binary scale) implies fixed-point binary numbers. The absence of all of these elements implies integers. Several more examples follow (see decimal literals for further explanation):

```
DEC   -1B17,-1.,1000
```

With the location counter at 1050, the above would generate:

Location	Contents	Relocation	
001050	777777000000	000	DEC -1B17,-1.,1000
001051	001000000000	000	
001052	000000001750	000	

EXAMPLE OF DOUBLE-PRECISION DEC PSEUDO-OPERATION

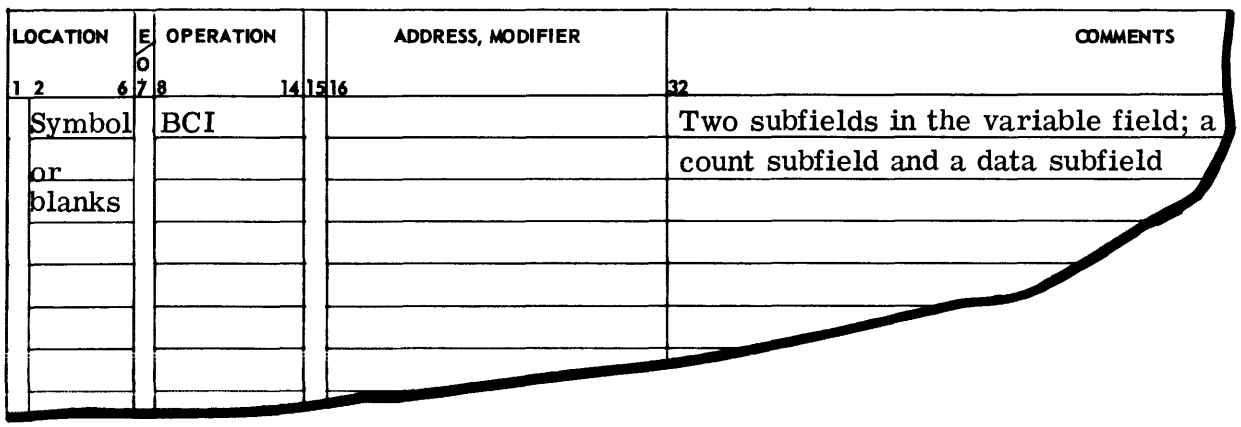
```
BETA   DEC    .3D0,0.D0,1.2D1B68,1D-1
```

The location counter is at the address BETA (1060); the above subfields generate the following double-words:

Location	Contents	Relocation	
001060	776463146314	000	BETA DEC .3D0,0.D0, 1.2D1B68,1D-1
001061	631463146314	000	

Location	Contents	Relocation
001062	400000000000	000
001063	000000000000	000
001064	000000000000	000
001065	000000000140	000
001066	772631463146	000
001067	314631463146	000

c. BCI (Binary Coded Decimal Information)



The BCI pseudo-operation is used by the programmer to enter Binary-Coded Decimal (BCD) character information into a program.

The first subfield is numeric and contains a count that determines the length of the data subfield. The count specifies the number of 6-character machine words to be generated; thus, if the count field contains n, then the data subfield contains 6n characters of data. The maximum value which n can be is 9. The minimum value for n is 0. If n is 0, no words will be generated.

The second subfield contains the BCD characters, six per machine word.

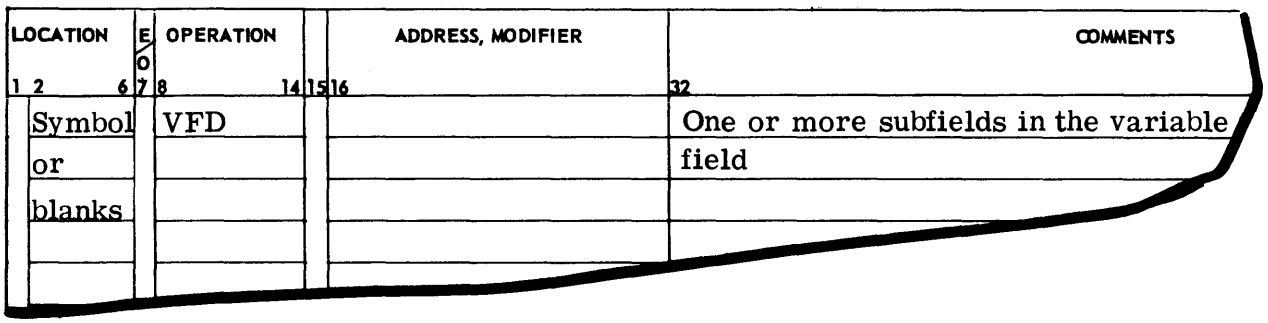
EXAMPLE OF BCI PSEUDO-OPERATION

```
BETA   BCI    3,NO ERROR CONDITION
```

Again assume the location counter set at 506 (location of BETA); the above would print out (less column headings):

Location	Contents	Relocation	
000506	454620255151	000	BETA BCI 3,NO ERROR CONDITION
000507	465120234645	000	
000510	243163314645	000	

d. VFD (Variable Field Definition)



LOCATION 1 2 6	E/O 7	OPERATION 8 14	15	ADDRESS, MODIFIER 16	COMMENTS 32
Symbol		VFD			One or more subfields in the variable
or					field
blanks					

The VFD pseudo-operation is used for generation of data where it is essential to define the data word in terms of individual bits. It is used to specify by bit count certain information to be packed into words.

In considering the definition of a subfield, it is understood that the unit of information is a single bit (in contrast with the unit of information in the BCI pseudo-operation which is six bits). Each VFD subfield is one of three types: an algebraic expression, a Boolean expression, or alpha-numeric. Each subfield contains a conversion type indicator and a bit count, the maximum value of which is 36. The bit count is an unsigned integer which defines the length of the subfield; it is separated from the data subfield by a slash (/). If the bit count is immediately preceded by on O or H, the variable-length data subfield is either Boolean or alphanumeric, respectively. In the absence of both the type indicators, O and H, the data subfield is an algebraic field. A Boolean subfield contains an expression that is evaluated using the Boolean operators (*, /, +, -).

The data subfield is evaluated according to its form: algebraic, Boolean, or alphanumeric. A 36-bit field results. The low-order n bits of the algebraic or Boolean expression determine the resultant field value; whereas for the alphanumeric subfield the high-order n bits are used.

If the required subfields cannot be contained on one card, they may be continued by the use of the ETC pseudo-operation. This is done by terminating the variable field of the VFD pseudo-operation with a comma. The next subfield is then given as the beginning expression in the variable field of an ETC card. If necessary, subsequent subfields may be continued onto

following ETC cards in the same manner. The scanning of the variable field is terminated upon encountering the first blank character.

The VFD may generate more than one machine word; if the sum of the bit counts is not a multiple of a discrete machine word, the last partial string of bits will be left-justified and the word completed with zeros.

EXAMPLES OF VFD PSEUDO-OPERATION

Assume one would like to have the address ALPHA packed in the first 18 bits of a word, octal 3 in the next 6 bits, the literal letter B in the next 6 bits, and an octal 77 in the last 6 bits. One could easily define it as follows:

```
VFD  18/ALPHA,6/3,H6/B,06/77
```

With the location counter at 1053 and the location 731_8 assigned for ALPHA, this would print out (without column headings):

Location	Contents	Relocation	
001053	000731032277	000	VFD 18/ALPHA,6/3,H6/B,06/77

NOTE: Relocation digits 000 refer to binary code data for A, BC, and DE of the relocation scheme.

If ALPHA had been a relocatable element, the relocation bits would have been 010; that is, the relocation scheme would have specified the left half of the word as containing a relocatable address. The relocation is only assigned if the programmer specifies a field width of 18 bits and has it left- or right-justified; in all other cases the fields are considered absolute. The total number of bits under a VFD need not be a multiple of full words nor is the total field (sum of all subfields) restricted to one word. The total field width, however, for a single subfield is 36 bits.

Consider a program situation where one wishes to generate a three-word identifier for a table. Assume n is the word length of the table and is equal to 12. You wish to place twice the length of the table in the first 12 bits, the name of the table in the next 60 bits, the location of the table (where TABLE is a relocatable symbol equal to 2351_8) in the next 18 bits, zero in the next 8 bits, and -1 in the next 6 bits--all in a three-word key.

With the location counter at 1054,

```
VFD   12/2*12,H36/PRESSU,H24/RE,18/TABLE,8/,6/-1
```

will generate

Location	Contents	Relocation		
001054	003047512562	000	VFD	12/2*12,H36/PRESSU, H24/RE,18/TABLE,8/ ,6/-1
001055	626451252020	000		
001056	002351001760	010		

where 010 specifies the relocatability of TABLE.

e. DUP (Duplicate Cards)

LOCATION 1 2 6	E/O 7	OPERATION 8 14	15	ADDRESS, MODIFIER 16	COMMENTS 32
Symbol		DUP			Two subfields in the variable field,
or					separated by a comma.
blanks					

The DUP pseudo-operation provides the programmer with an easy means of generating tables and/or data. It causes the Assembler to duplicate a sequence (range) of instructions or pseudo-operations a specified number of times.

The first subfield in the variable field is an absolute expression which defines the count. The value of the count field specifies the number of cards, following the DUP pseudo-operation, that are included in the group to be duplicated. The value in the count field must be a decimal integer less than or equal to ten.

The second subfield of the pseudo-operation is an absolute expression which specifies the number of iterations. The value in the iteration field specifies the number of times the group of cards, following the DUP pseudo-operation, is to be duplicated. This value can be any positive integer less than $2^{18}-1$. The groups of duplicated cards appear in the assembled listing immediately behind the original group.

If either the count field or the iteration field contains 0 (zero) or is null, the DUP pseudo-operation will be ignored.

If a symbol appears in the location field of the pseudo-operation it is given the address of the next location to be assigned by the Assembler.

If an odd/even address is specified for an instruction within the range of a DUP pseudo-operation, the instruction will be placed in odd/even address and a filler used when needed. The filler for a nondata-generating instruction will be an NOP instruction. No filler for a data-generating instruction is needed.

All symbols appearing in the variable field of the DUP pseudo-operation must have been previously defined. Any symbols appearing in the location field of cards in the range of DUP are defined only on the first iteration, thus avoiding multiply-defined symbols (the SET pseudo-operation is the only exception).

The only instructions or pseudo-operations which may not appear in the range of a DUP instruction are END, MACRO, and DUP. ETC may not appear as the first card after the range of a DUP.

6. Storage Allocation Pseudo-Operations

These pseudo-operations are used to reserve specified core memory storage areas within the coding sequence of a program for use as storage areas or work areas.

a. BSS (Block Started by Symbol)

LOCATION	E/O	OPERATION	ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15 16	32
Symbol		BSS		A permissable expression in the
or				variable field defines the amount
blanks				of storage to be reserved.

The BSS pseudo-operation is used by the programmer to reserve an area of memory within his assembled program for working and for data storage. The variable field contains an expression that specifies the number of locations the Assembler must reserve in the program.

If a symbol is entered in the location field, it is assigned the value of the first location in the block of reserved storage. If the expression in the variable field contains symbols, they must have been previously defined and must fall under the same location counter. No binary cards are generated by this pseudo-operation.

b. BFS (Block Followed by Symbol)

LOCATION	E/O	OPERATION	ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15 16	32
Symbol		BFS		A permissible expression in the
or				variable field defines the amount of
blanks				storage to be reserved.

The BFS pseudo-operation is identical to BSS with one exception. If a symbol appears in the location field, it is assigned the value of the first location after the block of reserved storage has been assigned; if the expression in the variable field contains symbols, they must have been previously defined and must fall under the same location counter.

c. BLOCK (Block Common)

LOCATION	E/O	OPERATION		ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15	16	32
Blanks		BLOCK			A symbol in the variable field

The purpose of the BLOCK pseudo-operation is to specify that program data following the BLOCK entry is to be assembled in the LABELED COMMON region of the user program under the symbol appearing in the variable field. BLOCK is, in effect, another location counter external to the text of the program.

A BLOCK pseudo-operation continues in effect until another BLOCK is encountered, or until a USE pseudo-operation appears (specifying return of control to the program located counter or another counter), or until the END pseudo-operation occurs.

The symbol in the variable field specifies the label of the COMMON area to be assembled. If the variable field is left blank, the normal FORTRAN BLANK COMMON is specified, and temporary storage will be reserved in the unlabeled (BLANK COMMON) memory area of the user program.

d. LIT (Literal Pool Origin)

LOCATION	E/O	OPERATION		ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15	16	32
Symbol		LIT			Column 16 must be blank
or					
blanks					

The LIT pseudo-operation causes the Assembler to punch and print out at assembly time all the previously developed literals. If the LIT instruction occurs in the middle of the program, the literals up to that point are output and printed out starting with the first available location after LIT; the literal pool is reinitialized as if the assembly had just begun.

If no LIT instruction is encountered by the Assembler, the origin of the literal pool will be one location past the final word defined by the program.

7. Conditional Pseudo-Operations

The pseudo-operations INE, IFE, IFL, and IFG to follow are especially useful within MACRO prototypes to gain additional flexibility in variable-length or conditional expansion of the MACRO prototype. Their use, however, is not limited to MACROS: they can be employed elsewhere in coding a subprogram to effect conditional assembly of segments of the program.

The programmer is responsible for avoiding noncomparable elements within these pseudo-operations. In addition, symbols used in the variable field will normally have been previously defined. On the other hand, one of the primary uses of conditionals is to test whether or not a symbol has been defined at a given point in an assembly. Consequently, undefined symbols within a conditional are not flagged in the left margin of the listing. However, if the symbol is never defined within the assembly, the symbol will be listed as undefined at the end of the listing; if the symbol is defined later in the assembly, it is not listed as undefined. Alpha-numeric literals as used with these pseudo-operations differ from those described under literals earlier in this section. The literal information used with the conditional pseudo-operations is right-justified with leading zeros.

a. INE (If Not Equal)

LOCATION	E/O	OPERATION		ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15	16	32
Blanks		INE			Two or three subfields in the variable field

The INE pseudo-operation provides for conditional assembly of the next n instructions, depending on the value of the first two subfields of the variable field.

The value of the expression in the first subfield is compared to the value of the expression in the second subfield. If they are not equivalent, the next n cards are assembled, where n is specified in the third subfield; otherwise, the next n cards are bypassed, resumption beginning at the (n+1)th card. If the third subfield is not present, n is assumed to be one.

Two types of comparisons are possible in the subfields of the INE pseudo-operation. The first is a straight numeric comparison after the expression has been evaluated. The second is alphanumeric comparison and the relation is the collating sequence. Alphanumeric literals in the variable field of INE are denoted by placing the subfield within apostrophe marks. If either the first or second subfield is designated as an alphanumeric literal, the other will automatically be classified as such.

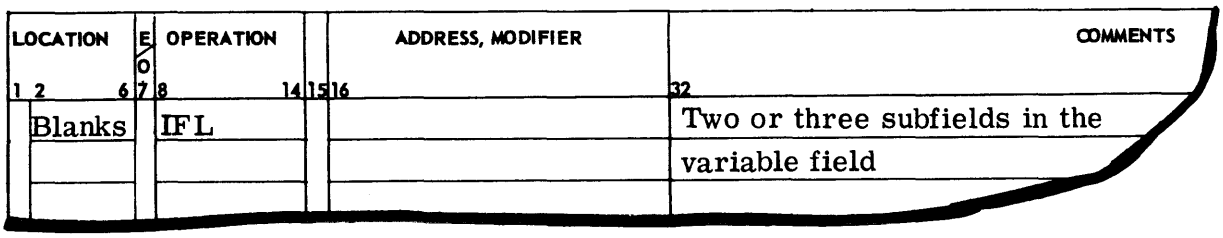
b. IFE (If Equal)

LOCATION	E/O	OPERATION		ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15	16	32
Blanks		IFE			Two or three subfields in the variable field

The IFE pseudo-operation provides for conditional assembly of the next n cards depending on the value of the first two subfields of the variable field. The next n cards are assembled if and only if the expression or alphanumeric literal in the first subfield is equal to the expression or alphanumeric literal in the second subfield. The n is specified in the third subfield and assumed to be one if not present. If the compared subfields are not equal, the next n cards are bypassed.

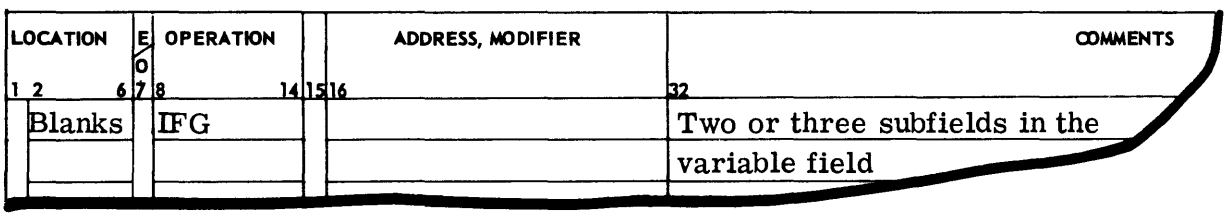
Alphanumeric literals in the variable field of IFE are denoted by placing the subfield within apostrophe marks. If either the first or second subfield is designated as an alphanumeric literal, the other will automatically be classified as such.

c. IFL (If Less Than)



The IFL pseudo-operation provides for conditional assembly of the next n cards depending on the value of the first two subfields of the variable field. The next n cards are assembled if the expression or alphanumeric literal in the first subfield is algebraically less than the expression or alphanumeric literal in the second subfield; otherwise, the next n cards are bypassed. The n is specified in the third subfield and assumed to be one if not present. Alphanumeric literals in the variable field of IFL are denoted by placing the subfield within apostrophe marks. If either the first or second subfield is designated as an alphanumeric literal, the other will automatically be classified as such.

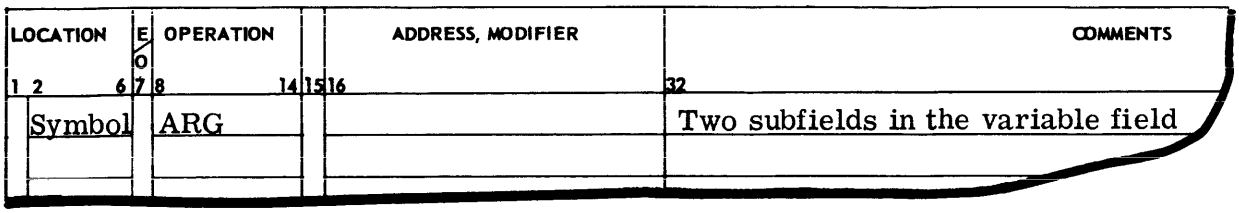
d. IFG (If Greater Than)



The IFG pseudo-operation provides for conditional assembly of the next n cards depending on the value of the first two subfields of the variable field. The next n cards are assembled if the expression or alphanumeric literal in the first subfield is algebraically greater than the expression or alphanumeric literal in the second subfield; otherwise, the next n cards are bypassed. The n is specified in the third subfield and assumed to be one if not present. Alphanumeric literals in the variable field of IFG are denoted by placing the subfield within apostrophe marks. If either the first or second subfield is designated as an alphanumeric literal, the other will automatically be classified as such.

8. Special Word Formats

a. ARG A, M (Argument--Generate Zero Operation Code Computer Word)



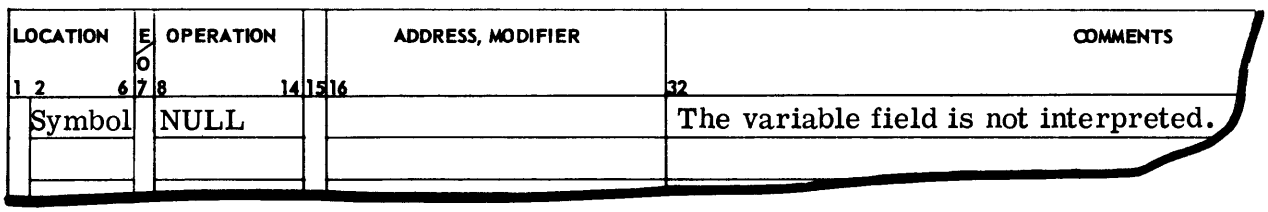
LOCATION	E/O	OPERATION	ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15 16	32
Symbol		ARG		Two subfields in the variable field

The use of ARG in the operation field causes the Assembler to generate a binary word with bit configuration in the general instruction format. The operation code 000 is placed in the operation field. The variable field is interpreted in the same manner as a standard machine instruction.

b. NONOP (Undefined Operation)

When an undefined operation is encountered, NONOP is looked up in the operation table and used in place of the undefined operation. NONOP is initially set as an error routine, but the programmer through the use of OPSYN or MACRO may redefine NONOP to his own purpose. For example, NONOP could be redefined by the use of a MACRO to be a MME to GECHEK with a dump sequence.

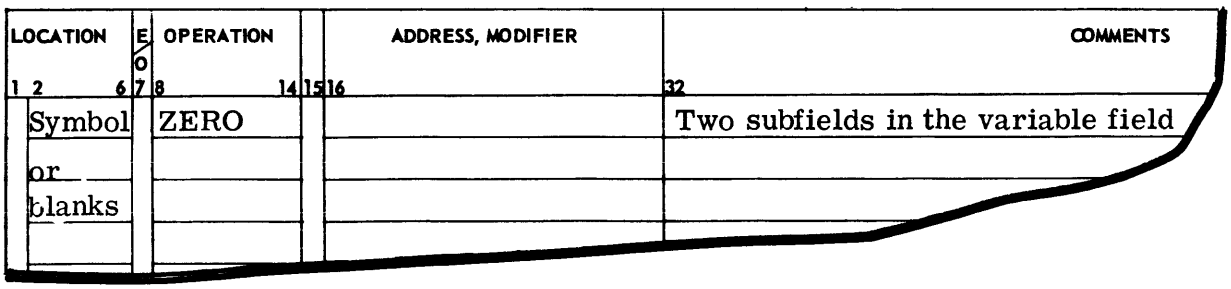
c. NULL (Null)



LOCATION	E/O	OPERATION	ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15 16	32
Symbol		NULL		The variable field is not interpreted.

The NULL pseudo-operation acts as an NOP machine instruction to the Assembler in that no actual words are assembled. A symbol on a NULL will be defined as current value of the location counter.

d. ZERO B, C (Generate One Word With Two Specified 18-bit Fields)



LOCATION	E/O	OPERATION	ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15 16	32
Symbol		ZERO		Two subfields in the variable field
or				
blanks				

The pseudo-operation ZERO is provided primarily for the definition of values to be stored in either or both the high- or low-order 18-bit halves of a word. The Assembler will generate

the binary word divided into the two 18-bit halves; bit positions 0-17 and 18-35. The equivalent binary value of the expression in the first subfield will be in bit positions 0-17. The equivalent binary value of the expression in the second subfield will be in bit positions 18-35.

e. MAXSZ (Maximum Size of Assembly)

LOCATION	E/O	OPERATION		ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15	16	32
Blanks		MAXSZ			A decimal number in the variable field

The decimal number represents the programmer's estimate of the largest number of assembled instructions and data in his program or subprogram. The variable field number is evaluated, saved, and printed out at the end of the assembly listing. It can then be compared with the actual size of the assembly.

MAXSZ is provided as a programmer convenience and can be inserted anywhere in his coding.

9. Address Tally Pseudo-Operations

The Indirect Then Tally (IT) type of address modification in several cases requires special word formats which are not instructions and do not follow the standard word format. The following pseudo-operations are for this purpose.

a. TALLY A, T, B, (Tally)

Used for ID, DI, and SC type of tally modification. A is the address, T is the tally count, and B is the character position. In ID and DI, the third subfield B is not specified. Character from indirect (CI) may be denoted with tally by allowing T to be zero. A six bit character is specified for the SC and CI modifications.

b. TALLYB A, T, B

Same as TALLY pseudo-operation except a nine-bit character is specified for the SC and CI modifications.

c. TALLYD A, T, D, (Tally and Delta)

Used for Add Delta (AD) and Sequence Delta (SD) modification. A is the address, T the tally, and D the delta of incrementing.

d. TALLYC A, T, mod (Tally and Continue)

Used for Address, Tally, and Continue. A is the address, T the tally count, and mod the address modification as specified under normal instructions.

10. Repeat Instruction Coding Formats

The machine instructions Repeat (RPT), Repeat Double (RPD), (macro operation), and Repeat Link (RPL) use special formats and have special tally, terminate repeat, and other conditions associated with them. The Assembler coding formats for the several RPT, RPD, and RPL options follow.

a. RPT N,I,k1,k2,.....,kj

The command generated by the Assembler from the above format will cause the instruction immediately following the command to be iterated N times and the increment value for each iteration set to I. The range for N is 0-255. If N=0, the instruction will be iterated 256 times. The fields k1, k2.....,kj may or may not be present. They are conditions for termination. These fields may contain the allowable codes of TOV, TNC, TRC, TMI, TPL, TZE, and TNZ.

It is also possible to use an octal number rather than the special symbols to denote termination conditions. Thus if field k1 is found to be numeric, it will be interpreted as octal; the low-order seven bits will be ORed into positions 11-17 of the instruction. The variable field scan will be terminated with the octal field.

b. RPTX ,I

This instruction behaves just as the RPT instruction with the exception that N and the conditions of termination will be found in index register zero instead of imbedded in the instruction.

c. RPD N,I,k1,k2,.....,kj

The command generated by the Assembler from the above format will cause the two instructions immediately following the RPD instruction to be iterated N times and the increment value for each iteration set to I. The increment I will apply to both instructions being repeated.

The variables k1,.....,kj are identical to those explained in the RPT instruction. Since the double repeat must fall in an odd location, the Assembler will force this condition and use an NOP instruction for a filler when needed.

d. RPDX ,I

This instruction behaves just as the RPD instruction with the exception that N and the conditions of termination will be found in index register zero instead of imbedded in the instruction.

e. RPDB N,I,k1,k2,.....,kj

This is the same as the RPD instruction except that only the address of the second instruction following the RPDB instruction will be incremented by I on each iteration.

f. RPDA N,I,k1,k2,.....,kj

This is the same as the RPD instruction except that only the address of the first instruction following the RPDA instruction will be incremented on each iteration by I.

g. RPL N, k1, k2, ..., kj

The instruction above will cause the instruction immediately following it to be repeated N times or until one of the conditions specified in k1, ..., kj are satisfied. The relation of k1, ..., kj is the same as in RPT. The address effectively used by the repeated instruction is the linked address. (See RPL instruction description.)

h. RPLX

This instruction behaves just as the RPL instruction except that N and conditions of termination will be found in index register zero instead of imbedded in the instruction.

11. Program Linkage Pseudo-Operations

The CALL, SAVE, RETURN and ERLK pseudo-operations are used in such a way that each generates many lines of coding in the assembly program from a single instruction input to the Assembler; they are therefore considered to be system MACROS.

a. CALL (Call-Subroutines)

LOCATION	E/O	OPERATION		ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15	16	32
Symbol		CALL			Subfields in the variable field with
or					contents and delimiters as
blanks					described below

The CALL pseudo-operation is used to generate the standard subroutine calling sequence.

The first subfield in the variable field of the instruction is separated from the next n subfields by a left parenthesis. This subfield contains the symbol which identifies the subroutine being called. It is possible to modify this symbol by separating the symbol and the modifier with a comma. (The symbol entered in this subfield is treated as if it were entered in the variable field of a SYMREF instruction.)

The next n subfields are separated from the first subfield by a left parenthesis and from subfield n+1 by a right parenthesis. Thus the next n subfields are contained in parentheses and

are separated from each other by commas. The contents of these subfields are arguments which will be used in the subroutine being called.

The next m subfields are separated from the previous subfields by a right parenthesis and from each other by commas. These subfields are used to define locations for error returns from the subroutine. If no error returns are needed, then m=0. In addition, if the programmer has placed all data under BLOCK pseudo-operations, the automatic generation of error linkage words is suppressed. The programmer must then supply his own error linkages. (See ERLK following.)

The last subfield is used to contain an identifier for the instruction. This identifier is used when a trace of the path of the program is made. The identifier must be a number contained in apostrophes. Thus the last subfield is separated from the previous subfields by an apostrophe. If the last subfield is omitted, the assembly program will provide an identifier.

In the examples following, the calling sequences generated by the pseudo-operation are listed below the CALL pseudo-operation. For clarification AAAAA defines the location the CALL instruction; SUB is the name of the subroutine called; MOD is an address modifier; A1 through An are arguments; E1 through Em define error returns; E. I. is an identifier; and E. L. defines a location where error linkage information is stored. E. L. is automatically defined by the Assembler after the END card is encountered unless previously defined by the ERLK pseudo-operation. The number sequences 1, 2, . . . , n and 1, 2, . . . , m designate argument positions only.

```
AAAAA    CALL    SUB, MOD(A1, A2, . . . . , An)E1, E2, . . . . . , Em' E. I. '

AAAAA    TSX1    SUB, MOD
         TRA     *+2+n+m
         ZERO    E. L. , E. I.
         ARG     A1
         ARG     A2
           .
           .
           .
         ARG     An
         TRA     Em
           .
           .
         TRA     E2
         TRA     E1
```

The preceding example of instructions generated by the CALL pseudo-operation was in the relocatable mode. The following example is in the absolute mode.

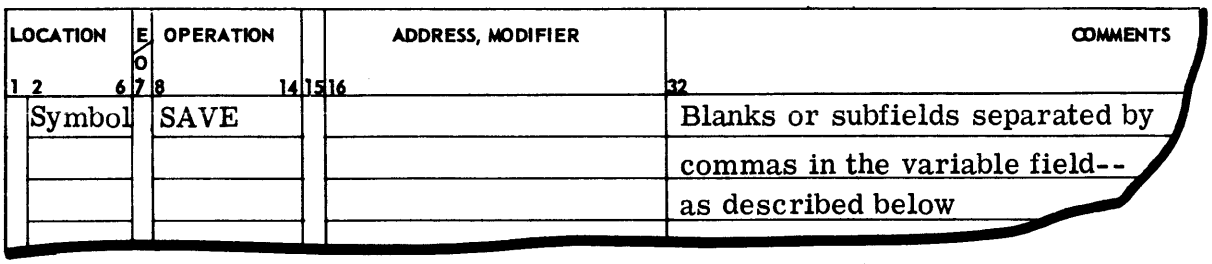
```
AAAAA    CALL    SUB, MOD(A1, A2, . . . . , An)E1, E2, . . . . . , Em' E. I. '

AAAAA    TSX1    SUB, MOD
         TRA     *+2+n+m
         ZERO    0, E. I.
         ARG     A1
         ARG     A2
           .
           .
           .
```

```
ARG     An
TRA     Em
 .
 .
 .
TRA     E2
TRA     E1
```

If the variable field of the CALL cannot be contained on a single line of the coding sheet, it may be continued onto succeeding lines by use of the ETC pseudo-operation. This is done by terminating the variable field of the CALL instruction with a comma (,). The next subfield is then placed as the first subfield of the ETC pseudo-operation. Subsequent subfields may be continued onto following lines in the same manner.

b. SAVE (Save--Return Linkage Data)



The SAVE speudo-operation is used to produce instructions necessary to save specified index registers and the contents of the error linkage index register.

The symbol in the location field of the SAVE instruction is used for referencing by the RETURN instruction. (This symbol is treated by the Assembler as if it had been coded in the variable field of a SYMDEF instruction when the Assembler is in the relocatable mode.)

The subfields in the variable field, if present, will each contain an integer 0-7. Thus, each subfield specifies one index register to be saved.

The instructions generated by the SAVE pseudo-operation are listed below. The argument symbols i_1 through i_n are integers 0-7. E.L. defines the location provided for the contents of the error linkage register. If the programmer has placed all program data under BLOCK pseudo-operations, automatic generation of error linkage words is suppressed.

BBBBB is a symbol that must be present; it is always a primary SYMDEF. Example one is in the relocatable mode, and example two is in the absolute mode.

EXAMPLE ONE

BBBBB	SAVE	$i_1, i_2, \ldots i_n$
	SYMDEF	BBBBB
BBBBB	TRA	*+2+n
	LDX(i_1)	**, DU
	.	
	.	
	.	
	LDX(i_n)	**, DU
	RET	E. L.
	STI	E. L.
	STX1	E. L.
	STX(i_1)	BBBBB+1
	STX(i_2)	BBBBB+2
	.	
	.	
	.	
	STX(i_n)	BBBBB+n

EXAMPLE TWO

BBBBB	SAVE	$i_1, i_2, \ldots, i_n$
BBBBB	TRA	*+3+n
	ZERO	
	LDX(i_1)	**, DU
	LDX(i_2)	**, DU
	.	
	.	
	.	
	.	
	LDX(i_n)	**, DU
	RET	BBBBB+1
	STI	BBBBB+1
	STX1	BBBBB+1
	STX(i_1)	BBBBB+2
	STX(i_2)	BBBBB+3
	.	
	.	
	.	
	STX(i_n)	BBBBB+n+1

c. RETURN (Return--From Subroutines)

LOCATION	E/O	OPERATION	ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15 16	32
Symbol		RETURN		One or two subfields in the
or				variable field
blanks				

The RETURN pseudo-operation is used for exit from a subroutine. The instructions generated by a RETURN pseudo-operation must make reference to a SAVE instruction within the same subroutine. This is done by the first subfield of RETURN. The first subfield in the variable field must always be present. This subfield must contain a symbol which is defined by its presence in the location field of a SAVE instruction.

The second subfield is optional and, if present, specifies the particular error return to be made; that is, if the second subfield contains the value k, then the return is made to the kth error return. If the programmer has placed all program data under BLOCK pseudo-operations, automatic generation of error linkage words is suppressed.

In the examples following, the assembled instructions generated by RETURN are listed below the RETURN instruction. For both examples the group of instructions on the left are generated when the Assembler is in the relocatable mode, and the instructions on the right when the Assembler is in the absolute mode.

EXAMPLE ONE

RETURN			BBBBB		
TRA	BBBBB+1	Generated Instruction	TRA	BBBBB+2	Generated Instruction

EXAMPLE TWO

RETURN			BBBBB, k		
LDX1	E. L., *	Generated Instructions	LDX1	BBBBB+1, *	Generated Instructions
SBX1	k, DU		SBX1	k, DU	
STX1	E. L.		STX1	BBBBB+1	
TRA	BBBBB+1		TRA	BBBBB+2	

d. ERLK (Error Linkage--between Subroutines)

LOCATION	E/O	OPERATION		ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15	16 32	
Blanks		ERLK			Column 16 must be blank

The normal operation of the Assembler is to assign a location for error linkage information, as shown in the examples of the CALL, SAVE, and RETURN pseudo-operations. However, if the programmer wishes to specify the location for error linkage information, he can do so by using ERLK. Thus, ERLK makes the location of the error linkage register known and available to the programmer. The appearance of ERLK causes the Assembler to generate two words of the following form:

```
E. L.    ZERO
         BCI      1, NAME
```

These words will be placed in the assembly at the point the Assembler encountered ERLK. Note that if the programmer has placed all program data under the BLOCK pseudo-operation, he must use ERLK since in this case automatic error linkage is suppressed. (See CALL, SAVE, and RETURN.)

In the example, the location symbol NAME must appear under the coded SYMDEF pseudo-operation (1) if ERLK is used within CALL, or (2) if not using CALL, the programmer generates his own subroutine calling sequence. If ERLK appears within the SAVE, SYMDEF need not be coded since SAVE automatically generates a SYMDEF.

NAME, as generated by the Assembler, is the first symbol defined under the first SYMDEF of the program containing ERLK.

D. MACRO OPERATIONS

1. Introduction

Programming applications frequently involve (1) the coding of a repeated pattern of instructions that within themselves contain variable entries at each iteration of the pattern and (2) basic coding patterns subject to conditional assembly at each occurrence. The macro operation gives the programmer a shorthand notation for handling (1) and (2) through the use of a special type of pseudo-operation referred to in the Macro Assembler as a MACRO. Having once determined the iterated pattern, the programmer can, within the MACRO, designate selectable fields of any instruction of the pattern as variable. Thereafter, by coding a single MACRO instruction, he can use the entire pattern as many times as needed, substituting different parameters for the selected subfields on each use.

When he defines the iterated pattern, the programmer gives it a name, and this name then becomes the operation code of the MACRO instruction by which he subsequently uses the macro operation.

As a generative operation, the macro operation causes n card images (where n is normally greater than one) to be generated; these may have substitutable arguments. The MACRO is known as the prototype or skeleton, and the card images that may be defined are relatively unrestricted as to type.

They can be:

- Any processor instruction
- Most Assembler pseudo-operations
- Any previously defined macro operation (such as the GE-635 instructions handled by software in certain models of the M-605).

Card images of these types are subject to the same conditions and restrictions when generated by the macro processor as though they had been produced directly by the programmer as in-line coding.

To use the MACRO prototype, once named, the programmer enters the macro operation code in the operation field and arguments in the variable field of the MACRO instruction. (The arguments comprise variable field subfields and refer directly to the argument pointers specified in the fields of the card images of the prototype.) By suitably selecting the arguments in relation to their use in the prototype, the programmer causes the Assembler to produce in-line coding variations of the n card images defined within the prototype.

The effect of a macro operation is the same as an open subroutine in that it produces in-line code to perform a predefined function. The in-line code is inserted in the normal flow of the program so that the generated instructions are executed in-line with the rest of the program each time the macro operation is used.

An important feature in specifying a prototype is the use of macro operations within a given prototype. The Assembler processes such "nested" macro operations at expansion time only. The nesting of one prototype within another prototype is not permitted. If macro operation codes are arguments, they must be used in the operation field for recognition. Thus, the MACRO must be defined before its appearance as an argument; that is, the prototype must be available to the Assembler before encountering a demand for its usage.

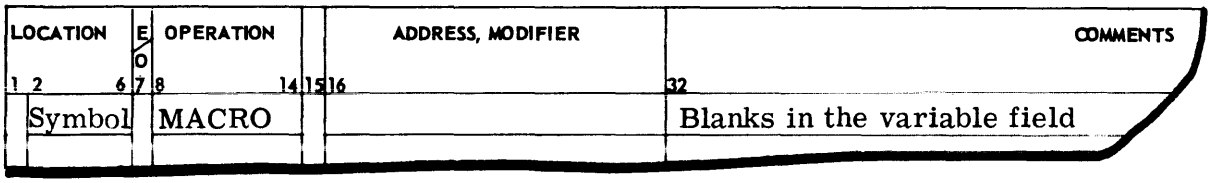
2. Definition of the Prototype

The definition of a MACRO prototype is made up of three parts:

- Creation of a heading card that assigns the prototype a name
- Generation of the prototype body of n card images with their substitutable arguments
- Creation of a prototype termination card

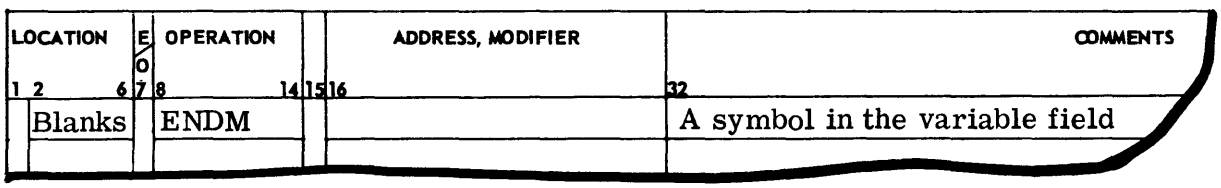
These parts are described in the following three subparagraphs.

a. MACRO (MACRO Identification) PSEUDO-OPERATION



The MACRO pseudo-operation is used to define a macro operation by symbolic name. The symbol in the location field can contain up to six allowable alphanumeric characters and defines the name of a MACRO whose prototype is given on the next n lines. (The prototype definition continues until the Assembler encounters the proper ENDM pseudo-operation.) The name of the MACRO is a required entry. If the symbol is identical to an operation code already in the table, then the macro operation will be used as a new definition for that operation code. It is entered in the Assembler operation table with a reference to its associated prototype that is entered in the MACRO skeleton table.

b. ENDM (End MACRO) PSEUDO-OPERATION



The symbol in the variable field is the symbolic name of the MACRO instruction as defined in the location field of the corresponding MACRO heading card. Every MACRO prototype must contain both the terminal ENDM pseudo-operation and the MACRO pseudo-operation.

Thus, every prototype will have the form

Heading card	OPNAME	MACRO	

Prototype body		-------	
		.	
		.	
		.	

Terminal card		ENDM	OPNAME

where OPNAME represents the prototype name that is placed in the Assembler operation table.

c. PROTOTYPE BODY

The prototype body contains a sequence of standard source-card images (of the types listed earlier) that otherwise would be repeated frequently in the source program. Thus, for example, if the iterated coding pattern

LOCATION	E/O	OPERATION	ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15 16	32
		:		
		LDA	5,DL	
		LDQ	13, DL	
		CWL	ALPHA, 2	
		TZE	FIRST	
		:		
		:		
		:		
		LDA	U	
		LDQ	V	
		CWL	BETA, 4	
		TZE	SCND	
		:		
		:		
		:		
		LDA	W+X	
		LDQ	Y+Z	
		CWL	GAMMA	
		TZE	NEXT 1	

appeared in a subprogram, it could be represented by the following prototype body (preceded by the required prototype name):

LOCATION	E/O	OPERATION	ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15 16	32
CMPAR		MACRO		MACRO prototype with substitutable
		LDA	#1	arguments in the variable field
		LDQ	#2	
		CWL	#3	
		TZE	#4	
		ENDM	CMPAR	

Then the previous coding examples could be represented by the macro operation CMPAR as follows:

```
CMPAR       (5, DL), (13, DL), (ALPHA, 2), FIRST
  ---
  ---
CMPAR       U, V, (BETA, 4), SCND
  ---
  ---
CMPAR       W+X, Y+Z, GAMMA, NEXT1
```

The Assembler recognizes substitutable arguments by the presence of the number-sign identifier (#). Having sensed this identifier, it examines the next one or two digits. (Sixty-three is the maximum number of arguments usable in a single prototype.)

MACRO prototype arguments can appear in the location field, in the operation field, in the variable field, and coincidentally in combinations of these fields within a single card image. Substitutions that can be made in these fields are:

- Location field--any permissible location symbol (see comments below)
- Operation field--all machine instruction, all pseudo-operations (except the MACRO pseudo-operation) and previously defined macro operations
- Variable field--any allowable expression followed by an admissible modifier tag and separated from the expression by a delimiting comma.

In general, anything appearing to the right of the first blank in the variable field will not be copied into the generated card image. For example, a substitutable argument appearing in the comments field of a card image--that is, separated from the variable field by one or more blanks--will not be interpreted by the Assembler (except in the case of the BCI, REM, TTL, and TTLS pseudo-operations). This means that only pertinent information in the location, operation, and variable fields is recognized, that internal blanks are not allowed in these fields, and that the first blank in these fields causes field termination.

When specifying a symbol in a location field of an instruction within a prototype the programmer must be aware that this MACRO can be used only once since on the second use the same symbol will be assigned a different location, causing a multiply-defined symbol. Consequently, the use of location symbols within the prototype is discouraged. Alternatively, for cases where repeated use of a prototype is necessary, two techniques are available: (1) use of Created Symbols and (2) placement of substitutable argument in the location field and use of a unique symbol in the argument of the macro operation each time the prototype is used. (These techniques are described under Using a Macro Operation, following below.)

The location field, operation field, and variable field may contain text and arguments which can be linked together (concatenated) by simply entering the substitutable argument (for example, AB#3) directly in the text with no blanks or special symbols preceding or following the entry. Concatenation is especially useful in the operation field and in the partial subfields of the variable field. (Refer to the discussion of BCI, REM, TTL, and TTLS immediately following.) As an example of the first use, consider a machine instruction such as LD(R) where R can assume the designators A, Q, AQ, and X0-X7.

The prototype NAME

```
NAME        MACRO
            -------
            -------
            LD#2
            -------     A,#1
            -------
```

contains a partial operation field argument; and when the in-line coding is generated, LD#2 becomes LDA, LDQ, etc., as designated by the argument used in the macro operation.

The BCI, REM, TTL, and TTLS pseudo-operations used within the prototype are scanned in full for substitutable arguments. The variable field of these pseudo-operations can contain blanks and argument pointers. The following illustrates a typical use:

```
ALPHA       MACRO
            -------
            -------
            -------
NOTE#1      REM         IGNORE␣#2␣ERRORS␣ON␣#3
            -------
            -------
```

(Note: b = blank)

An asterisk (*) type comment card cannot appear in a MACRO prototype.

3. Using a Macro Operation

Use of a Macro operation can be divided into two basic parts; definition of the prototype and writing the Macro operation. The first part has been described on the preceding pages; writing the Macro operation to call upon the prototype is the process of using the Macro and is described in the following paragraphs.

The Macro operation card is made up of two basic fields; the operation field that contains the name of the prototype being referenced and the variable field that contains subfield arguments relating to the argument pointers of the prototype on a sequential, one-to-one basis. For example, the defined prototype CMPAR, mentioned earlier, could be called for expansion by the MACRO instruction

```
CMPAR        U, V, (BETA, 4), SCND
```

where the variable field arguments, separated by commas and taken left-to-right, correspond with the prototype pointers #1 through #4. These arguments are then substituted in their corresponding positions of the prototype to produce a sequence of instructions using these arguments in the assigned location, operation, and variable fields of the prototype body. (The above MACRO instruction expands to the coding shown on page IV-67.)

The maximum number of MACRO-call arguments is 63; arguments greater than 63 are treated modulo 64. For example, the 70th argument is the same as the 6th argument and would be so recognized by the Assembler. Each such argument can be a literal, a symbol, or an expression (delimited by commas) that conforms to the restrictions imposed upon the field of the machine instruction or pseudo-operation within the prototype where the argument will be inserted.

The following conditions and restrictions apply to the expansion of MACROS:

- Anything appearing in the location field of a prototype card image, whether text or a substitutable argument, causes generation to begin in column 1 for that text or argument.
- Location field text generated from an argument pointer (in a prototype location field) so as to produce a resultant field extending beyond column 8 causes the operation field to begin in the next position after the generated text. Normally, the operation field will begin in column 8.
- Operation field text generated from an argument pointer (in a prototype operation field) so as to produce a resultant field extending beyond column 16 causes the variable field to start in the next position after the generated text. Normally, the variable field will begin in column 16.
- The variable field may begin after the first blank that terminates the operation field but not later than column 16 in the absence of the condition in 3 above.
- No generated card image can have more than 72 characters recorded; that is, the capacity of one card image cannot be exceeded (columns 73-80 are not part of the card image).
- No argument string of alphanumeric characters can exceed 57 characters.
- Up to 63 levels of MACRO nesting are permitted.

An argument can also be declared null by the programmer when writing the MACRO instruction; however, it must be declared explicitly null. Explicitly null arguments of the MACRO instruction argument list can be specified in either of two ways; by writing the delimiting commas in succession with no spaces between the delimiters or by terminating the argument list with a comma with the next normal argument of the list omitted. (Refer to the CRSM description, following.) A null argument means that no characters will be inserted in the generated card image wherever the argument is referenced. When a macro operation argument relates to an

argument pointer and the pointer requires the argument to have multiple entries or contains blanks, the corresponding argument must be enclosed within parentheses with the parenthetical argument set off by the normal comma delimiters. The parenthetical argument can contain commas as separators. Examples of prototype card images that require the use of parentheses in the MACRO call are pseudo-operations such as IDRP, VFD, BCI, and REM, as well as the variable field of an instruction where the address and tag may be one argument.

It is also possible to enclose an argument within brackets, making them subarguments, in which case blanks are ignored as part of the argument. For example the MACRO call of the MACRO named ABC can be written as

```
ABC     [A,
ETC     24,
ETC     2*D]
```

and is equivalent to

```
ABC     (A, 24, 2*D)
```

even though numerous blanks occur after the arguments A, and 24,. Thus, the Assembler packs everything it finds within brackets and suppresses all blanks therein. The above manner of writing the MACRO call permits the programmer additional flexibility in placing one sub-argument per card by means of using ETC, the blanks no longer being significant.

It can happen that the argument list of a macro operation extends beyond the capacity of one card. In this case, the ETC pseudo-operation is used to extend the list on to the next card. In using ETC, the last argument entry of the macro operation is delimited by a following comma, and the first entry of the ETC card is the next argument in the list. Within the prototype, as many ETC cards as required can be used for internal MACROS or VFD pseudo-operations.

4. Pseudo-Operations Used Within Prototypes

a. NEED FOR PROTOTYPE CREATED SYMBOLS

In case of a MACRO prototype in which an argument pointer is used in the location field, the programmer must specify a new symbol each time the prototype is called. In addition, for those cases where a nonsubstitutable symbol is used in a prototype location field, the programmer can use the macro operation only once without incurring an Assembler error flag on the second and all subsequent calls to the prototype (multiply-defined symbol). Primarily to avoid the former task (having to repeatedly define new symbols on using the macro operation) and to enable repeated use of a prototype with a location field symbol (nonsubstitutable), the created symbol concept is provided.

b. USE OF CREATED SYMBOLS

Created symbols are of the type .xxx. where xxx runs from 001 through 999, thus making possible up to 999 created symbols for an assembly. The periods are part of the symbol. The Assembler will generate a created symbol only if an argument in the macro operation is implicitly null; that is, only if the macro operation defines fewer arguments than given in the related MACRO prototype or if the designator # is used as an argument. Explicitly null arguments will not cause created symbols to be generated. The example given clarifies these ideas.

Assume a MACRO prototype of the form

```
NAME        MACRO
            -------     #1, #2
  #4        -------     X
  #5        -------     ALPHA, #3
            -------     #4
            TMI         #5
            ENDM        NAME
```

with five arguments, 1 through 5. The macro operation NAME in the form

```
            NAME        A, 7, , , B
```

specifies the third and fourth arguments as explicitly null; consequently, no created symbols would be provided. The expansion of the operation would be

```
            -------     A, 7
            -------     X
  B         -------     ALPHA,
            -------
            TMI         B
```

The macro operation card

```
            NAME        A, 7,
```

indicates the third argument is explicitly null, while arguments four and five are implicitly null. Consequently, created symbols would be provided for arguments four and five but not for three. This is shown in the expansion of the macro operation as follows:

```
            -------     A, 7
 .011.      -------     X
 .012.      -------     ALPHA,
            -------     .011.
            TMI         .012.
```

A created symbol could be requested for argument three simply by omitting the last comma. The programmer can conveniently change an explicitly null argument to an implicitly null one by inserting the # designator in an explicitly null position. Thus, for the preceding example

```
            NAME        A, 7, , #, B
```

the fourth argument becomes implicitly null and a created symbol will be generated.

c. CRSM ON/OFF (Created Symbols)

LOCATION	E/O	OPERATION	ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15 16	32
Blanks		CRSM	ON	Normal mode
Blanks		CRSM	OFF	
Blanks		CRSM	SAVE, ON	
Blanks		CRSM	SAVE, OFF	
Blanks		CRSM	RESTORE	

Created symbols are generated only within MACRO prototypes. They can be generated for argument pointers in the location, operation, and variable fields of instructions or pseudo-operations that use symbols. Accordingly, the created symbols pseudo-operation affects only such coding as is produced by the expansion of MACROS. CRSM ON causes the Assembler to initiate or resume the creation of symbols: CRSM OFF terminates the symbol creation if CRSM ON was previously in effect. The SAVE option in the variable field causes the present mode of the CRSM pseudo-operation to be saved and then the mode specified by the second term in the variable field is set. The RESTORE option causes the saved status to be reset as the mode of CRSM.

d. ORGCSM (Origin Created Symbols)

LOCATION	E/O	OPERATION	ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15 16	32
Blanks		ORGCSM		One expression in the variable field

The variable field is evaluated and becomes the new starting value between the decimal points of the created symbols.

e. IDRP (Indefinite Repeat)

LOCATION	E/O	OPERATION	ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15 16	32
Blanks		IDRP	# 3	An argument number or blanks in the
				variable field, depending on the IDRP
				of the IDRP pair

The purpose of the IDRP is to provide an iteration capability within the range of the MACRO prototype by letting the number of grouped variables in an argument pointer determine the iteration count.

The IDRP pseudo-operation must occur in pairs, thus delimiting the range of the iteration within the MACRO prototype. The variable field of the first IDRP must contain the argument number that points to the particular argument used to determine the iteration count and the variables to be affected. The variable field of the second IDRP must be blank.

At expansion time, the programmer denotes the grouping of the variables (subarguments) of the iteration by placing them, contained in parentheses, as the nth argument where n was the argument value contained in the initial IDRP variable field entry.

IDRP is limited to use within the MACRO prototype, and nesting is not permitted. However, as many disjoint IDRP pairs may occur in one MACRO as the programmer wishes.

For example, given the MACRO skeleton

```
NAME        MACRO
              .
              .
              .
            IDRP        #2
            ADA         #2
            IDRP
              .
              .
            ENDM        NAME
```

the MACRO call (with variables X1, X2, and X3)

```
A           NAME   Q+2, (X1, X2, X3), B
```

would generate

```
A             .
              .
              .
            ADA         X1
            ADA         X2
            ADA         X3
              .
              .
```

In the example, arguments #1 and #3, Q+2, and B respectively, are used in the skeleton ahead of and after the appearance of the IDRP, range-iteration pair.

f. DELM (Delete MACRO)

LOCATION	E/O	OPERATION		ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15	16	32
Symbol		DELM			A symbol in the variable field
or					
Blanks					

The function of this pseudo-operation is to delete the MACRO named in the variable field from the MACRO prototype area, and disable its corresponding operation table entry. Through the use of this pseudo-operation, systems which require many, or large MACRO prototypes, or which have minimal storage allocation at assembly time, can re-use storage in the prototype area for redefining or defining new MACROSs. Redefinition of a deleted MACRO will not produce an M multiply defined flag on the assembly listing.

g. PUNM (Punch MACRO Prototypes and Controls)

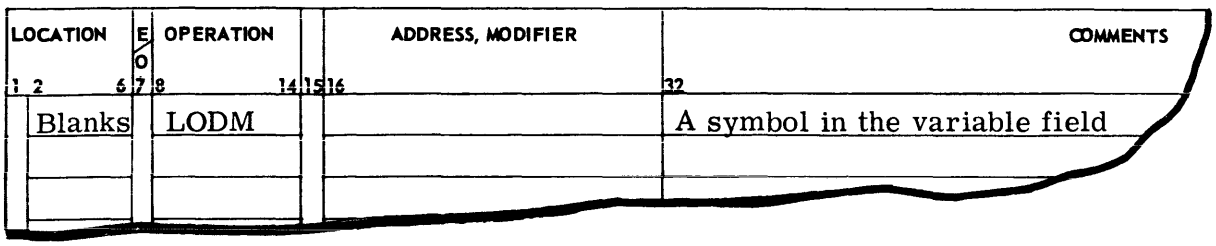
LOCATION	E/O	OPERATION		ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15	16	32
Blanks		PUNM			The variable field is not examined

This pseudo-operation causes the Assembler, in pass one, to scan the operation table for all MACROs defined. It then appends their definitions to the end of the prototype table and constructs a control word specifying the length of this area and the number of MACROs defined therein.

At the beginning of pass two, this information is punched onto relocatable binary instruction cards, along with $ OBJECT, preface, and $ DKEND cards. The primary SYMDEF of this deck will arbitrarily be . MACR. .

In the normal preparation of System MACROs, it would not be desirable to include the GMAP System MACROs. For this reason, the assembly of a set of System MACROs should have NGMAC elected on its $ GMAP card.

h. LODM (Load System MACROs)



LOCATION	E/O	OPERATION	ADDRESS, MODIFIER	COMMENTS
1 2 6	7	8 14	15 16	32
Blanks		LODM		A symbol in the variable field

This pseudo-operation causes the Assembler to issue an MME GECALL for a set of System MACROs. The name used in the GECALL sequence is the symbol taken from the variable field of the LODM pseudo-operation. MACROs thus loaded will be appended to (not overlay) the MACRO prototype table. They will be defined and made available for immediate use. If a MACRO is redefined by this operation the LODM instruction will be flagged with an M.

5. Notes and Examples On Defining A Prototype

The examples following show some of the ways in which MACROS can be used.

a. FIELD SUBSTITUTION

Prototype definition:

```
        ADDTO       MACRO
                    LDA         #1
                    ADA         #2
                    STA         #3
                    ENDM        ADDTO
```

Use:

```
                    ADDTO       A, (1, DL), B+5
```

b. CONCATENATION OF TEXT AND ARGUMENTS

Prototype definition:

```
        INCX        MACRO
                    ADLX#2      #3, DU
                    INE         #1, '*+1'
                    TRA         #1
                    ENDM        INCX
```

Use:

```
                    INCX        LOCA, 4, 1
```

or

```
                    INCX        *+1, 4, 1
```

c. ARGUMENT IN A BCI PSEUDO-OPERATION

Prototype definition:

```
        ERROR       MACRO
                    TSX1        DIAG
                    ARG         #1
                    BCI         5, ERROR␣#1␣CONDITION␣IGNORED
                    ENDM        ERROR
```

Use:

```
                    ERROR       5
```

d. MACRO OPERATION IN A PROTOTYPE

Prototype definition:

```
        TEST        MACRO
                    LDA         #1
                    CMPA        #2
                    #3          #4
                    ERROR       #5
                    ENDM        TEST
```

Use:

```
                    TEST        A, B, TZE, ALPHA, 3
```

e. INDEFINITE REPEAT

Prototype definition (for generating a symbol table):

```
        SYMGEN      MACRO
                    IDRP        #1
        #1          BCI         1,#1
                    IDRP
                    ENDM        SYMGEN
Use:
                    SYMGEN      (LABEL, TEST, ERROR, MACRO)
```

f. SUBROUTINE CALL MACRO

Prototype definition:

```
        DOO         MACRO
        K           SET         0
                    IDRP        #2
        K           SET         K+1
                    IDRP
                    TSX1        #1
                    TRA         *+1+K
                    IDRP        #2
                    ARG         #2
                    IDRP
                    ENDM        DOO
Use:
                    DOO         SRT, (ARG1, ARG2, ARG3)
```

6. System (Built-In) MACROS and Symbols

GMAP has been implemented with the facility for loading a unique set (or sets) of MACROs, under control of a pseudo-operation. This permits the various language processors to uniquely identify those standard system MACROs that are required for the assembly of their generated code.

System MACROs are located on the system file in mass storage. They are put there by the System Editor, in System Loadable Format, as a free-standing system program. Their catalog name is that which is to be used by GMAP in the loading operation. For proper implementation, the MASTER option of the System Editor parameters card must be elected. It may be in absolute or relocatable System Loadable Format.

This implementation technique permits any unit, or functionally related group of users of GMAP to define and implement a unique set of System MACROs; or on a larger scale, it allows various M-605 installations to install local standard sets of MACROs,without changing the Assembler.

E. SOURCE PROGRAM INPUT

The input job stream managed by the Comprehensive Operating Supervisor (GECOS, GEFLOW module) can comprise assembled object programs, Macro Assembler language source programs, and FORTRAN compiler-language source programs. Such programs of a job are referred to as activities or as subprograms. A source program input to the Assembler written in the M-605 machine language is an Assembler language input subprogram. Comments to follow in this section pertain to this subprogram, as opposed to the others noted above.

The Assembler language subprogram is composed of the following parts, in order:

- $ GMAP control card (calls the Assembler into Memory from external storage and provides Assembler output options; refer to the paragraph following)
- Text of the subprogram (one instruction per card)
- END pseudo-operation card (terminates the input subprogram)

The $ GMAP control card is prepared as shown below:

CARD COLUMN	1	8	16
SYMBOLIC EXAMPLE	$	GMAP	OPTION 1, OPTION 2,...
ACTUAL EXAMPLE	$	GMAP	NDECK, LSTOU

The operand field specifies the system options listed in any random order. When an option, or its converse, does not appear in the operand field, there is a standard entry which is assumed. (The standard entries are asterisked below.)

The options available with GMAP are as follows:

- LSTOU--A listing of the output will be prepared.
- NLSTOU--No listing of the output will be prepared.
- DECK--A program deck will be prepared as part of the output of this processor.
- NDECK--No program deck will be prepared.

The content of columns 73-80 is used as an identifier to uniquely identify the binary object programs resulting from the assembly.

F. RELOCATABLE AND ABSOLUTE ASSEMBLIES

The normal operating mode of the Assembler in processing input subprograms is relocatable; that is, each subprogram in a job stream is handled individually and is assigned memory locations nominally beginning with zero and extending to the upper limit required for that subprogram. Since a job stream can contain many such subprograms, it is apparent that they cannot all be loaded into a memory area starting with location zero; they must be loaded into different memory areas. Furthermore, they must be movable (relocatable) among the areas. Then for relocatable subprograms, the Assembler must provide (1) delimiters identifying each subprogram, (2) information specifying that the subprogram is relocatable, (3) the length of the subprogram, and (4) relocation control bits for both the upper and lower 18 bits of each assembled word.

Subprogram delimiters are the Assembler output cards $ OBJECT, heading the subprogram assembly, and $ DKEND, ending the assembly. An assembly is designated as relocatable on a card-to-card basis by a unique 3-bit Assembler punched code value in each binary output card. (See descriptions of Binary Punched Cards, page IV-78 and following.) The subprogram length is punched in the preface card(s) which immediately follows the $ OBJECT card of each subprogram. The relocation control bits are grouped together on the binary card and are referenced by GELOAD/605 while it is loading the subprogram into absolute memory locations.

The Assembler designates that the assembly output is absolute on a card-to-card basis by punching a unique 3-bit code value in each card. This value causes GELOAD/605 to regard all addresses on a card as actual (physical) memory addresses and to load accordingly. Each absolute subprogram assembly begins with a $ OBJECT card and terminates with the $ DKEND card, as in the case of relocatable assemblies.

The normal Assembler operating mode is relocatable; it is set to the absolute mode by programmer use of the ABS pseudo-operation.

G. ASSEMBLY OUTPUTS

1. Binary Decks

When the $ GMAP control card specifies the DECK option, the Assembler punches a binary assembly output deck. Since the normal mode of the Assembler is relocatable, all addresses punched in the output cards are normally relative to the blank location counter (relative to zero) and the text is described as relocatable. Alternatively, still considering the DECK option, the Assembler can operate in the absolute mode and punch only absolute addresses in the output cards.

Relocatable or absolute addresses can be punched in four types of binary cards. These cards and their uses are summarized below. The user subprogram memory map blocks are (1) the subprogram region, (2) the LABELED COMMON region, and (3) the BLANK COMMON region.

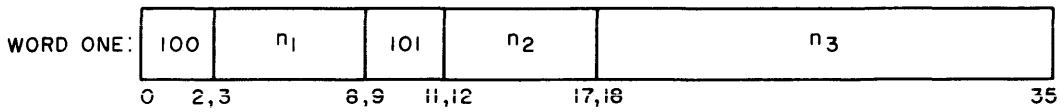
CARD TYPE	USE
Preface	Provides the Loader with (1) the length of the subprogram text region; (2) the length of the BLANK COMMON region; (3) the total number of SYMDEF, SYMREF, and LABELED COMMON symbols; (4) the type identification of each symbol in (3); and (5) the relative entry value or the region length for each symbol in (3).
Relocatable	Supplies the Loader with relocatable binary text by using preface card information and relocation identifiers, where the relocation identifiers specify whether the 18-bit field refers to a subprogram, LABELED COMMON, or BLANK COMMON regions (of the assembly core-storage area) and will allow the loader to relocate these fields by an appropriate value.
Absolute	Provides the Loader with absolute binary text and the absolute starting-location value for Loader use in assigning core-storage addresses to all words on the card.
Transfer	Can be generated only in an absolute assembly and causes the Loader to transfer control to the routine at the location given on the card. (The transfer card is generated automatically as the last card of an absolute subprogram assembly by the END pseudo-operation; however, use of the TCD pseudo-operation can cause the card to appear anywhere in the assembly.)

The formats in which the Assembler punches the above cards are described in the paragraphs to follow.

2. Preface Card Format

Preface card symbolic entries are primary SYMDEF symbols, secondary SYMDEF symbols, SYMREF symbols, LABELED COMMON symbols (from the BLOCK pseudo-operation), and the .SYMT. LABELED COMMON symbol. These symbols appear on the card in a precise order. All SYMDEF symbols appear before any other symbol. Following the SYMDEF symbols are any LABELED COMMON symbols that may have relocatable binary data loaded into that region. The SYMREF symbols are then recorded followed by the remaining LABELED COMMON symbols.

The format and content of the preface card are summarized as follows:

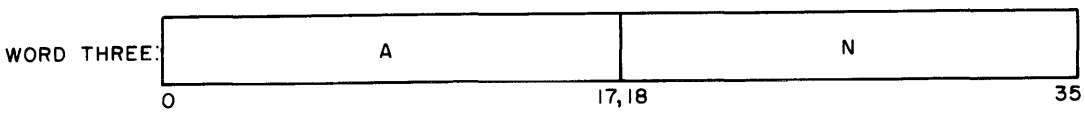


n_1--V is a value within the range $5 \leq V \leq 35$ and represents the size of the field within a special relocation entry needed to point the specific preface card entry. Thus, $V = \log_2 N + 1$, where N is the number of LABELED COMMON and SYMREF entries.

n_2--Word count of the preface card text

n_3--Length of the subprogram

Word Two: Checksum of columns 1-3 and 7-72



The value A is the length of BLANK COMMON; and N is two times the total number of SYMDEFs, SYMREFs, and LABELED COMMONs.

Words Four, Five: $Symbol_1$: A_1, K_1

Words Six, Seven: $Symbol_2$: A_2, K_2

The even-numbered word contains the symbol in BCD. The value K defines the type symbol in the even-numbered word; A is a value associated with K, as explained in the following list.

If K equals zero, then the symbol is a primary SYMDEF symbol; A is the entry value relative to the subprogram region origin.

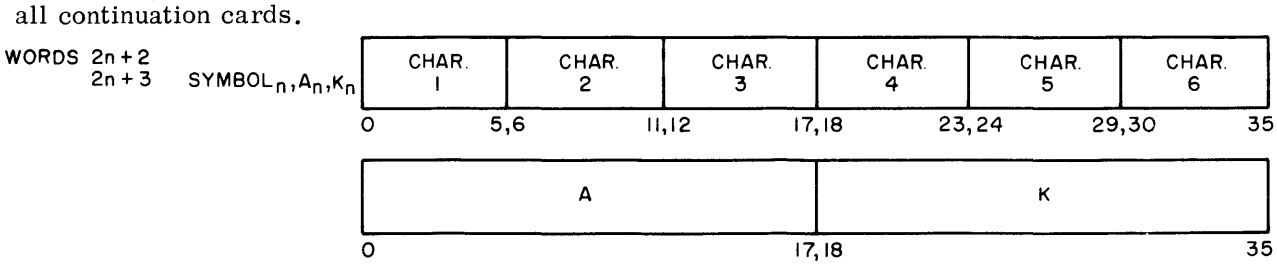
If K equals one, then the symbol is a secondary SYMDEF symbol; A is the entry value relative to the subprogram region origin.

If K equals five, then the symbol is a SYMREF symbol; A is zero.

If K equals six, then the symbol is a LABELED COMMON symbol; A is the length of the region.

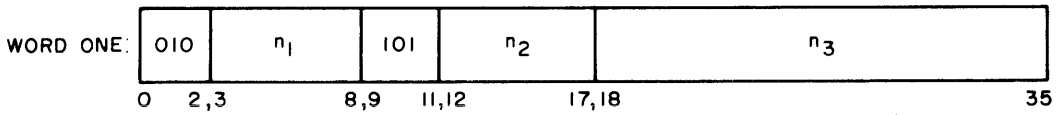
If K equals seven, then the symbol is a .SYMT. LABELED COMMON symbol; A is the length of the region reserved for debug information.

NOTE: If preface continuation cards are necessary, word three will be repeated unchanged on all continuation cards.



3. Relocatable Card Format

A relocatable assembly card has the format and contents summarized in the following comments.



n_1--0 indicates that loading is within the subprogram region of the user subprogram core-storage area

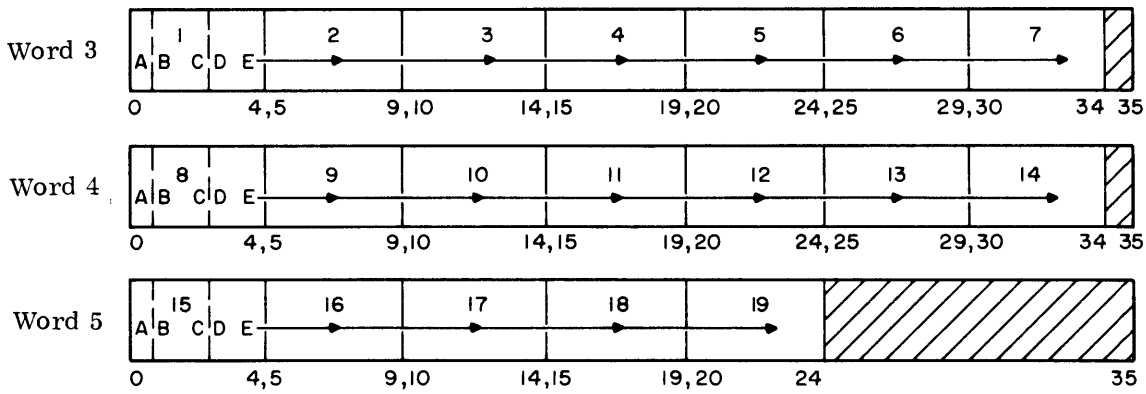
n_2--Word count of the data words to be loaded using the origin and relative address in this control word

n_3--Loading address, relative to the subprogram region origin.

or for the alternative cases:

n_1--i, where $i \neq 0$ indicates that the ith entry (beginning with the first LABELED COMMON or SYMREF entry in the preface card text has been used and that n_3 is relative to the origin of that entry.

Word Two: Checksum of columns 1-3 and 7-72



Relocation data--words three and four comprise seven 5-bit relocation identifiers, while word five holds 5 such identifiers. The five bits of each identifier carry relocation scheme data for each of the card words (7 + 7 + 5 = 19, or fewer). The identifiers are placed in bit positions 0-34 of words three and four and in 0-24 of word five. (Refer to the Relocation Scheme description in the paragraph following.)

Words Six-Twenty-Four: Instructions and data (up to 19 words per card). If the card is not complete and at least two words are left vacant, then after the last word entered, word one may be repeated with a new word count and loading address. The loading is then continued with the new address, and the relocation bits are continuously retrieved from words three through five. This process may be repeated as often as necessary to fill a card.

4. Relocation Scheme

For each binary text word in a relocatable card, the five bits--A, BC, and DE--of each relocation scheme identifier are interpreted by the Loader as follows:

Bit A--0 (reserved for future use)

Bits BC--Left half-word

Bits DE--Right half-word

To every 18-bit half-word one of four code values apply; these are:

CODE VALUE	
XX = 00	Absolute value that is not be to modified by the Loader.
= 01	Relocatable value that is to be added to the origin of the subprogram region by the Loader.
= 10	BLANK COMMON, relative value that is to be added to the origin of the BLANK COMMON region by the Loader.
= 11	Special entry value (to be interpreted as described in the next paragraph)

apply where XX stands for BC or DE.

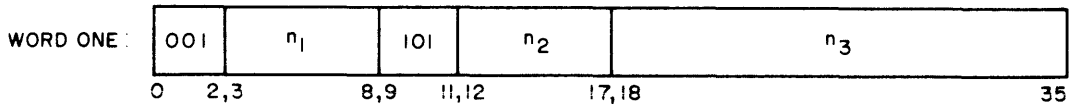
If special entry is required, the Loader decodes and processes the text and bits of the 18-bit field (left/right half of each relocatable card word) as follows:

Bit 1	--This is the sign of the addend; 0 implies a plus (+) and 1 implies a minus (-).
Bits 2→V + 1	--The value V that was specified in word 1 of the preface card dictates the length of the field. The contents of the field is a relative number which points to a LABELED COMMON region or a SYMREF that appeared in the preface card. The value one in this field would point to the first symbol entry after the last SYMDEF.
Bits V + 2→18	--The value in this field is the addend value that appeared in the expression. If the field is all bits then the corresponding 18 bits of the next data word are interpreted as the addend.

All references to each undefined symbol are chained together. When the symbol is defined, the Loader can rapidly insert the proper value of the symbol in all relocatable fields that were specified in the chain.

5. Absolute Card Format

The absolute binary text card appears as shown below.



n_1--0

n_2--Word count of the card text

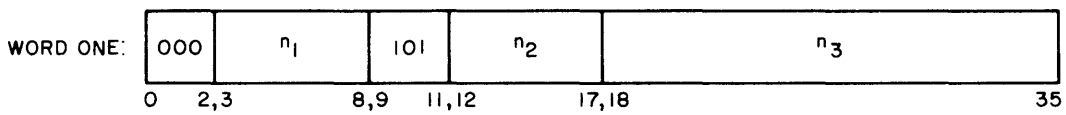
n_3--Loading address relative to the absolute core-storage origin zero

Word Two: Checksum of columns 1-3 and 7-72

Words Three-Twenty-Four: Instructions and text (22 words per card, maximum). If the card is not complete and at least two words are left vacant, then after the last word entered word one may be repeated with a new word count and loading address.

6. Transfer Card Format

The transfer card is generated by the Assembler only in an absolute assembly deck. Its format and contents are:



n_1--0

n_2--0

n_3--Transfer address (in absolute only).

Words Two-Twenty-Four: Not used

7. Assembly Listings

Each Assembler subprogram listing is made up of the following parts:

- The sequence of instructions in order of input to the Assembler
- The contents of all preface cards (primary SYMDEF symbols, secondary SYMDEF symbols, SYMREF symbols, LABELED COMMON symbols (from the BLOCK pseudo-operation), and the . SYMT . LABELED COMMON symbol)
- The symbolic reference table

a. FULL LISTING FORMAT

Each instruction word produced by the Assembler is individually printed on a 120-character line. The line contains the following items for each such word of all symbolic cards:

1. Error flags--one character for each error type (see Error Codes below).
2. Octal location of the assembled word
3. Octal representation of the assembled word
4. Relocation bits for the assembled word (see the topic, Relocation Scheme, Loader manual)
5. Reproduction of the symbolic card, including the comments and identification fields, exactly as coded

The exact format of the full listing is shown below.

FIELDS	A	B	C	D	E	F	G	H
PRINT LINE COLUMNS	1-6	7-12	15-20	22-25	27,28	31-33	35-39	41-120
			MACHINE INSTRUCTION					SOURCE CARD IMAGE

A--ERROR FLAGS
B--RELATIVE/ABSOLUTE LOCATION
C--OPERAND ADDRESS
D--OPERATION CODE
E--TAG FIELD MODIFIER
F--RELOCATION BITS
G--ALTER STATEMENT NUMBER
H--CARD IMAGE

Several variations appear for bit positions 15 through 28. (The six, four, two subfield groups C, D, and E shown above is the octal configuration for machine instructions.) These are summarized in the table below in which the X represents one octal digit.

Type of Machine Word	Listing Format	Source Program Instruction
Processor instruction and indirect address	XXXXXX XXXX XX	Processor instruction and indirect address word
Data	XXXXXXXXXXXX	Data generating pseudo-operations (OCT, DEC, BCI, etc.)
Data Control	XXXXXX XX XXXX	Data Control Word (DCW)
Special 18-bit field data	XXXXXX XXXXXX	ZERO pseudo-operation
Input/output command	XX XXXX XXXXXX	Input/output pseudo-operation

Error flags are summarized at the end of this section. The interpretation of the relocation bits is described in the Loader manual.

b. PREFACE CARD LISTING

The contents of one or more preface cards are listed using a self-explanatory format. The LABELED COMMON symbols are listed according to type in the same order as presented on single or multiple cards: SYMDEFs, SYMREFs, LABELED COMMON, and .SYMT.

c. BLANK COMMON ENTRY

Following the LABELED COMMON symbols, the Assembler enters a statement of the amount of BLANK COMMON storage requested by the subprogram. The statement format is self-explanatory.

d. SYMBOLIC REFERENCE TABLE

The symbol table listing contains all symbols used, their octal values (normally, the location value), and the alter numbers of all instructions that referenced the symbol. The table format is as follows:

Definition	Symbol	Alter Numbers
00364	BETA	00103, 00103, 01027, 01761, 03767, 07954

The above sample indicates that the symbol BETA has been assigned the value 364_8 and is referenced in five places: namely, at alter number positions 00103, 01027, 01761, 03767, and 07954 in the listing of instructions. The first alter number is the point in the instruction listing where the symbol was defined. If an instruction contains a symbol twice, the alter number for that point in the instruction listing is given twice. The alter numbers are assigned sequentially in the subprogram listing, one per instruction. Because of this fact, it is easy for the programmer to locate in the listing those card images that referenced any particular symbol as well as locate the card image that caused the symbol to be defined.

e. ERROR CODES

The following list comprises the error flags for individual instructions and pseudo-operations.

ERROR	FLAG	CAUSE
Undefined	U	Undefined symbol(s) appear in the variable field.
Multidefined	M	Multiple-defined symbol(s) appear in the location field and/or the variable field.
Address	A	Illegal value or variable appears in the variable field. Also used to denote lack of a required field.

ERROR	FLAG	CAUSE
Index	X	Illegal index or address modification.
Relocation	R	Relocation error; expression in the variable field will produce a relocatable error upon loading.
Phase	P	Phase error; this implies undetected machine error or symbols becoming defined in Pass Two which were undefined in Pass One.
Even	E	Address in the variable field is odd, the current instruction requires an even reference.
Conversion	C	Error in conversion of either a literal constant or a subfield of a data-generative pseudo-operation.
Location	L	Error in the location field
Operation	O	Illegal operation
Table	T	An assembly table overflowed not permitting proper processing of this card completely. Table overflow error information will appear at the end of testing.

H. MACRO ASSEMBLER IMPLEMENTATION

This Assembler is implemented in the classic format of Macro Assemblers with several variations. The Assembler makes two passes over the external text. During pass one, all symbols are collected and assigned their absolute or relocatable values relative to the current location counter. MACRO prototypes are processed and placed in the MACRO skeleton table immediately ready for expansion. All MACRO calls, therefore, are expanded in pass one, allowing the MACRO skeleton table to be destroyed prior to pass two.

Machine operation codes, pseudo-operations, and MACRO names are all carried in the operation table during pass one. This implies that all operation codes, machine or pseudo, along with MACROS are looked up during pass one, and that the general operation table is destroyed at the end of pass one. The literal pool is completely expanded during pass one, avoiding duplicates (except for V, M, and nH literals where n is greater than 12), which are assigned unique locations in pass one and will be later expanded in pass two. Double-precision numbers in the literal pool start at even locations.

At the end of pass one, the symbol table is sorted; and a complete readjustment of symbols by their relative location counter is performed. The preface card is then punched.

All instructions are generated during pass two. This is accomplished by performing a scan over the variable fields and address modifications. This information is then combined with the operation code from pass one by using a Boolean OR function. Apparent errors are flagged.

The symbolic cross-reference table is created as the variable fields are scanned and expanded. The final edit of the symbol table is done at the end of pass two. Generative pseudo-operations are processed with the conversion being done in pass two. Pseudo-operations are available to control punching of binary cards and printing images of source cards. Images of source cards in error will be printed, regardless of control pseudo-operations. Multidefined symbols, undefined symbols, and error conditions will be noted at the end of the printer listing.

The following is a summary of Pass 1 and Pass 2 functions.

PASS 1

1. Location symbols are placed in the symbol table along with their definitions.
2. The operation code is looked up in the operation table and the operation control word passed on to Pass 2. Pseudo-operations requiring Pass 1 processing are processed.
3. Literals that can be evaluated in Pass 1 are converted to binary and placed in the literal pool. Literals M, V, and nH, where $n > 12$, are not processed until Pass 2.
4. Macro definitions are entered in the macro prototype table.
5. Card images required by DUP and macro expansions are produced.

6. Tables are formed from information supplied by certain pseudo-operations (USE, BEGIN, SYMDEF, BLOCK, LIT).

Housekeeping at End of Pass 1:

7. The USE tables are processed in conjunction with the BEGIN information to determine the origin of each USE.

8. The location of the literal pool, the error linkage and the program break are computed.

9. The symbol table is sorted, the symbols are given their true definitions based on the origin of their associated USE and the table is checked for multidefined symbols which are flagged.

10. The LIT table is processed to set the origins of each literal pool based on the origin of the USE under which the LIT occurred.

11. The values for the preface are computed.

12. Pass 2 is called.

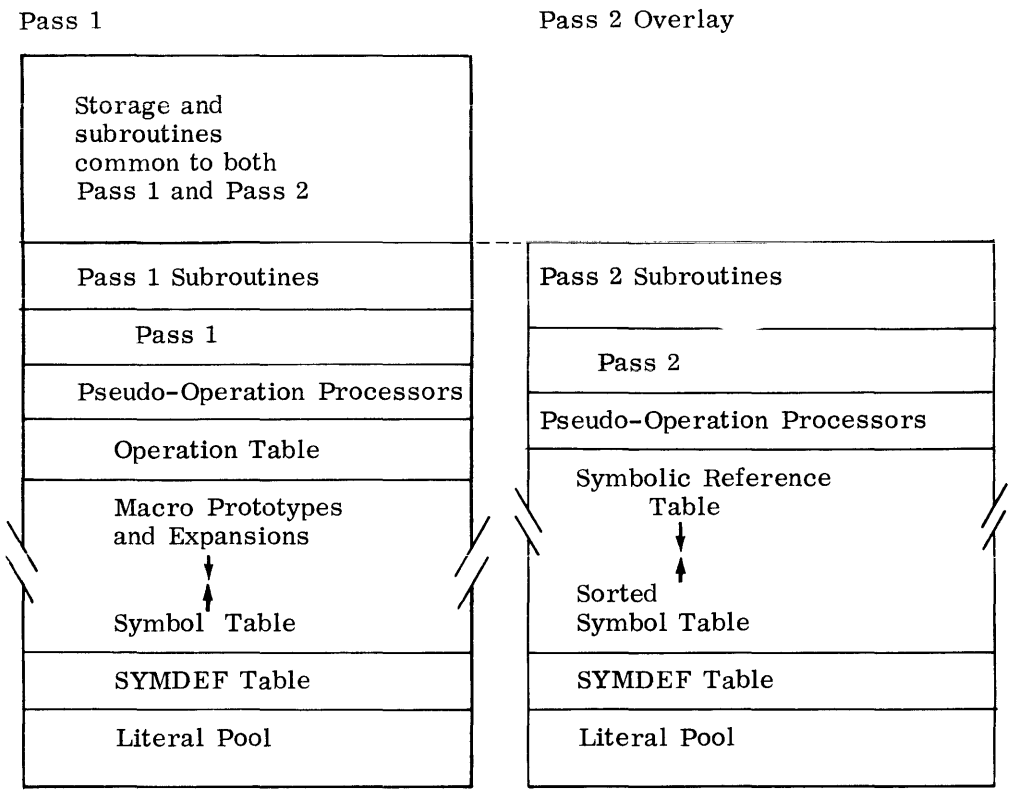
PASS 2

1. The preface is punched and listed for relocatable programs.

2. The binary deck and listing are produced using information from the intermediate file and the tables built by Pass 1.

3. The symbolic reference table is formed.

CORE MEMORY ALLOCATION FOR GMAP



PASS 1 CONTROL LOGIC

1. Initialize GMAP table locations and I/O routines. Read GMAP system macros.
2. Read a record; go to step 9.
3. If not in DUP mode, go to step 7.
4. If not the first time through the range of the DUP, go to step 6.
5. Save record for succeeding times through DUP range. Strip location symbol if not a SET and go to step 7.
6. Write intermediate file, retrieve next record from those previously saved and go to step 10.
7. Write intermediate file.
8. If expanding a macro, get next record from macro processor and go to step 10.
9. Move record into working storage and read next record.
10. If any records are to be skipped, reduce count and go to step 9.

11. Set up controls for processing the record.
12. If processing a macro prototype, pack record in prototype storage and go to step 3.
13. Look up operation code and, if a pseudo-operation, go to appropriate processor.
14. Enter location symbol in the symbol table.
15. Increase location counter, process literal if it exists and go to step 3.

PASS 2 LOGIC

1. Punch system macros, if required.
2. List and punch preface, if required.
3. Read a record from the intermediate file.
4. If not a BCD card to be punched by the DCARD pseudo-operation, go to 6.
5. Punch and list BCD card and go to 3.
6. If pseudo-operation, go to appropriate processor.
7. Check location symbol for phase error.
8. If a literal is present, get address and go to 12.
9. If I/O-type instruction, go to 17.
10. Evaluate symbolic index, if required.
11. Evaluate address field, if present.
12. Assemble operation code.
13. Evaluate tag field, if present.
14. List and punch instruction; increase location counter.
15. If literal is not to be assembled at this point, go to 3.
16. Assemble required literal and go to 3.
17. Assemble I/O-type instruction word and go to 14.

I. RELOCATABLE AND ABSOLUTE EXPRESSIONS

Expression evaluation can result in either relocatable or absolute values. There are three types of relocatable expressions: program relocatable (R), BLANK COMMON relocatable (C), and LABELED COMMON relocatable (L). The rules by which the assembler determines the relocation validity of an expression are of necessity a little complex, and the presence of multiple location counters compounds the problem somewhat. Certain of the principle pseudo-operations impose restriction as to type of expression that is permissible; these are described separately under each of the affected pseudo-operations. These are:

EQU	MAX	BFS	LOC
SET	BOOL	ORG	
MIN	BSS	BEGIN	

The following ten rules summarize the conditions and restrictions governing the admissibility of relocation:

1. The sum, difference, product, or quotient of two different types of relocatable elements is not valid.
2. An absolute element is an absolute expression.
3. A relocatable element is a relocatable expression.
4. An expression containing only absolute terms is absolute.
5. The difference between two relocatable elements is an absolute expression.
6. The asterisk (*) symbol (implying current location counter) is a relocatable element.
7. The sum, product, or quotient of two relocatable elements is not valid for relocation.
8. The product or quotient of an absolute element and a relocatable element is not valid.
9. The complement of a relocatable element is not valid.
10. The sum or difference of a relocatable element and an absolute element is relocatable.

These ten rules are not a complete set of determinants but do serve as a basis for establishing a method of defining relocation admissibility of an expression.

Let R_r denote a program-text relocatable element, R_c denote a BLANK COMMON element, and R_l denote a LABELED COMMON element. Next, take any expression and process it as follows:

1. Replace all absolute elements with their respective values.
2. Replace any relocatable element with the proper R_i, where i = r, c, or l. This yields a resulting expression involving only numbers and the terms R_r, R_l, and R_c.
3. Discard all terms in which all elements are absolute.

4. Evaluate the resulting expression. If it is zero or numeric, the original expression is absolute; if it is explicitly R_r, R_c, or R_l, then the original expression is normal relocatable, BLANK COMMON relocatable, or LABELED COMMON relocatable, respectively.

5. If the resulting expression is not as given in 4 above, it is a relocation error and/or an invalid expression.

In the illustrative examples following, assume ALPHA and BETA to be normal relocatable elements (R_r), GAMMA and DELTA to be BLANK COMMON relocatable elements (R_c), and EPSILON and ZETA to be LABELED COMMON relocatable elements (R_l). Let N and K be absolutely equivalent to 5 and 8, respectively.

1. 4*ALPHA-7-4*BETA
   reduces to
   $4*R_r - 4*R_r = 0$,
   thus indicating a valid absolute expression.

2. N*ALPHA + 8*GAMMA + 21 - K*DELTA
   reduces to
   $5*R_r+8*R_c-8*R_c = 5*R_r$,
   thus indicating an invalid expression.

3. EPSILON+N-ZETA
   reduces to
   $R_l+5-R_l = 5$,
   thus indicating a valid absolute expression.

4. ALPHA-GAMMA+DELTA+7
   reduces to
   $R_r-R_c+R_c = R_r$,
   thus indicating a valid relocatable expression.

V. PROGRAMMING EXAMPLES

EXAMPLE 1: ACCUMULATIVE SUMMATION

Two 100 word blocks of variables, a_i and b_i, start at locations A and B, respectively. It is required to compute:

a. $\sum_{i=1}^{100} a_i$ store in SA

b. $\sum_{i=1}^{100} ia_i$ store in SIA

c. $\sum_{i=1}^{100} b_i$ store in SB

d. $\sum_{i=1}^{100} ib_i$ store in SIB

```
        STZ     SIA     Clear for sum IA(I)
        STZ     SIB     Clear for sum IB(I)
        LDA     =0, DU  Clear for sum A(I)
        LDQ     =0, DU  Clear for sum B(I)
        LDX1    =100, DU Set Index to 100
LOOP    ADA     A-1, 1  Add A(I)
        ASA     SIA     Add sum of A(I)
        ADQ     B-1, 1  Add B(I)
        ASQ     SIB     Add sum of B(I)
        SBX1    =1, DU  Decrement Index
        TNZ     LOOP    Continue if I not zero
        STA     SA      Store sum of A(I)
        STQ     SB      Store sum of B(I)
```

The four summations are formed simultaneously in a loop controlled by index register 1. Problem a is accumulated in the A register. Problem c is accumulated in the Q register. Problem b is accumulated in memory location SIA. Problem d is accumulated in memory location SIB.

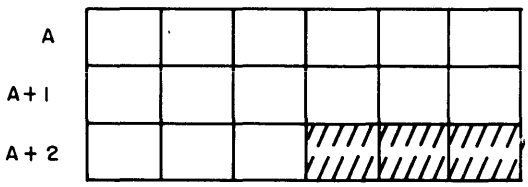
The algorithm for problems b and c is to add a_{100} to SIA 100 times, add a_{99} to SIA 99 times, etc. This is accomplished by accumulating the summation of the a_i's starting with a_{100} and adding the partial summation to SIA each time through the loop. This algorithm comes about by expanding the summation:

$$\sum_{i=1}^{100} ia_i = a_1 + 2a_2 + 3a_3 + \ldots\ldots\ldots\ldots\ldots\ldots\ldots 99a_{99} + 100a_{100}$$

$$= a_{100} + (a_{100} + a_{99}) + (a_{100} + a_{99} + a_{98}) + \ldots\ldots\ldots\ldots\ldots\ldots$$

EXAMPLE 2: CHARACTER MOVEMENT

At A and following is a string of up to 15 six-bit characters ending in a slash, /. The number of characters before the slash is unknown. Move the string of characters excluding the slash to location B and following. Store the number of characters moved in an index register.



LOOP	LDA	A1, SC	Load char. in A reg., right adjusted
	CMPA	SLASH, DL	Is char. a SLASH?
	TZE	*+3	Yes, exit LOOP
	STA	B1, SC	Store char. in B block
	TRA	LOOP	Return to get next char.
	LDA	=15, DL	Max. char. count = 15
	SBA	A2, CI	Char. count = 15 - tally
	EAX1	0, AL	Move count to XR1
A1	TALLY	A, 15, 0	IND word - A Block
B1	TALLY	B, 15, 0	IND word - B Block
A2	TALLY	B1, 0, 4	IND word - char. TALLY
SLASH	BOOL	61	SLASH = 61,GE char. set

This example illustrates both forms of character address modification, Sequence Character, and Character from Indirect. The Sequence Character modification is used to obtain each character from the A block and to store each non-slash character in the B block using indirect words A1 and B1 respectively. The Character from Indirect modification is used to obtain the tally from the A1 indirect word which tells how many characters are left in A that have not been examined.

EXAMPLE 3: LIST COMPARISON

A table of 100 data words is stored in locations A to A + 99. Find the first number in the table whose value is between two numbers stored in locations L (lower limit) and U (upper limit).

```
LDA     L       Lower limit in A reg.
LDQ     U       Upper limit in Q reg.
LDX7    =0,DU   Initialize XR7
RPT     100,1,TZE  Repeat next inst. 100 times,
                   increment address by 1,
                   terminate on hit
CWL     A,7     Compare table entries with limits
TNZ     NONE    None within limits
LDA     -1,7    Hit in A register
```

This example shows the technique used with the repeat instruction. Index register 7 contains the address of the next entry in the table to be compared. Using XR7-1 as the address of the successful hit, the successful hit is loaded into the A Register. The terminate condition specified in the Repeat Instruction (TZE) is the zero indicator condition for a successful comparison within the limits set in the A and Q registers.

EXAMPLE 4: GRAY CODE TO BINARY

An unsigned, Gray-coded binary integer is stored in bits 0 - 19 of location DATA. Extract the word, convert it to binary and store it as an integer in location DATA1.

```
LDA     DATA    Load Data
ARL     16      Shift Integer Right
GTB             Convert to Binary
STA     DATA1   Store converted integer
```

The logical shift to the right brings the integer to the lower accumulator and fills the remaining 16 bits with 0. The conversion is done in one step with the Gray to Binary instruction.

EXAMPLE 5: BINARY TO BINARY CODED DECIMAL (BCD)

This example illustrates a method of converting a number from binary to BCD. The example converts a number that is in the range of $-10^6 + 1$ to $+10^6 - 1$, inclusive.

01	LDX2	0,DU	Place zeros in X2
02	LDA	X	Load accumulator with value to be converted
03	RPT	6,1	Repeat 6 times, increment by 1
04	BCD	TAB,2	Divide by TAB, TAB + 1, etc.
05	STQ	Y	Store converted number in Y
	.		
	.		
	.		
	.		
06 TAB	DEC	800000,640000,512000,409600,327680,	
	DEC	262144	

Steps 03 and 04 perform the conversion of the binary number in the accumulator to the Binary-Coded Decimal equivalent. Step 03 will repeat step 04 six times. It will also increment the contents of index register 2 by one after each execution.

The BCD instruction, step 04, is designed to convert the magnitude of the contents of the accumulator to the Binary-Coded Decimal equivalent. The method employed is to effectively divide a constant into this number, place the result in bits 30-35 of the quotient register, and leave the remainder in the accumulator. The execution of the BCD instruction will then allow the user to convert a binary number to BCD, one digit at a time, with each digit coming from the high-order part of the number. The address of the BCD instruction refers to a constant to be used in the division, and a different constant would be needed for each digit. In the process of the conversion, the number in the accumulator is shifted left three positions. The $C(Q)_{0-35}$ are shifted left 6 positions before the new digit is stored.

In this example, the constants used for dividing are located at TAB, TAB + 1, TAB + 2, ..., TAB + 5. If the value in X were 000000522241_8, the quotient register would contain 010703020107_8 at the completion of the repeat sequence. Step 05 stores the quotient register in Y.

The table in Appendix A gives the conversion constants to be used with the binary to BCD instruction. Each vertical column represents the set of constants to be used depending on the initial value of the binary number to be converted to its decimal equivalent. The instruction is executed once per digit, using the constant appropriate to the conversion step with each execution.

An alternate use of the table for conversion involves the use of the constants in the row corresponding to conversion step 1. If after each conversion, the contents of the accumulator are shifted right 3 positions, the constants in the conversion step 1 row may be used one at a time in order of decreasing value until the conversion is complete.

EXAMPLE 6: BCD ADDITION

This example illustrates the addition of two words containing BCD integers. The example limits the result to 999999. Add the BCD numbers in locations A and B and store the result in C.

```
01      LDA     A
02      ADLA    B               Compute A + B
03      ADLA    =O666666666666  Add octal 66 to each digit to force
                                carries
04      STA     C
05      ANA     =O606060606060  Extract octal 60 from each non-
                                carry
06      ERSA    C               Subtract octal 60 from each non-
                                carry
07      ARL     3               Subtract octal
08      NEG                     06 from each
09      ASA     C               Non-carry
```

ADDITIONAL RESULTS

Line	V	W	X	Y	Z
0	00	66	60	6	00
1	01	67	60	7	01
2	02	70	60	10	02
3	03	71	60	11	03
4	04	72	60	12	04
5	05	73	60	13	05
6	06	75	60	14	06
7	07	75	60	15	07
8	10	76	60	16	10
9	11	77	60	17	11
10	12	00	00	0	00
11	13	01	00	1	01
12	14	02	00	2	02
13	15	03	00	3	03
14	16	04	00	4	04
15	17	05	00	5	05
16	20	06	00	6	06
17	21	07	00	7	07
18	22	10	00	10	10
19	--	11	00	11	11

Step 01 places the number in A into the accumulator.

Step 02 adds the number in B to the accumulator, Column V in the table, following, shows the possible results for any digit. It should be noted that there are 19 possible results, indicated by lines 0-18.

Step 03 forces any carries into the units position of the next digit. Lines 10-18 of Column V contain the sums that will carry into the next digit. Column W contains the 20 possible results for each digit position. The additional possibility (line 19) arises from the fact that there can be a carry of one into a digit.

Step 04 stores the intermediate result in C.

Step 05 extracts an octal 60 from each non-carry digit. The results are indicated in column X. The digits that did not force a carry (lines 0-9) result in an octal 60, the digits that had a carry into the next digit (lines 10-18) result in 00.

Step 06 performs an EXCLUSIVE OR of the contents of the accumulator with the contents of C. This in effect subtracts octal 60 from each digit that did not have a carry (lines 0-9). The results are indicated in column Y.

Step 07 shifts the octal 60's to the right three places.

Step 08 negates the contents of the accumulator.

Step 09 is an add to storage the contents of the accumulator to the contents of C. This in effect subtracts a 06 from each digit that did not have a carry, the results of which are indicated in Column Z.

EXAMPLE 7: BCD SUBTRACTION

The BCD number in B is subtracted from the BCD number in A and the result is stored in C. The contents of A must be equal to or greater than the contents of B.

```
01      LDA     A
02      SBLA    B               Compute A-B
03      STA     C
04      ANA     =O606060606060  Extract octal 60 from each
                                borrow
05      ERSA    C               Subtract octal 60 from each
                                borrow
06      ARL     3               ⎫ Subtract octal
07      NEG                     ⎬ 06 from each
08      ASA     C               ⎭ Borrow
```

SUBTRACTION RESULTS

Line	W	X	Y	Z
0	11	0	11	11
1	10	0	10	10
2	07	0	07	07
3	06	0	06	06
4	05	0	05	05
5	05	0	04	04
6	03	0	03	03
7	06	0	02	02
8	01	0	01	01
9	00	0	00	00
10	77	60	17	11
11	76	60	16	10
12	75	60	15	07
13	74	60	14	06
14	73	60	13	05
15	72	60	12	04
16	71	60	11	03
17	70	60	10	02
18	67	60	7	01
19	66	60	6	00

Step 01 loads the accumulator with the contents of A.

Step 02 subtracts the contents of B from the accumulator. The possible results for each digit are indicated in Column W of the table that is included with this example.

Step 03 stores the intermediate result in C.

Step 04 extracts an octal 60 from each digit that required a borrow. This will leave an octal 60 in each digit position where there was a borrow. The possible results of this instruction are indicated in Column X, lines 0-19 (10-19 refer to those which result in octal 60).

Step 05, an EXCLUSIVE OR to storage, in effect subtracts the octal 60's in the accumulator from the corresponding digit in C. The possible results for each digit are displayed in Column Y.

Step 06 shifts the octal 60's in the accumulator right three places.

Step 07 negates the contents of the accumulator.

Step 08, an add to storage, is in effect a subtraction of 06 from each digit that required a borrow, the result being placed in C. Column Z of the table reflects the possible results for each digit.

EXAMPLE 8: FIXED-POINT INTEGER TO FLOATING-POINT CONVERSION

The integer to be converted is in location M

```
TOV     1,IC        Reset overflow indicator
LDA     M           Load integer in A reg.
LDQ     ,DL         Clear Q reg.
LDE     =35B25,DU   Set exponent to 35
FNO                 Normalize
```

The Floating Normalize instruction completes the conversion by shifting the AQ left while adjusting the exponent until $C(AQ_1) = 1$.

For example, if the contents of M = $000000000002_8 = +2_{10}$, then the contents of the floating point register (EAQ) will be E = $+2_{10}$, AQ = $200000000000000000000000_8 = +0.1_2$ or EAQ = +2.

EXAMPLE 9: CHARACTER TRANSLITERATION

This illustrates a method of transliterating each character of a card image that has been punched in the FORTRAN Character Set to the octal value of the corresponding character in the General Electric Standard Character Set. There are 48 characters in the FORTRAN Set and 64 characters in the General Electric Standard Character Set. Each character that is punched invalidly (not a standard punch combination in the FORTRAN Set) is converted to a blank. The card is stored in the first 80 character positions of block location IMAGE.

The table, TABLE, is 64 locations long. The character in each location is a General Electric standard character that corresponds to a FORTRAN character in the following manner. The relative location of a particular character to the start of the table is equal to the binary value of the corresponding FORTRAN character. For example, an A punched in the FORTRAN Character Set has the octal value $21 = 17_{10}$. The relative location 17 to TABLE contains an A in the General Electric Standard Character Set. A 3-8 punch in the FORTRAN Set represents an = character. The 3-8 punch would be read as an octal 13 (11_{10}). The relative location 11 to TABLE contains an octal 75 (see line 21) which represents the = character in the General Electric Standard Character Set.

Note: Character transliteration is normally handled by the M-605 software package.

```
01          LDA     TALLY1      Initialize TALLY word
02          STA     TALLY2
03    LOOP  LDA     TALLY2,CI   Pick up character to be trans-
                                literated
04          LDQ     TABLE,AL    Load QR with transliterated
                                character
```

```
05                  STQ       TALLY2,SC     Store back on card image
06                  TTF       LOOP          If tally has not run out, continue
                                            LOOP
                    .
                    .
                    .
07    TALLY1        TALLY     IMAGE,80,0
08    TALLY2        ZERO
09    IMAGE         BSS       14
10    TABLE         OCT       0
11                  OCT       1
12                  OCT       2
13                  OCT       3
14                  OCT       4
15                  OCT       5
16                  OCT       6
17                  OCT       7
18                  OCT       10
19                  OCT       11
20                  OCT       20
21                  OCT       75            3-8 Punch = in FORTRAN set
22                  OCT       57            4-8 Punch ' in FORTRAN set
23                  OCT       20
24                  OCT       20
25                  OCT       20
26                  OCT       20
27                  OCT       21
28                  OCT       22
29                  OCT       23
30                  OCT       24
31                  OCT       25
32                  OCT       26
33                  OCT       27
34                  OCT       30
35                  OCT       31
36                  OCT       60            12 punch + in FORTRAN set
37                  OCT       33            12-3-8 punch . in FORTRAN set
38                  OCT       55            12-4-8 punch ) in FORTRAN set
```

39	OCT	20	
40	OCT	20	
41	OCT	20	
42	OCT	20	
43	OCT	41	
44	OCT	42	
45	OCT	43	
46	OCT	44	
47	OCT	45	
48	OCT	46	
49	OCT	47	
50	OCT	50	
51	OCT	51	
52	OCT	52	11 punch - in FORTRAN set
53	OCT	53	11-3-8 punch $ in FORTRAN set
54	OCT	54	11-4-8 punch * in FORTRAN set
55	OCT	20	
56	OCT	20	
57	OCT	20	
58	OCT	20	
59	OCT	61	0-1 punch / in FORTRAN set
60	OCT	62	
61	OCT	63	
62	OCT	64	
63	OCT	65	
64	OCT	66	
65	OCT	67	
66	OCT	70	
67	OCT	71	
68	OCT	20	
69	OCT	73	0-3-8 punch , in FORTRAN set
70	OCT	35	0-4-8 punch (in FORTRAN set
71	OCT	20	
72	OCT	20	
73	OCT	20	

Steps 01 and 02 initialize the indirect word TALLY2.

Step 03 picks up the character to be transliterated by referencing the word TALLY2 with the Character from Indirect (CI) modifier. This will place the character specified by bits 33-35 of TALLY2 from a location specified by bits 0-17 of TALLY2 into the accumulator, bits 29-35. Bits 0-28 of the accumulator will be set to zero.

Step 04 picks up the corresponding General Electric standard character from the address TABLE modified by the contents of accumulator, bits 18-35.

Step 05 places the transliterated character back in the card image where it was originally picked up. The Sequence Character (SC) modifier increments the character specified in bits 33-35 of the word TALLY2.

Each time the character position becomes greater than 5, it is reset to zero; and the address specified in bits 0-17 of TALLY2 is incremented by one. The tally in bits 18-29 of the same word is decremented by 1 with each SC reference. Whenever a tally reaches zero, the Tally Runout Indicator is set ON.

Step 06 tests the Tally Runout Indicator. If it is OFF, the program transfers to LOOP; if not, the next sequential instruction is taken.

EXAMPLE 10: TABLE LOOKUP

This example illustrates a method of searching an unordered table for a value equal to the value in the accumulator. Prior to entering the routine given below, the user must load the accumulator with the search argument, load the quotient register with the size of the table to be searched (the size should be scaled at binary point 25), and initialize index register 1 with the first location of the table to be searched. The user enters the routine by executing a transfer and set index register 2 (TSX2) to the symbolic location TLU (see step 05, below). Return from the routine is to the instruction following the TSX2. The Zero Indicator will tell the user whether or not a match has occurred. Zero Indicator ON indicates a match; Zero Indicator OFF indicates no match. If a match was made, the contents of index register 1 will be W locations (W being the increment specified in the RPTX command, step 15) higher than the location of the equal argument.

```
01    *       CALLING SEQUENCE IS:
02    *       LDA     ITEM        Search item
03    *       LDQ     SIZE        Number of table entries---at B25.
04    *       LDX1    FIRST,DU    Location of first search word in table
05    *       TSX2    TLU         Call table lookup subroutine
06    *       TZE     FOUND       Transfer if search item is in table, or
07    *       TNZ     ABSENT      Transfer if search item is not in table
08    *
```

```
09  *       IF IN TABLE, C(X1)-W WILL BE THE LOCATION OF THE LAST WORD.
10  *       OTHERWISE, C(X1)-W WILL BE THE LOCATION OF THE LAST SEARCH
11  *       WORD IN THE TABLE.  W IS THE NUMBER OF WORDS PER ENTRY.
12  TLU     EAX0    64,QL       Pickup size (MOD 256) and TZE-BIT
13          SBLQ    1024,DL     Size = Size-1.
14          TMI     ,2          Exit if size was 0--empty table
15  TLU1    RPTX    ,W          Note that 0 represents 256 (MOD 256)
16          CMPA    ,1          Perform table lookup
17          TZE     ,2          Exit if search item is in table
18          SBLQ    1,DU        Size = Size-256
19          TPL     TLU1        Continue table lookup if more entries
20          TRA     ,2          Exit--Search item is not in table
```

Steps 01-11 are comment cards.

Step 12 places the contents of the lower half (bits 18-35) of the quotient register plus 64, in index register 0. The number 64, in effect, sets the TZE terminate repeat condition on. The instruction also places the last 8 bits of the size of the table in index register 0, bits 0-7. Thus if the size of the table is a multiple of 256 words, zeros will be loaded into bits 0-7 of index register 1. Zeros in those bit positions will cause the repeat to execute 256 times. If, however, the size of the table to be searched is of the form 256n+m, where $n \geq 0$, and $0 \leq m \leq 256$, the m would be placed in bits 0-7 of index register 0. This will cause the repeat instruction to be executed a maximum of m times on the first pass through.

Step 13 subtracts 1024 from the quotient register. This, in effect, subtracts 1 from the size of the table to be searched. The subtracting of 1 becomes meaningful in two places: (1) it provides a test to be sure the table is not zero words long (see step 14) and (2) if the table is a multiple of 256 words long, it effectively subtracts 1 from bits 0-17 (a look-ahead to steps 18 and 19 points out the importance of this).

Step 14 causes the routine to return to the main program if the size of the table was zero.

Step 15, an RPTX, executes step 16 a number of times equal to the contents of index register 0, bits 0-7, at the start of the instruction execution. Each time step 16 is executed, the contents of the accumulator (the search argument) are compared with the contents of the location specified by index register 1. At the same time, index register 1 is incremented by W as is specified in the repeat instruction; and the contents of index register 0, bits 0-7, are decremented by 1. The repeat sequence terminates when the compare causes the Zero Indicator to be set or when bits 0-7 of index register 0 are set to zero.

Step 17 tests the Zero Indicator and returns to the main program if it is set. It should be noted that index register 1 will be set W locations higher than when the equal argument was found because of the sequence of events described above.

If the Zero Indicator was not set by step 16, then step 18 will be executed. This instruction subtracts 1 from bits 0-17 of the quotient register. In effect, this is subtracting 256 from the size of the table. The size of the table can be expressed in the form 256n+m. If m=0 and n=1, then the contents of the quotient register would also go zero at this point. This is because step 13 would have caused a borrow of 1 from n when m equals zero. Further inspection of these instructions will reveal that positive values of n and m, other than those expressed above, will only cause the routine to loop until the contents of the quotient register are reduced to a negative value.

Step 19 transfers control to step 15 if the contents of quotient register remained positive. If the quotient register became negative, step 20 is executed and the routine returns to the main program.

It should be noted that when control is transferred back to step 15, index register 0, bits 0-7, contains zeros (causes the repeat to be executed a maximum of 256 times); and index register 1 contains the address of the next location in the table that is to be searched.

APPENDIX A
BINARY TO BCD CONVERSION

CONVERSION STEP \ STARTING RANGE OF C (ACC)	$10^9 \rightarrow 10^{10}-1$	$10^8 \rightarrow 10^9-1$	$10^7 \rightarrow 10^8-1$	$10^6 \rightarrow 10^7-1$	$10^5 \rightarrow 10^6-1$	$10^4 \rightarrow 10^5-1$	$10^3 \rightarrow 10^4-1$	$10^2 \rightarrow 10^3-1$	$10^1 \rightarrow 10^2-1$	$0 \rightarrow 10^1-1$
1	8×10^9	8×10^8	8×10^7	8×10^6	8×10^5	8×10^4	8×10^3	8×10^2	8×10^1	8
2	$8^2 \times 10^8$	$8^2 \times 10^7$	$8^2 \times 10^6$	$8^2 \times 10^5$	$8^2 \times 10^4$	$8^2 \times 10^3$	$8^2 \times 10^2$	$8^2 \times 10^1$	8^2	
3	$8^3 \times 10^7$	$8^3 \times 10^6$	$8^3 \times 10^5$	$8^3 \times 10^4$	$8^3 \times 10^3$	$8^3 \times 10^2$	$8^3 \times 10^1$	8^3		
4	$8^4 \times 10^6$	$8^4 \times 10^5$	$8^4 \times 10^4$	$8^4 \times 10^3$	$8^4 \times 10^2$	$8^4 \times 10^1$	8^4			
5	$8^5 \times 10^5$	$8^5 \times 10^4$	$8^5 \times 10^3$	$8^5 \times 10^2$	$8^5 \times 10^1$	8^5				
6	$8^6 \times 10^4$	$8^6 \times 10^3$	$8^6 \times 10^2$	$8^6 \times 10^1$	8^6					
7	$8^7 \times 10^3$	$8^7 \times 10^2$	$8^7 \times 10^1$	8^7						
8	$8^8 \times 10^2$	$8^8 \times 10^1$	8^8							
9	$8^9 \times 10^1$	8^9								
10	8^{10}									

The values in the above table are the conversion constants to be used with the Binary to BCD instruction. Each vertical column represents the set of constants to be used depending on the initial value of the binary number to be converted to its decimal equivalent. The instruction is executed once per digit using the constant appropriate to the conversion step with each execution.

An alternate use of the table for conversion involves the use of the constants in the row corresponding to conversion step 1. If, after each conversion, the contents of the Accumulator are shifted right three places, the constants in the conversion step one row may be used one at a time in order of decreasing value until the conversion is complete.

Refer to page V-4 for a programming example of this conversion.

APPENDIX B
GRAY CODE TO BINARY CONVERSION

The instruction GTB (gray to binary) will convert the gray code into the binary equivalent shown below:

Gray Code	Binary Equivalent	Decimal Equivalent
0 0 0 0	0 0 0 0	0
0 0 0 1	0 0 0 1	1
0 0 1 1	0 0 1 0	2
0 0 1 0	0 0 1 1	3
0 1 1 0	0 1 0 0	4
0 1 1 1	0 1 0 1	5
0 1 0 1	0 1 1 0	6
0 1 0 0	0 1 1 1	7
1 1 0 0	1 0 0 0	8
1 1 0 1	1 0 0 1	9
1 1 1 1	1 0 1 0	10
1 1 1 0	1 0 1 1	11
1 0 1 0	1 1 0 0	12
1 0 1 1	1 1 0 1	13
1 0 0 1	1 1 1 0	14
1 0 0 0	1 1 1 1	15

Codes of up to 36 bits in length can be accommodated.

Gray code is a cyclic binary code in which only one bit at a time changes as the total number increases or decreases. Analog to digital angular shaft encoders often employ Gray code devices. This technique results in less errors for angular digital read-outs. The GTB instruction thus facilitates the real-time data processing of radar angle data and other devices that use Gray code encoders.

The Gray to binary conversion is defined by the following algorithm, where R_i and S_i denote the contents of bit positions i of the A Register before and after the conversion:

$$S_o = R_o$$

$$S_i = (R_i \text{ AND } \overline{S_{i-1}}) \text{ OR } (\overline{R_i} \text{ AND } S_{i-1}) \text{ for } i = 1,2,3....35$$

Refer to page V-3 for a programming example of this conversion.

APPENDIX C
M-605 STANDARD
CHARACTER SET

M-605 STANDARD CHARACTER SET

Standard Character Set	GE-Internal Machine Code	Octal Code	Hollerith Card Code	Standard Character Set	GE-Internal Machine Code	Octal Code	Hollerith Card Code
0	00 0000	00	0	↑	10 0000	40	11-0
1	00 0001	01	1	J	10 0001	41	11-1
2	00 0010	02	2	K	10 0010	42	11-2
3	00 0011	03	3	L	10 0011	43	11-3
4	00 0100	04	4	M	10 0100	44	11-4
5	00 0101	05	5	N	10 0101	45	11-5
6	00 0110	06	6	O	10 0110	46	11-6
7	00 0111	07	7	P	10 0111	47	11-7
8	00 1000	10	8	Q	10 1000	50	11-8
9	00 1001	11	9	R	10 1001	51	11-9
[	00 1010	12	2-8	-	10 1010	52	11
#	00 1011	13	3-8	$	10 1011	53	11-3-8
@	00 1100	14	4-8	*	10 1100	54	11-4-8
:	00 1101	15	5-8	)	10 1101	55	11-5-8
>	00 1110	16	6-8	;	10 1110	56	11-6-8
?	00 1111	17	7-8	'	10 1111	57	11-7-8
(blank)	01 0000	20	(blank)	,	11 0000	60	12-0
A	01 0001	21	12-1	/	11 0001	61	0-1
B	01 0010	22	12-2	S	11 0010	62	0-2
C	01 0011	23	12-3	T	11 0011	63	0-3
D	01 0100	24	12-4	U	11 0100	64	0-4
E	01 0101	25	12-5	V	11 0101	65	0-5
F	01 0110	26	12-6	W	11 0110	66	0-6
G	01 0111	27	12-7	X	11 0111	67	0-7
H	01 1000	30	12-8	Y	11 1000	70	0-8
I	01 1001	31	12-9	Z	11 1001	71	0-9
&	01 1010	32	12	←	11 1010	72	0-2-8
.	01 1011	33	12-3-8	,	11 1011	73	0-3-8
]	01 1100	34	12-4-8	%	11 1100	74	0-4-8
(	01 1101	35	12-5-8	=	11 1101	75	0-5-8
<	01 1110	36	12-6-8	"	11 1110	76	0-6-8
\	01 1111	37	12-7-8	!	11 1111	77	0-7-8

APPENDIX D
PSEUDO-OPERATIONS
BY FUNCTIONAL CLASS
WITH PAGE REFERENCES

PSEUDO-OPERATIONS

APPENDIX E
CONVERSION TABLE OF
OCTAL-DECIMAL
INTEGERS AND FRACTIONS

OCTAL-DECIMAL INTEGER CONVERSION TABLE

0000 (Octal)	0000 (Decimal)
to	to
0777 (Octal)	0511 (Decimal)

Octal	Decimal
10000 -	4096
20000 -	8192
30000 -	12288
40000 -	16384
50000 -	20480
60000 -	24576
70000 -	28672

	0	1	2	3	4	5	6	7
0000	0000	0001	0002	0003	0004	0005	0006	0007
0010	0008	0009	0010	0011	0012	0013	0014	0015
0020	0016	0017	0018	0019	0020	0021	0022	0023
0030	0024	0025	0026	0027	0028	0029	0030	0031
0040	0032	0033	0034	0035	0036	0037	0038	0039
0050	0040	0041	0042	0043	0044	0045	0046	0047
0060	0048	0049	0050	0051	0052	0053	0054	0055
0070	0056	0057	0058	0059	0060	0061	0062	0063
0100	0064	0065	0066	0067	0068	0069	0070	0071
0110	0072	0073	0074	0075	0076	0077	0078	0079
0120	0080	0081	0082	0083	0084	0085	0086	0087
0130	0088	0089	0090	0091	0092	0093	0094	0095
0140	0096	0097	0098	0099	0100	0101	0102	0103
0150	0104	0105	0106	0107	0108	0109	0110	0111
0160	0112	0113	0114	0115	0116	0117	0118	0119
0170	0120	0121	0122	0123	0124	0125	0126	0127
0200	0128	0129	0130	0131	0132	0133	0134	0135
0210	0136	0137	0138	0139	0140	0141	0142	0143
0220	0144	0145	0146	0147	0148	0149	0150	0151
0230	0152	0153	0154	0155	0156	0157	0158	0159
0240	0160	0161	0162	0163	0164	0165	0166	0167
0250	0168	0169	0170	0171	0172	0173	0174	0175
0260	0176	0177	0178	0179	0180	0181	0182	0183
0270	0184	0185	0186	0187	0188	0189	0190	0191
0300	0192	0193	0194	0195	0196	0197	0198	0199
0310	0200	0201	0202	0203	0204	0205	0206	0207
0320	0208	0209	0210	0211	0212	0213	0214	0215
0330	0216	0217	0218	0219	0220	0221	0222	0223
0340	0224	0225	0226	0227	0228	0229	0230	0231
0350	0232	0233	0234	0235	0236	0237	0238	0239
0360	0240	0241	0242	0243	0244	0245	0246	0247
0370	0248	0249	0250	0251	0252	0253	0254	0255

	0	1	2	3	4	5	6	7
0400	0256	0257	0258	0259	0260	0261	0262	0263
0410	0264	0265	0266	0267	0268	0269	0270	0271
0420	0272	0273	0274	0275	0276	0277	0278	0279
0430	0280	0281	0282	0283	0284	0285	0286	0287
0440	0288	0289	0290	0291	0292	0293	0294	0295
0450	0296	0297	0298	0299	0300	0301	0302	0303
0460	0304	0305	0306	0307	0308	0309	0310	0311
0470	0312	0313	0314	0315	0316	0317	0318	0319
0500	0320	0321	0322	0323	0324	0325	0326	0327
0510	0328	0329	0330	0331	0332	0333	0334	0335
0520	0336	0337	0338	0339	0340	0341	0342	0343
0530	0344	0345	0346	0347	0348	0349	0350	0351
0540	0352	0353	0354	0355	0356	0357	0358	0359
0550	0360	0361	0362	0363	0364	0365	0366	0367
0560	0368	0369	0370	0371	0372	0373	0374	0375
0570	0376	0377	0378	0379	0380	0381	0382	0383
0600	0384	0385	0386	0387	0388	0389	0390	0391
0610	0392	0393	0394	0395	0396	0397	0398	0399
0620	0400	0401	0402	0403	0404	0405	0406	0407
0630	0408	0409	0410	0411	0412	0413	0414	0415
0640	0416	0417	0418	0419	0420	0421	0422	0423
0650	0424	0425	0426	0427	0428	0429	0430	0431
0660	0432	0433	0434	0435	0436	0437	0438	0439
0670	0440	0441	0442	0443	0444	0445	0446	0447
0700	0448	0449	0450	0451	0452	0453	0454	0455
0710	0456	0457	0458	0459	0460	0461	0462	0463
0720	0464	0465	0466	0467	0468	0469	0470	0471
0730	0472	0473	0474	0475	0476	0477	0478	0479
0740	0480	0481	0482	0483	0484	0485	0486	0487
0750	0488	0489	0490	0491	0492	0493	0494	0495
0760	0496	0497	0498	0499	0500	0501	0502	0503
0770	0504	0505	0506	0507	0508	0509	0510	0511

1000 (Octal)	0512 (Decimal)
to	to
1777 (Octal)	1023 (Decimal)

	0	1	2	3	4	5	6	7
1000	0512	0513	0514	0515	0516	0517	0518	0519
1010	0520	0521	0522	0523	0524	0525	0526	0527
1020	0528	0529	0530	0531	0532	0533	0534	0535
1030	0536	0537	0538	0539	0540	0541	0542	0543
1040	0544	0545	0546	0547	0548	0549	0550	0551
1050	0552	0553	0554	0555	0556	0557	0558	0559
1060	0560	0561	0562	0563	0564	0565	0566	0567
1070	0568	0569	0570	0571	0572	0573	0574	0575
1100	0576	0577	0578	0579	0580	0581	0582	0583
1110	0584	0585	0586	0587	0588	0589	0590	0591
1120	0592	0593	0594	0595	0596	0597	0598	0599
1130	0600	0601	0602	0603	0604	0605	0606	0607
1140	0608	0609	0610	0611	0612	0613	0614	0615
1150	0616	0617	0618	0619	0620	0621	0622	0623
1160	0624	0625	0626	0627	0628	0629	0530	0631
1170	0632	0633	0634	0635	0636	0637	0638	0639
1200	0640	0641	0642	0643	0644	0645	0646	0647
1210	0648	0649	0650	0651	0652	0653	0654	0655
1220	0656	0657	0658	0659	0660	0661	0662	0663
1230	0664	0665	0666	0667	0668	0669	0670	0671
1240	0672	0673	0674	0675	0676	0677	0678	0679
1250	0680	0681	0682	0683	0684	0685	0686	0687
1260	0688	0689	0690	0691	0692	0693	0694	0695
1270	0696	0697	0698	0699	0700	0701	0702	0703
1300	0704	0705	0706	0707	0708	0709	0710	0711
1310	0712	0713	0714	0715	0716	0717	0718	0719
1320	0720	0721	0722	0723	0724	0725	0726	0727
1330	0728	0729	0730	0731	0732	0733	0734	0735
1340	0736	0737	0738	0739	0740	0741	0742	0743
1350	0744	0745	0746	0747	0748	0749	0750	0751
1360	0752	0753	0754	0755	0756	0757	0758	0759
1370	0760	0761	0762	0763	0764	0765	0766	0767

	0	1	2	3	4	5	6	7
1400	0768	0769	0770	0771	0772	0773	0774	0775
1410	0776	0777	0778	0779	0780	0781	0782	0783
1420	0784	0785	0786	0787	0788	0789	0790	0791
1430	0792	0793	0794	0795	0796	0797	0798	0799
1440	0800	0801	0802	0803	0804	0805	0806	0807
1450	0808	0809	0810	0811	0812	0813	0814	0815
1460	0816	0817	0818	0819	0820	0821	0822	0823
1470	0824	0825	0826	0827	0828	0829	0830	0831
1500	0832	0833	0834	0835	0836	0837	0838	0839
1510	0840	0841	0842	0843	0844	0845	0846	0847
1520	0848	0849	0850	0851	0852	0853	0854	0855
1530	0856	0857	0858	0859	0860	0861	0862	0863
1540	0864	0865	0866	0867	0868	0869	0870	0871
1550	0872	0873	0874	0875	0876	0877	0878	0879
1560	0880	0881	0882	0883	0884	0885	0886	0887
1570	0888	0889	0890	0891	0892	0893	0894	0895
1600	0896	0897	0898	0899	0900	0901	0902	0903
1610	0904	0905	0906	0907	0908	0909	0910	0911
1620	0912	0913	0914	0915	0916	0917	0918	0919
1630	0920	0921	0922	0923	0924	0925	0926	0927
1640	0928	0929	0930	0931	0932	0933	0934	0935
1650	0936	0937	0938	0939	0940	0941	0942	0943
1660	0944	0945	0946	0947	0948	0949	0950	0951
1670	0952	0953	0954	0955	0956	0957	0958	0959
1700	0960	0961	0962	0963	0964	0965	0966	0967
1710	0968	0969	0970	0971	0972	0973	0974	0975
1720	0976	0977	0978	0979	0980	0981	0982	0983
1730	0984	0985	0986	0987	0988	0989	0990	0991
1740	0992	0993	0994	0995	0996	0997	0998	0999
1750	1000	1001	1002	1003	1004	1005	1006	1007
1760	1008	1009	1010	1011	1012	1013	1014	1015
1770	1016	1017	1018	1019	1020	1021	1022	1023

OCTAL-DECIMAL INTEGER CONVERSION TABLE (Cont.)

	0	1	2	3	4	5	6	7
2000	1024	1025	1026	1027	1028	1029	1030	1031
2010	1032	1033	1034	1035	1036	1037	1038	1039
2020	1040	1041	1042	1043	1044	1045	1046	1047
2030	1048	1049	1050	1051	1052	1053	1054	1055
2040	1056	1057	1058	1059	1060	1061	1062	1063
2050	1064	1065	1066	1067	1068	1069	1070	1071
2060	1072	1073	1074	1075	1076	1077	1078	1079
2070	1080	1081	1082	1083	1084	1085	1086	1087
2100	1088	1089	1090	1091	1092	1093	1094	1095
2110	1096	1097	1098	1099	1100	1101	1102	1103
2120	1104	1105	1106	1107	1108	1109	1110	1111
2130	1112	1113	1114	1115	1116	1117	1118	1119
2140	1120	1121	1122	1123	1124	1125	1126	1127
2150	1128	1129	1130	1131	1132	1133	1134	1135
2160	1136	1137	1138	1139	1140	1141	1142	1143
2170	1144	1145	1146	1147	1148	1149	1150	1151
2200	1152	1153	1154	1155	1156	1157	1158	1159
2210	1160	1161	1162	1163	1164	1165	1166	1167
2220	1168	1169	1170	1171	1172	1173	1174	1175
2230	1176	1177	1178	1179	1180	1181	1182	1183
2240	1184	1185	1186	1187	1188	1189	1190	1191
2250	1192	1193	1194	1195	1196	1197	1198	1199
2260	1200	1201	1202	1203	1204	1205	1206	1207
2270	1208	1209	1210	1211	1212	1213	1214	1215
2300	1216	1217	1218	1219	1220	1221	1222	1223
2310	1224	1225	1226	1227	1228	1229	1230	1231
2320	1232	1233	1234	1235	1236	1237	1238	1239
2330	1240	1241	1242	1243	1244	1245	1246	1247
2340	1248	1249	1250	1251	1252	1253	1254	1255
2350	1256	1257	1258	1259	1260	1261	1262	1263
2360	1264	1265	1266	1267	1268	1269	1270	1271
2370	1272	1273	1274	1275	1276	1277	1278	1279

	0	1	2	3	4	5	6	7
2400	1280	1281	1282	1283	1284	1285	1286	1287
2410	1288	1289	1290	1291	1292	1293	1294	1295
2420	1296	1297	1298	1299	1300	1301	1302	1303
2430	1304	1305	1306	1307	1308	1309	1310	1311
2440	1312	1313	1314	1315	1316	1317	1318	1319
2450	1320	1321	1322	1323	1324	1325	1326	1327
2460	1328	1329	1330	1331	1332	1333	1334	1335
2470	1336	1337	1338	1339	1340	1341	1342	1343
2500	1344	1345	1346	1347	1348	1349	1350	1351
2510	1352	1353	1354	1355	1356	1357	1358	1359
2520	1360	1361	1362	1363	1364	1365	1366	1367
2530	1368	1369	1370	1371	1372	1373	1374	1375
2540	1376	1377	1378	1379	1380	1381	1382	1383
2550	1384	1385	1386	1387	1388	1389	1390	1391
2560	1392	1393	1394	1395	1396	1397	1398	1399
2570	1400	1401	1402	1403	1404	1405	1406	1407
2600	1408	1409	1410	1411	1412	1413	1414	1415
2610	1416	1417	1418	1419	1420	1421	1422	1423
2620	1424	1425	1426	1427	1428	1429	1430	1431
2630	1432	1433	1434	1435	1436	1437	1438	1439
2640	1440	1441	1442	1443	1444	1445	1446	1447
2650	1448	1449	1450	1451	1452	1453	1454	1455
2660	1456	1457	1458	1459	1460	1461	1462	1463
2670	1464	1465	1466	1467	1468	1469	1470	1471
2700	1472	1473	1474	1475	1476	1477	1478	1479
2710	1480	1481	1482	1483	1484	1485	1486	1487
2720	1488	1489	1490	1491	1492	1493	1494	1495
2730	1496	1497	1498	1499	1500	1501	1502	1503
2740	1504	1505	1506	1507	1508	1509	1510	1511
2750	1512	1513	1514	1515	1516	1517	1518	1519
2760	1520	1521	1522	1523	1524	1525	1526	1527
2770	1528	1529	1530	1531	1532	1533	1534	1535

2000 to 2777 (Octal)	1024 to 1535 (Decimal)

Octal	Decimal
10000 -	4096
20000 -	8192
30000 -	12288
40000 -	16384
50000 -	20480
60000 -	24576
70000 -	28672

	0	1	2	3	4	5	6	7
3000	1536	1537	1538	1539	1540	1541	1542	1543
3010	1544	1545	1546	1547	1548	1549	1550	1551
3020	1552	1553	1554	1555	1556	1557	1558	1559
3030	1560	1561	1562	1563	1564	1565	1566	1567
3040	1568	1569	1570	1571	1572	1573	1574	1575
3050	1576	1577	1578	1579	1580	1581	1582	1583
3060	1584	1585	1586	1587	1588	1589	1590	1591
3070	1592	1593	1594	1595	1596	1597	1598	1599
3100	1600	1601	1602	1603	1604	1605	1606	1607
3110	1608	1609	1610	1611	1612	1613	1614	1615
3120	1616	1617	1618	1619	1620	1621	1622	1623
3130	1624	1625	1626	1627	1628	1629	1630	1631
3140	1632	1633	1634	1635	1636	1637	1638	1639
3150	1640	1641	1642	1643	1644	1645	1646	1647
3160	1648	1649	1650	1651	1652	1653	1654	1655
3170	1656	1657	1658	1659	1660	1661	1662	1663
3200	1664	1665	1666	1667	1668	1669	1670	1671
3210	1672	1673	1674	1675	1676	1677	1678	1679
3220	1680	1681	1682	1683	1684	1685	1686	1687
3230	1688	1689	1690	1691	1692	1693	1694	1695
3240	1696	1697	1698	1699	1700	1701	1702	1703
3250	1704	1705	1706	1707	1708	1709	1710	1711
3260	1712	1713	1714	1715	1716	1717	1718	1719
3270	1720	1721	1722	1723	1724	1725	1726	1727
3300	1728	1729	1730	1731	1732	1733	1734	1735
3310	1736	1737	1738	1739	1740	1741	1742	1743
3320	1744	1745	1746	1747	1748	1749	1750	1751
3330	1752	1753	1754	1755	1756	1757	1758	1759
3340	1760	1761	1762	1763	1764	1765	1766	1767
3350	1768	1769	1770	1771	1772	1773	1774	1775
3360	1776	1777	1778	1779	1780	1781	1782	1783
3370	1784	1785	1786	1787	1788	1789	1790	1791

	0	1	2	3	4	5	6	7
3400	1792	1793	1794	1795	1796	1797	1798	1799
3410	1800	1801	1802	1803	1804	1805	1806	1807
3420	1808	1809	1810	1811	1812	1813	1814	1815
3430	1816	1817	1818	1819	1820	1821	1822	1823
3440	1824	1825	1826	1827	1828	1829	1830	1831
3450	1832	1833	1834	1835	1836	1837	1838	1839
3460	1840	1841	1842	1843	1844	1845	1846	1847
3470	1848	1849	1850	1851	1852	1853	1854	1855
3500	1856	1857	1858	1859	1860	1861	1862	1863
3510	1864	1865	1866	1867	1868	1869	1870	1871
3520	1872	1873	1874	1875	1876	1877	1878	1879
3530	1880	1881	1882	1883	1884	1885	1886	1887
3540	1888	1889	1890	1891	1892	1893	1894	1895
3550	1896	1897	1898	1899	1900	1901	1902	1903
3560	1904	1905	1906	1907	1908	1909	1910	1911
3570	1912	1913	1914	1915	1916	1917	1918	1919
3600	1920	1921	1922	1923	1924	1925	1926	1927
3610	1928	1929	1930	1931	1932	1933	1934	1935
3620	1936	1937	1938	1939	1940	1941	1942	1943
3630	1944	1945	1946	1947	1948	1949	1950	1951
3640	1952	1953	1954	1955	1956	1957	1958	1959
3650	1960	1961	1962	1963	1964	1965	1966	1967
3660	1968	1969	1970	1971	1972	1973	1974	1975
3670	1976	1977	1978	1979	1980	1981	1982	1983
3700	1984	1985	1986	1987	1988	1989	1990	1991
3710	1992	1993	1994	1995	1996	1997	1998	1999
3720	2000	2001	2002	2003	2004	2005	2006	2007
3730	2008	2009	2010	2011	2012	2013	2014	2015
3740	2016	2017	2018	2019	2020	2021	2022	2023
3750	2024	2025	2026	2027	2028	2029	2030	2031
3760	2032	2033	2034	2035	2036	2037	2038	2039
3770	2040	2041	2042	2043	2044	2045	2046	2047

3000 to 3777 (Octal)	1536 to 2047 (Decimal)

OCTAL-DECIMAL INTEGER CONVERSION TABLE (Cont.)

4000 (Octal)	2048 (Decimal)
to	to
4777 (Octal)	2559 (Decimal)

Octal	Decimal
10000 -	4096
20000 -	8192
30000 -	12288
40000 -	16384
50000 -	20480
60000 -	24576
70000 -	28672

	0	1	2	3	4	5	6	7
4000	2048	2049	2050	2051	2052	2053	2054	2055
4010	2056	2057	2058	2059	2060	2061	2062	2063
4020	2064	2065	2066	2067	2068	2069	2070	2071
4030	2072	2073	2074	2075	2076	2077	2078	2079
4040	2080	2081	2082	2083	2084	2085	2086	2087
4050	2088	2089	2090	2091	2092	2093	2094	2095
4060	2096	2097	2098	2099	2100	2101	2102	2103
4070	2104	2105	2106	2107	2108	2109	2110	2111
4100	2112	2113	2114	2115	2116	2117	2118	2119
4110	2120	2121	2122	2123	2124	2125	2126	2127
4120	2128	2129	2130	2131	2132	2133	2134	2135
4130	2136	2137	2138	2139	2140	2141	2142	2143
4140	2144	2145	2146	2147	2148	2149	2150	2151
4150	2152	2153	2154	2155	2156	2157	2158	2159
4160	2160	2161	2162	2163	2164	2165	2166	2167
4170	2168	2169	2170	2171	2172	2173	2174	2175
4200	2176	2177	2178	2179	2180	2181	2182	2183
4210	2184	2185	2186	2187	2188	2189	2190	2191
4220	2192	2193	2194	2195	2196	2197	2198	2199
4230	2200	2201	2202	2203	2204	2205	2206	2207
4240	2208	2209	2210	2211	2212	2213	2214	2215
4250	2216	2217	2218	2219	2220	2221	2222	2223
4260	2224	2225	2226	2227	2228	2229	2230	2231
4270	2232	2233	2234	2235	2236	2237	2238	2239
4300	2240	2241	2242	2243	2244	2245	2246	2247
4310	2248	2249	2250	2251	2252	2253	2254	2255
4320	2256	2257	2258	2259	2260	2261	2262	2263
4330	2264	2265	2266	2267	2268	2269	2270	2271
4340	2272	2273	2274	2275	2276	2277	2278	2279
4350	2280	2281	2282	2283	2284	2285	2286	2287
4360	2288	2289	2290	2291	2292	2293	2294	2295
4370	2296	2297	2298	2299	2300	2301	2302	2303

	0	1	2	3	4	5	6	7
4400	2304	2305	2306	2307	2308	2309	2310	2311
4410	2312	2313	2314	2315	2316	2317	2318	2319
4420	2320	2321	2322	2323	2324	2325	2326	2327
4430	2328	2329	2330	2331	2332	2333	2334	2335
4440	2336	2337	2338	2339	2340	2341	2342	2343
4450	2344	2345	2346	2347	2348	2349	2350	2351
4460	2352	2353	2354	2355	2356	2357	2358	2359
4470	2360	2361	2362	2363	2364	2365	2366	2367
4500	2368	2369	2370	2371	2372	2373	2374	2375
4510	2376	2377	2378	2379	2380	2381	2382	2383
4520	2384	2385	2386	2387	2388	2389	2390	2391
4530	2392	2393	2394	2395	2396	2397	2398	2399
4540	2400	2401	2402	2403	2404	2405	2406	2407
4550	2408	2409	2410	2411	2412	2413	2414	2415
4560	2416	2417	2418	2419	2420	2421	2422	2423
4570	2424	2425	2426	2427	2428	2429	2430	2431
4600	2432	2433	2434	2435	2436	2437	2438	2439
4610	2440	2441	2442	2443	2444	2445	2446	2447
4620	2448	2449	2450	2451	2452	2453	2454	2455
4630	2456	2457	2458	2459	2460	2461	2462	2463
4640	2464	2465	2466	2467	2468	2469	2470	2471
4650	2472	2473	2474	2475	2476	2477	2478	2479
4660	2480	2481	2482	2483	2484	2485	2486	2487
4670	2488	2489	2490	2491	2492	2493	2494	2495
4700	2496	2497	2498	2499	2500	2501	2502	2503
4710	2504	2505	2506	2507	2508	2509	2510	2511
4720	2512	2513	2514	2515	2516	2517	2518	2519
4730	2520	2521	2522	2523	2524	2525	2526	2527
4740	2528	2529	2530	2531	2532	2533	2534	2535
4750	2536	2537	2538	2539	2540	2541	2542	2543
4760	2544	2545	2546	2547	2548	2549	2550	2551
4770	2552	2553	2554	2555	2556	2557	2558	2559

5000 (Octal)	2560 (Decimal)
to	to
5777 (Octal)	3071 (Decimal)

	0	1	2	3	4	5	6	7
5000	2560	2561	2562	2563	2564	2565	2566	2567
5010	2568	2569	2570	2571	2572	2573	2574	2575
5020	2576	2577	2578	2579	2580	2581	2582	2583
5030	2584	2585	2586	2587	2588	2589	2590	2591
5040	2592	2593	2594	2595	2596	2597	2598	2599
5050	2600	2601	2602	2603	2604	2605	2606	2607
5060	2608	2609	2610	2611	2612	2613	2614	2615
5070	2616	2617	2618	2619	2620	2621	2622	2623
5100	2624	2625	2626	2627	2628	2629	2630	2631
5110	2632	2633	2634	2635	2636	2637	2638	2639
5120	2640	2641	2642	2643	2644	2645	2646	2647
5130	2648	2649	2650	2651	2652	2653	2654	2655
5140	2656	2657	2658	2659	2660	2661	2662	2663
5150	2664	2665	2666	2667	2668	2669	2670	2671
5160	2672	2673	2674	2675	2676	2677	2678	2679
5170	2680	2681	2682	2683	2684	2685	2686	2687
5200	2688	2689	2690	2691	2692	2693	2694	2695
5210	2696	2697	2698	2699	2700	2701	2702	2703
5220	2704	2705	2706	2707	2708	2709	2710	2711
5230	2712	2713	2714	2715	2716	2717	2718	2719
5240	2720	2721	2722	2723	2724	2725	2726	2727
5250	2728	2729	2730	2731	2732	2733	2734	2735
5260	2736	2737	2738	2739	2740	2741	2742	2743
5270	2744	2745	2746	2747	2748	2749	2750	2751
5300	2752	2753	2754	2755	2756	2757	2758	2759
5310	2760	2761	2762	2763	2764	2765	2766	2767
5320	2768	2769	2770	2771	2772	2773	2774	2775
5330	2776	2777	2778	2779	2780	2781	2782	2783
5340	2784	2785	2786	2787	2788	2789	2790	2791
5350	2792	2793	2794	2795	2796	2797	2798	2799
5360	2800	2801	2802	2803	2804	2805	2806	2807
5370	2808	2809	2810	2811	2812	2813	2814	2815

	0	1	2	3	4	5	6	7
5400	2816	2817	2818	2819	2820	2821	2822	2823
5410	2824	2825	2826	2827	2828	2829	2830	2831
5420	2832	2833	2834	2835	2836	2837	2838	2839
5430	2840	2841	2842	2843	2844	2845	2846	2847
5440	2848	2849	2850	2851	2852	2853	2854	2855
5450	2856	2857	2858	2859	2860	2861	2862	2863
5460	2864	2865	2866	2867	2868	2869	2870	2871
5470	2872	2873	2874	2875	2876	2877	2878	2879
5500	2880	2881	2882	2883	2884	2885	2886	2887
5510	2888	2889	2890	2891	2892	2893	2894	2895
5520	2896	2897	2898	2899	2900	2901	2902	2903
5530	2904	2905	2906	2907	2908	2909	2910	2911
5540	2912	2913	2914	2915	2916	2917	2918	2919
5550	2920	2921	2922	2923	2924	2925	2926	2927
5560	2928	2929	2930	2931	2932	2933	2934	2935
5570	2936	2937	2938	2939	2940	2941	2942	2943
5600	2944	2945	2946	2947	2948	2949	2950	2951
5610	2952	2953	2954	2955	2956	2957	2958	2959
5620	2960	2961	2962	2963	2964	2965	2966	2967
5630	2968	2969	2970	2971	2972	2973	2974	2975
5640	2976	2977	2978	2979	2980	2981	2982	2983
5650	2984	2985	2986	2987	2988	2989	2990	2991
5660	2992	2993	2994	2995	2996	2997	2998	2999
5670	3000	3001	3002	3003	3004	3005	3006	3007
5700	3008	3009	3010	3011	3012	3013	3014	3015
5710	3016	3017	3018	3019	3020	3021	3022	3023
5720	3024	3025	3026	3027	3028	3029	3030	3031
5730	3032	3033	3034	3035	3036	3037	3038	3039
5740	3040	3041	3042	3043	3044	3045	3046	3047
5750	3048	3049	3050	3051	3052	3053	3054	3055
5760	3056	3057	3058	3059	3060	3061	3062	3063
5770	3064	3065	3066	3067	3068	3069	3070	3071

OCTAL-DECIMAL INTEGER CONVERSION TABLE (Cont.)

	0	1	2	3	4	5	6	7
6000	3072	3073	3074	3075	3076	3077	3078	3079
6010	3080	3081	3082	3083	3084	3085	3086	3087
6020	3088	3089	3090	3091	3092	3093	3094	3095
6030	3096	3097	3098	3099	3100	3101	3102	3103
6040	3104	3105	3106	3107	3108	3109	3110	3111
6050	3112	3113	3114	3115	3116	3117	3118	3119
6060	3120	3121	3122	3123	3124	3125	3126	3127
6070	3128	3129	3130	3131	3132	3133	3134	3135
6100	3136	3137	3138	3139	3140	3141	3142	3143
6110	3144	3145	3146	3147	3148	3149	3150	3151
6120	3152	3153	3154	3155	3156	3157	3158	3159
6130	3160	3161	3162	3163	3164	3165	3166	3167
6140	3168	3169	3170	3171	3172	3173	3174	3175
6150	3176	3177	3178	3179	3180	3181	3182	3183
6160	3184	3185	3186	3187	3188	3189	3190	3191
6170	3192	3193	3194	3195	3196	3197	3198	3199
6200	3200	3201	3202	3203	3204	3205	3206	3207
6210	3208	3209	3210	3211	3212	3213	3214	3215
6220	3216	3217	3218	3219	3220	3221	3222	3223
6230	3224	3225	3226	3227	3228	3229	3230	3231
6240	3232	3233	3234	3235	3236	3237	3238	3239
6250	3240	3241	3242	3243	3244	3245	3246	3247
6260	3248	3249	3250	3251	3252	3253	3254	3255
6270	3256	3257	3258	3259	3260	3261	3262	3263
6300	3264	3265	3266	3267	3268	3269	3270	3271
6310	3272	3273	3274	3275	3276	3277	3278	3279
6320	3280	3281	3282	3283	3284	3285	3286	3287
6330	3288	3289	3290	3291	3292	3293	3294	3295
6340	3296	3297	3298	3299	3300	3301	3302	3303
6350	3304	3305	3306	3307	3308	3309	3310	3311
6360	3312	3313	3314	3315	3316	3317	3318	3319
6370	3320	3321	3322	3323	3324	3325	3326	3327

	0	1	2	3	4	5	6	7
6400	3328	3329	3330	3331	3332	3333	3334	3335
6410	3336	3337	3338	3339	3340	3341	3342	3343
6420	3344	3345	3346	3347	3348	3349	3350	3351
6430	3352	3353	3354	3355	3356	3357	3358	3359
6440	3360	3361	3362	3363	3364	3365	3366	3367
6450	3368	3369	3370	3371	3372	3373	3374	3375
6460	3376	3377	3378	3379	3380	3381	3382	3383
6470	3384	3385	3386	3387	3388	3389	3390	3391
6500	3392	3393	3394	3395	3396	3397	3398	3399
6510	3400	3401	3402	3403	3404	3405	3406	3407
6520	3408	3409	3410	3411	3412	3413	3414	3415
6530	3416	3417	3418	3419	3420	3421	3422	3423
6540	3424	3425	3426	3427	3428	3429	3430	3431
6550	3432	3433	3434	3435	3436	3437	3438	3439
6560	3440	3441	3442	3443	3444	3445	3446	3447
6570	3448	3449	3450	3451	3452	3453	3454	3455
6600	3456	3457	3458	3459	3460	3461	3462	3463
6610	3464	3465	3466	3467	3468	3469	3470	3471
6620	3472	3473	3474	3475	3476	3477	3478	3479
6630	3480	3481	3482	3483	3484	3485	3486	3487
6640	3488	3489	3490	3491	3492	3493	3494	3495
6650	3496	3497	3498	3499	3500	3501	3502	3503
6660	3504	3505	3506	3507	3508	3509	3510	3511
6670	3512	3513	3514	3515	3516	3517	3518	3519
6700	3520	3521	3522	3523	3524	3525	3526	3527
6710	3528	3529	3530	3531	3532	3533	3534	3535
6720	3536	3537	3538	3539	3540	3541	3542	3543
6730	3544	3545	3546	3547	3548	3549	3550	3551
6740	3552	3553	3554	3555	3556	3557	3558	3559
6750	3560	3561	3562	3563	3564	3565	3566	3567
6760	3568	3569	3570	3571	3572	3573	3574	3575
6770	3576	3577	3578	3579	3580	3581	3582	3583

6000	3072
to	to
6777	3583
(Octal)	(Decimal)

Octal	Decimal
10000 -	4096
20000 -	8192
30000 -	12288
40000 -	16384
50000 -	20480
60000 -	24576
70000 -	28672

	0	1	2	3	4	5	6	7
7000	3584	3585	3586	3587	3588	3589	3590	3591
7010	3592	3593	3594	3595	3596	3597	3598	3599
7020	3600	3601	3602	3603	3604	3605	3606	3607
7030	3608	3609	3610	3611	3612	3613	3614	3615
7040	3616	3617	3618	3619	3620	3621	3622	3623
7050	3624	3625	3626	3627	3628	3629	3630	3631
7060	3632	3633	3634	3635	3636	3637	3638	3639
7070	3640	3641	3642	3643	3644	3645	3646	3647
7100	3648	3649	3650	3651	3652	3653	3654	3655
7110	3656	3657	3658	3659	3660	3661	3662	3663
7120	3664	3665	3666	3667	3668	3669	3670	3671
7130	3672	3673	3674	3675	3676	3677	3678	3679
7140	3680	3681	3682	3683	3684	3685	3686	3687
7150	3688	3689	3690	3691	3692	3693	3694	3695
7160	3696	3697	3698	3699	3700	3701	3702	3703
7170	3704	3705	3706	3707	3708	3709	3710	3711
7200	3712	3713	3714	3715	3716	3717	3718	3719
7210	3720	3721	3722	3723	3724	3725	3726	3727
7220	3728	3729	3730	3731	3732	3733	3734	3735
7230	3736	3737	3738	3739	3740	3741	3742	3743
7240	3744	3745	3746	3747	3748	3749	3750	3751
7250	3752	3753	3754	3755	3756	3757	3758	3759
7260	3760	3761	3762	3763	3764	3765	3766	3767
7270	3768	3769	3770	3771	3772	3773	3774	3775
7300	3776	3777	3778	3779	3780	3781	3782	3783
7310	3784	3785	3786	3787	3788	3789	3790	3791
7320	3792	3793	3794	3795	3796	3797	3798	3799
7330	3800	3801	3802	3803	3804	3805	3806	3807
7340	3808	3809	3810	3811	3812	3813	3814	3815
7350	3816	3817	3818	3819	3820	3821	3822	3823
7360	3824	3825	3826	3827	3828	3829	3830	3831
7370	3832	3833	3834	3835	3836	3837	3838	3839

	0	1	2	3	4	5	6	7
7400	3840	3841	3842	3843	3844	3845	3846	3847
7410	3848	3849	3850	3851	3852	3853	3854	3855
7420	3856	3857	3858	3859	3860	3861	3862	3863
7430	3864	3865	3866	3867	3868	3869	3870	3871
7440	3872	3873	3874	3875	3876	3877	3878	3879
7450	3880	3881	3882	3883	3884	3885	3886	3887
7460	3888	3889	3890	3891	3892	3893	3894	3895
7470	3896	3897	3898	3899	3900	3901	3902	3903
7500	3904	3905	3906	3907	3908	3909	3910	3911
7510	3912	3913	3914	3915	3916	3917	3918	3919
7520	3920	3921	3922	3923	3924	3925	3926	3927
7530	3928	3929	3930	3931	3932	3933	3934	3935
7540	3936	3937	3938	3939	3940	3941	3942	3943
7550	3944	3945	3946	3947	3948	3949	3950	3951
7560	3952	3953	3954	3955	3956	3957	3958	3959
7570	3960	3961	3962	3963	3964	3965	3966	3967
7600	3968	3969	3970	3971	3972	3973	3974	3975
7610	3976	3977	3978	3979	3980	3981	3982	3983
7620	3984	3985	3986	3987	3988	3989	3990	3991
7630	3992	3993	3994	3995	3996	3997	3998	3999
7640	4000	4001	4002	4003	4004	4005	4006	4007
7650	4008	4009	4010	4011	4012	4013	4014	4015
7660	4016	4017	4018	4019	4020	4021	4022	4023
7670	4024	4025	4026	4027	4028	4029	4030	4031
7700	4032	4033	4034	4035	4036	4037	4038	4039
7710	4040	4041	4042	4043	4044	4045	4046	4047
7720	4048	4049	4050	4051	4052	4053	4054	4055
7730	4056	4057	4058	4059	4060	4061	4062	4063
7740	4064	4065	4066	4067	4068	4069	4070	4071
7750	4072	4073	4074	4075	4076	4077	4078	4079
7760	4080	4081	4082	4083	4084	4085	4086	4087
7770	4088	4089	4090	4091	4092	4093	4094	4095

7000	3584
to	to
7777	4095
(Octal)	(Decimal)

OCTAL-DECIMAL FRACTION CONVERSION TABLE

OCTAL	DEC.	OCTAL	DEC.	OCTAL	DEC.	OCTAL	DEC.
.000	.000000	.100	.125000	.200	.250000	.300	.375000
.001	.001953	.101	.126953	.201	.251953	.301	.376953
.002	.003906	.102	.128906	.202	.253906	.302	.378906
.003	.005859	.103	.130859	.203	.255859	.303	.380859
.004	.007812	.104	.132812	.204	.257812	.304	.382812
.005	.009765	.105	.134765	.205	.259765	.305	.384765
.006	.011718	.106	.136718	.206	.261718	.306	.386718
.007	.013671	.107	.138671	.207	.263671	.307	.388671
.010	.015625	.110	.140625	.210	.265625	.310	.390625
.011	.017578	.111	.142578	.211	.267578	.311	.392578
.012	.019531	.112	.144531	.212	.269531	.312	.394531
.013	.021484	.113	.146484	.213	.271484	.313	.396484
.014	.023437	.114	.148437	.214	.273437	.314	.398437
.015	.025390	.115	.150390	.215	.275390	.315	.400390
.016	.027343	.116	.152343	.216	.277343	.316	.402343
.017	.029296	.117	.154296	.217	.279296	.317	.404296
.020	.031250	.120	.156250	.220	.281250	.320	.406250
.021	.033203	.121	.158203	.221	.283203	.321	.408203
.022	.035156	.122	.160156	.222	.285156	.322	.410156
.023	.037109	.123	.162109	.223	.287109	.323	.412109
.024	.039062	.124	.164062	.224	.289062	.324	.414062
.025	.041015	.125	.166015	.225	.291015	.325	.416015
.026	.042968	.126	.167968	.226	.292968	.326	.417968
.027	.044921	.127	.169921	.227	.294921	.327	.419921
.030	.046875	.130	.171875	.230	.296875	.330	.421875
.031	.048828	.131	.173828	.231	.298828	.331	.423828
.032	.050781	.132	.175781	.232	.300781	.332	.425781
.033	.052734	.133	.177734	.233	.302734	.333	.427734
.034	.054687	.134	.179687	.234	.304687	.334	.429687
.035	.056640	.135	.181640	.235	.306640	.335	.431640
.036	.058593	.136	.183593	.236	.308593	.336	.433593
.037	.060546	.137	.185546	.237	.310546	.337	.435546
.040	.062500	.140	.187500	.240	.312500	.340	.437500
.041	.064453	.141	.189453	.241	.314453	.341	.439453
.042	.066406	.142	.191406	.242	.316406	.342	.441406
.043	.068359	.143	.193359	.243	.318359	.343	.443359
.044	.070312	.144	.195312	.244	.320312	.344	.445312
.045	.072265	.145	.197265	.245	.322265	.345	.447265
.046	.074218	.146	.199218	.246	.324218	.346	.449218
.047	.076171	.147	.201171	.247	.326171	.347	.451171
.050	.078125	.150	.203125	.250	.328125	.350	.453125
.051	.080078	.151	.205078	.251	.330078	.351	.455078
.052	.082031	.152	.207031	.252	.332031	.352	.457031
.053	.083984	.153	.208984	.253	.333984	.353	.458984
.054	.085937	.154	.210937	.254	.335937	.354	.460937
.055	.087890	.155	.212890	.255	.337890	.355	.462890
.056	.089843	.156	.214843	.256	.339843	.356	.464843
.057	.091796	.157	.216796	.257	.341796	.357	.466796
.060	.093750	.160	.218750	.260	.343750	.360	.468750
.061	.095703	.161	.220703	.261	.345703	.361	.470703
.062	.097656	.162	.222656	.262	.347656	.362	.472656
.063	.099609	.163	.224609	.263	.349609	.363	.474609
.064	.101562	.164	.226562	.264	.351562	.364	.476562
.065	.103515	.165	.228515	.265	.353515	.365	.478515
.066	.105468	.166	.230468	.266	.355468	.366	.480468
.067	.107421	.167	.232421	.267	.357421	.367	.482421
.070	.109375	.170	.234375	.270	.359375	.370	.484375
.071	.111328	.171	.236328	.271	.361328	.371	.486328
.072	.113281	.172	.238281	.272	.363281	.372	.488281
.073	.115234	.173	.240234	.273	.365234	.373	.490234
.074	.117187	.174	.242187	.274	.367187	.374	.492187
.075	.119140	.175	.244140	.275	.369140	.375	.494140
.076	.121093	.176	.246093	.276	.371093	.376	.496093
.077	.123046	.177	.248046	.277	.373046	.377	.498046

OCTAL-DECIMAL FRACTION CONVERSION TABLE (Cont.)

OCTAL	DEC.	OCTAL	DEC.	OCTAL	DEC.	OCTAL	DEC.
.000000	.000000	.000100	.000244	.000200	.000488	.000300	.000732
.000001	.000003	.000101	.000247	.000201	.000492	.000301	.000736
.000002	.000007	.000102	.000251	.000202	.000495	.000302	.000740
.000003	.000011	.000103	.000255	.000203	.000499	.000303	.000743
.000004	.000015	.000104	.000259	.000204	.000503	.000304	.000747
.000005	.000019	.000105	.000263	.000205	.000507	.000305	.000751
.000006	.000022	.000106	.000267	.000206	.000511	.000306	.000755
.000007	.000026	.000107	.000270	.000207	.000514	.000307	.000759
.000010	.000030	.000110	.000274	.000210	.000518	.000310	.000762
.000011	.000034	.000111	.000278	.000211	.000522	.000311	.000766
.000012	.000038	.000112	.000282	.000212	.000526	.000312	.000770
.000013	.000041	.000113	.000286	.000213	.000530	.000313	.000774
.000014	.000045	.000114	.000289	.000214	.000534	.000314	.000778
.000015	.000049	.000115	.000293	.000215	.000537	.000315	.000782
.000016	.000053	.000116	.000297	.000216	.000541	.000316	.000785
.000017	.000057	.000117	.000301	.000217	.000545	.000317	.000789
.000020	.000061	.000120	.000305	.000220	.000549	.000320	.000793
.000021	.000064	.000121	.000308	.000221	.000553	.000321	.000797
.000022	.000068	.000122	.000312	.000222	.000556	.000322	.000801
.000023	.000072	.000123	.000316	.000223	.000560	.000323	.000805
.000024	.000076	.000124	.000320	.000224	.000564	.000324	.000808
.000025	.000080	.000125	.000324	.000225	.000568	.000325	.000812
.000026	.000083	.000126	.000328	.000226	.000572	.000326	.000816
.000027	.000087	.000127	.000331	.000227	.000576	.000327	.000820
.000030	.000091	.000130	.000335	.000230	.000579	.000330	.000823
.000031	.000095	.000131	.000339	.000231	.000583	.000331	.000827
.000032	.000099	.000132	.000343	.000232	.000587	.000332	.000831
.000033	.000102	.000133	.000347	.000233	.000591	.000333	.000835
.000034	.000106	.000134	.000350	.000234	.000595	.000334	.000839
.000035	.000110	.000135	.000354	.000235	.000598	.000335	.000843
.000036	.000114	.000136	.000358	.000236	.000602	.000336	.000846
.000037	.000118	.000137	.000362	.000237	.000606	.000337	.000850
.000040	.000122	.000140	.000366	.000240	.000610	.000340	.000854
.000041	.000125	.000141	.000370	.000241	.000614	.000341	.000858
.000042	.000129	.000142	.000373	.000242	.000617	.000342	.000862
.000043	.000133	.000143	.000377	.000243	.000621	.000343	.000865
.000044	.000137	.000144	.000381	.000244	.000625	.000344	.000869
.000045	.000141	.000145	.000385	.000245	.000629	.000345	.000873
.000046	.000144	.000146	.000389	.000246	.000633	.000346	.000877
.000047	.000148	.000147	.000392	.000247	.000637	.000347	.000881
.000050	.000152	.000150	.000396	.000250	.000640	.000350	.000885
.000051	.000156	.000151	.000400	.000251	.000644	.000351	.000888
.000052	.000160	.000152	.000404	.000252	.000648	.000352	.000892
.000053	.000164	.000153	.000408	.000253	.000652	.000353	.000896
.000054	.000167	.000154	.000411	.000254	.000656	.000354	.000900
.000055	.000171	.000155	.000415	.000255	.000659	.000355	.000904
.000056	.000175	.000156	.000419	.000256	.000663	.000356	.000907
.000057	.000179	.000157	.000423	.000257	.000667	.000357	.000911
.000060	.000183	.000160	.000427	.000260	.000671	.000360	.000915
.000061	.000186	.000161	.000431	.000261	.000675	.000361	.000919
.000062	.000190	.000162	.000434	.000262	.000679	.000362	.000923
.000063	.000194	.000163	.000438	.000263	.000682	.000363	.000926
.000064	.000198	.000164	.000442	.000264	.000686	.000364	.000930
.000065	.000202	.000165	.000446	.000265	.000690	.000365	.000934
.000066	.000205	.000166	.000450	.000266	.000694	.000366	.000938
.000067	.000209	.000167	.000453	.000267	.000698	.000367	.000942
.000070	.000213	.000170	.000457	.000270	.000701	.000370	.000946
.000071	.000217	.000171	.000461	.000271	.000705	.000371	.000949
.000072	.000221	.000172	.000465	.000272	.000709	.000372	.000953
.000073	.000225	.000173	.000469	.000273	.000713	.000373	.000957
.000074	.000228	.000174	.000473	.000274	.000717	.000374	.000961
.000075	.000232	.000175	.000476	.000275	.000720	.000375	.000965
.000076	.000236	.000176	.000480	.000276	.000724	.000376	.000968
.000077	.000240	.000177	.000484	.000277	.000728	.000377	.000972

OCTAL-DECIMAL FRACTION CONVERSION TABLE (Cont.)

OCTAL	DEC.	OCTAL	DEC.	OCTAL	DEC.	OCTAL	DEC.
.000400	.000976	.000500	.001220	.000600	.001464	.000700	.001708
.000401	.000980	.000501	.001224	.000601	.001468	.000701	.001712
.000402	.000984	.000502	.001228	.000602	.001472	.000702	.001716
.000403	.000988	.000503	.001232	.000603	.001476	.000703	.001720
.000404	.000991	.000504	.001235	.000604	.001480	.000704	.001724
.000405	.000995	.000505	.001239	.000605	.001483	.000705	.001728
.000406	.000999	.000506	.001243	.000606	.001487	.000706	.001731
.000407	.001003	.000507	.001247	.000607	.001491	.000707	.001735
.000410	.001007	.000510	.001251	.000610	.001495	.000710	.001739
.000411	.001010	.000511	.001255	.000611	.001499	.000711	.001743
.000412	.001014	.000512	.001258	.000612	.001502	.000712	.001747
.000413	.001018	.000513	.001262	.000613	.001506	.000713	.001750
.000414	.001022	.000514	.001266	.000614	.001510	.000714	.001754
.000415	.001026	.000515	.001270	.000615	.001514	.000715	.001758
.000416	.001029	.000516	.001274	.000616	.001518	.000716	.001762
.000417	.001033	.000517	.001277	.000617	.001522	.000717	.001766
.000420	.001037	.000520	.001281	.000620	.001525	.000720	.001770
.000421	.001041	.000521	.001285	.000621	.001529	.000721	.001773
.000422	.001045	.000522	.001289	.000622	.001533	.000722	.001777
.000423	.001049	.000523	.001293	.000623	.001537	.000723	.001781
.000424	.001052	.000524	.001296	.000624	.001541	.000724	.001785
.000425	.001056	.000525	.001300	.000625	.001544	.000725	.001789
.000426	.001060	.000526	.001304	.000626	.001548	.000726	.001792
.000427	.001064	.000527	.001308	.000627	.001552	.000727	.001796
.000430	.001068	.000530	.001312	.000630	.001556	.000730	.001800
.000431	.001071	.000531	.001316	.000631	.001560	.000731	.001804
.000432	.001075	.000532	.001319	.000632	.001564	.000732	.001808
.000433	.001079	.000533	.001323	.000633	.001567	.000733	.001811
.000434	.001083	.000534	.001327	.000634	.001571	.000734	.001815
.000435	.001087	.000535	.001331	.000635	.001575	.000735	.001819
.000436	.001091	.000536	.001335	.000636	.001579	.000736	.001823
.000437	.001094	.000537	.001338	.000637	.001583	.000737	.001827
.000440	.001098	.000540	.001342	.000640	.001586	.000740	.001831
.000441	.001102	.000541	.001346	.000641	.001590	.000741	.001834
.000442	.001106	.000542	.001350	.000642	.001594	.000742	.001838
.000443	.001110	.000543	.001354	.000643	.001598	.000743	.001842
.000444	.001113	.000544	.001358	.000644	.001602	.000744	.001846
.000445	.001117	.000545	.001361	.000645	.001605	.000745	.001850
.000446	.001121	.000546	.001365	.000646	.001609	.000746	.001853
.000447	.001125	.000547	.001369	.000647	.001613	.000747	.001857
.000450	.001129	.000550	.001373	.000650	.001617	.000750	.001861
.000451	.001132	.000551	.001377	.000651	.001621	.000751	.001865
.000452	.001136	.000552	.001380	.000652	.001625	.000752	.001869
.000453	.001140	.000553	.001384	.000653	.001628	.000753	.001873
.000454	.001144	.000554	.001388	.000654	.001632	.000754	.001876
.000455	.001148	.000555	.001392	.000655	.001636	.000755	.001880
.000456	.001152	.000556	.001396	.000656	.001640	.000756	.001884
.000457	.001155	.000557	.001399	.000657	.001644	.000757	.001888
.000460	.001159	.000560	.001403	.000660	.001647	.000760	.001892
.000461	.001163	.000561	.001407	.000661	.001651	.000761	.001895
.000462	.001167	.000562	.001411	.000662	.001655	.000762	.001899
.000463	.001171	.000563	.001415	.000663	.001659	.000763	.001903
.000464	.001174	.000564	.001419	.000664	.001663	.000764	.001907
.000465	.001178	.000565	.001422	.000665	.001667	.000765	.001911
.000466	.001182	.000566	.001426	.000666	.001670	.000766	.001914
.000467	.001186	.000567	.001430	.000667	.001674	.000767	.001918
.000470	.001190	.000570	.001434	.000670	.001678	.000770	.001922
.000471	.001194	.000571	.001438	.000671	.001682	.000771	.001926
.000472	.001197	.000572	.001441	.000672	.001686	.000772	.001930
.000473	.001201	.000573	.001445	.000673	.001689	.000773	.001934
.000474	.001205	.000574	.001449	.000674	.001693	.000774	.001937
.000475	.001209	.000575	.001453	.000675	.001697	.000775	.001941
.000476	.001213	.000576	.001457	.000676	.001701	.000776	.001945
.000477	.001216	.000577	.001461	.000677	.001705	.000777	.001949

APPENDIX F
TABLE OF POWERS OF TWO AND BINARY-DECIMAL EQUIVALENTS

TABLE OF POWERS OF 2

2^n	n	2^{-n}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 281 25
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625
137 438 953 472	37	0.000 000 000 007 275 957 614 183 425 903 320 312 5
274 877 906 944	38	0.000 000 000 003 637 978 807 091 712 951 660 156 25
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125
1 099 511 627 776	40	0.000 000 000 000 909 494 701 772 928 237 915 039 062 5

BINARY AND DECIMAL EQUIVALENTS

Maximum Decimal Integral Value	Number of Decimal Digits	Number of Bits	Maximum Decimal Fractional Value
1		1	.5
3		2	.75
7		3	.875
15	1	4	.937 5
31		5	.968 75
63		6	.984 375
127	2	7	.992 187 5
255		8	.996 093 75
511		9	.998 046 875
1 023	3	10	.999 023 437 5
2 047		11	.999 511 718 75
4 095		12	.999 755 859 375
8 191		13	.999 877 929 687 5
16 383	4	14	.999 938 964 843 75
32 767		15	.999 969 482 421 875
65 535		16	.999 984 741 210 937 5
131 071	5	17	.999 992 370 605 468 75
262 143		18	.999 996 185 302 734 375
524 287		19	.999 998 092 651 367 187 5
1 048 575	6	20	.999 999 046 325 683 593 75
2 097 151		21	.999 999 523 162 841 796 875
4 194 303		22	.999 999 761 581 420 898 437 5
8 388 607		23	.999 999 880 790 710 449 218 75
16 777 215	7	24	.999 999 940 395 355 244 609 375
33 554 431		25	.999 999 970 197 677 612 304 687 5
67 108 863		26	.999 999 985 098 838 806 152 343 75
134 217 727	8	27	.999 999 992 549 419 403 076 171 875
268 435 455		28	.999 999 996 274 709 701 538 085 937 5
536 870 911		29	.999 999 998 137 354 850 769 042 968 75
1 073 741 823	9	30	.999 999 999 068 677 425 384 521 484 375
2 147 483 647		31	.999 999 999 534 338 712 692 260 742 187 5
4 294 967 295		32	.999 999 999 767 169 356 346 130 371 093 75
8 589 934 591		33	.999 999 999 883 584 678 173 065 185 546 875
17 179 869 183	10	34	.999 999 999 941 792 339 086 532 592 773 437 5
34 359 738 367		35	.999 999 999 970 896 169 543 266 296 386 718 75
68 719 476 735		36	.999 999 999 985 448 034 771 633 148 193 359 375
137 438 953 471	11	37	.999 999 999 992 724 042 385 816 574 096 679 687 5
274 877 906 943		38	.999 999 999 996 362 021 192 908 287 048 339 843 75
549 755 813 887		39	.999 999 999 998 181 010 596 454 143 524 169 921 875
1 099 511 627 775	12	40	.999 999 999 999 090 505 298 227 071 762 084 960 937 5
2 199 023 255 551		41	.999 999 999 999 545 252 649 113 535 881 042 480 468 75
4 398 046 511 103		42	.999 999 999 999 772 626 324 556 767 940 521 240 234 375
8 796 093 022 207		43	.999 999 999 999 886 313 162 278 383 970 260 620 117 187 5
17 592 186 044 415	13	44	.999 999 999 999 943 156 581 139 191 985 130 310 058 593 75
35 184 372 088 831		45	.999 999 999 999 971 578 290 569 595 992 565 155 029 296 875
70 368 744 177 663		46	.999 999 999 999 985 789 145 284 797 996 282 577 514 648 437 5
140 737 488 355 327	14	47	.999 999 999 999 992 894 572 642 398 998 141 288 757 324 218 75
281 474 976 710 655		48	

This chart provides the information necessary to determine:

a. The number of bits needed to represent a given decimal number. Use columns one and three or four and three.

b. The number of bits needed to represent a given number of decimal digits (all nines). Use columns two and three.

c. The maximum decimal value represented by a given number of bits, use columns one and three or three and four.

APPENDIX G
M-605 INSTRUCTION MNEMONICS WITH ALLOWABLE ADDRESS MODIFICATIONS

Mnemonic	Modifications Allowed
ADA	All
ADAQ	All except DU, DL, CI, SC
ADE	All except CI, SC
ADL	All except CI, SC
ADLA	All
ADLAQ	All except DU, DL, CI, SC
ADLQ	All
ADLXn	All except CI, SC
ADQ	All
ADXn	All except CI, SC
ALR	All except DU, DL, CI, SC
ALS	All except DU, DL, CI, SC
ANA	All
ANAQ	All except DU, DL, CI, SC
ANQ	All
ANSA	All except DU, DL, CI, SC
ANSQ	All except DU, DL, CI, SC
ANSXn	All except DU, DL, CI, SC
ANXn	All except CI, SC
AOS	All except DU, DL, CI, SC
ARL	All except DU, DL, CI, SC
ARS	All except DU, DL, CI, SC
ASA	All except DU, DL, CI, SC
ASQ	All except DU, DL, CI, SC
ASXn	All except DU, DL, CI, SC
AWCA	All
AWCQ	All
BCD	All except CI, SC
CANA	All
CANAQ	All except DU, DL, CI, SC
CANQ	All
CANXn	All except CI, SC
CIOC	All except DU, DL, CI, SC
CMG	All
CMK	All
CMPA	All
CMPAQ	All except DU, DL, CI, SC
CMPQ	All
CMPXn	All except CI, SC
CNAA	All
CNAAQ	All except DU, DL, CI, SC
CNAQ	All
CNAXn	All except CI, SC
CWL	All

Mnemonic	Modifications Allowed
DFAD	All except DU, DL, CI, SC
DFCMG	All except DU, DL, CI, SC
DFCMP	All except DU, DL, CI, SC
DFDI	All except DU, DL, CI, SC
DFDV	All except DU, DL, CI, SC
DFLD	All except DU, DL, CI, SC
DFMP	All except DU, DL, CI, SC
DFSB	All except DU, DL, CI, SC
DFST	All except DU, DL, CI, SC
DIS	No effect on operation
DIV	All
DRL	No effect on operation
DUFA	All except DU, DL, CI, SC
DUFM	All except DU, DL, CI, SC
DUFS	All except DU, DL, CI, SC
DVF	All
EAA	All except DU, DL
EAQ	All except DU, DL
EAXn	All except DU, DL
ERA	All
ERAQ	All except DU, DL, CI, SC
ERQ	All
ERSA	All except DU, DL, CI, SC
ERSQ	All except DU, DL, CI, SC
ERSXn	All except DU, DL, CI, SC
ERXn	All except CI, SC
FAD	All except CI, SC
FCMG	All except CI, SC
FCMP	All except CI, SC
FDI	All except CI, SC
FDV	All except CI, SC
FLD	All except CI, SC
FMP	All except CI, SC
FNEG	No effect on operation
FNO	No effect on operation
FSB	All except CI, SC
FST	All except DU, DL, CI, SC
FSZN	All except CI, SC
GTB	No effect on operation
LBAR	All except CI, SC
LCA	All
LCAQ	All except DU, DL, CI, SC
LCQ	All
LCXn	All except CI, SC
LDA	All
LDAQ	All except DU, DL, CI, SC
LDE	All except CI, SC

Mnemonic	Modifications Allowed
LDI	All except CI, SC
LDLXn	All except CI, SC
LDT	All except CI, SC
LDQ	All
LDXn	All except CI, SC
LLR	All except DU, DL, CI, SC
LLS	All except DU, DL, CI, SC
LREG	All except DU, DL, CI, SC
LRL	All except DU, DL, CI, SC
LRS	All except DU, DL, CI, SC
MME	No effect on operation
MPF	All except CI, SC
MPY	All except CI, SC
NEG	No effect on operation
NEGL	No effect on operation
NOP	All (See notes under instruction)
ORA	All
ORAQ	All except DU, DL, CI, SC
ORQ	All
ORSA	All except DU, DL, CI, SC
ORSQ	All except DU, DL, CI, SC
ORSXn	All except DU, DL, CI, SC
ORXn	All except CI, SC
QLR	All except DU, DL, CI, SC
QLS	All except DU, DL, CI, SC
QRL	All except DU, DL, CI, SC
QRS	All except DU, DL, CI, SC
RET	All except DU, DL, CI, SC
RMCM	All except DU, DL, CI, SC
RMFP	All except DU, DL, CI, SC
RPD	None
RPL	None
RPT	None
SBA	All
SBAQ	All except DU, DL, CI, SC
SBAR	All except DU, DL, CI, SC
SBLA	All
SBLAQ	All except DU, DL, CI, SC
SBLQ	All
SBLXn	All except CI, SC
SBQ	All

Mnemonic	Modifications Allowed
SBXn	All except CI, SC
SMCM	All except DU, DL, CI, SC
SMFP	All except DU, DL, CI, SC
SMIC	All except DU, DL, CI, SC
SREG	All except DU, DL, CI, SC
SSA	All except DU, DL, CI, SC
SSQ	All except DU, DL, CI, SC
SSXn	All except DU, DL, CI, SC
STA	All except DU, DL
STAQ	All except DU, DL, CI, SC
STBA	None
STBQ	None
STC1	All except DU, DL, CI, SC
STC2	All except DU, DL, CI, SC
STCA	None
STCQ	None
STE	All except DU, DL, CI, SC
STI	All except DU, DL, CI, SC
STLXn	All except DU, DL, CI, SC
STQ	All except DU, DL
STT	All except DU, DL, CI, SC
STXn	All except DU, DL, CI, SC
STZ	All except DU, DL, CI, SC
SWCA	All
SWCQ	All
SZN	All
TEO	All except DU, DL, CI, SC
TEU	All except DU, DL, CI, SC
TMI	All except DU, DL, CI, SC
TNC	All except DU, DL, CI, SC
TNZ	All except DU, DL, CI, SC
TOV	All except DU, DL, CI, SC
TPL	All except DU, DL, CI, SC
TRA	All except DU, DL, CI, SC
TSS	All except DU, DL, CI, SC
TSXn	All except DU, DL, CI, SC
TTF	All except DU, DL, CI, SC
TZE	All except DU, DL, CI, SC
UFA	All except CI, SC
UFM	All except CI, SC
UFS	All except CI, SC
XEC	All except DU, DL, CI, SC
XED	All except DU, DL, CI, SC

APPENDIX H

M-605 INSTRUCTION MNEMONICS CORRELATED WITH THEIR OPERATION CODES

M-605 INSTRUCTION MNEMONICS CORRELATED WITH THEIR OPERATION CODES

M-605 Mnemonics and Operation Codes										GENERAL ELECTRIC RADIO GUIDANCE OPERATION						
	000	001	002	003	004	005	006	007	010	011	012	013	014	015	016	017
000		MME	DRL							NϕP				CIϕC		
020	ADLX0	ADLX1	ADLX2	ADLX3	ADLX4	ADLX5	ADLX6	ADLX7				ADL		ADLA	ADLQ	ADLAQ
040	ASX0	ASX1	ASX2	ASX3	ASX4	ASX5	ASX6	ASX7					AϕS	ASA	ASQ	
060	ADX0	ADX1	ADX2	ADX3	ADX4	ADX5	ADX6	ADX7		AWCA	AWCQ	LREG		ADA	ADQ	ADAQ
100	CMPX0	CMPX1	CMPX2	CMPX3	CMPX4	CMPX5	CMPX6	CMPX7		CWL				CMPA	CMPQ	CMPAQ
120	SBLX0	SBLX1	SBLX2	SBLX3	SBLX4	SBLX5	SBLX6	SBLX7						SBLA	SBLQ	SBLAQ
140	SSX0	SSX1	SSX2	SSX3	SSX4	SSX5	SSX6	SSX7						SSA	SSQ	
160	SBX0	SBX1	SBX2	SBX3	SBX4	SBX5	SBX6	SBX7		SWCA	SWCQ			SBA	SBQ	SBAQ
200	CNAX0	CNAX1	CNAX2	CNAX3	CNAX4	CNAX5	CNAX6	CNAX7		CMK				CNAA	CNAQ	CNAAQ
220	LDX0	LDX1	LDX2	LDX3	LDX4	LDX5	LDX6	LDX7	LBAR			RMCM	SZN	LDA	LDQ	LDAQ
240	ϕRSX0	ϕRSX1	ϕRSX2	ϕRSX3	ϕRSX4	ϕRSX5	ϕRSX6	ϕRSX7						ϕRSA	ϕRSQ	
260	ϕRX0	ϕRX1	ϕRX2	ϕRX3	ϕRX4	ϕRX5	ϕRX6	ϕRX7						ϕRA	ϕRQ	ϕRAQ
300	CANX0	CANX1	CANX2	CANX3	CANX4	CANX5	CANX6	CANX7						CANA	CANQ	CANAQ
320	LCX0	LCX1	LCX2	LCX3	LCX4	LCX5	LCX6	LCX7						LCA	LCQ	LCAQ
340	ANSX0	ANSX1	ANSX2	ANSX3	ANSX4	ANSX5	ANSX6	ANSX7						ANSA	ANSQ	
360	ANX0	ANX1	ANX2	ANX3	ANX4	ANX5	ANX6	ANX7						ANA	ANQ	ANAQ
400		MPF	MPY			CMG				LDE				ADE		
420		UFM		DUFM		FCMG		DFCMG	FSZN	FLD		DFLD		UFA		DUFA
440	STLX0	STLX1	STLX2	STLX3	STLX4	STLX5	STLX6	STLX7	STZ	SMIC		SMFP	STT	FST	STE	DFST
460		FMP		DFMP										FAD		DFAD
500	RPL					BCD	DIV	DVF				FNEG		FCMP		DFCMP
520	RPT					FDI		DFDI		NEG		NEGL		UFS		DUFS
540									SBAR	STBA	STBQ	XMCM	STCA			
560	RPD					FNI		[illegible]						FST		DFSB
600	TZE	TNZ	TNC	TRC	TMI	TPL		TTF					TEϕ	TEU	DIS	TϕV
620	EAX0	EAX1	EAX2	EAX3	EAX4	EAX5	EAX6	EAX7	RET			RMFP	LDI	EAA	EAQ	LDT
640	ERSX0	ERSX1	ERSX2	ERSX3	ERSX4	ERSX5	ERSX6	ERSX7						ERSA	ERSQ	
660	ERX0	ERX1	ERX2	ERX3	ERX4	ERX5	ERX6	ERX7						ERA	ERQ	ERAQ
700	TSX0	TSX1	TSX2	TSX3	TSX4	TSX5	TSX6	TSX7	TRA					TSS	XEC	XED
720	LDLX0	LDLX1	LDLX2	LDLX3	LDLX4	LDLX5	LDLX6	LDLX7		ARS	QRS	LRS		ALS	QLS	LLS
740	STX0	STX1	STX2	STX3	STX4	STX5	STX6	STX7	STC2	STCA	STCQ	SREG	STI	STA	STQ	STAQ
760										ARL	QRL	LRL	GTB	ALR	QLR	LLR
	000	001	002	003	004	005	006	007	010	011	012	013	014	015	016	017

APPENDIX I
M-605 MNEMONICS
IN ALPHABETICAL ORDER
WITH PAGE REFERENCES

APPENDIX J
M-605 INSTRUCTIONS LISTED BY FUNCTIONAL CLASS WITH PAGE REFERENCES AND TIMING

DATA MOVEMENT			M-605 TIMING 2 μsec	M-605 TIMING 1 μsec	Reference (Page)
Load					
LDA	235	Load A	3.6	2.8	III-3
LDQ	236	Load Q	3.6	2.8	3
LDAQ	237	Load AQ	*3.9	3.1	83
LDXn	22n	Load Xn	3.6	2.8	3
LDLXn	72n	Load Xn from Lower	3.6	2.8	4
LREG	073	Load Registers	10.0	8.0	5
LCA	335	Load Complement A	3.6	2.8	5
LCQ	336	Load Complement Q	3.6	2.8	6
LCAQ	337	Load Complement AQ	*3.9	3.1	83
LCXn	32n	Load Complement Xn	3.6	2.8	6
EAA	635	Effective Address to A	2.2	2.0	7
EAQ	636	Effective Address to Q	2.2	2.0	7
EAXn	62n	Effective Address to Xn	2.2	2.0	8
LDI	634	Load Indicator Register	3.6	2.8	4
Store					
STA	755	Store A	3.2	2.6	9
STQ	756	Store Q	3.2	2.6	9
STAQ	757	Store AQ	*4.5	3.7	84
STXn	74n	Store Xn	3.2	2.6	9
STLXn	44n	Store Xn in Lower	3.2	2.6	9
SREG	753	Store Register	12.0	10.0	10
STCA	751	Store Character of A (6 Bit)	3.2	2.6	10
STCQ	752	Store Character of Q (6 Bit)	3.2	2.6	11
STBA	551	Store Character of A (9 Bit)	3.2	2.6	12
STBQ	552	Store Character of Q (9 Bit)	3.2	2.6	13
STI	754	Store Indicator Register	3.8	3.2	14
STT	454	Store Timer Register	3.2	2.6	15
SBAR	550	Store Base Address Register	3.2	2.6	15
STZ	450	Store Zero	3.2	2.6	15
STC1	554	Store Instruction Counter plus 1	3.8	3.2	16
STC2	750	Store Instruction Counter plus 2	3.8	3.2	17
Shift					
ARS	731	A Right Shift	3.6	3.6	18
QRS	732	Q Right Shift	3.6	3.6	18
LRS	733	Long Right Shift	3.6	3.6	19
ALS	735	A Left Shift	3.6	3.6	19
QLS	736	Q Left Shift	3.6	3.6	20
LLS	737	Long Left Shift	3.6	3.6	20

* Performed by macro-operation or hardware option. Timing listed is for optional hardware operation.

DATA MOVEMENT			M-605 TIMING 2 μsec	M-605 TIMING 1 μsec	Reference (Page)
Shift					
ARL	771	A Right Logic	3.6	3.6	III-20
QRL	772	Q Right Logic	3.6	3.6	21
LRL	773	Long Right Logic	3.6	3.6	21
ALR	775	A Left Rotate	3.6	3.6	21
QLR	776	Q Left Rotate	3.6	3.6	22
LLR	777	Long Left Rotate	3.6	3.6	22
FIXED-POINT ARITHMETIC					
Addition					
ADA	075	Add to A	3.6	2.8	23
ADQ	076	Add to Q	3.6	2.8	23
ADAQ	077	Add to AQ	*3.9	3.1	85
ADXn	06n	Add to Xn	3.6	2.8	24
ASA	055	Add Stored to A	4.5	3.5	24
ASQ	056	Add Stored to Q	4.5	3.5	25
ASXn	04n	Add Stored to Xn	4.5	3.5	25
ADLA	035	Add Logic to A	3.6	2.8	26
ADLQ	036	Add Logic to Q	3.6	2.8	26
ADLAQ	037	Add Logic to AQ	*3.9	3.1	85
ADLXn	02n	Add Logic to Xn	3.6	2.8	27
AWCA	071	Add with Carry to A	3.6	2.8	27
AWCQ	072	Add with Carry to Q	3.6	2.8	28
ADL	033	Add Low to AQ	3.6	2.8	28
AOS	054	Add One to Storage	4.5	3.5	29
Subtraction					
SBA	175	Subtract from A	3.6	2.8	30
SBQ	176	Subtract from Q	3.6	2.8	30
SBAQ	177	Subtract from AQ	*3.9	3.1	86
SBXn	16n	Subtract from Xn	3.6	2.8	31
SSA	155	Subtract Stored from A	4.5	3.5	31
SSQ	156	Subtract Stored from Q	4.5	3.5	32
SSXn	14n	Subtract Stored from Xn	4.5	3.5	32

* Performed by macro-operation or hardware option. Timing listed is for optional hardware operation.

FIXED-POINT ARITHMETIC			M-605 TIMING 2 μsec	M-605 TIMING 1 μsec	Reference (Page)
Subtraction					
SBLA	135	Subtract Logic from A	3.6	2.8	III-33
SBLQ	136	Subtract Logic from Q	3.6	2.8	33
SBLAQ	137	Subtract Logic from AQ	*3.9	3.1	86
SBLXn	12n	Subtract Logic from Xn	3.6	2.8	34
SWCA	171	Subtract with Carry from A	3.6	2.8	34
SWCQ	172	Subtract with Carry from Q	3.6	2.8	35
Multiplication					
MPY	402	Multiply Integer	10.5	10.0	36
MPF	401	Multiply Fraction	10.5	10.0	37
Division					
DIV	506	Divide Integer	18.0	17.5	III-38
DVF	507	Divide Fraction	18.0	17.5	39
Negate					
NEG	531	Negate A	2.2	2.0	40
NEGL	433	Negate Long	2.2	2.0	40
BOOLEAN OPERATIONS					
AND					
ANA	375	AND to A	3.6	2.8	41
ANQ	376	AND to Q	3.6	2.8	41
ANAQ	377	AND to AQ	*3.9	3.1	87
ANXn	36n	AND to Xn	3.6	2.8	41
ANSA	355	AND to Storage A	4.5	3.5	42
ANSQ	356	AND to Storage Q	4.5	3.5	42
ANSXn	34n	AND to Storage Xn	4.5	3.5	42
OR					
ORA	275	OR to A	3.6	2.8	43
ORQ	276	OR to Q	3.6	2.8	43
ORAQ	277	OR to AQ	*3.9	3.1	88
ORXn	26n	OR to Xn	3.6	2.8	43

* Performed by macro-operation or hardware option. Timing listed is for optional hardware operation.

BOOLEAN OPERATIONS			M-605 TIMING 2 μsec	M-605 TIMING 1 μsec	Reference (Page)
OR					
ORSA	255	OR to Storage A	4.5	3.5	III-44
ORSQ	256	OR to Storage Q	4.5	3.5	44
ORSXn	24n	OR to Storage Xn	4.5	3.5	44
EXCLUSIVE OR					
ERA	675	EXCLUSIVE OR to A	3.6	2.8	45
ERQ	676	EXCLUSIVE OR to Q	3.6	2.8	45
ERAQ	677	EXCLUSIVE OR to AQ	*3.9	3.1	89
ERXn	66n	EXCLUSIVE OR to Xn	3.6	2.8	45
ERSA	655	EXCLUSIVE OR to Storage A	4.5	3.5	46
ERSQ	656	EXCLUSIVE OR to Storage Q	4.5	3.5	46
ERSXn	64n	EXCLUSIVE OR to Storage Xn	4.5	3.5	46
COMPARISON					
Compare					
CMPA	115	Compare with A	3.6	2.8	47
CMPQ	116	Compare with Q	3.6	2.8	48
CMPAQ	117	Compare with AQ	*3.9	3.1	90
CMPXn	10n	Compare with Xn	3.6	2.8	49
CWL	111	Compare with Limits	3.8	3.4	50
CMG	405	Compare Magnitude	3.6	2.8	51
SZN	234	Set Zero and Negative Indicators from Memory	3.6	2.8	52
CMK	211	Compare Masked	3.8	3.4	52
Comparative AND					
CANA	315	Comparative AND with A	3.6	2.8	53
CANQ	316	Comparative AND with Q	3.6	2.8	53
CANAQ	317	Comparative AND with AQ	*3.9	3.1	91
CANXn	30n	Comparative AND with Xn	3.6	2.8	53
Comparative NOT AND					
CNAA	215	Comparative NOT AND with A	3.6	2.8	54
CNAQ	216	Comparative NOT AND with Q	3.6	2.8	54
CNAAQ	217	Comparative NOT AND with AQ	*3.9	3.1	92
CNAXn	20n	Comparative NOT AND with Xn	3.6	2.8	54

* Performed by macro-operation or hardware option. Timing listed is for optional hardware operation.

FLOATING POINT			M-605 TIMING 2 μsec	M-605 TIMING 1 μsec	Reference (Page)
Load					
FLD	431	Floating Load	*3.8	3.3	III-93
DFLD	433	Double-Precision Floating Load	*4.1	3.6	93
LDE	411	Load Exponent Register	3.6	2.8	55
Store					
FST	455	Floating Store	*3.2	2.6	94
DFST	457	Double-Precision Floating Store	*4.5	3.7	94
STE	456	Store Exponent Register	3.2	2.6	55
Addition					
FAD	475	Floating Add	*6.5	6.2	95
UFA	435	Unnormalized Floating Add	*6.5	6.2	95
DFAD	477	Double-Precision Floating Add	*6.8	6.5	96
DUFA	437	Double-Precision Unnormalized Floating Add	*6.8	6.5	96
ADE	415	Add to Exponent Register	3.6	2.8	55
Subtraction					
FSB	575	Floating Subtract	*6.5	6.2	97
UFS	535	Unnormalized Floating Subtract	*6.5	6.2	97
DFSB	577	Double-Precision Floating Subtract	*6.8	6.5	98
DUFS	537	Double-Precision Unnormalized Floating Subtract	*6.8	6.5	98
Multiplication					
FMP	461	Floating Multiply	*10.0	9.4	99
UFM	421	Unnormalized Floating Multiply	*10.0	9.4	99
DFMP	463	Double-Precision Floating Multiply	*10.3	9.7	100
DUFM	423	Double-Precision Unnormalized Floating Multiply	*10.3	9.7	100
Division					
FDV	565	Floating Divide	*19.2	18.7	101
FDI	525	Floating Divide Inverted	*19.2	18.7	102
DFDV	567	Double-Precision Floating Divide	*19.5	19.0	103
DFDI	527	Double-Precision Floating Divide Inverted	*19.5	19.0	104

* Performed by macro-operation or hardware option. Timing listed is for optional hardware operation.

FLOATING POINT			M-605 TIMING 2 μsec	1 μsec	Reference (Page)
Negate, Normalize					
FNEG	513	Floating Negate	*6.5	6.2	III-105
FNO	573	Floating Normalize	6.5	6.2	56
Compare					
FCMP	515	Floating Compare	*6.5	6.2	106
FCMG	425	Floating Compare Magnitude	*6.5	6.2	107
DFCMP	517	Double-Precision Floating Compare	*6.8	6.5	107
DFCMG	427	Double-Precision Floating Compare Magnitude	*6.8	6.5	108
FSZN	430	Floating Set Zero and Negative Indicators from Memory	*3.8	3.3	108
TRANSFER OF CONTROL					
Transfer					
TRA	710	Transfer Unconditionally	2.0	1.9	57
TSXn	70n	Transfer and Set Xn	2.0	1.9	57
TSS	715	Transfer and Set Slave Mode	2.0	1.9	57
RET	630	Return	4.0	3.6	58
Conditional Transfer					
TZE	600	Transfer on Zero	2.0	1.9	59
TNZ	601	Transfer on Not Zero	2.0	1.9	59
TMI	604	Transfer on Minus	2.0	1.9	59
TPL	605	Transfer on Plus	2.0	1.9	59
TRC	603	Transfer on Carry	2.0	1.9	60
TNC	602	Transfer on No Carry	2.0	1.9	60
TOV	617	Transfer on Overflow	2.0	1.9	60
TEO	614	Transfer on Exponent Overflow	2.0	1.9	60
TEU	615	Transfer on Exponent Underflow	2.0	1.9	61
TTF	607	Transfer on Tally-Runout Indicator OFF	2.0	1.9	61
MISCELLANEOUS OPERATIONS					
NOP	011	No Operation	2.0	1.6	62
DIS	616	Delay Until Interrupt Signal	2.0	1.9	62

* Performed by macro-operation or hardware option. Timing listed is for optional hardware operation.

MISCELLANEOUS OPERATIONS			M-605 TIMING 2 μsec	M-605 TIMING 1 μsec	Model 60 Reference (Page)
BCD	505	Binary to Binary-Coded-Decimal	6.8	6.5	III-63
GTB	774	Gray to Binary	9.8	9.5	64
XEC	716	Execute	2.0	1.9	64
XED	717	Execute Double	2.0	1.9	65
MME	001	Master Mode Entry	2.0	1.9	66
DRL	002	Derail	2.0	1.9	67
RPT	520	Repeat	2.0	1.9	68
RPD	560	Repeat Double	2.0	1.9	73
RPL	500	Repeat Link	2.0	1.9	70
MASTER MODE OPERATIONS					
LBAR	230	Load Base Address Register	3.6	2.8	76
LDT	637	Load Timer Register	3.6	2.8	76
SMIC	451	Set Memory Controller Interrupt Cells	3.2	2.6	77
RMCM	233	Read Memory Controller Mask Registers	3.9	3.1	78
RMFP	633	Read Memory File Protect Register	3.9	3.1	79
SMCM	553	Set Memory Controller Mask Registers	4.5	3.7	80
SMFP	453	Set Memory File Protect Register	4.5	3.7	81
CIOC	015	Connect I/O Channel	3.6	2.6	82

An explanation of instruction execution timing is given in paragraph 5, page II-36.

* Performed by macro-operation or hardware option. Timing listed is for optional hardware operation.

Progress Is Our Most Important Product

GENERAL ELECTRIC

RADIO GUIDANCE OPERATION • SYRACUSE, N. Y.

RGO-157