PART I

General Description of the Maniac III Computer

A. Introduction

Maniac III is an automatic digital computer designed and constructed by the staff of the Institute for Computer Research at the University of Chicago. It is composed of the following units: (1) a central processor; (2) an associated rapid-access memory; (3) a primary input/output system consisting of a paper tape reader, a paper tape punch, a console typewriter, and a line printer; (4) additional high-speed memory shared with auxiliary devices and other computers. The additional memory (4) not only extends the storage available to the central processor, but also provides a means by which Maniac III can communicate with the auxiliary devices and other computers.

The basic unit of information for Maniac III is the word, which is a string of 48 bits, each 0 or 1. Word transfer within the computer is in parallel, and arithmetic is performed in the binary system.

Provision is made for rapid access to 16384($=2^{14}$) storage words, in locations designated by sequentially numbered addresses. Most addresses refer to locations in memory, but certain registers in the central processor are also addressable. The latter essentially comprise a set of 48-bit A registers, which include the following: three constant registers, denoted V, W, and Z, whose contents are unalterable; eight temporary storage registers, denoted T_0, T_1, . . . , T_7; and two operation registers,

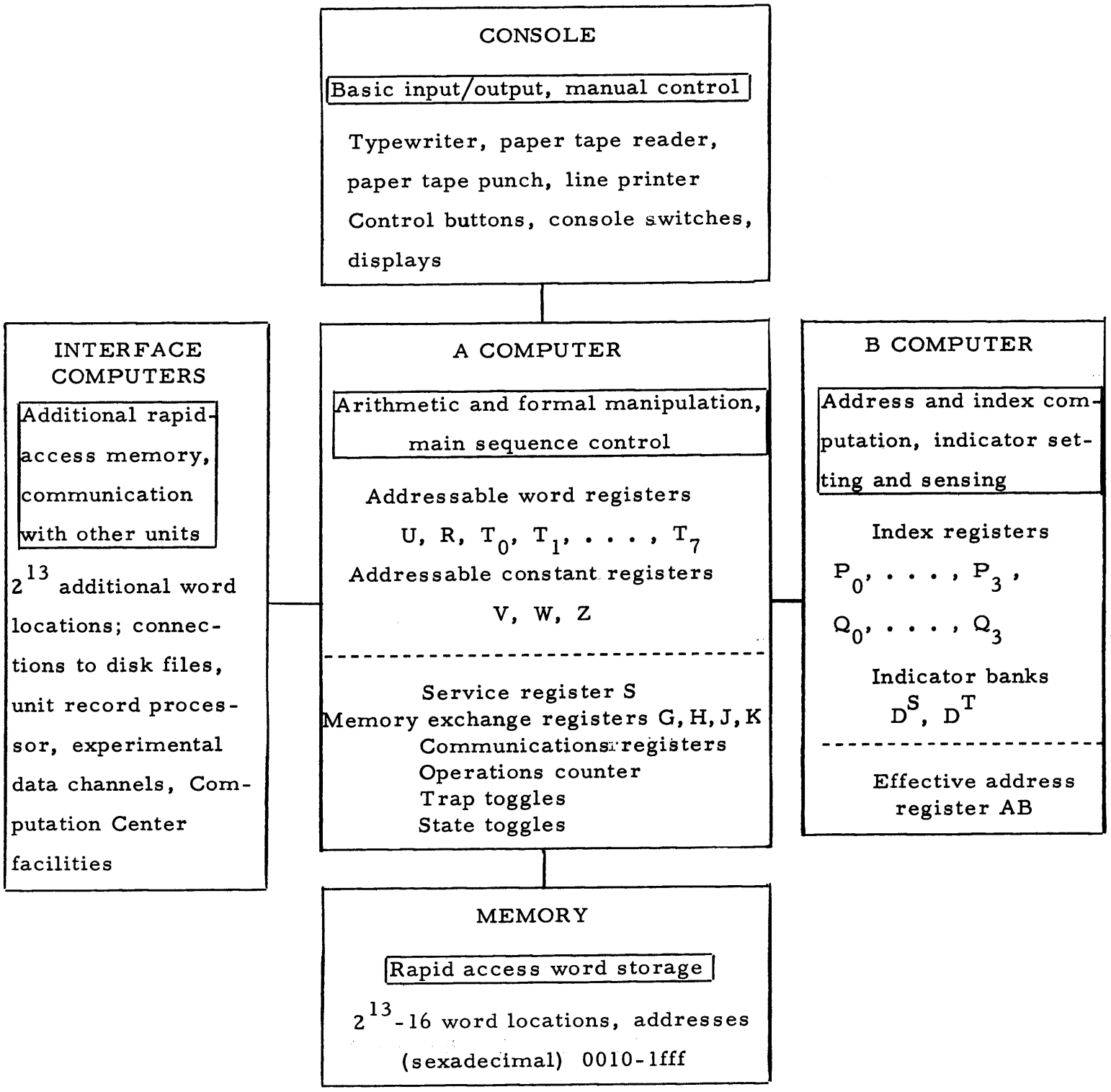

Figure I-A1. Maniac III block diagram

denoted U (for Universal) and R (for Residual). The Z register contains a word of all bits 0, and the W register contains a word of all bits 1; the V register contains a special numeric word (representing the integer -1 in the internal number system of the computer). Arithmetic and formal operations carried out by computer instructions combine operands residing in U (and possibly R) with operands from memory or addressable registers, and form primary results in U, residual results (such as low order parts and remainders) in R. The addresses assigned to these A registers are listed in Appendix I.

In addition to these 48-bit registers there are a number of special-purpose nonaddressable 14-bit B registers. Included among these are eight index registers, divided into two groups of four each: group P, containing index registers P_0, P_1, P_2 and P_3, and group Q, containing index registers Q_0, Q_1, Q_2 and Q_3. Two of these, P_3 and Q_3, have a special control function; they are called the reference register (link register or path finder) and the sequence register (instruction location or control counter), respectively. Also included as B registers are two indicator registers, D^S and D^T; their individual bit positions are referred to as sense indicators and trap indicators, respectively.

Other registers, which are ordinarily of no concern to the user except in special cases, or when console operation is involved, are the 48-bit service register S, the instruction register I', and various special 48-bit communications registers; there is also a 14-bit effective address

register AB, which records the currently addressed memory location.

Figure I-A1 gives a block diagram illustrating the general structure and organization of the computer. It is convenient to think of the Maniac III as composed of several independent but interacting units, as shown. In normal automatic operation the A computer, which contains an instruction register, can be considered to be primary controller. Its action is determined, however, by the contents of the sequence register Q_3, which is in the B computer; this records the location in memory of a coded instruction which is to be fetched to the instruction register. The instruction specifies an operation to be performed and associated address information, the latter to be processed by the B computer and used to obtain operands from memory, or to store results in memory. The A computer carries out the instruction, and following this the sequence register, unless specifically altered by the instruction performed, has its contents augmented by 1 and is used to obtain the next instruction from memory.

Instructions for the A computer are 48-bit words, whose interpretation is given in a standard instruction set or vocabulary. This set is to be distinguished from a particular set of instructions actually stored in memory, comprising a program representing a particular computational task. When residing in memory, instruction words are objectively indistinguishable from data words, representing information to be processed; it is only their treatment in the course of computation that differs (i.e., they are routed to the instruction register).

For engineering and mnemonic convenience, the 128 possible instructions are divided into 8 classes of 16 instructions each. Three classes are for performing full word numeric operations; these are designated the Floating Point Arithmetic, Numerical Manipulation, and Specified Point Arithmetic classes. Four classes are for performing nonnumeric and index word A computer operations; these are the Index Computation, Formal Manipulation, Data Transmission, and Miscellaneous classes. Finally, there is the Auxiliary System Operation class for performing operations associated with the input/output and auxiliary equipment.

A single form of number representation is used for all arithmetic operands; the 48-bit word is partitioned into a 40-bit coefficient part and an 8-bit exponent part. An exponent part is interpreted as an integer e, while a coefficient part is interpreted as a fraction f in the range $-1 \leq f < 1$; the pair (e, f) then corresponds to the number $f.2^e$. Complement representation is used for negative numbers and there is a special exponent symbol to indicate an "absolute zero" value.

Floating Point Arithmetic in Maniac III differs from the procedure so called in most computers, in that coefficient parts are handled in unnormalized form, according to a system which allows an estimation of the precision of results. Specified Point Arithmetic is a generalization of what is usually referred to as "fixed point" operation, which affords arbitrary but controlled positioning of the binary point. The use of the

single number representation eliminates any need for distinguishing between "fixed point numbers" and "floating point numbers".

Numerical Manipulation instructions include operations which normalize results (after the manner of traditional floating point arithmetic).

Index Computation instructions serve to set and alter the contents of the index registers, and to effect jumps in the normal instruction sequence (the latter because the sequence and reference registers are included in the index category). Such jumps may be made conditional, depending on the outcome of a comparison between the contents of specified registers or locations, or on the contents of the two indicator registers. The primary use of the other index registers is in connection with address modification at the time an instruction is performed.

The individual bit positions of the indicator registers can be set in a variety of ways to indicate the occurence of events during the course of a program, and to allow manual or programmed imposition of procedural variations.

The Formal Manipulation instructions are designed to permit the common variety of internal "bit processing" to be carried out efficiently, by means of shifting and combining words without special numeric interpretation. Also in this class are instructions for independently manipulating exponents, and for converting exponent and index representations to standard numeric form.

This gives a brief picture of the computer and its operation; a more detailed discussion is given in the subsequent sections.

B. Basic Information Patterns

Within the computer system, symbolic information is coded in terms of the bits 0, 1. A sequence of 48 bits comprises a word; the bits of a word are numbered 47, 46, . . . 1, 0 from left to right for identification purposes, as illustrated in Figure I-B1.

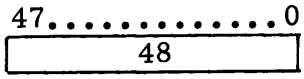

Figure I-B1. Word format

1. Tetrad symbols

It is often convenient to consider a word as a sequence of 12 tetrads, or 4-bit groups. There are 16 possible distinct tetrads, and they are represented by the symbols 0, 1, . . . , f as shown in Table I-B1. This alphabet of tetrad symbols affords a compact written representation of computer words; for example, the word

1010011100.....0,

which has first tetrad 1010 and second tetrad 0111, followed by ten tetrads 0000, is represented by the sequence

a70000000000.

Shorter sequences of bits may also be expressed in terms of tetrad symbols; if the number of bits is not divisible by 4, the tetrad grouping is started from the right, and the leftmost tetrad completed by considering the string to be headed with an appropriate number of bits 0. Thus, for example, the 14-bit sequence

01111001100001

is represented by

1e61.

2. Sexadecimal representation

It is possible to interpret any sequence of bits as a numeral in the binary (base 2) system, wherein the bits function as binary digits weighted by powers of 2 increasing from right to left. For example, the 4-bit sequence 1101 represents the number 13, since

$$1\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^0 = 8 + 4 + 1 = 13.$$

The 16 tetrads listed in Table 1 represent, in order, the numbers 0, 1, . . ., 15 in binary form; the associated tetrad symbol is seen to be the natural (decimal) one for the first 10 of these. The assignment of a single symbol to the remaining 6 cases allows the complete set of 16 tetrad symbols to be considered as a set of digits suitable for representing numbers in the sexadecimal (base 16) system. The symbol 1e61, interpreted as a sexadecimal expression, represents the (decimal) number 7777, for

$$1\cdot 16^3 + (14)\cdot 16^2 + 6\cdot 16^1 + 1\cdot 16^0 = 4096 + 3584 + 96 + 1 = 7777.$$

Tetrad	Symbol
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	a
1011	b
1100	c
1101	d
1110	e
1111	f

Table I-B1. Assignment of tetrad symbols

As indicated, the tetrad symbols a, b, . . ., f are replaced by their decimal equivalents in computing the decimal value of a sexadecimal expression. It can be verified that a similar computation on the 14-bit sequence represented by 1e61 gives the same decimal result; thus the tetrad symbol shorthand retains its usefulness even when sequences of bits are interpreted numerically.

In the computer system, 14-bit binary numbers are referred

to as designators. When interpreted as positive binary integers as described above, they are used as addresses to designate memory locations; the number of possible addresses is therefore $2^{14} = 16384$, and these are naturally ordered 0000, 0001, . . . , 3fff.

The natural interpretation of a 14-bit sequence is thus as an unsigned designator, an integer in the range from 0 to $2^{14}-1$, as shown in Table I-B2. In many contexts, however, it is convenient to regard 14-digit quantities as signed designators in the range from -2^{13} to $+2^{13}-1$. The nonnegative integers are represented just as in the unsigned case; they therefore all have a leftmost digit of 0. Negative integers all have a leftmost digit of 1 and are related to their absolute values according to the (true or 2-) complement scheme; this interpretation is also shown in Table I-B2.

The complement of a binary integer of any length may be formed by first reflecting (changing 0 to 1 and vice versa) each bit, and then adding 1 at the right. Cognizance may or may not be taken of whether this addition causes a carry (overflow) beyond the left end of the number or into the sign position, depending on the particular application. Arithmetic (ignoring overflows) may be done completely without regard for whether the binary numbers involved are regarded as signed or unsigned or even mixed -- e.g. 1 may be subtracted from 16383 by adding 3fff (as -1) to 3fff (as 16383), to give 3ffe (=16382).

Designator (sexadecimal)	Unsigned Interpretation	Signed Interpretation
0000	0	0
0001	+1	+1
⋮	⋮	⋮
1fff	$+2^{13}-1$	$+2^{13}-1$
2000	2^{13}	-2^{13}
2001	$2^{13}+1$	$-2^{13}+1$
⋮	⋮	⋮
3fff	$2^{14}-1$	-1

Table I-B2. Unsigned and signed designators.

3. Octad symbols as character codes

To allow the representation of an extended set of symbols, octads, or 8-bit groups, are employed. These are used to code a set of characters, which can thus be manipulated within the computer system. In this code the character represented is determined by the rightmost 7 bits of an octad; the leftmost bit is normally 0 for a character residing in computer memory. When characters are recorded in coded form on external media such as paper tape, however, the leftmost bit is set according to a parity criterion, so that the total number of bits 1 in each octad is even; this provides a means for checking the transmission of characters between the central processor and auxiliary devices.

There are 128 possible distinct groups of 7 bits; of these, 88 are used to represent actual symbols producible on the console typewriter associated with the computer, and several more serve to represent typewriter carriage control actions. This set of assigned characters is called the alphabet of the computer; a table of the coding for this alphabet is given in Appendix I. It can be seen from this table that the 16 tetrad symbols 0, 1, . . ., f are represented in octad (actually, 7-bit) form as 001 followed by the four bits of the associated tetrad, so that conversion of these symbols between tetrad and octad representation is straight-forward. Also note that the value of the leftmost bit of the group of 7 distinguishes between the lower and upper cases of the typewriter; characters for which this bit is 0 are typed lower case, characters for which this bit is 1 are typed upper case. The characters corresponding to carriage control actions have two codes since they can be invoked with the typewriter carriage in either position.

Any octad can be considered to be composed of two tetrads; hence it is sometimes convenient to think of the characters of the computer alphabet as coded in terms of tetrads rather than bits. Thus the octad characters 0, 1, . . . , f are represented by the tetrad-pairs 10, 11, . . . , 1f, and any sequence of k octads is represented by a sequence of 2k tetrads. For example, the tetrad sequence

5b1e2022270f

represents the sequence of characters

BeginΔ

(where Δ stands for the lower case version of the "space" character).

Via the octad coding a 48-bit computer word may be interpreted as a symbolic word of 6 characters. Care must be taken to distinguish this interpretation from that which is based on tetrad coding, since the symbols 0, 1, . . . , f are associated with both systems. Frequently a symbolic word may be intended to represent a string of less than 6 characters; the excess positions (normally grouped at the right) are then occupied by the "blank" character (octad 00...0), which is vacuous as far as typewriter action is concerned (and hence is not the same as "space").

4. Exponent and coefficient

From what has been said, it can be seen that a computer word can be attributed significance in either its basic (tetrad) or its symbolic (octad) interpretation. The former is appropriately used to represent integers (base 16) in certain "formal" contexts, while the latter serves to represent general numerical information in "edited" form, which is naturally more convenient from the viewpoint of the computer user. Now, for internal arithmetic manipulation within the operations unit of the computer, the 48-bit word is interpreted in a third way, as a numeric word. In this interpretation the leftmost 8 bits (2 tetrads) represent an integral exponent e in the range $-127 \leq e \leq +127$, while the rightmost

40 bits (10 tetrads) represent a fractional coefficient f in the range $-1 \leq f < 1$; the entire numeric word then represents a number in the form $2^e \cdot f$. For example, the word

a7000000000

in this interpretation represents a "relative zero" with e = 39 (represented by a7 in a sexadecimal "excess-128" code) and f = 0. Details of the exponent and coefficient coding are given subsequently, in connection with the exposition of arithmetic processes as they are defined for the computer; see Section I-H, Number Representation.

Still a fourth interpretation of the computer word is as an instruction word; this is discussed in section I-D.

C. Register Structure and Nomenclature

Stages in a 48-bit register are numbered 47, 46, . . . , 1, 0 from left to right, corresponding exactly to the numbering of bit positions in the word format (see Figure I-B1). These stages are referred to by subscripting the register label; thus, single stages in U, reading from left to right, are

$$U_{47}, U_{46}, \ldots, U_0.$$

Sections of U are identified by giving the numbers of the leftmost and rightmost positions in subscript form; for example

$$U_{19\ldots0}$$

denotes the section of U corresponding to the rightmost five tetrads, while

$$U_{47\ldots40}$$

denotes the section of U corresponding to leftmost octad (or two tetrads). The latter is also precisely the section corresponding to the exponent in the numeric format; it is convenient in arithmetic contexts to simply reger to $U_{47\ldots40}$ as U^E, and the coefficient section $U_{39\ldots0}$ as U^F, renumbering stages within these sections as

$$U_7^E,\ U_6^E,\ \ldots,\ U_0^E$$

for exponent, and

$$U_0^F,\ U_1^F,\ \ldots,\ U_{39}^F$$

for coefficient, reading from left to right in each case (the reason for the renumbering is made clear in Section I-H).

Similar notation applies to the other registers R, T_0, T_1, . . ., etc. In describing computer action, it is convenient to adopt the convention that the information stored in a register or part thereof is denoted by the lower case letter corresponding to the register, section, or stage; for example, u is the 48-bit word residing in U, u^E is its exponent residing in U^E, and $u_7^E,\ u_6^E,\ \ldots,\ u_0^E$ are the individual bits of the exponent. Similarly u^F is the coefficient, with bits $u_0^F,\ u_1^F,\ \ldots\ u_{39}^F$; in particular, the numeric representation is such that u_0^F is the bit which gives the algebraic sign of u^F (and hence of u, interpreted as a number).

Such conventions also can be used, when necessary, to denote 14-bit registers and their contents. Thus P_k is the 14-bit designator residing in P_k, and its jth bit is $P_{k,\,j}$. The individual stages of the sense and trap indicator registers are denoted D^S_{13}, D^S_{12}, . . . D^S_0 and D^T_{13}, D^T_{12}, . . . D^T_0; the indicator bit settings are thus d^S_{13}, d^S_{12}, . . . d^S_0 and d^T_{13}, d^T_{12}, . . . , d^T_0. When considering the stages of the indicator registers individually, they themselves are called indicators, and the indicator registers are called banks.

D. Instruction Format

For computer control purposes the 48-bit word is interpreted in an instruction format composed of eight fields, as shown in Figure I-D1. The names of the fields and their lengths are listed in Table I-D1.

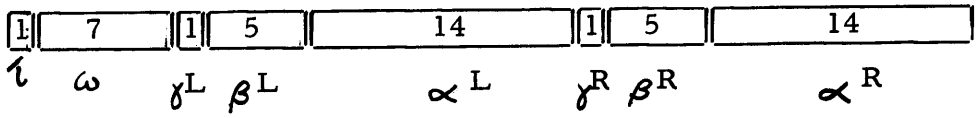

Figure I-D1. Instruction format

Within each field, bit positions are conventionally numbered from right to left starting from 0; thus field ω consists of bit positions ω_6, ω_5, . . . , ω_0, and so forth.

The tag bit τ is used for "breakpointing" instructions in a;way to be described subsequently; see Section K, Trapping. Since the tag does not affect the interpretation of the remainder of the instruction, its use is not considered further in this section.

Field	Length	Name
τ	1	Tag
ω	7	Operation Code
γ^L	1	Left Inflector
β^L	5	Left Modifier
α^L	14	Left Designator
γ^R	1	Right Inflector
β^R	5	Right Modifier
α^R	14	Right Designator

Table I-D1. Instruction fields

Informally, the general significance of an instruction word in terms of computer action may be characterized as follows: "Do operation ω, in a manner specified by γ^L and γ^R, on information designated by α^L modified by β^L, and α^R modified by β^R". The details of, and exceptions to, this characterization are included with the descriptions of individual instructions; certain general conventions, however, are appropriately discussed in the present and following sections.

1. Operation code

The basic function of an instruction is determined by its 7-bit

operation code ω. The 128(=2^7) possible operation codes are grouped into 8 classes of 16 codes each. The leftmost 3 bits ω_6, ω_5, ω_4, interpreted as integer 0, . . ., 7, specify one of the classes 0, 1, . . ., 7; these classes are denoted as shown in Table I-D2.* The remaining 4 bits ω_3, ω_2, ω_1, ω_0 similarly identify the operation as one of 0, 1, . . ., f within a class. Thus any operation is specified by two tetrad symbols. For example, the code 13 specifies instruction 3 in class 1; this instruction is named Floating Add-Store, and it specifies that a number from memory be added to the number in the U register, and the result stored in memory. Note that the corresponding instruction word actually begins with tetrad symbols 13, if the tag bit τ = 0; if τ = 1, however, the first two symbols are 93.

Class	Name
0	Miscellaneous
1	Floating Point Arithmetic
2	Numerical Manipulation
3	Specified Point Arithmetic
4	Formal Manipulation
5	Data Transmission
6	Index Computation
7	Auxiliary System Operation

Table I-D2. Instruction classes

*Since the initial design of the machine, some instructions have been added which violate the characterizations of the instruction classes given in the table. These are mostly complex auxiliary system instructions which are not directly used in a typical program.

2. Inflectors

The two 1-bit inflectors γ^L and γ^R determine a mode in which the instruction is to be performed. By way of illustration, in the Floating Add-Store instruction already mentioned, γ^L has the interpretation + (coded $\gamma^L = 0$) or - (coded $\gamma^L = 1$), and γ^R has the interpretation H or C ("hold" or "clear", coded correspondingly). The first allows the addition operation to be performed positively or negatively, and thus to be in actuality either an addition or a subtraction. The second allows the U register to retain its contents after storing then, or else to be cleared to zero. There are seven standard inflector interpretations; these are discussed in detail in Section F, Use of Inflectors, and in connection with the specific instructions to which they are applicable.

3. Designators

The two 14-bit designators α^L and α^R are used as 14-bit operands for the instruction or as bases for forming, in conjunction with the corresponding modifier field, modified designators as discussed below. A 14-bit designator is ordinarily represented by 4 tetrad symbols, the first of which is necessarily 0, 1, 2, or 3.

One use of modified designators is as the addresses by which instructions refer to 48-bit words in memory. Registers R, S, T_0, T_1, . . . , T_7, U, V, W, Z also have addresses of this type, and can be used to supply or store information as if they were memory locations.

As mentioned earlier, however, most of these are special-purpose registers: V, W, Z hold constant words; U, R, figure in all arithmetic and formal operations in a prescribed way; S is used as an intermediary in transmitting information along the various units of the computer, and in general need not be addressed by program. Only T_0, T_1, . . ., T_7 can be used without restriction as if they were memory locations; however, their use is conventionally restricted to certain utility programs and engineering tests.

4. Modifiers

The modifiers β^L and β^R are generally used to specify one or more index registers, which either serve to modify the associated designator each time the instruction is executed or are themselves the objects of the instruction. The 5 bits of either modifier are identified as β_4, β_3, β_2, β_1, β_0, reading from left to right. The bit β_4 of the modifier specifies an index register group (0 for group P, 1 for group Q), while β_3, β_2, β_1, β_0 specify an arbitrary subset of the four registers in the group, in the obvious manner: P_k or Q_k is included in the subset if $\beta_k = 1$ $(0 \leq k \leq 3)$ and excluded otherwise. For example, the configuration 10110, where β_4, β_2 and β_1 are 1, specifies Q_1 and Q_2.

In a few instructions a different interpretation is attached to the modifier β^L; this interpretation is discussed in connection with the specific instructions to which it applies.

If a β,α pair is used to determine a modified designator at the time of instruction execution, the contents of the index registers specified by β are added to α to form δ. The use of the modified designator δ depends on the instruction and position (right or left) in which β,α appears. Typically δ functions as an address (of a memory location or register), an operand (to be used in 14-bit index arithmetic), a parameter (specifying e.g. the number of places in a shift operation), or a selection pattern for one of the indicator banks. The first of these is perhaps the most common use; in this case one may further distinguish the function of a designator as a "fetch address" (a word is to be transferred to the central processor) or a "store address" (a word is to be transferred from the central processor).

Index register modification is discussed more fully in Section G, Use of Modifiers.

5. Instruction operation

Insight into the way that the coding discussed actually determines instruction operation can be gained by careful perusal of the following examples, which are designed to reveal typical uses of the more important of the conventions described.

Example 1. Consider as a particular instance of an instruction word the string

1394093e0006.

In terms of the instruction format, this is broken down into bit-groups

as shown in Figure I-D2.

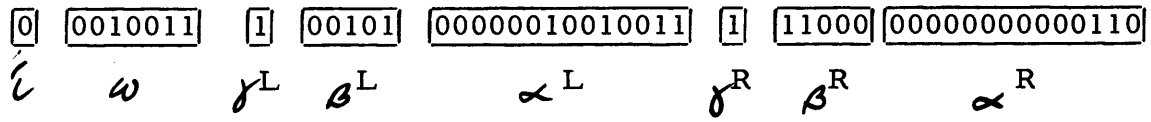

Figure I-D2. A typical instruction

The tag bit ι is seen to be 0 indicating that the instruction is "untagged". The configuration 0010011 of the operation code signifies class 1 (Floating Point Arithmetic), instruction number 3 -- i.e. the Floating Add-Store (FAS) instruction mentioned earlier. With this operation code the inflector γ^L has the ± interpretation; $\gamma^L = 1$ thus specifies a minus sign -, which effectively converts the floating point addition into a subtraction. The configuration 00101 of modifier β^L specifies selection of index registers P_0 and P_2. As the instruction is carried out, the 14-bit numbers residing in the selected index registers are added to the base address (sexadecimal) 0093 appearing as α^L; in this way the location of the subtrahend for the arithmetic operation is designated.

When the arithmetic operation is complete, the result is to be stored in the location specified by α^R modified by β^R. The configuration 11000 of the latter dictates selection of index register Q_3, the sequence register. Since α^R represents the integer 6, the location effectively addressed is Q_3+6, six locations beyond the instruction itself.

Finally, the inflector γ^R has the interpretation HC for this instruction, and the value $\gamma^R = 1$ signifies the C option. This causes register U to be cleared (all positions set to 0) following the store operation.

Example 2. A second example demonstrates a basically different use of the α and β sections of an instruction word, and also illustrates the function of two other inflector options. A Jump instruction is one which affords the possibility of changing the setting of the sequence register Q_3, thereby giving rise to a "jump" in the sequence of instructions performed. The computer vocabulary has a variety of such instructions, differing in the conditions under which the jump will be effective. One such instruction is the Jump on Index Negative instruction, which might appear as

677FFF80001,

which breaks down as shown in Figure I-D3.

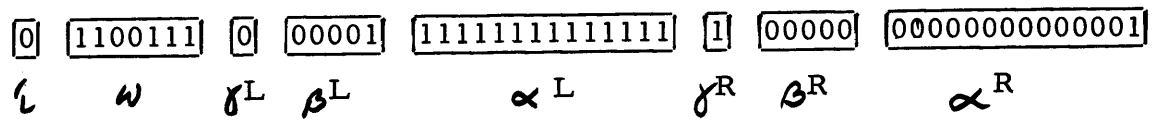

Fig. I-D3. Another typical instruction

Here the tag bit ι is again 0, as indeed it usually is, the contrary case causing an instruction trap (see Section K, Trapping). The operation code ω specifies instruction 7 of class 6.

The interpretation of γ^R, β^R, α^R in this case is common to

all jump instructions. The pair β^R, α^R is used to form a modified designator (here sexadecimal 0001, since no index registers are specified), and γ^R is interpreted as the Operand or Address option. It specifies whether the modified designator δ^R is itself the jump destination ($\gamma^R = 0$) or is instead the address of the location in memory whose α^R field gives the jump designation ($\gamma^R = 1$). In the example α^R is 1, which is the address of the U register, and $\gamma^R = 1$, so the jump, if effective, will set the sequence register Q_3 to the designator residing in the 14 rightmost bit positions of U.

In conditional jump instructions such as this one the left inflector γ^L gives the option Jump or Proceed. If $\gamma^L = 0$, the jump is effective if the condition (described subsequently) is met, and processing proceeds in the normal sequential order if the condition is not met. If $\gamma^L = 1$, normal processing proceeds when the condition is met, otherwise the jump is effective. In effect, a γ^L bit of 1 would convert Jump on Index Negative to a "jump on index not negative". However, $\gamma^L = 0$, so the jump will take place if the condition is met.

The modifier β^L in this case is used to specify a selection of index registers whose sum is to be regarded as a signed designator. The α^L field is an unmodifiable designator which is used immediately as a 14-bit operand. This operand is added to each of the index registers specified in β^L, and then these index registers are summed.

If the sum is negative, the condition is met. In the example, index register P_0 is specified, and α^L = 3fff, which is -1 as a signed designator.

The operation of this second sample instruction can be summarized: Decrement P_0 by 1. If the result is negative, take the next instruction from the location given by the α^R field of the U register. Otherwise, increment Q_3 by 1 and take the next instruction from the location now addressed by Q_3.

E. Function of Indicators

The rightmost four sense indicators D_3^S, D_2^S, D_1^S, D_0^S have the special function of recording the character of a word or string of bits. The character of the string is a four bit quantity with exactly one bit 1, which depends on the configuration of leftmost and remaining bits as shown in Table 1-E1. The leftmost bit is singled out here since it often has a special significance; it represents the sign in a numeric interpretation, the tag of an instruction, and so forth. The notation $\chi(s)$ stands for the character of the bit string s. Note that there are two uses for the word "character", the present one and the eight-bit characters for input and output discussed in Section B3. It should always be clear from the context which "character" is meant.

This definition of the character may be applied to any string of bits, but particular instructions exist for recording the character

of a word, an exponent, a coefficient, or a designator in the indicators $D^S_{3\ldots0}$.

Several of the trap indicators D^T_{13}, D^T_{12}, . . . D^T_0 also have special functions; they record the occurrence of anomalous events, or tagged instructions; see Section K, Trapping. Because of the unanticipated nature of most trap conditions, it is recommended that these indicators not be used for purposes other than those associated with trap control.

D^S_3, D^S_2, D^S_1, D^S_0	Leftmost bit	Remaining bits
0 0 0 1	0	all 0
0 0 1 0	0	some 1
0 1 0 0	1	all 0
1 0 0 0	1	some 1

Table I-E1. Character coding.

The sense and trap indicators can be set collectively by the Set Sense and Set Trap instructions or individually by the Jump on Pattern Comparison and Execute on Pattern Comparison instructions. The latter two also permit setting the indicators from sense switches on the console and permit conditional jumps to be made on the contents of selected indicators.

F. Use of Inflectors

There are seven basic interpretations of inflectors, as shown in Table I-F1.

Variable	Description	Value for = 0	= 1
±	Plus/Minus	+	-
RL	Right/Left	R	L
JP	Jump/Proceed	J	P
KT	Keys/Tape	K	T
OA	Operand/Address	O	A
HC	Hold/Clear	H	C
IE	Ignore/Examine	I	E

Table I-F1. Inflector interpretations

The effect of these in various contexts in which they may occur is summarized as follows:

a) ±

Depending upon the instruction in which a "±" inflection bit occurs, it is given one of three interpretations. It may be interpreted arithmetically, to indicate whether or not an operand in exponent, coefficient form fetched from memory is (-) or is not (+) to be negated. It may be interpreted formally in one of two ways: to indicate that the (numeric) complement of an operand is (-) or is

not (+) to be formed, or to indicate that the reflection (bitwise complement) of an operand is (-) or is not (+) to be formed. The operand is normally that specified by the designator accompanying the inflector (only exception: instructions CNM and CRM in class 4, where the operand is u).

b) HC (with store address)

The contents (or partial contents) of register U are stored, after which U may (C) or may not (H) be cleared.

c) HC (with fetch address)

An operand is fetched, and register U may (C) or may not (H) be cleared before the operation proceeds.

d) RL

A 14-bit designator is substituted into either the α^R field (R) or the α^L field (L) of some location.

e) OA

The modified designator functions as a 14-bit operand (O), or else (A) gives the address of the location where the 14-bit operand resides (in this case the reference is always to the right designator (α^R) field of the cited location).

f) JP

(used as a left inflector in conditional jump instructions.)

If γ^L is J: the instruction effects a program jump if a specified

condition is met, and the program proceeds in normal sequence if the condition is not met.

If γ^L is P: the program proceeds in normal sequence if the condition is met, and the instruction effects a program jump if the condition is not met.

g) KT

The instruction invokes either the console typewriter (K) or the paper tape equipment (T).

h) IE

At some point in the instruction the character of a specified string of bits (representing word, coefficient or exponent) may (E) or may not (I) be examined; if examined, its character is recorded in the indicators D_3^S, D_2^S, D_1^S, D_0^S.

There are a few instructions in which the inflection bits have interpretations other than those listed above. Such exceptional interpretations are described in Part II in the description of the relevant instruction.

G. Use of Modifiers

The manner in which the modifier bits β_4, β_3, β_2, β_1, β_0 specify the selection of one or more index registers in either of the

groups P or Q has already been mentioned; Table I-G1 gives the selection coding explicitly.

Bit	$\beta_4 = 0$ (group P)	$\beta_4 = 1$ (group Q)
0	P_0	Q_0
1	P_1	Q_1
2	P_2	Q_2
3	P_3 (reference register)	Q_3 (sequence register)

Table I-G1.
Index register selection coding

In most cases the modifier β is used in conjunction with its associated designator α to form a modified designator δ ; this is accomplished by adding to α the contents of the index registers specified by β, the addition being performed modulo 2^{14}.

The β configuration 00000 is interpreted consistently with the above selection scheme; i.e., to specify selection of no (the empty set of) index registers. The configuration 10000, which would be equivalent in effect to 00000 if interpreted naturally, is used for a special purpose to call for indirect designator specification. If β is 10000, a new pair β', α', is obtained (at execution time) from the β^R, α^R fields of the location whose address is α, and this new pair is used to form a modified designator δ'. If β' is

10000, a third pair β'', α'' is obtained, and this process may be repeated indefinitely. Note that indirect specification is subject to the following limitations: the initial address cannot be modified in the conventional way (since β is used to indicate indirect specification), and subsequent modified designators are restricted to the right-hand fields of the words they occupy.

Note further that if an OA inflector option is associated with a β, α pair it takes effect after the modified designator δ is formed; thus one might have one more level of memory reference on top of that specified by the β patterns. To distinguish the two kinds of address specification involved here, it is convenient to think of the modifier pattern β as giving a choice between <u>direct</u> ($\beta \neq$ 10000) and <u>indirect</u> (β = 10000) addressing, while the OA option gives a choice between <u>immediate</u> (O - the operand is contained in the instruction) and <u>standard</u> (A - the operand is addressed by the instruction) addressing forms. (Note that the OA option references only α^R of the addressed word, whereas indirect addressing uses β^R and α^R.)

When a β field is not used to modify the associated designator, but rather to specify a selection of index registers to be used in some other manner, the modifier 10000 is equivalent to 00000 and specifies no index registers.

H. Number Representation

As stated earlier, the 48-bit word is partitioned into an 8-bit exponent part and a 40-bit coefficient part for purposes of numerical manipulation. The exponent bits are numbered from right to left, and the coefficient bits from left to right, starting from 0 in either case. The format is diagrammed in Figure I-H1.

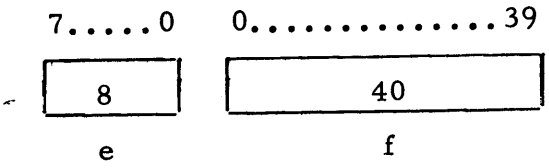

Figure I-H1. Numeric format

Any sequence $f_0, f_1, \ldots f_{39}$ of bits in the coefficient field represents a number f in the range $-1 \leq f \leq +1-2^{-39}$ according to the formula

$$f = -f_0 + \sum_{k=1}^{39} f_k \cdot 2^{-k}$$

(see Table I-H1). In this scheme nonnegative numbers are represented in standard binary form (with implicit binary point between f_0 and f_1), while negative numbers are represented in the corresponding (true) complement form. The bit f_0 is called the sign bit of the representation; $f_0 = 0$ when $f \geq 0$ and $f_0 = 1$ when $f < 0$. If f is negative, the sequence of bits to the right of f_0 represents (in standard binary form) the positive number f* satisfying $f^* + |f| = 1$, so that $f = -1 + f^*$. From this it is

Code	Number
10 00	-1
10.....01	$-1+2^{-39}$
10.....10	$-1+2^{-38}$
. . .	. . .
11.....11	-2^{-39}
00.....00	0
00.....01	$+2^{-39}$
. . .	. . .
01.....10	$+1-2^{-38}$
01.....11	$+1-2^{-39}$

Table I-H1. Coefficient coding

easily seen that, as indicated by the table, every positive number f which can be represented has a negative -f which can also be represented, and the only negative number for which the reverse does not nold is f = -1.

For f > 0, the number $d \geq 0$ of leading digits of f is defined as the number of consecutive bits 0 appearing to the right of the binary point in the representation of f. Evidently d = 0 when $1/2 \leq f < 1$, d = 1 when $1/4 \leq f < 1/2$, and so forth. When f = 0,

$d = 39$ by convention, and for $-1 < f < 0$, d is defined to be the number of leading digits in the positive number -f. For all negative $f \neq -2^{-k}$, this is just the number of consecutive bits 1 appearing to the right of the binary point in the representation of f; if $f = -2^{-k}$, d is one less than this number. If $f = -1$, d is taken as -1 by convention.

Any sequence $e_7, e_6, \ldots, e_0$ of bits in the exponent section, except the sequence 00 . . . 0, represents an integer e in the range $-127 \leq e \leq +127$ according to the formula

$$e = (e_7 - 1) \cdot 2^7 + \sum_{k=0}^{6} e_k \cdot 2^k$$

(see Table I-H2). This is an "excess-128" scheme, in which each component e is represented by the sequence of bits which would represent e + 128 in standard binary form. The bit e_7 is called the exponent sign; $e_7 = 1$ signifies $e \geq 0$, while $e_7 = 0$ signifies $e < 0$. For each possible positive exponent e there is a corresponding negative -e, and vice versa, without exception. The sequence 00 . . . 0, however, is not assigned a numerical value; it is denoted e_Z and called the exponent for "absolute zero," as explained below.

If $e \neq e_Z$, the pair (e, f) is taken to represent a number x,

$$x = f \cdot 2^e .$$

If $f_1 = f_0$, the same number x is obtained by shifting the coefficient pattern one place left (i.e., replacing f_k with f_{k+1} for $k = 0, 1, \ldots, 38$, and f_{39} with 0), and subtracting 1 from the

exponent, since

$$2f \cdot 2^{e-1} = f \cdot 2^{e} = x,$$

and the left shift effectively doubles the coefficient. If $f \neq 0$, this process can be repeated until $f_1 \neq f_0$, in which case the number x is said to be in normalized form. In general, d = 0 for a normalized

Code	Number
00...01	-127
00...10	-126
.	.
.	.
.	.
01...11	-1
10...00	0
10...01	+1
10100111	+39
.	.
.	.
.	.
11...10	+126
11...11	+127

Table I-H2. Exponent coding

form; the only exception occurs when $x = -2^{-k}$ for some $k > 0$, in which case f = -1 in the normalized form and d = -1. For f = 0, the normalized form itself is undefined, since the number x = 0 is

represented by f = 0 with any numerical exponent e whatever; for purposes of standardization, however, e = -127, f = 0 is adopted as the "normalized" representation of zero. The number x = 0 is also denoted by $e = e_Z$ and any coefficient f whatever; this version of 0 is termed absolute zero, and is subject to different arithmetic rules than the versions with coefficient 0 and $e \neq e_Z$, any of which is termed relative zero.

The difference between the two types of zero will become clearer in the treatment of arithmetic manipulation on numbers in exponent-coefficient form which follows.

The (sexadecimal) a7 exponent, i.e. (decimal) 39, effectively places the binary point at the right end of the coefficient. This exponent is therefore often possessed by numbers being manipulated as integers. The V register mentioned in Section A as containing -1 has this exponent; its contents, in sexadecimal, is a7fffffffff.

I. Types of Arithmetic

The foregoing discussion shows that number representation in the coefficient-exponent system is, in general, not unique. It has been customary to take the normalized representation as standard, with special rules to cover the case of zero coefficient. Thus "floating point" arithmetic units are conventionally designed to handle operands in normalized form, and to generate normalized results. In Maniac III, by contrast, the arithmetic unit handles unnormalized as well as

normalized operands in a meaningful way, and generates results whose form depends not only on the form of the operands, but on which of several types of arithmetic is called for: Floating Point, Specified Point, or Normalized. Floating Point results are adjusted according to a "significant digit" criterion, while the results of Normalized operations are adjusted to normalized form, regardless of whether the operands were normalized or not. Specified Point operations generate results whose form depends on the operand exponents. It should be noted that Maniac III has no "fixed Point" arithmetic which operates on numbers in pure coefficient form. Hence the usual distinction between "fixed point" and "floating point" numbers need not be made in numerical work, and no formal conversion is necessary to employ the result of one type of arithmetic operation as operand for another.

The computer operations which are relevant to the discussion of this section are: Add, Multiply, Divide (Floating Point); Add, Multiply, Divide (Specified Point); and Normalized Add. All Add, Multiply and Divide operations can be performed with a sign (±) option for the second operand; hence subtraction is included as a case of addition.

Normally, two words are used to express the complete result of an arithmetic or numerical operation in the computer. These are developed in the U and R registers. In some operations the result u, r is to be interpreted as u + r (with r to a low order part, small in magnitude compared to u), in others as u with remainder r. In

any case u is the approximate result generally used in further computation.

For each of the operations under consideration there is defined a "true" (arithmetically exact) result, expressible as a single number in the case of Add and Multiply, or as two numbers one of which is a remainder, in the case of Divide. Furthermore, these numbers are all representable with a finite number of binary digits, and their adjustment (binary point positioning as expressed by the exponent) is uniquely determined. Sometimes, however, this "true" result is not produced by the computer, for reasons as follow:

(a) The desired exponent is out of the representable range

$-127 \leq e \leq +127$;

(b) The desired coefficient is out of the representable range

$-1 \leq f \leq +1-2^{-39}$;

(c) The coefficient is truncated so that some of its rightmost digits are lost.

The three effects are, of course, not unrelated; in fact, a case of (a) where $e < -127$, if it arises during the course of an operation, is generally converted to a case of (c) by successively halving the coefficient (by right-shifting, so that low order digits are lost), and augmenting the exponent until it reaches the value -127.

During the running of a program, if a final result is generated in which either (a) or (b) obtains, a trap condition is initiated, unless

specifically overridden by instructions available for the purpose; see Section K, Trapping. In case of (c), however, no special computer action is invoked, since the numerical error committed, unlike that of the other two cases, is small. In all three situations, moreover, the result actually developed by the computer can be interpreted in terms of the "true" one; in (a), all digits of the exponent except the sign are correct, in (b) and (c) the coefficient sign is the actual one, and all other digits are those which would appear in the corresponding part of the coefficient if it were to be developed in full.

A trap condition also occurs in special cases where no "true" result is defined (e.g. because of zero divisor). In some such situations a partial or related result is generated, depending on the nature of the anomaly.

In an operation whose result is defined as u + r, the nonsign coefficient positions of R generally contain the low-order continuation of the coefficient residing in U; that is, the coefficient of the result is represented by the 69-bit sequence $u_0^F, u_1^F, \ldots u_{39}^F, r_1^F, \ldots, r_{39}^F$. The u + r interpretation is consistent with this if $r^E = u^E - 39$ and $r_0^F = 0$. This rule for the exponent is the one actually used, unless $r_0^F < -127$; in this case, r is adjusted to exponent $r^E = -127$ as explained earlier. The u + r interpretation then still holds (although r^F may have lost some nonzero digits at the right), but the 79-bit interpretation must be modified.

The 79-bit interpretation is also affected when $r^F_0 = 1$, which occurs frequently because of the method of rounding employed in the Maniac III system; see Section J, Rounding Rules.

Because numbers are not required to be normalized, relative zero results may be expected to occur more often in this arithmetic system. Cases where one or both of u and r are relative zero as the result of the operation arise naturally, and are subject to no conventions additional to those described above. Absolute zero (exponent e_Z) can in general occur as a result only if one or possibly both operands are absolute zero (specifically, both operands in addition, either operand in multiplication, and one operand (dividend) in division). In these cases, both u^E and r^E are set to e_Z, and both u^F and r^F to 0.

One further property of absolute zero deserves mention. In the natural ordering of computer numbers, absolute zero is taken as greater than any number with negative sign and less than any number with positive sign; the latter category includes all relative zeros. The effect of this convention is only apparent in the result of the (quasi-arithmetic) instruction Jump on Arithmetic Comparison, in which a jump condition is considered to be met if and only if $u \geq x$, where x is a number in memory. If u is absolute zero while x is relative zero, the condition is considered not to be met, since by the above definition $u < x$ under these circumstances.

The operations Add, Multiply, and Divide each require two operands; one of these is taken from register U, the other from memory. The operand from memory is first fetched to register S. The procedure then implemented can be symbolized by

$$u, r \leftarrow u\omega\ (\pm s)$$

where ω is either +, ·, or /, and the ± alternative is specified (by an inflector) in the instruction calling for the operation.

The operations designated Floating Point are defined as follows: Let (e_1, f_1) and (e_2, f_2) be operands with d_1 and d_2 leading digits, respectively; then a result $(e \cdot f)$ with d leading digits is developed according to the rule $e = \max(e_1, e_2)$ for Add, and $d = \max(d_1, d_2)$ for Multiply and Divide. In each case the rule given is enough to uniquely determine the form of the result; for Add, the final d is generally unpredictable (and indeed, it may be that $d > d_1$ and $d > d_2$ by considerable amounts, if the operands are nearly equal and the operation is in effect a subtraction). For a more detailed description of the results of these operations, including the remainder of a division, see the relevant instructions in Part II.

If the value of each operand is regarded as uncertain in the 39th place, then the transmitted uncertainty in the result, formed according to these rules, will also be approximately in the 39th place; hence, the informal characterization "significant-digit arithmetic."

Exception is made to the rules as given when following them would give a coefficient out of range, or an exponent less than -127; to remedy

either of these conditions requires a shifting of coefficient which is in the direction of decreasing apparent significance. Adjustment for exponent exceeding +127, however, requires a shift in the opposite direction; even if this would not take the coefficient out of range, it would increase its apparent significance, and so the step is not undertaken. The case therefore causes a trap condition.

Relative zero operands in multiplication and division are handled in a way consistent with the interpretation d = 39; the result is always zero with an appropriate exponent (even in the case of relative zero divisor, which is a trap condition). Absolute zero operands, however, act like the true zero of the number system, obeying the rules

$$x + 0 = 0 + x = x$$

$$x \cdot 0 = 0 \cdot x = 0$$

$0/x = 0$; $x/0$ undefined (trap, no operation).

When absolute zero is produced as a result, it is always with coefficient 0, regardless of any nonzero coefficients which may be attached to the absolute zero operands involved.

The multiplication and division operations described above have the property that if they are performed on normalized operands, the results are generally produced in normalized form also, unless the normalized result would have exponent less than -127. (A further exception, of generally trivial consequence, arises for a result which

is exactly the negative of a power of 2, since the d = 0 form is not normalized in this case.) Hence only one supplementary operation, Normalized Add, is needed to permit calculation to be carried out essentially in standard normalized fashion. The major qualification "essentially" points up the exceptions just mentioned; in particular, when results computed in this way become small or zero, they are systematically "denormalized" so that the condition $e \geq -127$ remains satisfied. Normalized Add gives a normalized, rounded result regardless of the original adjustment of the operands, whenever this is possible keeping the exponent within range. If the normalized result would have an exponent $e > +127$, the coefficient is still in normalized form, and all digits of the exponent except the sign are correct, but as in the case of Floating Point Add a trap condition occurs. If the normalized result would have an exponent $e < -127$, the result is presented at the exponent -127 in unnormalized form, as in the case of Floating Point Multiply and Divide. As a special case of this, a zero result of a Normalized Add operation always appears as a relative zero with $e = -127$.

The third type of arithmetic to be discussed is Specific Point, which supplants the conventional "fixed-point" operation. Here Add, Multiply and Divide are all three subject to the same type of rule: the result is adjusted so that its exponent is equal to the exponent of the first operand (that originally residing in U). The only exceptions

occur when absolute zero operands are involved, in which case the behavior is the same as for the Floating Point operations.

Following the Specified Point rules obviously never leads to out-of-range exponents, but the required adjustment may cause the coefficient to exceed its permitted magnitude. When this happens the sign and remaining bits which do appear are correct (i.e. are part of the true result), but since high order digits have been lost a trap condition obtains.

J. Rounding Rules

For Add or Multiply, rounding is defined as follows: Consider a coefficient sequence $f_0, f_1, \ldots, f_n$ of arbitrary length n, and suppose it is desired to replace this with a shorter sequence whose last bit is in place p. If (i) $f_p = 1$ or (ii) $f_{p+1} = \ldots = f_n = 0$, the rounded sequence is taken as $f_0, f_1, \ldots, f_{p-1}, 1$. Thus the last bit of the rounded sequence is always 1, except in the case where it originally is 0 and the truncation involves no approximation (i.e. only bits 0 are dropped).* It should be noted that under these rules the

*This rounding scheme has twice the spread ("variance") of the more conventional one where u is altered by addition of 1 in the pth place if the part dropped exceeds 1 in the (p + 1)st place; however, the sceme described here also has some advantages, both theoretical and practical, over the traditional one. First, it is symmetric ("unbiased"); the mean of the numbers which round to a particular value u^F is u^F. Second, it is nonpropagative; only the rightmost digit of the number u^F is altered in rounding, and this both simplifies the mechanization of the computer operations and permits their extension to multiprecision work.

operations of rounding and negation commute, i.e., the negative of a rounded value is the rounded value of the corresponding negative.

In practice, a rounded result u is computed from a full result (u + r) by "forcing" u^F_{39} to 1 if any digits 1 have been "shifted off" into the low-order region. If $u^F_{39} = 0$, originally, this amounts to adding 2^{u^E-39} to u; if $r^E = u^E - 39$, this incrementation can be compensated for by setting r^F_0 to 1, thereby subracting 2^{u^E-39} from r and leaving the u + r interpretation valid. This step is always taken before any necessary adjustment of r for $r^E < -127$; thus it can be said that the (u + r) interpretation always holds, up to truncation at the right end of the representation of r, given only that the exponent and coefficient of u have themselves stayed within range.

For Divide, the quotient u is effectively rounded by the same rule, by "forcing" u^F_{39} to 1 if the remainder is nonzero. The remainder r is then compensated so that the relation dividend = quotient ✕ divisor + remainder is exactly satisfied, except in extreme cases such as when $r^E < -127$ necessitates adjustment of r so that low-order bits are lost.