ANL-5995
Physics and Mathematics
(TID-4500, 14th Ed.)
AEC Research and
Development Report

ARGONNE NATIONAL LABORATORY
P. O. Box 299
Lemont, Illinois

GEORGE PROGRAMMING MANUAL

by

Loretta Kassel

with

Appendices by Donald A. Flanders

May, 1959

Operated by The University of Chicago
under
Contract W-31-109-eng-38

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

TABLE OF CONTENTS

	Page
I. GENERAL INFORMATION	3
A. Introduction	3
B. GEORGE Components	7
C. GEORGE Operations	11
D. GEORGE Words	13
E. Console Operations	15
II. GEORGE ORDER STRUCTURE	22
A. Index of Orders	22
B. The Tag	23
C. Orders	27
D. Order Decoding Chart	52
E. Overflow Control	53
F. Timing	54
III. PROGRAMMING TECHNIQUES	55
A. Machine Language	55
B. GEORGE Assembly Routine (GAR I)	82
IV. AUXILIARY EQUIPMENT	94
A. Off Line	94
B. On Line	103
V. BIBLIOGRAPHY OF GEORGE ROUTINES	105
APPENDIX A. Boolean Algebra	115
APPENDIX B. Representation and Notation for Numbers	123
APPENDIX C. GEORGE Logic	149

GEORGE PROGRAMMING MANUAL

Loretta Kassel
with
Appendices by Donald A. Flanders

I. GENERAL INFORMATION

A. Introduction

The purposes of this manual are to provide a description of the operating features of the digital computer GEORGE and to present a practical set of instructions that will enable a prospective user to construct codes for GEORGE, operate the machine and its auxiliary equipment, use the basic routines available in the routine library, and decide whether a particular problem is suitable for GEORGE on the basis of capacity, speed, and auxiliary equipment. As additional auxiliary equipment and/or general purpose routines become available for usage, supplements to this manual will be issued.

GEORGE is an automatic high-speed electronic digital computer designed and constructed by the Argonne National Laboratory. When the initial design work and the early construction work were begun by the Computer Group and the Mathematics Group, these groups were parts of the Physics Division under the direction of L. A. Turner. The Computer Group was directed by J. C. Chu and the Mathematics Group by D. A. Flanders. Subsequently the Computer Group became a part of the Electronics Division under the direction of Thomas Brill, and the Mathematics Group became a separate division, the Applied Mathematics Division, directed by the late D. A. Flanders. The order, structure, and logic of the main frame of GEORGE were the end product of discussions by the following mathematicians and engineers: J. C. Chu, D. A. Flanders, Ray Kramer, Sr., Jean Hall, Jim Butler, Joe Cook, Herbert Gray, Lester Kornblith, and Margaret Butler. In addition to these, the following people made major contributions to the construction of GEORGE and to the design and construction of its auxiliary equipment: D. H. Jacobsohn, L. C. Merrill, T. Brill, Loren Collins, Stanley Hanson, James Whalen, Forrest Salter, R. Aschenbrenner, and Curtis Rockwood.

In August, 1957, the basic components of GEORGE became available for use on a part-time basis, and in October, 1957, they came into full-time operation. The separate auxiliary units, such as the floating point unit, the wide magnetic tape supplementary storage, and the narrow magnetic tape input and output units, are still under construction.

The remainder of this section is a brief, general introduction to the use of high-speed computers for the benefit of complete novices. Those readers who are familiar with computers may pass over the remaining paragraphs of this section.

A calculation is carried out on an automatic computer by providing the computer with an ordered set of instructions, called a code, which describes the operations necessary for carrying out the calculations, and a set of data on which the code operates. The instructions are of two kinds: arithmetic instructions (for example, additions and multiplications, and logical instructions, such as store orders) which cause the machine to put numbers (or instructions) in a specified place in the storage unit, and jump orders, which tell the machine to jump to a designated portion of the code. Altogether there are about 56 possible arithmetic instructions and about 163 possible logical instructions available on GEORGE. Thus, the first step in carrying out a calculation is the reduction of the calculation to the sequence of arithmetic and logical instructions which make up the code. The code is then put into the storage unit of the machine so that the machine operates on an internally stored code. Storage units are different for different machines. GEORGE, for example, has a 4096-word storage of the magnetic core type. That is, a total of 4096 words of instruction and words of data (numbers) may be stored in GEORGE at any one time. Each of the 4096 positions in the storage is composed of 40 magnetic cores which have two states of magnetization, representing either a one or a zero. If the word in storage is to be treated as an instruction, the state of each of the 40 magnetic cores contributes to determining what instruction will be carried out. If the word in storage is to be treated as a number, the composite of the 40 magnetic cores determines the number (see "Instructions" and "Numbers" under Section I. D. "GEORGE Words").

The mode of input, for "reading" code or data into the storage unit, also varies from machine to machine. The three most common input modes are punched cards, punched paper tape, and magnetic tape. Any given machine may have several optional modes of input. At present, the principal input into GEORGE is by means of punched paper tape. The instructions and data are punched in coded form on paper tapes which are then "read into" the computer by passing the punched tape through a photoelectric reader which reads the code on the tape and puts it in the storage unit. The computation is then started by manually instructing the machine to jump to the first instruction in the code; the machine then proceeds automatically and sequentially from instruction to instruction.

Output modes also vary with machines. Common modes of output are punched cards, punched paper tape, magnetic tape, or printers directly coupled to the machine. Output in the form of punched cards, punched paper tape, or magnetic tape must be "run through" auxiliary equipment which

interprets the information on the cards, etc., and activates a printer which prints this information. The principal output mode for GEORGE, at present, is the punched paper tape.

By means of the so-called jump orders the machine may be instructed to repeat a portion of the instruction code for a new set of data. One of the instructions may be to jump to an assigned part of the code if a certain condition is satisfied, but to continue to the next order in the code if the condition is not satisfied. For example, one such condition could be to jump if the result of an operation is a negative number. Repetition of a sequence of orders until a given condition is satisfied is called a loop. It is the fact that the machine can make conditional jumps and, therefore, go through loops that provides the automatic features of the computation.

In several respects a calculation is carried out on a high-speed computer in much the same manner as one carries out a calculation on a desk calculator. However, it is immediately clear that there should be several major differences between hand computing and machine computing processes. For example, the advantages of the high speed of the computer are lost if one has to inspect the results at intermediate steps. Thus, many of the decisions that may be made as the progress of hand computing unfolds must be coded into the problem for a high-speed machine. In a hand computation one might go to considerable effort to avoid frequent repetition of a set of simple operations, whereas in a machine code one might prefer to allow the machine to repeat the simple parts of the code.

The manner in which the code itself is constructed will vary greatly with the machine and the particular utility routines available for the machine. The most elementary procedure is to write the whole code directly in the machine language. It is quickly recognized that certain parts of a code will receive rather universal usage and can be put in a form for simple incorporation in any program, for example a square-root code of an exponential code. Such codes are called subroutines. Furthermore, one sees that most of the bookkeeping connected with constructing a code can be done by the machine itself - hence the advent of the assembly routine which, as the term implies, assembles separate parts of a code, inserts the appropriate addresses of variables in the code, and in general does the bookkeeping. One can go one step further and construct a code that will not only do the bookkeeping but will interpret certain mathematical statements and construct a code to carry out the computations indicated by the mathematical statement. Such utility codes are called compilers. The extent to which one can go in the construction of assembly routines and compilers depends on the basic machine available and its auxiliary equipment. Compilers generally require extensive auxiliary storage, for example. Some problems are better suited for using a particular one of the coding techniques than for the other techniques. Thus there is one more step in the process - that of deciding on the appropriate technique of coding.

The steps, then, in preparation and running of a problem on a high-speed computer are from beginning to end the following:

1. The problem to be solved must be reduced to a precise mathematical problem.
2. The mathematical problem must be reduced to a sequence of arithmetic and logical steps.
3. The arithmetic and logical steps must be incorporated into a machine code. Various coding techniques are available for this process.
4. The code must be tested to see that it does indeed carry out the arithmetic and logical steps prescribed and must be tested further to see that the set of arithmetic and logical steps actually solve the mathematical problem proposed.

B. GEORGE Components

1. Memory Unit

The GEORGE Memory (or Storage Unit) consists of three parts - the magnetic core storage, which is the principal storage unit, the magnetic tapes, which act as an auxiliary memory, and the memory register, which acts as a buffer unit between the core memory and the arithmetic, control and auxiliary memory units.

a. Magnetic Core Storage:

A program of instructions for a given problem and the associated input data must be stored in the magnetic core storage unit (core memory) before the code and data can be used by the control and arithmetic units in the solution of the problem. A total of 4096 cells, each cell capable of holding one 40-bit word (see Section I. D. "GEORGE Words"), are available for storage in the core memory.

Each cell in the core has a numeric address associated with it. An address is represented by twelve binary bits, or three hexadecimal characters. The cells are numbered consecutively from $(0000\ 0000\ 0000)_2$ or $(000)_{16}$ to $(1111\ 1111\ 1111)_2$ or $(FFF)_{16}$. Reference to any cell in the core can be made by using the appropriate address as a B or A address in an instruction.

Magnetic cores, i.e., rings of ferromagnetic material, are used to hold the bits of information constituting a stored word in the core memory. Passing a current through a wire inserted through the hollow of a core sets up a magnetic field around the wire. Reversing the direction of the current reverses the direction of the magnetic field. In either case, the core remains magnetized after the current is removed; by using one state to represent a one and the other a zero, the cores can, in combination, easily represent binary numbers. The cores can retain information indefinitely and recall it in a very minute amount of time (27 microseconds).

b. Wide Magnetic Tapes

Four wide magnetic tape units on GEORGE provide an auxiliary memory with storage capacity of one-million words each. Information is stored on the 1200-foot reels of tape in binary form by means of magnetized spots, 40 bits per word, in blocks of 128 words with gaps between blocks.

Since words stored in the auxiliary memory must be sent to the core memory before use in the calculations of a program, the chief uses of the tape units during a program are to hold data and instructions not currently relevant, but which, later during a program, will write over data

and/or instructions currently held in the core memory, or to hold blocks of data to be output-ed in a later part of a program. Tapes are also commonly used to store permanent programs, subroutines and service routines.

c. Memory Register (MR)

The Memory Register is a 40-bit register which acts as a buffer unit, or transfer register, between the magnetic core storage and the magnetic tape, arithmetic, and control units.

It is always the contents of MR that are sent to the memory when a Store, Input, or Read Tape instruction is executed, with the word(s) to be stored always initially sent to MR (one word at a time), then to the memory, during the instructions. Likewise, each word recalled from the memory for use by the control, arithmetic, output, or magnetic tape unit is always sent first to MR, and from there to the unit or register calling for it. The Memory Register is strictly an auxiliary register and is not addressable. However, several instructions are available which make use of the current contents of MR.

2. Arithmetic Unit

All arithmetic operations on GEORGE are performed with the use of three registers: the Accumulator (Ac), the Quotient Register (Qt), and the Plusser or Adder (P). All are 41-bit registers - 40 bits are arranged as in a normal GEORGE word (e.g., $a_0 a_1 \dots a_{39}$), and the other bit, called the x-bit (or a_x , q_x , etc.) immediately precedes the 0-bit of the register. The function of the x-bit is described later in this section and under the explanations of the orders which use it.

Since GEORGE uses complementations to represent negatives of numbers, all add-type operations can be reduced by the machine to simple additions - e.g., subtract becomes complement and add, add magnitude becomes add or complement and add.

References to additions in this section apply to all add operations, including preliminary adds, Σ orders, and the adds done in π orders.

a. Plusser (P)

In all GEORGE additions, the contents of MR and the contents of either Ac or Qt are added in the Plusser and the sum returned to either Ac or Qt. The selection of Ac or Qt is determined by the particular instruction being executed.

Since MR has no x-bit, during the addition MR is assumed to have an x-bit equal to its 0-bit. If a carry into the x-bit occurs in an addition, the carry overflow toggle (c_{OF}) is set. This toggle is examined by one class of Jump order.

The Plusser is neither addressable nor usable at the Programmer's discretion, but is used by the machine only as a tool in performing additions.

b. Accumulator and Quotient Registers (Ac and Qt)

All arithmetic operations and most store orders on GEORGE require use of either the Accumulator, the Quotient Register, or both. The registers are symmetric in all arithmetic operations except multiplication and division, as well as in all store orders except β orders. In general, one or two bits in the instruction determine which register will be used when both are equivalent.

Ac and Qt are 41-bit registers ($a_x a_0 a_1 \dots a_{39}$ and $q_x q_0 \dots q_{39}$, respectively). As arithmetic registers, either can be used to hold a summand and/or the sum. In multiplication, Qt holds the multiplier, MR the multiplicand, and the product, a 79-bit number, is left in Ac Qt - the 40 more significant bits in Ac, the last 39 bits in Qt ($q_0 - q_{38}$). Also, 2^{-39} times the original contents of Ac is added to the product. In division, Ac and Qt together form an 80-bit dividend (more significant part in Ac), MR holds the divisor, Qt holds the quotient and Ac the remainder.

Ac or Qt may be used as a transfer register - to move a word from one location to another - by adding the word to a cleared register (via the plusser) and storing the result. Ac and Qt are also used separately or together as shift registers. By an appropriate instruction, the contents of either register can be shifted left or right, to achieve a logical rearrangement of bits, or a multiplication of the register contents by a power of two. When Ac and Qt are treated as one 80-bit register in shifting, either can be chosen to hold the more significant part of the number to be shifted; the 0-bit of the other register is then considered the 40-bit of the double register, the 1-bit the 41-bit, etc.

The x-bit of Ac or Qt serves as:

1. the sign bit, with $a_x = a_0$ (or $q_x = q_0$) if the number is in range;
2. an overflow indicator, i.e., a number is considered out of range and a_{OF} or q_{OF} is set if $a_x \neq a_0$ or $q_x \neq q_0$, respectively (overflow may be set even when the 0 and x bits agree in the final result of an instruction, as in divides and shifts; see Section II. E. "Overflow Control"); and
3. means of extending the range of numbers used only in Ac or Qt in add operations (see Section III, "Programming Techniques").

3. Control Unit

Five registers and a special adder of the control unit are used in conjunction with other GEORGE components in the execution of a GEORGE instruction. The contents of the five registers are displayed in an array of neons on the console, as an aid to debugging and program monitoring.

a. Function Register (FR)

The 40-bit Function Register contains the current instruction being executed. FR is actually treated as four registers, FRB, FRO, FRT, FRA, holding the effective B-address, the Order, the Tag, and the unindexed A-address, respectively, of the current instruction.

b. Dispatch Register (DR)

The Dispatch Register is a 12-bit register used to hold the address of the most recent word consulted in any memory cycle of an instruction.

c. Control Counter (CC)

The Control Counter, made up of 12 bits, always holds 1 + the address of the instruction currently in FR. It is normally increased by one after each instruction is completed. If a jump occurs during an instruction, the address of the instruction jumped to is sent to CC before CC is increased and the next instruction begun. OOO is considered next in sequence to FFF by CC. The contents of CC can be stored with a Record order.

d. Shift Counter (SC)

The Shift Counter is used by Shift and Input-Output orders to count the number of shifts performed. 7 bits make up the Shift Counter - sufficient to count to 128, the maximum amount of shift allowable. A description of the use of SC appears under "Shift Orders" (Section II. C. 2). The Record order may also be used to store the contents of SC into the memory after any instruction.

e. Jump Register (JR)

The Jump Register is a 12-bit register which holds the address + 1 of the last instruction executed which called for a jump; that is, each time a Jump is executed, the contents of CC are sent to JR.

f. (A + B) Adder

The (A + B) Adder is a 12-bit adder which obtains the sum $FRA + FRB$ during an indexing operation. This sum is always sent to DR, not to FRA. As in CC, the sum obtained here is modulo 4096; e.g., $\underline{FFF} + \underline{001} = \underline{000}$.

C. GEORGE Operations

To execute a particular instruction, GEORGE must proceed through a sequence of events involving register clears and memory cycles. Four such cycles can occur within each instruction; three of the cycles are optional, and their occurrence or non-occurrence depends on the particular instruction. Only one cycle of each type can occur during a particular instruction.

Assuming the instruction in question is in the Function Register, the possible events in the order of their occurrence are as follows:

Preliminary Clear, followed by:

1. Preliminary Cycle
2. Operand Cycle
3. Result Cycle
4. Fetch Cycle.

Preliminary Clear (PCℓ)

The preliminary clear prepares one of the arithmetic registers for use during the remainder of the instruction - it must occur in each instruction. Four clears are available, none of which involve use of the memory. The Section II. B, "The Tag," contains an explanation of the types of clears and their use.

1. Preliminary Cycle (PC)

A preliminary cycle occurs during an instruction if either a preliminary addition or B-Boxing, or both, are asked for in the instruction's tag and always involves reading from the memory at B into MR. How the contents of MR are used after the word at B is sent there is described in the order list under the section on "The Tag."

2. Operand Cycle (OC)

The execution of the fundamental operations of most instructions occurs during the operand cycle. Only the Stop orders, the various Store orders, and the Jump orders omit the operand cycle, since these operations by their nature must occur during another cycle (e.g., α order - result cycle; Jump order - fetch cycle). If the operand cycle uses the memory, either the A or A-indexed address is consulted.

3. Result Cycle (RC)

A result cycle writes the contents of Ac, Qt, or MR into the memory. If the contents of Ac or Qt are being stored, they are first sent to MR; then MR (all or part of it) is stored in the memory as prescribed in the instruction. If only the contents of MR (or part of it) are being stored, Ac and Qt are not affected.

A result cycle may be asked for by the order or the tag and may require use of either the A, A-effective, B or B-effective address. (The A-effective and B-effective addresses arise from the use of B-boxing and are explained in that section of the write-up of "The Tag.")

4. Fetch Cycle (FC)

During the fetch cycle, the next instruction is brought into FR, ready to be executed. Thus, FC always uses the memory and must occur in each instruction. If no Jump is involved, the address consulted comes from CC, and hence is the address of the instruction located in the memory immediately after the current instruction. If a Jump occurs, the address consulted is either A, A-effective, B, or B-effective address of the current instruction.

The complete FC brings the next instruction from the memory to MR, and from MR to FR. The completion of FC normally starts the execution of the next instruction. If, however, a Stop occurs during the execution of the instruction - and the decision that a stop is to occur may be made at any time from PC to RC - then all cycles preceding FC are completed, the next instruction is sent to MR, but the transfer from MR to FR is not made, and FC is left uncompleted. The subsequent transfer from MR to FR (if desired) is made when GEORGE is manually restarted. Thus, in case of a stop, the current instruction is in FR and the new instruction is in MR.

D. GEORGE Words

A word, as used by GEORGE, is made up of 40 binary bits, say $a_0 a_1 \dots a_{39}$. For abbreviation, a word may be also treated as 10 hexadecimal characters (sigits) $A_0 A_1 \dots A_9$, where $A_0 = 8a_0 + 4a_1 + 2a_2 + a_3$, $A_1 = 8a_4 + 4a_5 + 2a_6 + a_7$, etc. For alpha-numeric purposes, a word may be used to represent five 8-bit (or 6-bit) characters. In this case, the first 2 bits of each character are irrelevant; the last 6 bits represent a decimal digit, a letter of the alphabet, a punctuation mark, a mathematical symbol, or a typewriter format control symbol, according to a uniformly specified code.

1. Numbers

A word ($a_0 a_1 a_2 \dots a_{39}$) used as a number in a program is interpreted as $a = -a_0 + a_1 \cdot 2^{-1} + a_2 \cdot 2^{-2} + \dots + a_{39} \cdot 2^{-39}$. The arithmetic operations of GEORGE are designed on the assumption that all machine numbers are of this form. Clearly, a machine number must satisfy the inequality:

$$-1 \leq x \leq 1 - 2^{-39},$$

and it also can be seen that the 0-bit of any number indicates the sign of that number: if $a_0 = 1$, $a < 0$; if $a_0 = 0$, $a \geq 0$. Thus, a_0 is often called the sign bit; e.g., $a = +1/4 = 2^{-2}$ is stored as 0.0100.....0(binary)

$$= 2000.....0(\text{sexadecimal})$$

$$a = -1/4 = -1 + 2^{-1} + 2^{-2} \text{ is stored as}$$

$$1.1100.....0(\text{binary})$$

$$= E000.....0(\text{sexadecimal})$$

(i.e., two's complement representation is used for negative numbers.)

2. Instructions

A GEORGE word acts as an instruction only when it is stored in the Function Register. (It is sent to FR during the fetch cycle of another instruction, or by a Special Order button, which causes a jump to that word.)

An instruction word will be denoted by the binary bits $F_0 F_1 \dots F_{39}$, or by the hexadecimal digits $S_0 S_1 \dots S_9$. When decoded by GEORGE, the instruction will be treated in four parts, as follows:

$F_0 \dots F_{11}$ ($= S_0 S_1 S_2$) forms the B-address, or B portion;

$F_{12} \dots F_{19}$ ($= S_3 S_4$) forms the Order, or O portion;

$F_{20} \dots F_{27}$ ($= S_5 S_6$) forms the Tag, or T portion; and

$F_{28} \dots F_{39}$ ($= S_7 S_8 S_9$) forms the A-address, or A portion.

B, O, T, A portions of an instruction have 12, 8, 8, 12 bits, respectively, and the contents of any one of them will usually be regarded as a non-negative integer. Thus

$$0 \leq B, A < 2^{12}, 0 \leq O, T < 2^8 \quad .$$

B and A usually are the addresses (identification numbers) of memory cells, each address having range 2^{12} , equal to the capacity of the core memory. O, the order, specifies the basic operation of the instruction and normally makes use of the A-address. T, the tag, acts as a helpmate to the order and indicates what uses are to be made of B. Occasionally, the B-portion of a word, if irrelevant in a particular instruction, can be used as a 12-bit register (since it includes the sign-bit) to hold counters, etc., which can be represented as integers $\times 2^{-11}$.

The sections on "The Tag" and "The Order" (see Section II, B and C) describe the available operations and the numerical codes associated with them.

E. Console Operations

The GEORGE Console (see illustration at the end of this section), by means of an array of lights and buttons, provides the operator with a convenient means of observing and controlling the course of the problem being run.

1. Register Display

The contents of the various arithmetic and control registers, counters, and toggles are displayed directly on the console, by an array of neon tubes, one for each bit. (A light on represents a 1; a light off represents a 0.) The lights are grouped in columns of 4, each column representing a sexa-decimal character.

The contents of the following are displayed on the console in this way:

Accumulator	Control Counter
Quotient Register	Jump Register
Memory Register	Dispatch Register
Function Register	Shift Counter

Also displayed are the contents of toggles a_x , q_x , a_{OF} , q_{OF} , and c_{OF} .

2. Stops

Since the appearance of the console relative to programmed stops, i.e., unconditional and breakpoint, is discussed under "Stop Orders," this section will treat only unprogrammed stops, i.e., error manual, and address recognition.

All stops (except emergency and parity error) occur after completion of the fetch cycle following the current instruction (i.e., after the next instruction is brought into MR), even if the condition causing the stop occurs during an earlier cycle. A panel light, labeled "STOP" and a light appropriately labeled to indicate the type of stop, are lit on the console when a stop occurs.

a. Manual Stops

The STOP button on the console, when depressed manually, stops GEORGE at the cycle time noted above.

The EMERGENCY STOP button, when depressed, causes the machine to stop immediately, regardless of what part of the instruction is currently being executed.

While the STOP button is the more commonly used means of manually stopping GEORGE, the EMERGENCY STOP button is especially useful when it is desirable to stop the machine while it is executing a time-consuming order (e.g., punch 4000 words) or an order which cannot be completed (e.g., Read to E_2 , when no E_2 appears on the tape).

b. Error Stops

Two types of error stops can occur during the execution of a GEORGE instruction. In both cases, the console lights labeled STOP and ERROR are lit.

An Overflow Stop occurs when $F_{20} = 0$ in the current instruction, and a preliminary add, Σ , π , or Left Shift (non-end-around) order in that instruction causes the OF toggle of the result or lead register to be set.

The Overflow Stop occurs after the instruction is completed (i.e., after the next instruction has been fetched). In case of an OFS, the result cycle is omitted, except on Jump, α , β , F8, and Record Orders. In this case, the panel light labeled OFS is lit also.

A Parity Error Stop occurs during a Read order when the character just read on the input tape (paper or magnetic) is seen by the reader to have an even number of holes (or magnetized dots).

When a parity error occurs, the reading process is temporarily discontinued, and the machine stops - still in the Continuous mode - with the Error and Load or Copy lights on. At this time, the erroneous character can be corrected and the tape replaced in the reader so that the corrected character is about to be read. Depressing GO will cause the machine to complete the read instruction correctly, then stop, with Single Cycle mode, Error, and Load or Copy lights on. This does not indicate an error. Depressing READY, CONTINUOUS, GO will cause the machine to continue the program. Note: The READY button should not be depressed before the instruction is completed (i.e., after the parity error is discovered) as depressing it may cause the contents of the memory to be altered.

c. Address Recognition Stops

With the use of the Address Recognition Buttons, it is possible to stop the program when a particular memory location is consulted during any memory cycle of the instruction.

The Address Recognition Buttons consist of:

- 1) A set of address selection buttons - i.e., three columns of 4 buttons each, used to represent a three-sigit address. (A button depressed represents a 1 in the corresponding bit of the address; a button not depressed represents a 0.)
- 2) A set of memory-use selection buttons, i.e., six buttons, each corresponding to a particular group of memory cycles.

When an address is set up on the address selection switches, the machine will stop if this address is consulted in any of the memory cycles controlled by the memory-use button which is depressed. (Only one such button can be depressed at one time: however, one of the button permits the selection of all cycles.)

In all cases, the stop will occur after completion of the instruction which uses the specified address, regardless of when during the instruction the address was used.

The buttons are as follows:

Button	Memory Cycles under its control
$\overline{FC}$ (Not Fetch Cycle)	Preliminary, operand, result cycles, and Input or Output orders. (i.e., a stop will occur if the given address is met in any way except as that of an instruction in the control sequence.)
FC (Fetch Cycle)	Fetch cycle only. (i.e., a stop will occur only if the word at the specified address is used as an instruction.)
FR M (From Memory)	Preliminary and operand cycles, and Output orders. (i.e., a stop will occur if a word is called <u>from</u> the specified memory location, to be used in a way other than as an instruction.)
TO M (To Memory)	Result cycle and Input Orders. (i.e., a stop will occur if information is put <u>into</u> the specified memory location.)
ALL	All uses of the memory. (i.e., stop will occur if the given address is consulted in <u>any</u> way.)
NONE	Ignore (i.e., <u>no</u> ADR stop will occur.)

The Address Recognition buttons may be pushed at any time, while the machine is running or stopped, without disturbing the program.

3. READY Button

After any GEORGE stop, the machine must be in a Ready state before going on. Depressing the READY button prepares GEORGE for a new instruction. Also the Ready state is achieved when any one of seven Special Order buttons is depressed. (An appropriately labeled panel light is lit when the machine is in the Ready state.)

4. Operation Modes and GO Button

A program of instructions on GEORGE can be executed in any one of three modes (or a combination of these): Minor Cycle, Single Cycle, and Continuous. The mode is selected by depressing the appropriate panel button, which also turns on a correspondingly labeled panel light.

In all cases, the GO button is used to initiate operation of GEORGE. (It is depressed after the Mode Selection and READY buttons.)

a. Single Cycle

The Single Cycle mode enables the operator to proceed through the code, one instruction at a time. With GEORGE in this mode and in the Ready state, depressing the GO button causes the instruction in MR to be brought into FR, interpreted and executed. After the fetch cycle of this instruction is completed, the machine will stop, with READY on, and the contents of the registers as described under "Stop Orders." As long as GEORGE is on Single Cycle, depressing GO causes one instruction at a time to be executed, as described above, and the machine stays on READY, unless the instruction or Address Recognition brings about a stop.

b. Continuous

When in the Continuous mode of operation, GEORGE will proceed automatically through the instructions of a program until stopped by the code, Address Recognition, or a push of the STOP button. Depressing GO, while in this mode, starts the machine.

c. Minor Cycle

The Minor Cycle operation mode is used when the operator wishes to observe the results of individual memory cycles of an instruction. While GEORGE is in this mode, depressing the GO button causes one memory cycle to take place. The console panel has a column of neons, one

corresponding to each memory cycle; any one of these, if lit, indicates that the corresponding memory cycle has just been completed.

While GEORGE is on Minor Cycle, it is also on either Single Cycle or Continuous, the only difference being that both READY and GO must be pressed to continue after a fetch cycle on Single Cycle. In either case, the READY button should never be depressed after any cycle other than fetch, as depressing it before an instruction is completed will alter the contents of the memory.

Once the Minor Cycle button is pushed, GEORGE will stay on Minor Cycle until the NOT button, alongside the Minor Cycle button, is depressed.

If Minor Cycle mode of operation is used, the register contents, after a given cycle, are as follows:

Let $W_0 = B_0 O_0 T_0 A_0$ = instruction being executed.
 $W_1 = B_1 O_1 T_1 A_1$ = contents of word at B_0 ($C(B_0)$).
 C = address of W_0 ($M(W_0)$).
 $W_2 = B_2 O_2 T_2 A_2$ = next instruction to be executed.
 D = address of W_2 ($M(W_2)$).

Cycle	FR	MR	DR	CC
PC ($F_{26} = 0$)	$B_0 O_0 T_0 A_0 = W_0$	$B_1 O_1 T_1 A_1 = W_1$	B_0	$C + 1$
($F_{26} = 1$)	$B_1 O_0 T_0 A_0$	$B_1 O_1 T_1 A_1 = W_1$	B_0	$C + 1$
OC ($F_{26} = 0$)	$B_0 O_0 T_0 A_0 = W_0$	$C(A_0)$	A_0	$C + 1$
($F_{26} = 1$)	$B_1 O_0 T_0 A_0$	$C(A_0 + B_1)$	$(A_0 + B_1)$	$C + 1$
RC ($F_{26} = 0$)	$B_0 O_0 T_0 A_0 = W_0$	$C(Ac)$ or $C(Qt)$	A_0 or B_0	$C + 1$
($F_{26} = 1$)	$B_1 O_0 T_0 A_0$	$C(Ac)$ or $C(Qt)$	$(A_0 + B_1)$ or B_1	$C + 1$
FC ($F_{26} = 0$)	$B_0 O_0 T_0 A_0 = W_0$	$B_2 O_2 T_2 A_2 = W_2$	D	$C + 1$
($F_{26} = 1$)	$B_1 O_0 T_0 A_0$	$B_2 O_2 T_2 A_2 = W_2$	D	$C + 1$

4. Special Order Buttons

The seven Special Orders provide a convenient means of initiating the execution of a program. When one of these Special Orders is executed by pressing the corresponding button and GO, an appropriate letter is printed by the typewriter, to provide a record of the console operation.

The buttons, their corresponding typewriter symbols, and the equivalent instructions are as follows:

Button	Instruction	Symbol
Type Qt (TYPE Q)	001 C9 01 000	W
Record (REC.)	001 FF 01 000	R
Jump to 001 (JUMP)	001 7D 01 000	T
Load from Wide Tape (LD WIDE)	001 80 01 001	M
Load from Keyboard (LD KB) to E ₂	001 D8 01 001	K
Load from Paper Tape (LD PAPER) Reader 1, to E ₂	001 D0 01 001	P
Jump to B (JUMP B)	FR ^B 7D 01 000	B

The Special Order buttons are operative only when GEORGE is in a Stop state.

Depressing a Special Order button causes the equivalent instruction to be brought into MR and puts the machine in the Ready state. When GO is depressed, the appropriate symbol is typed, the instruction is executed, and the machine proceeds in its selected mode.

Thus, since the instruction is only executed after GO is depressed, if several Special Order buttons are pushed sequentially, only the last one depressed before Go is effective.

5. Miscellaneous

a. Tape Feed Buttons

Pushing the button labeled TAPE FEED caused the punching of a leader of blank tape with only the sprocket holes punched.

Pushing the button labeled MT PARITY CHECK causes the punching of a leader of zeros. After this button is used, the parity bit of the next character punched will always be one, so a parity error may occur in that character.

b. Initial Clear Button

When the INITIAL CLEAR button is pressed, the current operation state is interrupted and GEORGE is reset to its normal starting state, i.e., the machine is returned to fetch cycle, STOP toggles are set to a "no-stop" condition, and Single Cycle mode is on. The various registers and the memory maintain their contents at the point of interruption. The only times the INITIAL CLEAR button should be used are:

1. after the machine has been turned on, to make it ready for the first program to be run;
2. to stop the machine when it cannot complete an instruction..

The normal means of starting GEORGE after an initial clear is with a Special Order.

c. Clock

The console clock is set up for use in timing machine operations. It operates while the machine is on GO and stops when the machine stops. The attached lever can be used to reset the clock to 0. The clock has a range of one hour.

II. GEORGE ORDER STRUCTURE

A. Order List Index

<u>Subject</u>	<u>Page</u>
The Tag	23
Orders	27
Arithmetic Operations	27
$\Sigma(1x)$: +, -, +m, -m.	27
$\pi(2x)$: x	29
Shifting Operations	30
Left Shifts (3x).	32
Right Shifts (4x)	33
Power	33
Combinatorial	34
Store Operations	38
$\alpha(5x)$: Partial Substitution.	38
$\beta(F9)$: Bit-wise Substitution	39
Record (Fx, x = 4, 5, 6, 7, C, D, E, F)	39
F8	40
Set (Fx, x = 0, 1, 2).	41
Wide Magnetic Tape Operations (Ax, Bx)	50
(Not complete)	
Jump Operations (7x)	42
Conditional	42
Sign	42
Overflow.	43
Carry Overflow	43
Unconditional.	44
Input-Output Operations.	44
Input (Dx)	46
6-bit	46
4-bit	47
Output (Cx)	49
6-bit	49
4-bit	50
Stop (0x)	51
Order Decoding Chart	52
Overflow Control	53
Timing	54

B. The Tag (S_5 and S_6 , $F_{20} - F_{27}$)

In general, the meaning of the tag is independent of the operation. The tag may be considered a "help-mate" to the order, controlling the preliminary cycle and extra uses of the memory.

In particular, S_5 ($F_{20} - F_{23}$) selects the initial register conditions for the instruction, and S_6 ($F_{24} - F_{27}$) controls the use of the B address and determines which extra memory cycles will be used in addition to those used by the order.

The following discussion of the tag describes the functions of its bits in most instructions. Exceptions and special uses of the tag with particular orders will be discussed later.

1. Preliminary Clear ($F_{21} - F_{23}$)

The preliminary clear occurs before any of the memory cycles required by the instruction are started. The value of the first sigit of the tag (S_5) indicates which register is to be cleared, which type of clear is to be effected, and the action to be taken if overflow occurs.

Only one of the two registers may be cleared at one time. The Accumulator (Ac) is cleared if $F_{23} = 0$, and the Quotient Register (Qt) is cleared if $F_{23} = 1$. The register not involved in the clear is unaffected at this time. Also, it is the cleared register which participates in the preliminary addition, if any.

F_{21} and F_{22} give the type of clear to be applied to the register, as follows:

F_{21}	F_{22}	Preliminary Clear
0	0	no clear
0	1	clear to zero
1	0	clear to 2^{-11}
1	1	clear to 2^{-39}

Note: a. The "no clear" leaves both registers unaffected, but F_{23} is still significant for preliminary addition.

b. A clear to 2^{-11} places a 1 in the least significant bit of the B portion of the register; clear to 2^{-39} places a 1 in the least significant bit of the A portion of the register. This provides a handy tool for increasing counters and indices, and for modifying the B or A addresses of instructions.

2. Overflow Stop (F_{20})

F_{20} in the tag selects the action to be taken by GEORGE if execution of the instruction causes the overflow toggle to be set. Should overflow occur (see Section II. E, "Overflow Control") as the result of an arithmetic operation in the instruction (i.e., preliminary add, Σ , π , or non-cyclic Shift order), $F_{20} = 0$ stops the machine after completing the instruction. The result cycle (if called for) is omitted if an overflow stop occurs, but the preliminary, operand, and fetch cycles are completed. The OFS console light indicates this stop.

If $F_{20} = 1$, overflow in the execution of the instruction will be ignored. The overflow toggle will still be set by overflow, but the instruction will be executed (all cycles called for completed) as in normal operation.

Thus, if $S_5 = 0, 1, 2, 3, 4, 5, 6, 7$, overflow will cause a stop after completion of the instruction. If $S_5 = 8, 9, A, B, C, D, E, F$, overflow will not cause a stop.

The code for S_5 of the tag is shown below. (Let a = initial contents of Ac, q = initial contents of Qt.)

S_5	After prel. clear	
	Ac	Qt
0,8	a	q
2,A	0	q
4,C	2^{-11}	q
6,E	2^{-39}	q

S_5	After prel. clear	
	Ac	Qt
1,9	a	q
3,B	a	0
5,D	a	2^{-11}
7,F	a	2^{-39}

3. Preliminary Addition (F_{25})

If $F_{25} = 0$, no preliminary addition is performed. $F_{25} = 1$ causes the preliminary cycle to occur, at which time b , the number at the B-address, is added to the contents of the cleared register.

In the preliminary cycle, b is looked up and placed in the memory register (MR). According to F_{23} , the following sum is formed in the adder:

F_{23}	Preliminary Sum
1	$Qt + b = Qt + MR$
0	$Ac + b = Ac + MR$

The result of this addition is put back into the cleared register, with two exceptions. These exceptions occur where the operation code calls for a multiplication or division ($S_3 = 2$). The section on π orders will explain these exceptions in detail.

4. Automatic Indexing (F_{26})

A second use of the preliminary cycle is in automatic indexing of the A-address of the instruction. This process is commonly called B-boxing.

B-boxing allows modification of an A-address during the execution of an instruction without altering the address in the memory. Only the index used for modification needs to be changed in the memory for each step of a loop. Since the left twelve bits (B-portion) of any memory position may be used to hold an index, no special B-boxes or index registers appear in the equipment.

If $F_{26} = 1$, B-boxing is called for, and the process is as follows: The information at the B-address is looked up and stored in the memory register (MR) during the preliminary cycle. After the preliminary cycle, the left 12 bits of MR (i.e., $MR^B = \text{index}$) are sent to FR^B , replacing the previous contents of FR^B . The new contents of FR^B are added to FRA in a special adder and this sum is sent to the dispatch register. It is this address in the dispatch register which is the effective A-address for the operand, result, and/or fetch cycles of the instruction being executed. It should also be noted that the effective B-address for the result and/or fetch cycles is not the same as the B-address used in the preliminary cycle, since B-boxing causes the original FR^B to be replaced by the index (MR^B).

No B-boxing takes place when $F_{26} = 0$.

5. Result to B (F_{24})

The F_{24} bit in the tag makes it possible to write the contents of one of the arithmetic registers into the memory without using a separate instruction for this purpose.

Normally, $F_{24} = 1$ causes the contents of the register containing the result of the operation performed to be written into the memory at the B-address of the instruction. F_{19} in the order indicates where the result is to be found, i.e., $F_{19} = 0$, result is in Ac, $F_{19} = 1$, result is in Qt.

Exceptions to this rule occur in the α orders ($S_3 = 5$) and the β and Set orders ($S_3 = F$) and will be discussed with those orders in a later section.

No result to B is available with the Stop, Copy, Load, and Magnetic Tape orders ($S_3 = 0, C, D, A$ and B , respectively). In these instances F_{24} should be set to 0 in case this bit takes on a new meaning at a later date.

6. Jump to B (F_{27})

$F_{27} = 1$ causes a jump to the instruction at the B-address after completion of the order, i.e., the instruction at the B-address is fetched.

If $F_{27} = 0$, no jump to B occurs, and the instruction whose address is in the control counter (i.e., the next instruction in the memory) is fetched, unless the order is a Jump order (see Jumps, $S_3 = 7$).

	F_{20}	F_{21}	F_{22}	F_{23}
1 0	Don't stop on OF Stop on OF	See "Clears"		Cl. Qt, Qt + MR Cl. Ac, Ac, + MR
	F_{24}	F_{25}	F_{26}	F_{27}
1 0	Result to B Not	$b + \begin{Bmatrix} Ac \\ Qt \end{Bmatrix} \rightarrow \begin{Bmatrix} Ac \\ Qt \end{Bmatrix}$ Not	$A_{eff} = A + c(B)^B$ $A_{eff} = A$	Jump to B Not

C. Orders (S_3 and S_4 , $F_{12} - F_{19}$)

The S_3 and S_4 sigits of the instruction word are the code for the order to be executed. S_3 selects the kind of operation (e.g., 1 = addition, 7 = jump, etc.). The bits $F_{16}-F_{19}$, making up S_4 of the order, select the particular species (e.g., use of the absolute value of the argument).

In the tag, if S_5 is even, Ac is involved; if S_5 is odd, Qt. S_4 in the order has a similar interpretation with regard to the register which is to hold the result of the order portion of the instruction.

The discussion of the particular GEORGE operation codes (orders) will include references to the tag only when exceptions to the normal functioning of the tag occur.

Whenever the A address is referred to, A-effective address is meant [i.e., A-eff. = $A(F_{26} = 0)$; A-eff. = A-indexed ($F_{26} = 1$)].

1. Σ and π orders ($S_3 = 1, 2$)

The basic arithmetic operations are performed normally with the Σ orders for addition and subtraction, and with π orders for multiplication and division.

Since execution of these orders occurs during the operand cycle, the A (or A-effective) address is consulted.

a. Result to A (F_{16})

F_{16} functions in the same manner for $S_3 = 1$ or 2. $F_{16} = 1$ causes the contents of the register containing the operation result to be stored in the memory at the A address. F_{19} in both cases determines which register contains the result. ($F_{19} = 1$, result in Qt; $F_{19} = 0$, result in Ac.) Should $F_{16} = F_{24} = 1$ (calling for a result to A and B) F_{24} takes precedence, and the result will be stored at B, since only one result cycle can occur per instruction. The functions of F_{17} and F_{18} in Σ and π orders are explained under the respective order discussions.

b. Σ Orders ($S_3 = 1$)

Σ orders include all additive type of operations, i.e., $+ a$, $- a$, $+ |a|$, $- |a|$. In Σ orders, as in preliminary additions, either register may be used as one of the summands. But in Σ orders, either register may also be selected as the recipient for the sum. This selection is independent of the selection by the tag of the summand register.

It must be noted that all additions are performed in the adder, not in Ac or Qt. For each addition, two adder input summands produce one output sum. The contents of MR (= the contents of B in P+, the contents of A in Σ) is always one of these summands. The other is the contents of either Ac or Qt, depending on F_{23} . ($F_{23} = 1$, Qt = adder input; $F_{23} = 0$, Ac = adder input.)

In Σ orders, F_{19} determines where the result of the addition is to go. $F_{19} = 1$ indicates that the sum is sent to Qt by the adder; $F_{19} = 0$ indicates that the sum is sent to Ac. The result of a preliminary addition in a Σ order is returned to the register designated by F_{23} (i.e., the register cleared).

The following chart shows the results of the various combinations of F_{19} and F_{23} , when $S_3 = 1$.

Let: a = contents of A p = contents of Ac after the clear
 b = contents of B q = contents of Qt after the clear

F_{19}	F_{23}	Reg. Cleared	Preliminary Cycle (if any)				Operand Cycle			
			Summand Register	Sum sent to -	After p +		Summand Register	Sum sent to -	After Σ	
					Ac =	Qt =			Ac =	Qt =
0	0	Ac	Ac	Ac	p + b	q	Ac	Ac	p + b + a	q
0	1	Qt	Qt	Qt	p	q + b	Qt	Ac	q + b + a	q + b
1	0	Ac	Ac	Ac	p + b	q	Ac	Qt	p + b	p + b + a
1	1	Qt	Qt	Qt	p	q + b	Qt	Qt	p	q + b + a

F_{17} F_{18} in Σ orders: F_{18} selects addition or subtraction. $F_{18} = 0$ selects addition; $F_{18} = 1$ selects subtraction. In subtraction a is replaced by -a (= two's complement of a) as adder input, and normal addition is performed.

F_{17} determines whether a itself or the magnitude of a (absolute value) will be used in the addition. $F_{17} = 0$ selects a itself; $F_{17} = 1$ selects the magnitude of a. When $F_{17} = 1$, |a| instead of a is used as adder input.

Thus, the code for S_4 , when $S_3 = 1$ is as follows:

	F_{16}	F_{17}	F_{18}	F_{19}
1	Result $\rightarrow$ A	a	-	$\Sigma \rightarrow$ Qt
0	not	a	+	$\Sigma \rightarrow$ Ac

c. π Orders ($S_3 = 2$)

The codes for multiplication and division make up the π orders. The actual mechanics of each operation are explained in the Appendix. For purposes of the description here, it will suffice to state that the product of two numbers is obtained by a series of additions and right shifts, the quotient by a series of subtractions and left shifts.

The symmetric use of Ac and Qt, possible in Σ orders, cannot be maintained here; e.g., in multiplication the multiplier must be in Qt.

Full and Half Precision (F_{17}): $F_{17} = 1$ selects a half precision operation. Here only the first twenty bits of the multiplier or the divisor are used, and only the first half-register of the result (product or quotient) is relevant. In calculations where six place accuracy is sufficient, half-precision can be used to increase the speed of the program. If $F_{17} = 0$, full register numbers are used as operands, producing a full register result.

Round-off (F_{18}): $F_{18} = 1$ selects a round-off operation. In either multiply or divide, the result is rounded in the 39th place for full-precision, or in the 19th place for half precision. The method of rounding, for the two operations, will be discussed below.

$F_{18} = 0$ selects a no-round-off operation.

F_{19} selects multiplication or division.

Multiplication ($F_{19} = 0$): In full precision multiplication, the number, m, in MR (= contents of the word at the A-eff. address) is multiplied by the contents of Qt, q, and to the resulting 79-digit product is added $2^{-39}p$ (p = contents of Ac at the start of multiplication). The result is left in ($a_0 a_1 a_2 \dots a_{39} q_0 q_1 \dots q_{38}$). q_{39} holds the sign-bit of the multiplier, originally in q_0 . ($a_0 a_1 \dots$ refer to the bits of Ac; $q_0 q_1 \dots$ refer to the bits of Qt.) Thus, after multiplication is completed, Ac Qt holds $mq + 2^{-39}p + 2^{-79}q_0$.

Half-precision multiplication works the same way, except that only half as many steps are performed, so that only the first 20 bits of the product (= $a_0 a_1 \dots a_{19}$) are relevant.

If round-off multiplication is selected, Ac is initially cleared to 2^{-1} . This causes rounding to the nearest unit in the 39th place (full precision) or the 19th place (half-precision) in Ac.

Division ($F_{19} = 1$): In all Divide orders, the dividend is assumed to be the 79-place number in Ac Qt (= $a_0 a_1 a_2 \dots a_{39} q_0 \dots q_{38}$). The divisor is m, the number is in MR (= the content of the word at the A-eff. address). The quotient is left in Qt.

After the completion of a full-precision division order, Qt holds the 40-bit result, and Ac holds the true remainder.

A half-precision division yields a 20-bit quotient. These 20 bits are held in $q_0 q_1 \dots q_{19}$. The rest of Qt, and Ac are irrelevant.

Round-off division forces the last bit of the quotient to be a "1." Thus, $q_{39} = 1$ after a full-precision r.o. divide; $q_{19} = 1$ after a half-precision r.o. divide.

When $S_3 = 2$, F_{19} selects multiplication or division, and it should be noted that it also indicates where the result of the operation is to be found for result cycle purposes ($F_{19} = 1$, division, result in Qt; $F_{19} = 0$, multiplication, result in Ac) in keeping with the normal function of F_{19} .

$S_3 = 2$, π Orders

	F_{16}	F_{17}	F_{18}	F_{19}
1	Result $\rightarrow$ A	Half-Prec.	Round-off	$\div$
0	Not	Full-Prec.	Not	x

Preliminary Addition with π Orders: An exception in the preliminary cycle occurs when the operation code calls for multiplication or division ($S_3 = 2$).

Normally, the result of the preliminary addition is sent to the register cleared. However, in multiplication, the preliminary sum, which will be the multiplier, is sent to Qt. In division, the sum, which will be the dividend, is sent to Ac. Thus, F_{23} determines which register is to be cleared and will participate in the preliminary addition, but when $S_3 = 2$, F_{19} determines where the preliminary sum will go.

Preliminary Addition when $S_3 = 2$

F_{19}	F_{23}	Preliminary Addition
1 ($= \div$)	1	Qt + b $\rightarrow$ Ac
	0	Ac + b $\rightarrow$ Ac
0 ($= x$)	1	Qt + b $\rightarrow$ Qt
	0	Ac + b $\rightarrow$ Qt

2. Shift Orders ($S_3 = 3, 4$)

Shift instructions are used to move the contents of one or both arithmetic registers to the left or to the right of the original position. The number of places moved (= amount of shift) is given by the A (or A-effective),

address of the instruction. The maximum amount of shift possible is 127_{10} ($=7F_{16}$). If the shift address is greater than $07F$, only 127 shifts will be performed.

A Shift instruction may be used to obtain a logical rearrangement of the register bits (as in moving the contents of the A portion of a register to the B portion), or to multiply the contents of the register by an integral power of two. The former is a combinatorial shift, the latter a power shift. Particular features of each type of shift will be discussed later.

a. All Shifts

Single or Double Register (F_{17}): $F_{17} = 1$ selects a two-register shift. Ac and Qt are shifted together as one double length (80-bit) register. The register holding the sign and next 39 bits of the number being shifted is called the lead register; the register holding the last 40 bits of the number is called the trailing register. The 0-bit of the trailing register functions as a 40-bit following the 39-bit of the lead register, not as a sign bit. If the tag calls for a result cycle after a two-register shift, the contents of the lead register are stored.

An overflow condition ($a_x \neq a_0$ or $q_x \neq q_0$) occurring at any point during the shift will cause a_{OF} or q_{OF} , respectively, to be set, whether the overflowed register is leading or trailing; but only overflow in the lead register can cause an OF stop.

$F_{17} = 0$ selects a one-register shift. Either Ac or Qt is shifted; the other register remains unchanged. The shifted register is stored in case a result cycle is called for in the tag of the instruction.

In the remainder of the discussion on shifts, remarks regarding the "register" will be applicable to single and double-length registers alike, unless otherwise specified.

Lead Register (F_{19}): If $F_{19} = 1$, Qt alone will be shifted if $F_{17} = 0$; Qt will lead and (Qt Ac) will be shifted if $F_{17} = 1$.

If $F_{19} = 0$, Ac alone will be shifted if $F_{17} = 0$; Ac will lead and (Ac Qt) will be shifted if $F_{17} = 1$.

$$(Qt \ Ac = q_x \ q_0 \ q_1 \ \dots \ q_{39} \ a_0 \ a_1 \ a_2 \ \dots \ a_{39};$$

$$Ac \ Qt = a_x \ a_0 \ a_1 \ \dots \ a_{39} \ q_0 \ q_1 \ q_2 \ \dots \ q_{39}).$$

F_{16} and F_{18} take on different meanings for left and right shifts and will be discussed in detail under those shifts.

b. Left Shifts ($S_3 = 3$)

In all left shifts, the bits of the register being shifted are moved simultaneously to the left A places. The 0-bit always feeds into the x-bit of the register.

Overflow is set but the shift is completed if, at any point during the left shift, the 0 and x-bits of either register disagree.

On all two-register left shifts, the 0-bit of the trailing register feeds both into the x-bit of its register and into the 39-bit of the lead register.

End-around vs. Power Shift (F_{18}): $F_{18} = 0$ selects a power shift, i.e., multiplication by a positive power of two. Here, 0's are fed into the vacated position of the right end of the register, and the bits shifted past the x-bit of the lead register are lost. An OF Stop will occur if $F_{20} = 0$ and overflow is set in the lead register.

$F_{18} = 1$ selects an end-around shift. Here all the bits of the register are preserved, since the 0-bit feeds into the 39-bit of the same register (1-register shift) or of the other register (2-register shift) during the shifting.

Since this is a combinatorial shift, overflow is ignored, regardless of the value of F_{20} , i.e., the OF toggle may be set by the shift, but no stop will occur.

Overflow Recognition Shift (F_{16}): $F_{16} = 1$ selects an overflow recognition shift. This is a normalizing left shift and is used chiefly to shift off insignificant zeros from the left of the register while keeping track of the number of places shifted.

The overflow recognition shift operates in the following manner.

The ones' complement of the effective A address is sent to the shift counter. Then the register is shifted left and one added to the SC for each shift until either:

- 1) an overflow is about to occur in the lead register
($a_0 \neq a_1, q_0 \neq q_1$), or
- 2) the Shift Counter becomes all ones (i.e., 7F).

One is then added to the shift counter, and the order is completed. Thus A -eff represents the maximum shift allowed. After the order is completed, if the register was shifted p places, the shift counter (SC) will hold $(p - A)$.

If A-effective is greater than or equal to 080, 0 goes to the shift counter as if the address were 07F, and thus a shift up to 127 can occur. After the shift of p places, the shift counter will hold $(p + 1) =$ the amount of shift plus 1.

Examples:

Before Shift		After Shift		
Ac	Inst.	Ac	SC	p =
2^{-30}	000 3800 000	2^{-30}	<u>00</u>	0
2^{-30}	000 3800 00A	2^{-20}	<u>00</u>	10 = <u>A</u>
2^{-30}	000 3800 020	2^{-1}	<u>7D</u>	29 = <u>1D</u>
2^{-30}	000 3800 100	2^{-1}	<u>1E</u>	29 = <u>1D</u>

The Record order, which will be discussed later, can be used to write the contents of the shift counter into the memory after an overflow recognition shift.

$F_{16} = 0$ selects a non-overflow recognition shift. Here, the register is always shifted A places to the left.

$S_3 = 3$ (Left Shift)

	F_{16}	F_{17}	F_{18}	F_{19}
1	OFR	2 reg.	End Around	Qt
0	Not	1 reg.	Not	Ac

c. Right Shifts ($S_3 = 4$)

In all right shifts, the bits of the register being shifted are moved simultaneously to the right A places.

On all two-register right shifts, the 39-bit of the lead register always feeds into the x and 0-bits of the trailing register.

1). Right Power Shift ($F_{16} = 0$)

When $S_3 = 4$, $F_{16} = 0$ selects a right power shift, i.e., the register will be multiplied by a negative integral power of two. The x-bit (sign bit) is preserved and propagates through the left part of the register as its contents are moved to the right, thus multiplying the contents of the register by 2^{-1} for each shift. The bits feeding out of the right end of the register are lost.

No end-around shift is available here.

Round-off (F_{18}): $F_{18} = 1$ selects a round-off shift, which forces a one into the least significant bit of the register after the shift. No round-off is performed, even if requested, on a shift where $A = 000$, since in this case no truncation error can possibly arise.

$F_{18} = 0$ selects a non-round-off shift.

2) Right Combinatorial Shift ($F_{16} = 1$)

When $S_3 = 4$, the setting $F_{16} = 1$ selects a combinatorial right shift. Since this shift is used to rearrange the contents of the register, the sign bit will not propagate, and end-around shifting is available.

End-Around Shift ($F_{18} = 1$): $F_{18} = 1$ selects an end-around shift. All the bits of both registers are preserved.

On a one-register shift, the 39-bit of the register feeds into both the x-bit and the 0-bit of the register. The 0-38 bits are shifted right as in all other right shifts.

On a two-register shift, the 39-bit of the trailing register feeds into both the x and 0-bits of the lead register; the 39-bit of the lead register feeds into the x and 0-bit of the trailing register. The 0-38 bits of both registers are shifted right as in all other right shifts.

Non-End-Around Shift ($F_{18} = 0$): When $F_{16} = 1$, the value $F_{18} = 0$ selects a non-end-around combinatorial right shift.

Here, the x-bit of the register is cleared to zero initially, and then shifted successively into the 0-bit of the register every time the contents of the register are moved to the right, thus replacing the vacated positions on the left with zeros. The bits feeding out of the right end of the register are lost.

Overflow can be set on a right combinatorial shift of 000; no OF stop will occur.

$S_3 = 4$ (Right Shift)

Combinatorial Shift

	F_{16}	F_{17}	F_{18}	F_{19}
1	1	2 reg.	End-around	Qt
0		1 reg.	not	Ac

Power Shift

	F_{16}	F_{17}	F_{18}	F_{19}
1	0	2 reg.	Round-off	Qt
0		1 reg.	not	Ac

LEFT SHIFTS: $S_3 = 3$

RIGHT SHIFTS (POWER): $S_3 = 4$

 S_4

Ac

Qt

0

1

2

3

4

5

6

7

RIGHT SHIFTS (COMBINATORIAL): $S_3 = 4$

 S_4

Ac

Qt

Pcl to 0

8

Pcl to 0

9

A

B

Pcl to 0

C

Pcl to 0

D

E

F

3. Store Orders

a. The α Order ($S_3 = 5$)

The α order is used to store Ac or Qt or parts thereof in the memory.

The tag determines which register is to be stored and whether the contents of this register are to be stored at the A or B address.

If $F_{23} = 0$, Ac will be stored; if $F_{23} = 1$, Qt will be stored (i.e., the register cleared is stored).

If $F_{24} = 0$, the register is stored at the A address. If $F_{24} = 1$, the register is stored at the B address, and the A address of the instruction is irrelevant.

The α order treats the register in question as four blocks of bits, as follows:

B portion = $a_0 a_1 \dots a_{11}$ (or $q_0 q_1 \dots q_{11}$) = 12 bits

O portion = $a_{12} a_{13} \dots a_{19}$ (or $q_{12} q_{13} \dots q_{19}$) = 8 bits

T portion = $a_{20} a_{21} \dots a_{27}$ (or $q_{20} q_{21} \dots q_{27}$) = 8 bits

A portion = $a_{28} a_{29} \dots a_{39}$ (or $q_{28} q_{29} \dots q_{39}$) = 12 bits

The bits F_{16} , F_{17} , F_{18} , F_{19} of S_4 control, respectively, the storage of the B, O, T, A portions of the register. If a bit of S_4 has the value "1," the corresponding portion of the register is written into the same portion of the memory word at A (or B). If a bit of S_4 has the value "0," writing does not occur, and the original contents of A (or B) in that portion stay intact.

Thus, a 51 order stores only the A portion of the register, a 56 stores the O and T portions, a 5F stores the entire register, and a 50 order is effectively a stall.

In any case, the contents of Ac, Qt, a_{OF} and q_{OF} are the same as those after the preliminary cycle.

α Order ($S_3 = 5$)

	F_{16}	F_{17}	F_{18}	F_{19}	F_{23}	F_{24}
1	Store B-Port.	Store O-Port.	Store T-Port.	Store A-Port	Qt Stored	Store at B
0	Not	Not	Not	Not	Ac Stored	Store at A

b. The β Order ($S_3 S_4 = F9$)

The β order is used to do a bit-wise substitution of the contents of Ac into the memory at the A or B address. Qt contains a stencil-type keyword, having ones in the positions corresponding to the positions of Ac to be stored and zeros elsewhere, and the order works as follows:

After the preliminary cycle (if any) is completed, the contents of the word at the A address are sent to MR.

Qt is examined. For every bit = 1 in Qt, the bit in the corresponding location in Ac replaces that bit in MR. For every bit = 0 in Qt, the corresponding bit in MR stays intact. The new contents of MR are then stored in the memory at A if $F_{24} = 0$, or at B if $F_{24} = 1$, and the order is complete.

The logical representation of the β order is $[a \cap \bar{Q}t \cup Ac \cap Qt \rightarrow A \text{ (or B)}]$ (a = contents of A). By appropriate choices of the quantities involved, we can perform the operations of Boolean algebra (see Appendix A). A few examples follow.

Initial Contents of			Quant. stored at A or B	$a \cap \bar{Q}t \cup Ac \cap Qt$
Ac	Qt	A		
0	q	1	$\bar{q}$	$1 \cap \bar{q} \cup 0 \cap q$
1	q	x	$x \cup q$	$x \cap \bar{q} \cup 1 \cap q$
Ac	q	0	$Ac \cap q$	$0 \cap \bar{q} \cup Ac \cap q$
0	q	x	$x \cap \bar{q}$	$x \cap \bar{q} \cup 0 \cap q$
Ac	q	1	$\bar{q} \cup Ac$	$1 \cap \bar{q} \cup Ac \cap q$
* 0	0	x	x	$x \cap 1 \cup 0 \cap 0$

*This possibility represents another way of transferring A to B.

c. The Record Order ($S_3 = F, F_{17} = 1$)

When $S_3 = F$, the situation $F_{17} = 1$ selects a record order, a special purpose order which can write the following information into the memory at A or B:

1. Contents of the control counter = CC
2. Contents of the shift counter = SC
3. A set of bits, t, representing the state of the toggles
 $a_{OF}, q_{OF}, c_{OF}, a_x, q_x$.
 (If a toggle is set, its corresponding bit = 1).

If $F_{24} = 0$, the information is stored at the A address. If $F_{24} = 1$, the information is stored at the B address and the A address of the instruction is irrelevant.

The order operates as follows:

After the preliminary cycle (if any) is completed, the Memory Register (MR) is cleared to zero. Then CC (12 bits) is sent to $m_0 - m_{11}$ of MR, SC (7 bits) to $m_{33} - m_{39}$, and t (5 bits) to $m_{12} - m_{16}$.

The Record order then treats MR as a register separable into three blocks of bits, as follows:

B portion, of 12 bits = $m_0 m_1 \dots m_{11} = CC$

OT portion, of 16 bits = $m_{12} m_{13} \dots m_{27} = aOF \ qOF \ cOF \ a_x \ q_x$
 $0 \ 0 \dots 0$

A portion, of 12 bits = $m_{28} m_{29} \dots m_{39} = 0 \ 0 \ 0 \ 0 \ 0 \ SC$

The bits F_{16} , F_{18} , and F_{19} of S_4 in the instruction control, respectively, the storage of the B, OT, and A portions of MR. If any of these bits has the value "1," the corresponding portions of MR are written into those portions of the memory word at A (or B). Otherwise, the original contents of that portion of A (or B) stay intact.

Thus, an FC order records the control counter, a convenient order if the current sequence of orders is to be temporarily interrupted and it is desired to regain it again. An F5 order can be used following an overflow recognition shift to record the amount of shift performed.

It should be noted that a preliminary addition increases SC by one on any order at preliminary cycle time which, in particular, precedes result cycle time of Record orders. Also, SC is only a 7-bit register, and zeros, rather than sign bits, are written in the first five bits of the A-address if SC is recorded.

d. The F8 order ($S_3 \ S_4 = F8$)

The F8 order is a special partial substitution order which allows changing of the first 12 bits of a word in the memory, without use of Ac or Qt.

The F8 order causes the B-portion of the word in the memory register (MR) at result cycle time to be sent to the B-portion of the word at either the A or B address of the instruction.

If no preliminary cycle (P + or B-box) is called for in the instruction, MR contains the instruction itself at result cycle time, and the B-portion of the instruction is stored as described. If a preliminary addition of B-boxing is called for, MR contains the contents of B at result cycle time, and the B-portion of the word at B is stored.

If $F_{24} = 0$, the information is written into the word at the A address; if $F_{24} = 1$, the information is written into the word at the B-address.

The following examples show how the F8 order works:

Let: 300 40 50 600 = contents of 100
 700 80 90 A00 = contents of 200
 B00 C0 D0 E00 = contents of AC.

Instruction	Cycle	Contents of (after given cycle)				
		MR	FR	100	200	AC
100 F8 00 200	FC	100 F8 00 200	100 F8 00 200	300 40 50 600	700 80 90 A00	B00 C0 D0 E00
	RC	"	"	"	100 80 90 A00	"
100 F8 04 200	FC	100 F8 04 200	100 F8 04 200	300 40 50 600	700 80 90 A00	B00 C0 D0 E00
	PC	300 40 50 600	"	"	"	E01 01 21 400
	RC	"	"	"	300 80 90 A00	"
100 F8 02 200	FC	100 F8 02 200	100 F8 02 200	300 40 50 600	700 80 90 A00	B00 C0 D0 E00
	PC	300 40 50 600	300 F8 02 200	"	"	"
	RC	"	"	"	"	"
		(300 = MR^B goes to $(500)^B$ ($500 = 300 + 200 = MR^B + FR^A$))				
100 F8 08 200	FC	100 F8 08 200	100 F8 08 200	300 40 50 600	700 80 90 A00	B00 C0 D0 E00
	RC	"	"	100 40 50 600	"	"

e. The Set order ($S_3 = F$; $S_4 = 0, 1, 2$)

The Set order is used to set artificially the overflow or carry toggles, whether or not they are already set.

F_{16} and F_{17} must equal 0 in a Set order.

If $F_{18} = 0$, the condition $F_{19} = 1$ will cause q_{OF} to be set and $F_{19} = 0$ will cause a_{OF} to be set.

If $F_{18} = 1$, then $F_{19} = 0$ will cause the carry overflow toggle (c_{OF}) to be set; $F_{19} = 1$ will cause a stall.

Thus:

F0 sets a_{OF}

F1 sets q_{OF}

F2 sets c_{OF}

F3 is a stall order.

No result to B is available in the Set Order; i.e., F_{24} is irrelevant.

The Set Order

	F_{16}	F_{17}	F_{18}	F_{19}
$S_3 = F, S_4 =$	1	0	C. OF	Qt
	0	0	OF	Ac

4. Jump Orders ($S_3 = 7$)

Normally, after GEORGE executes the instruction at location (n) in the memory, it fetches the word at location (n + 1) as the next instruction to be performed. By means of a Jump order, this process may be interrupted and the next instruction fetched from a location other than (n + 1).

Jump orders may be used to transfer control unconditionally, or upon the presence of one of a pair of mutually exclusive conditions. In all cases, the A-address gives the address of the instruction to be fetched if the jump occurs.

a. Conditional Jumps

In all Conditional Jump instructions, the instruction at the A address is fetched only if the specified condition is met; otherwise, the next instruction in the memory is fetched. If a jump to B is requested in the tag of a conditional jump instruction, the jump to B will occur only if no jump to A occurs.

In Conditional Jumps where the condition tested is the state of one of the registers (sign, OF), F_{19} selects the register to be examined. If $F_{19} = 1$, Qt is examined; if $F_{19} = 0$, Ac is examined. F_{19} serves also to indicate where the result is to be found, in case of a result to B in the tag.

Jump on Sign ($F_{16} = F_{17} = 0$): If $F_{16} = F_{17} = 0$, the x-bit (= sign bit) of the selected register is examined, and a jump to the instruction at the A address is executed if the x-bit has the tested value.

If $F_{18} = 0$, a jump to A occurs if the register is positive or zero; that is, the next instruction is fetched from A if the x-bit = 0. If $F_{18} = 1$, a jump to A occurs if the register is negative; that is, the next instruction is fetched from A if the x-bit = 1.

Jump on Sign

		F_{16}	F_{17}	F_{18}	F_{19}
$S_3 = 7, S_4 =$	1	0	0	-	Qt
	0			+	Ac

Jump on Overflow ($F_{16}, F_{17} = (0, 1)$): If $(F_{16}, F_{17}) = (0, 1)$, the overflow toggle of Ac ($F_{19} = 0$) or Qt ($F_{19} = 1$) is examined, and a jump to the instruction at the A-address is executed if a_{OF} (or q_{OF}) is in the tested state.

If $F_{18} = 1$, a jump to A will occur when a_{OF} (or q_{OF}) is set (= 1). If $F_{18} = 0$, a jump to A will occur when a_{OF} (or q_{OF}) is not set (= 0).

Jump on Overflow

		F_{16}	F_{17}	F_{18}	F_{19}
$S_3 = 7, S_4 =$	1	0	1	OF Set	Qt
	0			OF Not Set	Ac

Jump on Carry Overflow ($(F_{16}, F_{17}) = (1, 0)$): If $(F_{16}, F_{17}) = (1, 0)$, the carry overflow toggle (c_{OF}) of the adder is examined, and a jump to the instruction at the A address is executed if c_{OF} is in the tested state. The adder's c_{OF} is set when the addition being performed in the adder produces a carry out of the 0-bit of the adder into the x-bit.

If $F_{18} = 1$, a jump to A will occur when c_{OF} is set. If $F_{18} = 0$, a jump to A will occur when c_{OF} is not set.

Since the setting of c_{OF} depends on the last addition performed in the adder, not on the state of either Ac or Qt, F_{19} serves only to determine which register is to be stored in case a result to B is selected in the tag.

Jump on Carry

		F_{16}	F_{17}	F_{18}	F_{19}
$S_3 = 7, S_4 =$	1	1	0	c_{OF} Set	Qt
	0			c_{OF} Not Set	Ac

b. Unconditional Jump $((F_{16}, F_{17}) = (1, 1))$

If $(F_{16}, F_{17}) = (1, 1)$ in a Jump instruction, the state of the toggles or registers is irrelevant. F_{19} serves only to indicate the register to be stored if a result to B is called for in the tag.

$F_{18} = 1$ causes a jump to the instruction at the A address. No jump to B is possible with such orders (7E, 7F), since only one fetch cycle can occur per instruction.

If $F_{18} = 0$, no jump to A is executed. Thus the 7C and 7D orders work as fast stall orders. All the bits of the tag are relevant.

Unconditional Jump

	F_{16}	F_{17}	F_{18}	F_{19}
$S_3 = 7, S_4 =$	1	1	Jump to A	Qt
	0		Not	Ac

5. Input and Output Orders ($S_3 = D, C$)

Input and output orders are used to supply information to the machine and obtain information from the machine.

The following discussion explains the functions and particular features of the Input and Output orders. A more detailed description of the equipment appears later in this manual. Thus, facts regarding readers, tapes, etc., will be noted here only when they relate directly to the execution of a Cx or Dx order.

a. General Information

Punched Tapes: Input and output information on paper or narrow magnetic tapes is represented by holes or magnetized spots, respectively. For simplicity, only paper tapes will be described here, but the remarks apply also to magnetic tapes - with magnetized spots in place of punched holes. (At this writing, narrow magnetic tape input-output is not yet available.)

There are seven positions across the tape in which holes may be punched. Six of these holes are used to represent a character of information, the seventh is for parity and is punched or not to make the total number of holes odd. This feature is used by input and output equipment as a convenient check for tape errors. A hole in a given position represents a 1; no hole in that position represents a 0. The hole positions of a

character across a tape are identified as follows (an x represents the hole position):

x	hole 7	- parity
x	hole 6	- zone positions
x	hole 5	
x	hole 4	
.	Sprocket hole	sexadecimal character
x	hole 3	
x	hole 2	
x	hole 1	

Hole positions 1 to 4 represent a sexadecimal character, i.e., hole 1 represents 2^0 , hole 2 represents 2^1 , etc. For example, a C would be represented by holes punched in the 3, 4, and 7 positions ($2^2 + 2^3 = 12 = \underline{C}$). The hole 7 causes the number of holes to be odd (3). Hole positions 5 and 6, known as the zone positions, are used together with the sexadecimal character in holes 1 through 4 to represent non-numeric information. Zeros in positions 5 and 6 give a zone 0; a 1 in position 5 and a 0 in position 6 give a zone 1; a 0 in position 5 and a 1 in position 6 give a zone 2; 1's in both positions 5 and 6 give a zone 3.

End Symbols: Three end symbols are used to signal the input or output equipment that a word or message has been completed. The symbols and their functions are as follows:

1. End 1 (E_1) is represented on tape by a sexadecimal character 0 with a zone 3, and normally denotes the end of a word.
2. End 2 (E_2) is represented on tape by a sexadecimal character 3, with a zone 3, and denotes the end of a block of words or the end of an alphabetic message being read into the machine.
3. End 3 (E_3) is represented on tape by a sexadecimal character 6 with a zone 3. E_3 is used to indicate to the output equipment the end of the alphabetic message being punched.

Note that E_1 , E_2 , and E_3 , when used as signals, are not treated as part of the numeric or alphabetic information they accompany. However, E_3 , an output signal, can be a character of input information, and E_1 and E_2 , input signals, can be punched as output information.

Input and Output orders make use of Qt during their execution as a buffer between the memory and the input or output device. Ac is not affected by these orders.

No result to B is available on Input and Output orders, so F_{24} is irrelevant. The meanings of the order bits of the tag stay the same.

4-bit and 6-bit Information: Input and output information may be treated as either 4-bit or 6-bit information, depending on the use intended for the data.

4-bit information can represent only sexadecimal information (which includes decimal information). On tapes, only hole positions 1 through 4 (plus the hole 7 for parity) are relevant. The zone positions are ignored on input, and a zone 0 is punched on output. Thus, ten 4-bit characters constitute one 40-bit word of input or output.

6-bit information (sometimes called 8-bit information) uses hole positions 1 through 6, plus the parity. By appropriate combinations of sexadecimal characters and zone pulses, any letter of the alphabet plus several mathematical and control symbols can be represented. Each 6-bit character is represented by a block of 8 binary bits, the first two of which are irrelevant. Thus, five 6-bit characters can be held in one 40-bit word in the memory or Qt. Note that sexadecimal, as well as alphabetic, information can be represented as 6-bit characters (for 1-9, A-F, zone = 0), although this type of input and output is used most commonly for alphabetic messages.

b. Input Orders ($S_3 = D$)

Data are read into the machine through one of four input devices. Punched paper tape is read through either of two paper tape readers, narrow magnetic tape is read through a magnetic tape reader, and information may be typed directly into the machine from a console keyboard. The values assigned to F_{16} and F_{17} select the mode of input. The code for F_{16} and F_{17} when $S_3 = D$ is as follows:

Input Modes

Load from	F_{16}	F_{17}
Paper Tape, reader 1	0	0
Paper Tape, reader 2	0	1
Keyboard	1	0
Narrow Mag. Tape	1	1

(1) Load 6-bit Information ($F_{18} = 1$)

$F_{18} = 1$ causes each character of input to be treated as 6-bit information. Since the console keyboard has only keys for sexadecimal characters and the end symbols, alphabetic or symbolic characters not appearing on the keyboard (e.g., K, +, Tab) cannot be read as 6-bit information per se, but can be read from the keyboard if first converted into 4-bit information.

In all 6-bit Input orders, the information is read, six bits (i.e., one character) at a time, into the right end of Qt, Qt being shifted 8 places to the left before each character is read. Thus, after one character has been read, q_{34} and q_{35} hold zone pulses, q_{36} through q_{39} hold the sexadecimal character, and $q_{32} = q_{33} = 0$. After each group of five characters or portion thereof is read, Qt is stored in the memory. F_{19} determines whether one character or a block of characters is to be read.

Read One Character ($F_{19} = 1$): When $S_3 = D$ and $F_{18} = 1$, then $F_{19} = 1$ causes exactly one 6-bit character to be read into Qt and the memory, as follows: Qt is shifted left 8 places, one character from the tape (paper or magnetic) is read into Qt as described above, and the entire contents of Qt are stored in the memory at the A-address. Since no symbol is used here to indicate the end of input, any end symbol may be read, using this order, as ordinary 6-bit information.

Read Message, to E_2 ($F_{19} = 0$): When $S_3 = D$ and $F_{18} = 1$, the setting $F_{19} = 0$ causes a continuous alphabetic, or 6-bit, message to be read into consecutive locations in the memory, until an E_2 appears on the tape. The order works as follows:

The 6-bit characters are read into Qt, one at a time, Qt being shifted left 8 places before each character is admitted. After each group of 5 characters has been read and shifted through Qt, Qt is stored in consecutive memory locations, the A-address of the instruction being the address of the first location, until an E_2 appears on the tape. If, when E_2 appears, Qt contains k as yet unstored characters, where $k \leq 5$, then Qt is shifted left $(40 - 8k)$ places, so that the unstored characters are moved to the left-most part of Qt and the stored characters shifted off, and then Qt is stored. For example, if a message containing 13 6-bit characters followed by an E_2 is read, the first 5 characters will be stored at A, the next 5 at $A + 1$, then the next three will be read into Qt, and, after E_2 is recognized, Qt will be shifted left 16 places $(= 40 - 8 \cdot 3)$, and its contents (last 3 characters of the message in $q_0 - q_{23}$, 0's in $q_{24} - q_{39}$) stored at $A + 2$.

(2) Load 4-bit Information ($F_{18} = 0$)

$F_{18} = 0$ causes input data to be treated as 4-bit information. Only the information found in hole-positions 1 through 4 is stored. The zone pulses (positions 5, 6) and position 7 are examined only for parity check purposes, by the input equipment. 4-bit information may enter the machine through any of the input equipment described previously. E_1 and E_2 are used as end of word or message symbols, respectively, with all 4-bit Information Read orders. F_{19} determines whether one word or a block of words is to be read.

Read to E_1 (One Word) ($F_{19} = 1$): On Input orders where $F_{18} = 0$, the value $F_{19} = 1$ causes one word of data to be read, as 4-bit information, one character at a time, into the right end of Qt, Qt being shifted 4 places before each character is read, until an E_1 is recognized on the tape (or keyboard). Then the contents of Qt are stored in the memory, at A, and the order is complete.

Thus, ten 4-bit characters constitute 40 binary bits (= one word). If more than 10 characters are read before E_1 is recognized, only the last 10 characters preceding E_1 are stored. If $k < 10$ (where k = number of characters read), the stored word will begin with a portion of the previous contents of Qt.

An E_2 will act as an E_1 when E_1 is being sought in this order; that is, the reading process is completed when either an E_1 or an E_2 is recognized.

Load to E_2 (Block of Words) ($F_{19} = 0$): On Input orders where $F_{18} = 0$, the condition $F_{19} = 0$ causes a block of words to be read as four-bit information into consecutive memory locations. Each word of this block, except the last, must be followed by an E_1 , and an E_2 must follow the last word to be read. The order works as follows:

The data are read as described under "Read to E_1 ($F_{19} = 1$)" into the Qt. The contents of Qt are stored each time an E_1 is recognized. When the first E_1 appears, Qt (= first word) is stored at A; after the second E_1 appears, Qt (= second word) is stored at A + 1, and so on. When an E_2 is recognized, the contents of Qt (= last word of the block of k words) are stored at A + ($k - 1$) and the order is complete.

Input Orders ($S_3 = D$)

6-bit information

$S_4 =$	$F_{16} \quad F_{17}$		F_{18}	F_{19}
	1	See "Input Modes"	1	One Character
0	0			Message $\rightarrow E_2$

4-bit information

$S_4 =$	$F_{16} \quad F_{17}$		F_{18}	F_{19}
	1	See "Input Modes"	0	Read to E_1
0	0			Read to E_2

c. Output Orders ($S_3 = C$)

GEORGE output may be either sexadecimal or alphabetic, on punched paper tape, narrow magnetic tape, or typed directly onto a console typewriter. The mode of output desired is selected by assigning the appropriate values to F_{16} and F_{17} as follows:

Output Modes

Copy to	F_{16}	F_{17}
Paper Tape	0	0
Magnetic Tape	1	1
Console Typewriter	1	0

(1) Copy 6-bit (Alphabetic) Information ($F_{18} = 1$)

$F_{18} = 1$ causes information being punched or printed to be treated as 6-bit or alphabetic information. The desired output information is brought to Qt from the memory. The information is then punched, one character at a time, from the left of Qt, that is, q_0 through q_7 constitute one 6-bit character where q_0, q_1 , are ignored, q_2, q_3 are punched as zone positions, q_4 through q_7 are punched as the sexadecimal character. F_{19} determines whether one character or a message is to be punched.

Copy One Character ($F_{19} = 1$): When $F_{19} = 1$, the first 6-bit character, only, of the word at the A address is punched; that is, the contents of the word at the A-address are brought into Qt, the first character is punched (as described above), and the order is complete. Note that the punched character is not shifted out of Qt after the output process. Qt still holds the word at A upon completion of the order.

Copy Message (to E_3) ($F_{19} = 0$): $F_{19} = 0$ on 6-bit Output orders causes a sequence of 6-bit characters, stored 5 per word in consecutive memory locations A, A + 1, ..., to be punched until an E_3 is recognized. The order works as follows:

The word at the A address is brought into Qt. The characters are punched one at a time, as described previously, Qt being shifted left eight places after each character is punched, thus moving the next character into position. If the character in positions q_2 through q_7 is an E_3 , the E_3 is punched, and the order is complete; if not, the process is continued with succeeding words until E_3 does appear.

(2) Copy 4-bit Information ($F_{18} = 0$)

$F_{18} = 0$ causes the output information to be treated as 4-bit or numeric information. The information is punched, one word at a time, from Qt. Each word of 4-bit information is made up of ten 4-bit characters, followed by an E_1 symbol. Thus, positions q_0 through q_3 make up the first character, positions q_4 through q_7 the second, and so on.

Copy One Word from Qt ($F_{19} = 1$): When $F_{18} = 0$, then $F_{19} = 1$ causes the contents of Qt to be punched as one word of 4-bit information, followed by an E_1 . The A-address of this order is irrelevant. As Qt is shifted left 4 places after each character is punched, $q_0 - q_{39} = 0$ after the Output order is completed, but q_x is not necessarily 0.

Copy One Word from Memory ($F_{19} = 0$): When $F_{18} = 0$, the condition $F_{19} = 0$ causes the word stored at the A-effective address to be punched (or printed) as 4-bit information, followed by an E_1 . In this order, the word stored at A-eff is brought into Qt from the memory; then the process of shifting and punching proceeds as in "Copy One Word from Qt."

Output Orders ($S_3 = C$)

6-bit Information

		F_{16}	F_{17}	F_{18}	F_{19}
$S_4 =$	1	See "Output Modes"		1	1 Character
	0				Message to E_3

4-bit Information

		F_{16}	F_{17}	F_{18}	
$S_4 =$	1	See "Output Modes"		0	1 word from Qt
	0				1 word from M.

6. Wide Magnetic Tape Orders ($S_3 = A, B$)

When the additional storage capacity of the wide magnetic tape units is needed in a program, the Ax and Bx orders are used to transfer information between the tapes and the core memory, and to move or rewind the tapes backward or forward to position them for reading or writing.

At this writing, the tape units are not yet available for use by a program, so the description of the units and their use will be excluded at this time. A supplement to the manual containing this description will be published at a later date, when the units are ready for use.

7. Stop Orders ($S_3 = 0$)

The Stop and Recognition Breakpoint orders provide a convenient means of monitoring the program, especially valuable in debugging. Non-programmed stops (address recognition, error, manual) are discussed in the section on "Console Operations."

All Stop or Recognition Breakpoint orders are coded $0X$ ($X = 0, 1, \dots, F$). $X = 0$ (i.e., a 00 order) always stops the machine unconditionally. For $X \neq 0$, the interpretation of the order depends on the settings of the console buttons. The console holds 15 recognition breakpoint buttons, numbered 1 through F, and three breakpoint mode buttons, labeled STOP, IGNORE, and JUMP, respectively. For an $0X$ order, if the button corresponding to X is pushed, one of the following will occur, depending on the mode button:

- a. STOP mode on - $0X$ order stops the machine.
- b. IGNORE mode on - $0X$ order is treated as a Stall.
- c. JUMP mode on - $0X$ order is treated as a Stall, but a jump to B is forced.
- d. STOP and JUMP modes on together - $0X$ order stops the machine after the instruction at B is fetched (a "1" is forced into F_{27}).
When the machine is restarted, it will continue by first executing the instruction in MR, i.e., the instruction at the B address.

If the console breakpoint button X is not set, the $0X$ ($X \neq 0$) order is treated as a Stall order.

If STOP mode is used and a stop occurs, or if a Stop (00) occurs, a jump to the B address can be forced by pushing the JUMP to B console button (see "Console Operations").

The $0X$ order causes the machine to stop, if at all, after the preliminary and fetch cycles of the Stop instruction have been completed. No result cycle is available on a Stop order, but, except for F_{24} , the other bits of the tag are relevant.

Following a stop caused by an $0X$ order, the registers will hold the following:

Function Register - Contains the order just executed.

Memory Register - Contains the next order.

Control Counter - Contains 1 + the address of the order executed.

Dispatch Register - Contains the address of the next order.

Jump Register - Contains 1 + the address of the last Jump order executed.

D GEORGE ORDER DECODING TABLE

Order	Targ															
	F ₁₂ - F ₁₅	F ₁₆	F ₁₇	F ₁₈	F ₁₉	F ₂₀	F ₂₁	F ₂₂	F ₂₃	F ₂₄	F ₂₅	F ₂₆	F ₂₇			
Bits Order type																
Stop	0	1 0	1 0	1 0	1 0	Not OFS	See Pcl to chart below	Cl Qt Cl Ac	Cl Qt Cl Ac	Θ	P ₄ Not	B # Not	J to B Not			
Σ	1	Result to A Not	Mag # -	- +	To Qt To Ac	"	"	Cl Qt Qt Ac M Ac to B	Result to B Not	"	"	"	"			
π	2	Result to A Not	19 39	R O Not	- x	"	"	Cl Qt Cl Ac	"	"	"	"	"			
L	3	OFR Not	2 Reg 1 Reg	End Ard Not	Qt Ac	"	"	"	"	"	"	"	"			
R	4Comb 4Power	1 0	" "	" "	" "	"	"	"	"	"	"	"	"			
α	5	B-Port to M Not	O-Port to M Not	T-Port to M Not	A-Port to M Not	"	"	Cl Qt Qt Ac Ac to M	Result to B to A	"	"	"	"			
	6															
J	7Sign	0	0	- +	Qt Ac	"	"	Cl Qt Cl Ac	Result to B Not	"	"	"	"			
U	7OF	0	1	Yes No	" "	"	"	"	"	"	"	"	"			
M	7Carry	1	0	Yes No	" "	"	"	"	"	"	"	"	"			
P	7Uncond	1	1	Yes No	" "	"	"	"	"	"	"	"	"			
	8															
	9															
M V e	A	Bkwd Frwd	Re- wind #	1 0	1 0	"	"	"	"	Θ	"	"	"			
T read write	B	Θ	Write	1 0	1 0	"	"	"	"	Θ	"	"	"			
C	C ₆ -bit	See Copy to chart below	1	1	1Ch Mess to E ₃	"	"	"	"	Θ	"	"	"			
O	C ₄ -bit	"	"	0	1Wd to E ₃	"	"	"	"	Θ	"	"	"			
P	C ₄ -bit	"	"	0	1Wd to E ₃	"	"	"	"	Θ	"	"	"			
L	D ₆ -bit	See last from chart below	1	1Ch Mess to E ₃	"	"	"	"	"	Θ	"	"	"			
O	D ₄ -bit	"	"	0	E ₁ E ₂	"	"	"	"	Θ	"	"	"			
A	D ₄ -bit	"	"	0	E ₁ E ₂	"	"	"	"	Θ	"	"	"			
D	E															
	E															
β	F	1	0	0	1	"	"	"	"	Result Re- to suit B to A	"	"	"			
Record	F	C C → M B Not	1	Tog → M O T Not	S C → M A Not	"	"	"	"	"	"	"	"			
F8	F	1	0	0	0	"	"	"	"	"	"	"	"			
Set	F	0	0	C O F O F	Qt Ac	"	"	"	"	Θ	"	"	"			

Pcl to	F ₂₁	F ₂₂
Hold	0	0
0	0	1
2 ⁻¹¹	1	0
2 ⁻¹⁹	1	1

P ₄	F ₂₃
All Orders Except π	Cl Qt Qt + M → Qt Ac + M → Ac Cl Ac
x	Cl Qt Qt + M → Qt Ac + M → Qt Cl Ac
-	Cl Qt Qt + M → Ac Ac + M → Ac Cl Ac

Copy to	F ₁₆	F ₁₇
Paper Tape	0	0
	0	1
Type- Writer	1	0
Mag Tape	1	1

Load from	F ₁₆	F ₁₇
Paper Reader 1	0	0
Paper Reader 2	0	1
Keyboard	1	0
Mag Tape	1	1

E. GEORGE Overflow Control

- a_{OF} is cleared by any preliminary add or Σ , π , or Shift order whose results go to the Accumulator.
- q_{OF} is cleared by any preliminary add, or Σ , π , or Shift order whose results go to the Quotient Register.
- c_{OF} is cleared by any Σ , π , or Shift order, or by any preliminary add.
- a_{OF} is set when $a_x \neq a_0$ as a result of a preliminary add or any order except a divide; or by an a_{OF} Set order (F0).
- q_{OF} is set when $q_x \neq q_0$ as a result of a preliminary add or any order except a divide; or by the first step of a division where the absolute value of the divisor is less than that of the dividend or where the dividend is positive and equal to the absolute value of the divisor; or by a q_{OF} Set order (F1).
- c_{OF} is set whenever there is a carry into stage x of the Plusser; or by a c_{OF} Set order (F2).
- OFS (Overflow Stop) occurs when $F_{20} = 0$ in the current instruction, and a preliminary add, Σ , π , or Left Shift (non-end-around) order in that instruction causes either a_{OF} or q_{OF} to be set.

The overflow stop occurs after the instruction is completed (i.e., after the next instruction has been fetched). In case of an OFS, the result cycle is omitted, except on Jump, α , β , F8, and Record orders.

F. GEORGE Timing

The basic unit of GEORGE operation time is the memory cycle (m.c.) of 27 microseconds. Current order times are as follows:

PC, FC, RC	1 m.c.
+, -	1 m.c.
x	15 to 21 m.c. (average = 18 m.c.)
-	22 m.c.
Shift of n places	$(0.78 + 0.33n)$ m.c.
Input Paper Tape	200 characters per second
Output Paper Tape	60 characters per second
Print (Console Typewriter)	7 characters per second

500 microseconds (~ 24 m.c.) are available for computation between input paper tape characters.

III. PROGRAMMING TECHNIQUES

A. Machine Language

The process of preparing a problem for solution on the computer, once a precise mathematical statement has been formulated, involves a series of integrated steps. The purpose of this section is to outline and explain these steps, and to suggest a few checkpoints to facilitate their execution.

The steps involved in setting up a program are:

1. Statement of the Problem;
2. Scale Assignment and Scaled Equations;
3. Selection of Subroutines to be used;
4. Preparation of a Machine Code;
5. Checking Machine Code (Debugging);
6. Clear Operating Instructions;
7. Program Write-up.

1. Statement of the Problem

The statement of the problem is normally prepared at the time the problem is initiated and contains the following information:

- a. Description of the Problem;
- b. Formulas and/or method to be used in the solution;
- c. Desired output;
- d. Required input;
- e. Ranges of the parameters and variables involved.

2. Scale Assignment and Scaled Equations

Since a standard GEORGE machine number x must lie in the range $-1 \leq x \leq 1 - 2^{-39}$, and since most problems which treat physical data involve variables outside this range, scaling of some form must be done in these problems; that is, parameters and intermediate and final results which lie outside the range $[-1, 1 - 2^{-39}]$ must be multiplied by factors which will cause the numbers to stay within machine range. Since GEORGE represents numbers in the binary system, it is usually convenient to scale by powers of two, for then scaling and unscaling are accomplished by shifting, which is faster than multiplication or division.

It is possible to treat numbers of an extended range without scaling - by using floating point arithmetic via an interpretive routine. Here, all numbers are held in two parts - a fraction and an exponent - and the actual arithmetic is performed within the interpretive routine which operates on the fractional parts, then adjusts the exponent appropriately. Details on the use of floating point appear in the write-ups of the two floating point arithmetic routines available (D-1-115 and D-3-131).

Obviously, operations utilizing floating point arithmetic increases the computing time considerably. Problems in which the range of numbers can be closely approximated are more efficiently done without floating point by doing appropriate and careful scaling before writing the code.

Although the details of scaling differ with each program, a few general points, if noted, can facilitate the scaling process.

The most straightforward way of scaling is by finding the maximum and minimum absolute values of the quantity to be scaled, then multiplying the quantity by the smallest power of two (or ten) which makes the resulting scaled quantity within machine range.

Any one of several algebraic or analytic methods can be used to find the upper and lower bounds of the variables involved. However, it is advantageous to consider also the behavior of the variables and functions in physical reality, before choosing scale factors. Very often, in the particular experiment or circumstances which the problem represents, the variables involved in some function at a given time take on such values that they cancel one another, or act together to keep the function within a small range.

For example, take $f = 1 + a_1 x + a_2 x^2$, where $0 < x < 10$, $0 < |a_1| < 10^{-1}$, $0 < |a_2| < 10^{-1}$. Considering the domains of x , a_1 , and a_2 , we have $|f| < (1 + 10^{-1} \cdot 10 + 10^{-1} \cdot 10^2) = 12$. However, if the added information that $0 < f < 1$ is available, then f itself requires no scaling. Furthermore, since $f < 1$, we have $-1 < (a_1 x + a_2 x^2) < 0$, so that this sum is also in range and needs no scaling. Only x requires a scale factor. Suppose x is scaled by 2^{-4} , i.e., $\bar{x} = 2^{-4} x$ is used in the computation. Then we can compute $f = \left\{ [(\bar{x} a_2) 2^4 + a_1 \bar{x}] 2^4 + 1 \right\}$.

While in this example only a few bits of accuracy are saved by knowledge of the interrelation of a_1 , a_2 , and x , in many problems analysis of the physical relations of the quantities can save an appreciable amount of accuracy by reducing the range of numbers and thus avoiding unnecessary overscaling.

Another method of preserving accuracy is to write scaled equations so that the order in which a function is computed allows intermediate results to stay as close to one (in absolute value) as possible. Thus, loss of accuracy by truncation is cut down.

It is sometimes convenient to scale a function by a factor other than a power of two or ten. For example, one may scale $\arctan x = f$ by π instead of by a power of two, or a series involving exponentials may be scaled for convenience, by a power of e instead of a power of two.

Since counters are normally integers, scaling them by either 2^{-11} or 2^{-39} makes incrementing the counters simple. A counter can thus be increased by adding it to a register which has been cleared to 2^{-11} or 2^{-39} , or by incrementing the B or A portion of the word containing the counter.

Sometimes it is impossible to determine the exact upper bound of a quantity, or the quantity may reach this bound only in very rare cases. Here, a variable scale factor - or inner scaling - may prove advantageous; that is, the variables involved may be initially scaled to take care of the most common range, and the program coded such that, when these variables exceed the allowed range, they are rescaled [e.g., by shifting (multiplying by a negative power of two)] and the scale factors and other scaled variables involved adjusted accordingly. This testing and adjusting, while increasing the running time and amount of code necessary by a small amount, can allow a greater range for variables in a program without the usual accompanying decrease in accuracy.

Once scale factors are selected and scaled equations written, it is important that these be recorded as a permanent part of the program write-up, to facilitate future uses and revisions of the program. The write-up should contain a definition of the relations between the unscaled variables and equations and their scaled counterparts. The ranges before and after scaling should also be included.

3. Selection of Subroutines

Subroutines are codes written in a form convenient for incorporation into any program, and for evaluation of some much-used function or mathematical system. For example, subroutines are available for computing square roots, exponentials, sines or cosines of given arguments, finding the roots of a polynomial, and inverting a matrix.

Before preparing a code for a given program, investigation should be made into the library of subroutines available, to find those which can be incorporated into the program. If several subroutines are available for evaluating the same function or system, that subroutine should be selected which most closely approaches the speed and accuracy desired in the program.

Assigning one particular section of the memory for storage of subroutines and storing all the subroutines consecutively in this block usually proves quite convenient in arranging the remainder of the program, since the subroutines can be loaded with one load order, and the subroutine code tape need not be changed during debugging, since this code is already checked out. The Subroutine Loader (X-1-126) provides a convenient way of preparing subroutines for use in a program, since it can adjust addresses appropriately, store the subroutines, and punch the compiled subroutines.

Some of the conventions and definitions recommended for use in GEORGE Library subroutines follow.

a. Rel-000 of a subroutine is defined as the first word of the subroutine (regardless of where the subroutine is coded).

b. Length of a subroutine should be given in terms of relative addresses of the first and last word and in the number of words (e.g., Length: Rel 000-00F, 16 words) in the write-up.

c. Constants - If a constant used by a subroutine is one of the universal constants stored with the Command Routine (X-2-000), it should not be stored separately within the code, but should be used from its fixed storage in the list of universal constants. The write-up should indicate all constants used by the routine, as well as their relative or fixed addresses. (Addresses of universal constants may be omitted.)

d. Temporaries - The end of the memory, i.e., FFF, FFE, ..., in general is reserved for temporary storage. Therefore, where possible, subroutines should use these locations for needed temporary storage to leave as much other space as possible for the main program. Locations of all temporaries used by the routine should be given in the write-up.

e. Entrance-Exit -

(1) Fixed-Point Routines

The standard entrance to a subroutine which computes $f(2^P x)$ is the following:

$$Ac^B = *p(= 2^{-11} p), \quad Qt = x .$$

At R, record the control counter at and jump to Rel 000. When the computation is complete, the subroutine returns to R + 1, with:

$$Ac = 0, \quad Qt = f(2^P x); \text{ or } Ac = s*(= 2^{-39} s) \quad Qt = 2^{-S} f(2^P x).$$

Thus, the subroutine should be coded so that the code begins at Rel 000, and exits by jumping to the instruction at the address given in Rel 000B.

(2) Floating-Point (FP) Routines

To compute $f(y_i) = z$, where y and z are standard floating-point numbers and $i = i_{FP}$ (see D-1-115 write-up):

- (a) If the computation does not involve FP operations (i.e., computation is done in fixed point, and the FP routine is needed only for unpacking y_i , rescaling and packing z), the subroutine should be treated as a FP operation (FP*), and the standard FP entrance used, but with the argument (y_i) stored in the memory at $M(y) + i_{FP}$, and Rel 000 of the subroutine given as the operation address, $M(FP^*)$; i.e., entrance word, at $Q_1 = \text{Rel } \underline{000} \text{ FCCF } M(y)$.

After $z = f(y_i)$ is computed, it should be rescaled and packed, via the FP routine, and control returned to $Q + 1$, with z in Q_t and in Rel 037 (FP). To achieve this, Rel 000 of the subroutine should contain the following key word;

$$M(P_{FP}) \text{ } 00 \quad 00 \text{ } M(\beta \text{ SUBR}) ,$$

where

$$M(P_{FP}) = \text{Rel } \underline{02A} \text{ (FP)}$$

$$M(\beta \text{ SUBR}) = \text{the address of the subroutine's first word of code.}$$

Thus the program enters FP at P_{FP} , $y_{i_{FP}} = (y_f, y_p)$ is unpacked, and y_f and y_p^* are stored at Rel 03A, 03B, (FP), respectively, the exit $\omega = Q + 1$ is set up, and control transferred to $\beta \text{ SUBR}$, with $Q_t = y$, ready for computation.

After z_f and z_p $\left(z = z_f \cdot 2^{z_p - 128} \right)$ are obtained, the subroutine should jump to $S_1 [= \text{Rel } \underline{000}(\text{FP})]$ with z_f in Q_t , and z_p^* in Ac and in Rel 036(FP).

The FP routine will then rescale and pack z , and return to $\omega = Q + 1$, in the main program, with $z = f(y_i)$ in Q_t and in Rel 037(FP).

- (b) If FP operations are used within the routine, or if unpacking is not required, the standard fixed-point method of subroutine entrance-exit, described above, should be used.

f. Special Features - If, at no added word expense, the result of a subroutine can be left in a temporary, as well as in Q_t , this should be done, and this fact indicated in the write-up.

g. Stops - If an "error" condition is checked for in a subroutine, the subroutine should be coded to jump to the Subroutine Error Stop Routine (X-3-167) when the error condition occurs.

h. Time - Average running time, per argument, of a subroutine should be indicated on the subroutine write-up. This time can be obtained by actually timing the routine, using a large number of arguments (preferably random numbers) and then obtaining the mean time, or by summing the memory cycle times involved in computing one argument.

i. Mathematical Methods, including an explanation of the method used and an error analysis, should be included in the subroutine write-up.

Subroutines for which these conventions are not entirely feasible should still conform as closely as possible to the standards given here.

4. Preparation of a Machine Code

The preparation of a machine code involves rewriting the set of mathematical formulas making up the statement of the problem as a sequence of logical and arithmetic steps, and then converting these steps into a machine code, which the machine can interpret to produce the desired solution.

The preparation of such a code for a given machine can be done in varying degrees of sophistication, depending on the extent of the library of utility, assembly, and compiling routines available. The techniques discussed here are most applicable to the straightforward procedure of writing the whole code in basic machine language. However, the hints and suggestions are also applicable to programs written with the aid of subroutines and/or an assembly routine (a routine which does most of the code book-keeping, inserting appropriate address, assembling various parts of the code, etc.). Added points on the use of assemblers are given in the description of GAR (GEORGE Assembly Routine) in Section III.B.

The next degree of sophistication in coding is the use of compilers. A compiler not only does the code bookkeeping, but also constructs the code from a series of specially prepared mathematical statements, thus moving the job of coding from the programmer to the machine itself. At this writing, GEORGE programs are written via basic language and GAR, since a GEORGE compiler is not yet available. When a compiler is completed, its description and rules for its use will be added as a supplement to this manual.

In this section, a few of the more important considerations involved in code preparation will be discussed, and a few helpful hints on good coding will be given, but proficiency in programming comes mostly through experience. Thus it is advised that the programmer carefully consider making each code prepared as efficient as possible. In this way, he will become familiar with existing techniques, develop his own methods for writing a good program, and will learn to avoid common programming errors.

a. Flow Charts

A problem which involves any amount of testing or making of decisions to obtain its solution is usually too difficult to program unless its individual parts and branches can be visualized in relation to the problem as a whole. This can be accomplished with the use of a flow chart. A flow chart is merely a visual representation of the course of control the machine must follow as it moves through the various stages of the problem. It shows the exact sequence of operations on the data, shows how loops are tested, and in general allows one to see the over-all picture of a program.

A flow chart has added usefulness as a convenient communication between the original programmer of a problem and someone who takes over the problem to complete it or to modify it in some way. To understand the logic of a program when given only a listing of the instructions used is almost impossible. It is equally difficult to describe what has been done in a program by a few short sentences or formulas. But a newcomer to a problem may, by investigating a flow chart, see quickly how the computations are carried out and how the various alternatives fit together.

Basically, a flow chart consists of a series of boxes connected by lines, with arrows indicating the direction of the flow, and with the contents of the boxes indicating the operations to be performed.

For convenience, certain conventions for flow charts have been established at Argonne and are suggested for use in programs written for GEORGE and the IBM 704. While their use is not mandatory, these conventions do facilitate an understanding of the work of one programmer by another, thereby eliminating the need to interpret individual eccentricities in notation.

Except for the conventions taken from the "Report of AD HOC Committee on Flow Charts Symbols" (SHARE - Sept. 11, 1958), prepared from suggestions by users of the IBM 704, the suggestions outlined below are Argonne Flow Chart Conventions and not necessarily those used at other computer installations.

(1) Operation Boxes; Arrows

Arithmetic, substitution, and other operations which do not involve decisions on the part of the computer are indicated in operation boxes, as illustrated. The order of the operations being performed, if this is of concern, as well as the mathematical form of the expression being evaluated, are written in the box. Each such box has only one entrance and one exit, indicated by arrows leading to and from the box. For example, we may wish to set $x = 0.5$, $y = 0.1$ and then compute $f = x^2 + y + z$. We can indicate this as follows:

The arrow within the box means "replaces" or "goes into," as 0.5 replaces the former value of x . These operations could have been written all in one box, but it is easier to follow a flow chart when similar types of operations are separated by boxes from other types. Operation boxes may be numbered, and the numbers used may appear on the coding sheet to show the correspondence between the box and its code.

(2) Decision or Comparison Boxes

This type of box is used to indicate branch or conditional types of operations.

The symbol ":" represents "the relationship between" a and b , specified by the two exits. The illustration above indicates that if $a \leq b$, jump to box 9; if $a > b$, jump to box 10.

(3) Connectors

Two types of connectors are used.

A fixed connector is used to connect remote parts of a flow chart with one another without the use of long or crossing lines. Fixed connectors are normally indicated by a square with an arabic letter inside; e.g.,

This indicates that when the operations in box 10 are completed, the operations in box 19 are begun. One or more paths in a flow chart may lead to a single connector.

At a point where control is predetermined to transfer to one of several possible points, a variable connector is used. This is usually indicated by a circle enclosing a Greek letter:

The illustration shows that box 2 may exit to one of three boxes - 5, 21, or 40 - depending on how α is set at the time. The connector is set in a separate box, sometime before control is sent to box 2. For example $\alpha_1 \rightarrow \alpha$, or $\alpha = \alpha_1$, indicates that box 2 is being set to exit to box 5 (α_1). Here again, " $\rightarrow$ " means "replaces," " $=$ " means "is set equal to." Since setting variable connectors is done by adjusting addresses, $\alpha_1 \rightarrow \alpha$ means to place the address of the first instruction of box 5 into α , where α is the address of a jump instruction at the end of box 2.

(4) Subroutines

This type of box is used to indicate that a subroutine is being used to compute some function. The notation inside the hexagon may refer to a detailed flow chart contained elsewhere. Execution of the subroutine occurs at this point.

(5) Assertion Boxes

Various blocks may be attached to the logical flow lines by dotted or dashed lines to enhance the clarity of the flow chart, e.g.,

1. by indicating that a counter or instruction is in its initialized state;
2. to describe a coding trick; or
3. to make any parenthetical type of remark that may serve to clarify the program.

(6) Entrance, exit

The start and end of a program are usually indicated by boxes with the words IN or START, STOP or END written within them.

The important thing to remember in writing a flow chart is that it is a tool to aid in coding and to help visualize the program in its many paths. Thus long sequences of straightforward algebra should be written concisely, not "strung out" over many boxes, while decisions, order of operations (where important), and the various alternatives available should be emphasized and clearly indicated.

b. Loops

One of the most powerful features of stored program computers is the ability to repeat, with modification, basic parts of a program at electronic speeds. The sequence of operations which are to be retraced, and the bookkeeping operations of counting, testing, etc., make up what is called a loop.

A loop may be one of two kinds. In one type, the same procedure is repeated a predetermined number of times, perhaps using new parameters each time, as in the evaluation of a polynomial with a fixed number of terms. A counting procedure is built into the loop code, and the process is completed when the counter shows the loop has been repeated the proper number of times.

In the second type, a sequence of operations is repeated a variable number of times until some predetermined condition is reached, as, for example, convergence of successive approximations of a function. Here counting is not always necessary. The loop computations are ended when the specified criterion of convergence or accuracy is reached.

Both types of loop usually involve four steps - initialize, compute, modify, test - although the actual execution or order of these steps may differ with each type, or even with each loop of one type.

Initialization involves setting up the preliminaries necessary for proper execution of a loop. These preliminaries include storage of the starting values of counters and variables, clearing to zero those memory cells in which sums will be accumulated, and, in general, setting up all the conditions required at the start of the loop. This step is necessary in any type of loop and should precede the actual loop computations. Initialization steps are not normally repeated during the course of execution.

The computing step involves a new iteration, obtaining a new approximation to a function, adding another term to a series being computed, or the like. If the same memory cells are referred to in each cycle of the loop, then this step is treated in the same way as a similar computation outside a loop. If the same computations must be done in each cycle of a loop, but with the use of the contents of a new memory location for each new cycle, then addresses within the computing part of the loop must be arranged to vary for each cycle. This item is usually taken care of in the initializing and modifying steps of a loop.

Because of the automatic indexing (B-Boxing) available on GEORGE, the modifying and testing steps can usually be combined into one. Modifying involves an arrangement to provide that correct addresses are used during the next cycle. This can normally be done by increasing or decreasing counters. Testing determines when a loop is finished; it may be done by checking the value of a counter to see whether a predetermined number of cycles has been completed, or by checking the value of a computed function for convergence or desired accuracy by a predetermined criterion. The following examples illustrate various points regarding loops.

$$(1) \text{ Compute } f = K \sum_{i=1}^8 a_i \text{ and store result in } M(f).$$

Suppose $a_1, a_2, \dots, a_8$ are stored consecutively in the memory at locations 301, ... 308, respectively. K is stored at 309; f will be stored at 30A. The counter i will be stored in the B portion of 300. The flow chart and code are as follows:

Note that initializing is done in box 1, computing in box 2, modifying and testing in box 3. Box 4 is not technically a part of the loop.

Since the loop computation involves only addition, and thus only one register, it is convenient to store the running sum in Q_t . Both code and time are saved in this way by elimination of several store orders, including the storage of 0 in Σ .

The use of B-boxing simplifies the modifying and testing operations of the loop. By setting the counter to (-8) initially, and adding one to it after each step and before testing for completion, we can increase the counter and test (on sign) all in one word. As long as the counter is negative, the loop computation is incomplete. When the counter becomes positive (i.e., $= 0$) after it is increased by 1, we have done the computation 8 times, and the loop is finished. $i + 1 \rightarrow i$ yields -7 after one step, -6 after 2, ..., -1 after 7, and 0 after 8. To assure use of the proper a_j in each step, the address of a is set permanently in the code to 309, i.e., $M(a_8) + 1$. With B-boxing, the first step uses the address $[309 + (-8)] = 301 = M(a_1)$, the second step uses $309 + (-7) = 302 = M(a_2)$, and so on.

Note, also, that, since Ac or Q_t can be cleared by the tag to 2^{-11} , counters can be conveniently stored in the B portion of a word and increased and replaced without use of the order, only the tag; [i.e., clear to 2^{-11} , preliminary add (i.e., $P +$) $C(B) = i$, send result to $B = M(i)$].

(2) Obtain a random vector, with coordinates $\tau_k = R_k / \sqrt{\tau_3}$ ($k = 0, 1, 2$), where R_k are random numbers generated by a Random Number Subroutine and chosen so that $\tau_3 = \sum_{k=0}^2 R_k^2 < 1$.

We must continually take random numbers until the sum of the squares of three successive random numbers is less than 1. Thus, the loop will involve testing on a counter and on a particular condition, but the number of times it is executed to obtain one vector cannot be predetermined. The flow chart and code are given below, with a commentary following.

Assume the following storage assignments:

τ_0 at 100 Random Number Subroutine starts at 200.
 τ_1 at 101 Square Root Subroutine starts at 210.
 $\tau_3, \sqrt{\tau_3}$ at 103 Main Code starts at 300.
 k at 301^B

The subroutines used are entered by storing the exit address in the B-portion of the subroutine's first word and jumping to the second word. They exit to the respective exit address, leaving the result in Qt. The square-root subroutine requires that the argument be in Qt upon entry.

In this example, Box 1, the initialization, may be repeated several times during the computation of the vector. Boxes 2 and 3 contain the computation and part of the testing portions of the loop. Since the vital condition is that the sum be less than 1, a test on overflow is the most convenient method of checking for completion of the loop. If the sum of at least two R^2 's is found to overflow, the current sum is rejected and the process started anew with the initialization.

Since three quantities are needed to complete the process, a count to three and a test on the counter is also necessary (Box 4).

Technically, boxes 5 and 6 are not a part of the loop, as they are entered only after the loop conditions are met. It is here that the final values of τ_0 , τ_1 , τ_2 are computed, as the value of τ_3 is < 1 as desired.

For further illustrations of loops, one may look at various subroutines in the GEORGE Library. E-3-105 (Square Root) illustrates a straight "convergence" loop, where the loop is repeated until an accurate approximation to $\sqrt{x}$ is obtained. E-1-103 (Exponential) obtains e^x by computing a polynomial, with a given number of precomputed coefficients.

When several separate operations must be done on each member of a set of consecutively stored parameters, it is normally more efficient to code one loop, in which all the operations are done using one parameter at a time, than to code several loops, each loop performing one operation on the parameters. If all the operations are consolidated into one loop, both code and time are saved by cutting down on the number of counting operations necessary.

c. Input - Output

Input-output techniques may differ with each new program. However, certain rules should be observed to assure a smoothly running, accurate program.

In general, all final numerical data to be used as input or output should be written externally as decimal, rather than sexadecimal, data, with conversions to and from sexadecimal being performed within the computer. In this way numerous errors are avoided, and much time is saved by elimination of the need to manually convert data to be read into a program as input or which comes out of a program as output. This does not necessarily apply to addresses, code words, keywords, etc., outputted in the course of code checking.

A variety of Input - Output Subroutines are available for use in GEORGE programs.

Most programs involve some logical input information which determines what path of many will be followed, what type of computations should be done, or the like. This type of input is handled best if arranged as concisely as possible in the form of keywords, if possible, with the value of certain bits in the keyword representing the various options available. The keyword is read by the program, and the machine deciphers the keyword, then sets connectors or computes various functions appropriately, e.g., a keyword may be set so that

000 0 0 0	means do not compute some quantity, say T_i ;
001 0 0 0	means compute T_i and output T_i ;
003 0 0 0	means compute T_i and output $(T_i)^2$.

The unused portions of the keyword, indicated by 0, may be used to select other distinct options. Thus redundant information on a data tape is avoided. A compact keyword replaces long lists of alphabetic information. Since the machine can decipher keywords, set up connectors, convert and scale numbers, etc., much more rapidly than it can read extra information, a considerable amount of machine time is saved. Also, all information which stays constant for every run of a program, including headings and alphabetic information, as well as constants, should be stored together with the code for the program instead of being read from the input data tape. This cuts down on the amount of tape punching necessary and thus again decreases the chance of errors entering into a program.

Once the form of the input data and the order in which it must be read are determined, an Input Data Form, to facilitate setting up the data tapes, is made up. Such a form includes spaces for writing each piece of decimal data and keywords, with each space identified. Also on the sheet, or accompanying it, there should be adequate but concise instructions for filling in the data, e.g., the meaning of the keyword(s), and how many decimal places are allowed for each item. If this data form is set up properly, the programmer need not be consulted as to its preparation once the program is coded and checked out.

An output tape, besides containing the desired decimal data, should also contain format information, arranged so that the tape can be printed on the off-line printers with a minimum amount of typewriter-setting; that is, the tape should contain appropriately placed titles, column headings, variable identification, as well as decimal point, space, tab, and carriage-return symbols.

d. Coding

Since one section (Section II) of this manual already treats the make-up of the GEORGE instructions, and the meanings of the various bits and combinations of bits of the instruction, the following discussion is intended only to provide some helpful coding hints, tricks, and check points that can be observed to conserve time, accuracy, and machine space, as well as work in coding and debugging.

One useful tool in coding is the storage chart. A storage chart is a sheet or set of sheets made up of identified squares, each square representing a memory cell and identified by the address of that cell. The name or symbol of each constant or variable stored in the memory during

the program is written in the square corresponding to the memory location on the storage chart. It is also convenient to indicate on the storage chart which sections of the memory are taken up with various subroutines and patches of code. Thus, one can see the various storage assignments at a glance.

In general, a good code is so arranged that a minimum amount of code and machine time is needed to obtain the desired results, with maximum accuracy maintained throughout the various calculations. Depending on the particular problem, it is sometimes advisable to sacrifice one of these factors (code, time, accuracy) to improve another; e.g., if accuracy is at a premium, extra machine time and code may be used to perform accuracy-saving double-precision arithmetic. If only a few figures are desired for accuracy as, say, for the bars of a histogram, machine time may be saved by performing half-precision arithmetic.

A few facts regarding multiplication and division should be kept in mind always. In division, the Qt acts as a continuation of the dividend, and in multiplication 2^{-39} times the initial value of Ac is added to the product. Thus care must be taken to clear Ac before a multiplication, or Qt before a division, unless the respective registers contain information significant to the operation. If the evaluation of some expression involves a series of multiplications and divisions whose order is irrelevant, the operations $\times$ and $\div$ can be alternated, to save accuracy and code. A no-round-off multiplication leaves the 78-bit product in Ac Qt, which could serve as the 2-register dividend for the following division. Likewise, Qt holds the quotient of a division operation and the multiplier of the next order, if it happens to be a multiplication order.

Round-off plays an important role in maintaining accuracy and should, except in special cases, be used in all operations involving truncation, i.e., multiplication, division, and shifting. The special cases arise when a two-register product is desired, the remainder after a division is to be used in later operations, when the operation is known in advance to be exact, or in various types of shifts.

Note: If a multiplication is to be followed by a division, the multiplication should be no-round-off, so that the two-register product obtained can be used as the dividend for the division. Naturally, in this case, Qt is not cleared before the division, since its contents are significant.

Two-register shifts are also convenient for decreasing truncation errors, especially when several left and right shifts must be done, with, perhaps, additions between, in evaluating some function.

Of course, no matter how carefully a code is prepared to preserve accuracy, the aim is defeated if the constants used are not stored accurately. Thus, a "double check" is suggested to be sure the constants stored with the code are:

1. accurately computed;
2. correctly converted to base 16, i.e., correct to 12 decimal places;
3. correctly copied into the code and onto the code tape.

Double-precision arithmetic, wherein all data is considered to be 80 bits long, can be used if more than normal accuracy is desired in all or parts of the computations of a program. A subroutine (D-2-110) is available for performing double-precision arithmetic. The write-up of the subroutine also contains an explanation of the double-precision method in case one wishes to code his own double-precision arithmetic instead of using the entire routine.

To conserve machine time for the running of a particular code, care should be taken in coding to seek that:

1. a minimum number of instructions - and memory cycles - is used to obtain the problem solution;
2. the instructions used to evaluate various functions are those requiring the least amount of over-all time.

One obvious way of saving memory cycles is through compact coding. Since each instruction to be obeyed by the machine must involve at least a fetch cycle, it follows that fewer instructions means fewer consultations of the memory and therefore less computing time.

The "preliminary clears" (Pcl) on GEORGE also help to eliminate many extra add orders; e.g., a Pcl to 2^{-11} or 2^{-39} automatically (without use of the memory) puts 2^{-11} or 2^{-39} into one of the registers, readying it for counter-increase operation or the like, and cutting out the need for a memory cycle to bring out 2^{-11} or 2^{-39} . Numerous result cycles can be eliminated if one can arrange to hold intermediate results in either Ac or Qt instead of storing them after each operation. This is especially true in certain types of loops where a sum can be accumulated in Ac or Qt instead of in the memory, or in computation of functions in which operations can be so ordered as to have the result of one operation be the argument for the next, e.g., alternating divisions and multiplications.

Furthermore, by holding sums in Ac or Qt instead of in the memory, these intermediate results can be allowed to overflow the normal interval of $[-1, 1 - 2^{-39}]$ without destruction of the accuracy of the final sum, provided:

1. the final result is known in advance to be an in-range number;
2. none of the "overflowed" sums are ever stored as such in the memory.

The following example illustrates this feature.

Consider obtaining $a + b + c = d$, where

$$a = 1/2 = 00.100 \dots 0 \quad (= a_x a_0 \cdot a_1 \dots a_{39})$$

$$b = 3/4 = 00.110 \dots 0$$

$$c = -1 = 11.000 \dots 0$$

$$\text{Take } a + b: 00.19 \dots 0$$

$$\begin{array}{r} +00.110 \dots 0 \\ \hline 01.010 \dots 0 \end{array}$$

This causes overflow, but if we consider

$$c(Ac) = (-2a_x + a_0 + 2^{-1} a_1 + \dots + 2^{-39} a_{39}) \bmod 2, \text{ we have}$$

$$a + b = 01.010 \dots = 1 + 2^{-2} = 5/4 = 1/2 + 3/4$$

Add c to this sum, and we have:

$$\begin{array}{r} 01.010 \dots 0 \\ +11.000 \dots 0 \\ \hline 00.0100 \dots 0 = d = 1/4 \end{array}$$

Thus $d (= 1/4)$ is a good machine number and is correct upon computing $a + b + c = d$, in spite of the overflow which occurs in the first step. Since the overflow does occur, and is expected, an "ignore Overflow" ($F_{20} = 1$) should be included in the first add instruction, to avoid an unnecessary overflow stop.

Since each multiplication and division involves 39 shifts, and 39 adds (or subtracts) as well as at least one memory consultation, π orders are quite time-consuming. Thus, where multiplication by $2^{\pm p}$, $p < 40$, is involved, the operation is more efficiently accomplished by a left or right shift, than by a π order.

In problems where only a few figures of accuracy are desired, or where it is desirable to sacrifice a few places of accuracy to decrease computing time, half-precision π orders can be used (see " π Orders," Section II.C.1)..

The running times of programs which require very much input and/or output can be greatly reduced by avoiding meaningless "extra decimal figures" and too much identification and format information on an output tape.

Loops and subroutines, as described previously, are two important code-saving devices for any program, because each loop or subroutine makes it possible to do calculations on several different parameters using the same block (blocks) of code. Similarly, it is often possible to arrange to use blocks of code in a program several times by proper selection of parameter-storage locations and arrangement of variable connectors.

For example, suppose, at different points in a program, the following functions are to be computed:

1. $f = [x(\Delta t) + \beta y]$
2. $g = [s(\Delta t) + \beta z]$
3. $h = [s(\Delta t) + \beta z - r]$.

If it can be arranged that, when g and h are being computed, s and z occupy the locations which x and y occupied, respectively, when f was being computed, then a common code can be set up to compute $[t_1(\Delta t) + \beta t_2] = t_3$, where $t_1 = x$, $t_2 = y$ for computation of f , and $t_1 = s$, $t_2 = z$ for computation of g and h . In addition, the code for computation of t_3 must be arranged so that it has a variable exit, or jumps to a variable address after completion. Before the code is entered each time, the address of the continuation of the code is placed into the exit word for the t_3 code. Thus, after computation of f , the code might jump to a word which stores t_3 as f and continues; after computation of g , it jumps to code which stores the result as g , and then goes on; after obtaining t_3 in the computation of h , the code would jump to that code which computes and stores $(t_3 - r) = h$. Section III.A.4.a, on "Flow Charts" has a further description of variable connectors.

The arrangements of the GEORGE instruction word, and, in particular, of the available "extra duties" of the tag simplify the task of writing concise codes. In coding any problem, one should strive to make full use of the tag.

B-boxing, while most commonly used as described under "Loops," also can simplify some other operations. For example, if the number of places of several quantities that are to be shifted agree with

each other, but vary from run to run, or iteration to iteration, one would normally arrange the code to change the amount of shift each time by changing the A addresses (shift addresses) of the shift orders before each new run or iteration. However, if this variable - the amount of shift - is stored in the B-portion of some memory cell instead, say 111, and the shift instructions coded with $F_{26} = 1$, the B-address = 111 (location of the amount of shift), and the A addresses = 000, the A addresses can remain unchanged; only the contents of 111 would need to be changed whenever the amount of shift is changed.

A variation of this technique is especially useful when several shift operations involve shifts which differ from a variable p by constants; suppose one instruction involves a shift of $(p + 1)$ places, another of $(p - 2)$ places, and another of p places. Then p can be stored, as an integer $\times 2^{-11}$, in some free B-address. B-boxing by $M(p)$ can be requested in the shift instructions, with the A-address = (000) when 2^p (or 2^{-p}) is desired, (001) to get $2^{(p+1)}$ (or $2^{-(p+1)}$), or (FFE) to obtain $2^{(p-2)}$ (or $2^{-(p-2)}$), and so on.

B-boxing can often be used conveniently with Jump orders as well.

Most of the uses of preliminary adds ($P+$) become apparent during coding of a particular program. One of the less obvious uses of $P+$ is in the preparation of one register (Ac or Qt) for a future instruction while testing or operating on the other. Another handy use is in testing Ac or Qt for zero. To perform the test, we can add -2^{-39} (via $P+$) to the register, and test for carry overflow. If the register is zero initially, no carry will be produced by the $P+$. If the register is non-zero, a carry will occur and the cOF toggle will be set.

Another code or storage-saving device is the use of irrelevant B-addresses as storage for counters, addresses, or other variables representable in 12 bits. A $Pc1$ to 2^{-11} allows a convenient way of increasing the B portion of any word. Having the sign bit as the first bit of the B address makes sign testing on counters simple, so a counter stored in the B address may be operated upon without disturbing the rest of the instruction.

Unless there is a definite shortage of storage space in a program, it is wise, when the computations for a problem can be grouped into sections, to reserve a separate memory block for each section, leaving sufficient space after each block for corrections. When errors are found while debugging and extra words of code are needed for the corrections, these words can be added without causing an overlap of sections, thus preventing over-complication of the code. Subroutines are most conveniently stored in a block - preferably before or after the main code - not mingled with the code, since the subroutines do not require debugging. Similarly, it is wise to store constants together and to arrange that other parameters used during the running of a program be stored in a separate section of the memory - sufficiently away from the code to prevent having to code around these variables.

For a smoothly running program, as many steps as possible needed for execution of a program on the computer should be coded into the program, so that the chances of human error are reduced to a minimum. The manual part of loading a code into the computer can be reduced to putting a tape into the reader and depressing three console buttons - LOAD PAPER, CONTINUOUS, GO. This can be accomplished by letting the code tape "bootstrap" itself, i.e., by having the tape punched as follows:

1. Each separate section of code (including subroutines) appears on the code tape as one block, followed by an E_2 . The blocks can be separated by gaps of zeros, for easy identification.
2. The last section of code on the tape is the Command Routine.
3. A loader routine followed by an E_2 appears as the first section of code on the tape. This loader code, coded to start at 001, contains a series of load orders (DO), one corresponding to each section of code appearing on the tape, and arranged in the same order that the code sections are arranged on the tape.

Thus, depressing LOAD PAPER, CONTINUOUS, GO with this tape in the reader will cause the computer to read and jump to this load code, then read each section into its proper location. If the last instruction is a "Read to 001" order, with a Jump to B in the tag, B being the address of the first word of code, this instruction will load the Command Routine (covering this loader routine) and jump to the start of the main code.

5. Checking a Machine Code (Debugging)

A program cannot be considered completed until it has been checked out, or "debugged," on the machine, to determine if it actually does the job it is designed for. Two major steps are involved in this checkout procedure. The code must be machine checked to make sure it accurately represents the mathematical statement of the problem, and the mathematical statement must be checked to be sure it accurately describes the physical experiment the program is to simulate or the calculations it is to complete.

The latter check, which is done only after the code has been completely debugged, involves running one or several sets of data for which the results are known or can be approximated in advance. These results should not be obtained through use of the formulas in the program, as this clearly would not check the mathematics. Rather, actual experimental

results, or results of computations by other mathematical methods, if possible, should be used to adequately check the accuracy of the mathematics of the program.

Because of the variety of errors that can occur between the mathematical statement and the code, the code-check process is more involved than the check of mathematics. Unless planned wisely, the time spent on this part of the check-out procedure may be long, tedious and wasteful. Thus several prescriptive warnings and suggestions are appropriate to facilitate the debugging process.

In order to fully ascertain the accuracy of the code, a test case, computed via desk calculator, can be used to compare with machine results of the program. The best data for use in such a test case are those data chosen to resemble the actual data that will be used in the running of the program.

However, in most long programs, it is advisable to check out individual sections of the code before attempting a complete test case. In this way all possible paths can be checked, errors can be more easily located, and the complete test case can then be used to check the general flow of information through the program. The remarks that follow provide hints on checking the program in its various phases.

In all cases, machine results should be checked against the formulas in the original mathematical statement rather than scaled or reworked equation. In this way errors made in scaling, or in rewriting equations, as well as those made in coding, can be detected.

In choosing data for test cases, it is best to select situations most likely to point out errors. In particular, care should be taken to avoid test values which cover up errors; e.g., if a function $f = (1 - a)e^{x^2/y}$ is computed, choosing $a = 1$ will not check the computation of $e^{x^2/y}$, since it will cause f to be zero, and perhaps allow a major error in the computation of $e^{x^2/y}$ to go undetected.

It is important to plan debugging to include a check on all possible branches of the code. This usually involves introducing "freak" or unusual data in some sections of the program to cause various function limits to be reached, etc.

Activity on the computer is most efficient when well-planned before hand. This means coming to the machine fully equipped with correctly punched tapes, adequate operating instructions for the code and all diagnostic routines to be used, and a good copy of the current program - equations, flow charts, code, etc. "Playing by ear" on the computer can be very wasteful of machine time.

Normally, code tapes for various tests used in debugging can be consolidated into one tape, with housekeeping; e.g., a loader routine, similar to that described under "Coding" (Section III.A.4.d) can precede a series of test codes on a tape.

Also included on this tape, there can be a correction code - i.e., a code which makes changes in the original code, then jumps to the main program. If control is transferred to this correction code after the main program is loaded, necessary code corrections can be made automatically before each new test is tried. Also, then, only the correction tape needs to be changed as new errors are discovered in the code, and the main code tape need only be corrected after the program is completely debugged.

This tape consolidation is but one step of many that can be taken to "let GEORGE do it," i.e., to avoid time-consuming manual operation as much as possible.

When in trouble, the best procedure is to get off the machine, but only after adequate memory dumps, decimal and sexadecimal, are secured, and addresses of trouble spots (overflow, unending loops, etc.) noted. The cause for the trouble can then be ascertained - at leisure - by analyzing the information secured through the dumps and other output. A memory dump is merely the printout of the contents of a specified section of the memory. Several dump routines, which print the memory contents in either decimal or sexadecimal with a minimum amount of machine manipulation and complete with formatting and identification on the dump output tapes, are available.

When it is suspected that an error lies within a small section of code, or if one desires to see exactly what happens during that section, use of the Program Trace (X-13-223) is recommended. The particulars of how to use it are included in the write-up for the routine. In general, a trace routine, as the name implies, follows through a specified section of code, punching for each instruction of the main program obeyed a line of information containing the following:

The address of the current instruction;

Contents of the various toggles after the instruction is executed;

The instruction itself;

Contents of Ac and Qt after the instruction's completion;

Contents of the words at the B and A addresses before the execution of the instruction.

Obviously, the trace with all its punch-out is very time-consuming and as such is not recommended on long sections of code. But it is often extremely helpful in finding hidden errors in short patches of code, particularly in sections which use many coding "tricks."

Many other good diagnostic routines are available for use in debugging. By investigating them before code-check is begun, several may be incorporated into the debugging process to make good use of check-out time.

Several features of the GEORGE Console were designed with code checking in mind. If memory space is not at a premium, a Breakpoint Stop can be coded at the end of each section of code, after some group of formulas is evaluated, or at some logical separation point in a code. The B address of the Breakpoint can be the address of some print-out routine. Then, during check-out time, if the Breakpoint(s) are set to JUMP, the routine can jump to a specially coded routine which prints out current pertinent information and returns to the code. If the BP is set on STOP, the program or the data in the computer can, after the STOP, be altered to prepare for the next test, dumps can be taken, or jumps to other sections of the program can be performed. Later, when the program is completely checked-out and production is run, the Breakpoints can be left off, and those Breakpoint instructions passed over as stalls by the main program.

When, for lack of memory space or for other reasons, coded Breakpoints are not convenient, address recognition switches may be used in debugging. In addition to making it possible to stop a code at any instruction without pre-coding this stop, AR switches also facilitate tracking down several types of coding errors - particularly address errors. For example, if the contents of some memory cell are being changed erroneously, one may set that cell's address, and To Memory on the AR switches, causing the machine to stop everytime something is put into that cell, and thus catching the code "in the act" of storing at the wrong address. Similarly, the switches can be used to find out exactly when a certain cell's contents are referred to, in computations or as an instruction. These switches and others are explained further under "Console Operations." (Section I.E).

While most of the "answer checking" is done best away from the computer, it is often quite advantageous to know immediately the decimal values of certain parameters or functions at a given point in the program, so as to know how to proceed. For example, if the answers at that point are correct and machine time is still available, one may wish to go on to testing another section of code; if the answers prove to be wrong, added tests or a trace on the same section of code may be run to determine the cause of the error or at least to isolate the trouble.

This "immediate printout" can be achieved by making use of the console typewriter, the Output Subroutine used in the program, and a short "tape to type" converter code which allows the Output Routine to type instead of punch results.

One such converter code may be set up to act as follows:

1. It may be entered, via the Command Routine, by bringing x, the number to be converted and typed, into Qt and jumping to the first word of its code.
2. Once control is transferred to this code, the code would temporarily store the contents of Qt (= x), then proceed to change all Punch orders within the given Output Subroutine to Type orders.
3. When all the Punch orders are adjusted, the routine would bring x into Qt again, and jump to the Output Subroutine.
4. After the desired x is typed in decimal, control is transferred back to the converter code which changes all the Type instructions in the Output Subroutine back to Punch instructions, and then jumps to the Command Routine (with the Output Routine back in its original state).
5. Via the Command Routine, another word can be outputted, control returned to the main code, a jump to a new section of code can be executed, or tracing begun, depending on the accuracy of the printed result.

Such a converter code may be modified, expanded, or shortened as desired for a particular program and/or Output Routine; e.g., provisions for outputting a block of words at one time may be made (by either setting up a loop in the above converter or by using a converter with a Decimal Dump Routine). If a keyword is required by the Output Routine, provision for storing this keyword before entering the converter can be made. Actually, the manipulating with orders that is necessary takes a negligible amount of machine time. Most of the time used is spent typing the results. Thus, typing long blocks of data in this manner is not recommended, as it is time-consuming; but looking at a small number of results in this way may prove quite beneficial and time-saving, as it may help the programmer to decide which of several tests to try next, and perhaps thereby eliminate unnecessary tests.

In spite of all the tricks, suggestions, and techniques for debugging described, the fact still remains that the best (and most) check-out work is done at one's desk - with the aid of the computer, but away from it;

that is, debugging time on the machine can be kept to a minimum if adequate planning of test cases is done beforehand, the machine time used intelligently, taking full advantage of diagnostic and dump routines and console facilities to obtain maximum usable information about a code, and this information analyzed carefully and thoughtfully.

6. Operating Instructions

After a program is debugged, instructions for the use of the program should be provided, so that the programmer need not be actively involved in running the program.

Three sets of instructions are needed, one set for each phase of processing the data through the computer and obtaining a printed copy of the results.

a. Input data preparation:

Included in this set are a copy of the data form with an explanation of how to complete it, and instructions for preparing (punching) the input tapes, with locations of decimal points, end symbols, etc., specified.

b. Computer operating instructions:

These should be as short and concise as possible, and include:

- (1) A list of the breakpoint settings required.
- (2) A step-by-step description of how, when and via which reader the various tapes are to be loaded.
- (3) A list of all possible stops (particularly error stops) with instructions on what must be done when a stop occurs.
- (4) Instructions on how to restart the program or how to start a new case, should trouble arise.
- (5) Details on how to stop the program without losing the computation results up to that time, if the program is lengthy, time-wise, and must be stopped.

c. Output data print-up:

This set should describe how machine results are to be printed, specifying the number and width of columns desired, type of format information existing on the output tape, the width of paper desired, stop codes, if any, to be used, and any other information vital to producing an attractive, neat, and correct output print-up.

7. Program Write-up

A clear, descriptive write-up of the program is the last step in the preparation of a problem for machine computation. Since such a write-up is intended chiefly as an aid for those who wish to use the program in the future without the aid of the original programmer, it should be as complete and up-to-date as possible, containing all that needs to be known about the existing program if changes, additions, etc., are desired. An accurate description of the following items will insure a good write-up.

- a. Statement of the problem, including its purpose, formulas used, restrictions, and options available.
- b. Copies of all the operating instructions.
- c. Description of input and output, including input data sheets with their instructions, enumeration of the output parameters, and an explanation or illustration of the printed format.
- d. Ranges of all parameters and functions used.
- e. Scale assignment and scaled equations as used in the actual computation.
- f. Intelligible and up to date copies of the flow charts and code used.

B. GEORGE Assembly Routine (GAR I)

Herbert L. Gray

GEORGE instructions to be assembled by this program are written with references expressed as arithmetic combinations of symbols and decimal and/or sexadecimal integers. A format is used in which the parts of the instruction are given in the order: symbolic location, B address, order and tag, and A address. In addition to instructions, data in decimal or sexadecimal form may be assembled, and library routines may be conveniently incorporated into the program being assembled.

In order to describe the use of this assembly program let us consider first a simplified explanation of symbolic assembly operation.

The procedure is divided into two parts. The first examines the program to be assembled in order to define each symbol used. The second part prepares the actual machine-language program, punches it in sexadecimal form on tape, and produces a printed copy of the program in symbolic form with the corresponding sexadecimal machine language program.

During the first part, a counter, L, is used to specify the absolute location of each word in the program. L is set initially to an integer supplied to the assembly program by the program being assembled. Thereafter, L is increased by one for each word used in the program.

Simultaneously with this counting procedure, a table is constructed. Each entry in this table defines a symbol used in the program as being equivalent to some integer which is usually an address. Entries to the table are made in two ways:

1. A symbolic address appears at the "symbolic location" of a statement in the program being assembled and is assigned the value of L.
2. A symbol is defined by a pseudo-operation.

It is important to note that the order of the absolute instructions produced by symbolic assembly corresponds exactly to the order in which the symbolic instructions are read by the assembly program.

During the second part of the assembly process, all symbols in the symbolic program are replaced by their integer equivalents given in the table formed during the first part, thus producing an absolute program. Note that this operation requires that each symbol be uniquely defined.

For use in the assembly program the following definitions are made:

Characters

- Type A: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, () · / Space = $\int^*$. All of these are printing characters. UC (upper case), LC (lower case) and IG (ignore) are nonprinting characters.
- Type S: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. These are sexadecimal characters.
- Type D: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. These are decimal characters.
- Type Z: , + - / TB (tab) CR (carriage return). These are used only as separators or operators.

All other characters are illegal. These include #, Δ , Σ , ϵ , |, back-space, upper case minus, black ribbon, red ribbon, and all zone 3 characters except UC, LC, IG, TB, and CR.

Symbols

Non-integer: One to six characters of type D and/or A.
At least one character must be of type A.
The last character of a non-integer must not be /.

Decimal Integer: One to six characters of type D. This integer is converted to sexadecimal mod 4096.

Sexadecimal Integer: One to five characters of type S terminated by a /. This integer is used mod 4096.

Symbolic Address: Any number of symbols separated by the operators + or -; e.g., X + B1 - 5, -AB + Z + 1 .

If in the symbolic location two or more expressions are separated by commas, each expression will be considered a symbolic address and will be operated on as if it were the only one. Each expression will be considered in the order in which it appears from left to right. As an example, let L = 211. If symbolic location contains 176, ABC; L and ABC are set equal to 176. (See "Location Counter.") If symbolic location contains ABC, 176; ABC is set equal to 211; L is set equal to 176.

The order and tag part of each instruction is specified by its hexadecimal equivalent.

The symbolic location, the B address, the order and tag, and the A address parts of a symbolic instruction are given in that order and are separated by tabs. In some cases certain parts are not necessary and only a tab need be punched. Those parts of the symbolic instruction which are not given will be assumed to be zero.

The following tape form is used by the assembly for a GAR statement:

```

symbolic location
tab
B address
tab
order and tag
tab
A address
tab
remark

```

The first tab to the right of the A address defines the end of the instruction. All punching to the right of such a tab is considered to be a remark and has no effect on the assembly process. A carriage return defines the end of the statement and may replace any of the tabs.

Arithmetic expressions

Arithmetic expressions in terms of symbols and integers may be used to define addresses of GEORGE instructions.

The following elementary operations may be used:

```

addition, indicated by +
subtraction, indicated by - .

```

No parenthetical expressions may be written.

Integral arithmetic modulo 2^{12} is used for addresses.

PSEUDO-OPERATIONS

Origin specification: ORG

The location counter L is set to the value of the expression appearing in the A address. Each symbol appearing in the expression must have been previously defined. If the B address contains a symbol,

then that symbol is used as the name for the code which follows the origin specification. This allows the code following the origin statement to be relocated independently of the main program. When creating a library subroutine, the symbol in the B address would be the chronological number of the subroutine. (See the description of the LIB instruction for the uses made of this number.)

If no origin specification is given for a program, the initial value of L shall be 100_{10} (064_{16}).

Origin specification instruction may be used at will.

Equals: EQU

The symbolic address(es) appearing in the symbol location are assigned the integer value given by the expression appearing in the A address. Each symbol used in this expression must have been previously defined.

Note that the pseudo-operation EQU is to be used only in those cases where the expression appearing in the A address specifies a preset program parameter such as the order of a matrix, the degree of a polynomial, the number of items in a group, or any other quantity which is invariant with respect to the location of the program in storage. If the expression specifies the location of a piece of data or an instruction, the pseudo-operation SYN should be used.

Synonym: SYN

The symbolic address(es) appearing in the symbolic location are assigned the integer value given by the expression appearing in the A address. Each symbol used in this expression must have been previously defined.

Note that the pseudo-operation SYN is to be used only in those cases where the expression appearing in the A address specifies the location of a piece of data, the location of an instruction, or any other quantity whose value depends upon the location of the program in storage. If the expression specifies a preset program parameter, the pseudo-operation EQU should be used.

Decimal data: DEC

The decimal number in the A address is converted to binary and assigned to location L. The first tab to the right of the number indicates that all punching to the right of this tab is a remark.

Signs are indicated by space, +, or - preceding the number, the exponent, or the binary scale factor.

The symbolic location identifies the decimal number on the tape. If the number N does not have a decimal exponent associated with it and neither of the characters . or B appears in the decimal number, the number is converted into a binary integer with the binary point at the right-hand end of the word (i.e., the integer N_{10} is converted to $N_2 \cdot 2^{-39}$).

If an exponent, or the character ., or both, appear in the decimal number and the character B does not appear, the number is converted to a GEORGE-type floating binary quantity. The decimal exponent used in this conversion is the number which follows immediately after the sign separating the digits. If an exponent does not appear it is assumed to be zero. If the decimal point does not appear, it is assumed to be at the right-hand end. For example, (space) 12.345, +12.345, 1.2345 1, +1.2345+1, +1234.5-2, and +12345-3 are all equivalent representations of the same floating-point number.

If the character B appears in a decimal number, the number is converted into a fixed-point binary fraction. The binary scale factor used in this conversion is the signed number which follows immediately after the character B and is positive for downscaling, negative for upscaling. If the decimal point does not appear in the decimal number, it is assumed to be at the right-hand end. If the decimal exponent does not appear, it is assumed to be 0. The order of the binary scale factor and the decimal exponent is not significant. For example, 12.345B+4, +1.2345+1B+4, and 12345B+4-3 are all equivalent representations of the same fixed-point quantity, $12.345/16$, while .12345B-2 and 12.345B-2-2 each represent $4 \times .12345$.

Sexadecimal data: SEXD

The sexadecimal number at the A address is changed to binary integer form in which the binary point is considered to be at the right-hand end of the GEORGE word, and assigned to storage location L.

The first tab to the right of the number indicates that all punching to the right is to be considered a remark.

The symbolic location identifies the sexadecimal word on the tape.

Alpha-numeric data: BCD

The five alpha-numeric characters in the A address are inserted into the program as one word of sexadecimal information.

The symbol appearing in the symbolic location identifies the BCD word.

Remarks: REM

Any punching in the remarks column (after the fourth tab) will be reproduced in the printed listing of the assembly, without otherwise affecting the assembly in any way.

End of program: OUT

This pseudo-operation must terminate each section of code including the last. A breakpoint F instruction with jump to B is punched. The B address of the instruction is set equal to the value of the expression in the B address of the pseudo-operation. Thus, whenever the code is read, this instruction will cause a jump to the beginning of that section. Location counter L is set back to 100_{10} .

End of punching: END 2

This pseudo-operation causes an End 2 to be punched at the end of the previous instruction instead of the normal End 1. All punching is stopped until a new instruction is reached. However, the location counter L continues to count. This pseudo-operation saves the punching of Storages whose contents are irrelevant.

Block started by symbol: SAVE

The block of storage extending from L to $L + N - 1$, where N is the value of the expression in the A address, is reserved by this pseudo-operation. Each symbol in the expression for N must have been previously defined.

If a symbol is punched in the symbolic location, it is assigned the value L, corresponding to the first word of the block reserved. Finally L is replaced by $L + N$.

Repeat: REP

Two expressions, the first in the B address and the second in the A address, define two integers M and N. The block of instructions and/or data preceding the REP operation in locations $L - M, L - M + 1, \dots, L - 1$ is repeated N more times, the repeated information being assigned to locations $L, L + 1, \dots, L + M, \dots, L + M \cdot N - 1$.

Library search: LIB

The library routine identified by its chronological number in the B address is obtained from a library tape, relocated, and inserted in the program being assembled. If the library routine requires k words of storage, it will occupy locations $L, L + 1, \dots, L + k - 1$, where L is set equal to the value of the expression in the A address. The expression in the symbolic location is entered in the table of symbols and assigned the value in the A address. There are no symbols on the library tape. The assembly program always keeps track of the position of the library tape and makes use of the information in the symbolic code in order to search the library in the most efficient manner. The setting of L equal to the value of the expression in the A address is equivalent to giving an ORG instruction.

Subroutine: SUB

The subroutine, in standard binary form headed by its code words, following this order will be inserted in the program being assembled. If the subroutine requires k words of storage, it will occupy locations $L, L + 1, \dots, L + k - 1$, where L is set to the value of the expression in the A address. The expression in the symbolic location is entered in the table of symbols and assigned the value in the A address. If a symbol appears in the B address, the subroutine is assigned this name or number. The assembly routine will then punch out the subroutine in such a form that it may be relocated independently of the main program.

End assembly: FIN

This must be the last statement read by the assembly routine.

Location counter

If an absolute decimal or sexadecimal number is punched in the symbolic location of any statement in the assembly, the location counter L will be set to that value. The effect of absolute location punching in the symbolic location is therefore identically the same as if the statement in question were to be placed immediately after an ORG statement having the same absolute location punched in its A address.

Operational features

As an aid to the programmer, this assembly program gives some indications of erroneously prepared programs. A print-out on the console typewriter of the following form is given for each error found:

EWX.XXXX Y WWWWWW

EWX = error number (See below)

XXXX = line number of program

Y = position within instruction

Y = L: symbolic location

Y = B: B add

Y = O: Order tag

Y = A: A add

WWWWW = quantity in error

Possible error indications and action assembly routine will take:

- E01 Too many characters in symbol. Only the last 6 characters are used.
- E02 More than one undefined symbol in expression at symbolic location. Values of all undefined symbols except the last are assumed to be zero. See "Method Used to Calculate the Value of the Symbolic Location."
- E03 Sexadecimal integer contains a non-sexadecimal character. It is assumed to be a non-integer symbol.
- E04 Symbol contains an illegal character. Illegal character is accepted by GAR, but, IBM-704 off-line printer will not accept the character.
- E05 Memory capacity exceeded (i.e., code is stored in 000). Only a warning.
- E06 Symbol defined twice. Contents of L (location counter) are set to the value of the symbol. See "Method Used to Calculate the Value of the Symbolic Location."
- E07 Non-existent order or tag. Order and tag are set to zero.
- E08 Address contains an undefined symbol. Value of symbol is assumed to be zero.
- E09 More than one symbolic address A or B. Uses only last one.
- E10 Wrong type of character in decimal or sexadecimal number. Assimilates it.
- E11 Expression in symbolic location of EQU or SYN was previously defined. Assigns to expression the new value.
- E15 Number of subroutines is greater than 14. Only the first 14 are retained.

- E16 Subroutine is not on reel. Length of subroutine is assumed to be zero.
- E17 Symbol table limit (2000) exceeded. Program should be modified.

This list of errors is printed, and, in addition, all symbols and their values (symbol table) are punched on tape prior to performing the second part of the assembly. Thus, if any errors have been indicated, the problem may be removed before the second pass is made.

Reassembly features

Additions to a program which has been assembled are easily accomplished if the table of symbols which was punched during the initial assembly process has been saved. It is then necessary only to reload this table (See "Operating Instruction") and to assemble the new parts of the program. The original program need not be reloaded.

Furthermore, any change in the original program which does not involve relocation of any part of the program or any reassignment of symbols may be made by performing only the final half of the assembly.

Other Convenient features

Punching of the entire symbolic program may be suppressed.
Punching of the subroutines copied from the library may be suppressed.

Sexadecimal punching is available in either non-relocatable or relocatable form. When punched as relocatable, not only may the entire program be relocated but the subroutines within a program may also be relocated, independently of the program.

If a statement is preceded by a + and a breakpoint (See "GAR Operating Instructions") is set to Jump, the statement will be ignored.

Method Used to Calculate the Value of the Symbolic Location

$$\begin{aligned}\text{Symbolic location} &= \text{Symbol}_1 \pm \text{Symbol}_2 \pm \text{Symbol}_3 \pm \dots \pm \text{Symbol}_i \pm \dots \\ &= L \text{ (Location Counter)}\end{aligned}$$

- a). If Symbol_i and only Symbol_i is undefined, then $\text{Symbol}_i = L - (\text{Symbol}_1 \pm \text{Symbol}_2 \pm \dots)$
- b). If $\text{Symbols}_{i,j,k}$ are all undefined and $i > j > k$, then $\text{Symbols}_{j,k}$ are set equal to zero and Symbol_i is calculated as above.
- c). If none of the symbols are undefined but the value of the expression does not equal L , then an error indication (E06) is printed, and L is set equal to the value of the expression.

Preassigned Symbols

Certain unique symbols have already had addresses assigned to them. They are as follows:

SYMBOL	ABSOLUTE ADDRESS	QUANTITY AT ADDRESS
ZERO	$010_s = 16_d$	zero
ONE	$011_s = 17_d$	$1 - 2^{-39}$
NEGONE or NBO	$012_s = 18_d$	$-1, -2^0$
B1 or $1/2$	$013_s = 19_d$	$2^{-1}, 1/2$
NB1 or $N1/2$	$014_s = 20_d$	$-2^{-1}, -1/2$
B11	$015_s = 21_d$	2^{-11}
NB11	$016_s = 22_d$	-2^{-11}
B39	$017_s = 23_d$	2^{-39}
NB39	$018_s = 24_d$	-2^{-39}
LN 2	$019_s = 25_d$	$\log_e 2$
$\pi/4$	$01A_s = 26_d$	$\pi/4$
SQRTB1	$01B_s = 27_d$	square root of $1/2$
SQR1/2		
N40	$01C_s = 28_d$	-40×2^{-39}
B32	$01D_s = 29_d$	2^{-32}
B18	$01E_s = 30_d$	2^{-18}
B8	$01F_s = 31_d$	2^{-8}
FP1 or FPONE	$020_s = 32_d$	floating point one
/	contents of location counter L.	
$\overline{UC} * \overline{LC}$	contents of location counter L.	
7FB39	$023_s = 35_d$	$7F_s \times 2^{-39}$
COMMAND	$001_s = 1_d$	entrance to Command, routine
Mu	$045_s = 69_d$	mu of the sub. error stop code

SYMBOL	ABSOLUTE ADDRESS	QUANTITY AT ADDRESS
SES	$040_s = 64_d$	entrance to sub. error stop code.
TO	$FFO_s = 4080_d$	temporary t_0
T1	$FF1_s = 4081_d$	temporary t_1
.		
.		
.		
.		
TF	$FFF_s = 4095_d$	temporary t_f

Special Restrictions

- 1) No more than fourteen subroutines may be used in any one program.
- 2) Any sign before a previously undefined symbol in the symbolic location is assumed to be positive; e.g., (10-A) is treated as $10 + A$ if A is undefined.
- 3) Any symbolic address that has a value less than zero can not be relocated.

GAR Operating Instructions

- 1) GAR in reader 1
- 2) Code in reader 2
- 3) "Load Paper," "Cont.," "Go."
- 4) At Stop EEE insert Code in reader 2 again
- 5) "Ready," "Cont.," "Go."
- 6) At Stop FFF assembly finished.

STOPS

CCC Insert additions to program in reader 2.

DDD Limit of symbol table (2000) exceeded. Get off machine.

EEE First pass of assembly completed. See 4) above.

FFF Assembly finished.

BREAKPOINTS

- B.P.1 Jump Mode Library subroutines are not punched.
- B.P.2 Jump Mode Symbolic program is not punched.
- B.P.3 Jump Mode Routine is punched in relocatable form. Otherwise it is punched in absolute form.
- B.P.4 Jump Mode OUT instruction is not punched. Used to create a subroutine.
- B.P.5 Jump Mode At step 2) above insert symbol table into reader instead of code tape.
- B.P.6 Jump Mode Statements preceded by a + sign are ignored.

IV. AUXILIARY EQUIPMENT

A. Off-Line Equipment

1. Paper Tape-Handling Equipment

Except for the console typewriter and keyboard, all input and output to GEORGE at present is in the form of seven-hole paper tape. To handle this tape, several off-line (i.e., not connected to computer) units are available. They are used for tape punching, reproducing, verifying, comparing, and printing. Each unit contains one or more of the following sub-units (all mechanical):

- a. Seven-hole paper-tape punch.
- b. Seven-hole paper-tape reader.
- c. Sexadecimal keyboard for punching tape with the characters 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, space, -, ., End 1, End 2, delete.
- d. Full alphabetic keyboard for punching all available characters (see table on a following page).
- e. Electric typewriter for printing tapes.

A unit may have all or some of the following control switches:

START READER 1 (or just START)

START READER 2

STOP: Stops both readers.

PUNCH ON-OFF: No tape punching can occur with punch OFF.

BP-STOP CONTROL: With this switch set on STOP, readers will stop whenever a STOP character appears on the tape being processed. With this switch at BP, readers will stop whenever a STOP or a Breakpoint character appears on the tape being processed.

READER TRANSFER: When ON, this switch causes a transfer at each End 1 or End 2 tape character from one reader to the other. Thus data on one tape can have the format controlled (e.g., carriage returns and tabs) by another tape. Also, two tapes can be combined, and printed or punched as one. If READER TRANSFER is not used, the tape must contain its own typewriter control characters. Control tapes for printing 1, 2, 3, 4, columns per page are already available.

$\left(\begin{smallmatrix} \text{READER A} \\ \text{ONLY} \end{smallmatrix}\right) - \left(\begin{smallmatrix} \text{FORMAT} \end{smallmatrix}\right) - \left(\begin{smallmatrix} \text{READER B} \\ \text{ONLY} \end{smallmatrix}\right)$: Three-position switch. The position FORMAT is equivalent to READER TRANSFER above.

TAPE FEED: Feeds Tape (without parity) out of the punch.

6-LEVEL CONTROL: When ON, characters with a zone 3 (non-printing characters) are changed to characters with zone 1 (printing characters) as they are read. Thus every character on the tape will be printed. This nullifies the action of the READER TRANSFER switch.

NORMAL-COMP-VERIFY: When this switch is set on NORMAL, tapes may be printed, punched, or reproduced. At COMP position, tapes in readers 1 and 2 are compared. Any difference in the two will cause the unit to stop and an error light to be lit. The tapes may also be repunched or printed at this time. At VERIFY position, the tape in the reader is compared against what is being typed on the keyboard. Disagreement will stop the unit before punching the character and turn on an error light.

NORMAL-VERIFY: Same as above, but only one reader is used. Therefore, tape comparison cannot be done.

RESET: When the error light is lit during tape verification, the key depressed is locked in the down position and no character is punched. Depressing RESET releases the key and readies the machine for a restart.

OVERRIDE: When the error light is on during tape verification, depressing the OVERRIDE switch will cause the character locked in the keyboard to be punched.

CODE CHANGE: In order to obtain certain characters when using an alphabetic keyboard, CODE CHANGE must be depressed. The upper case of the dark blue keys indicates to which characters this applies.

ADVANCE: Advances the tape in the reader one character.

The following units are available:

Two printing units with two readers and a typewriter each.

One printing unit with one reader and a typewriter.

Three punching, verifying, reproducing, units with one reader, a sexadecimal keyboard, and a punch each.

Two punching, verifying, comparing, and reproducing units with two readers, a sexadecimal or alphabetic keyboard, a punch, and a typewriter each, to obtain a printed record of what is typed from the keyboard.

2. Data Plotter

Graphs may be plotted from GEORGE paper tape or IBM cards on the DATAPLOTTER, built by Electronic Associates Inc. Instructions for operating the plotter and for preparing the data follow.

a. Operating Instructions

(1) Place suitable graph paper on table of plotter. By hand, move arm up and down over paper until paper is lined up properly (path of cursor should be parallel to a vertical line). Turn VACUUM switch ON. Smooth out paper.

(2) The following switches should be placed in the indicated positions:

Switch	Position
VACUUM	ON
LIGHT	ON
POWER	ON
BOARD	STANDBY
PEN	UP
INCREMENT ADVANCE	OFF
TAPE KEYBOARD: CARDS	TAPE KEYBOARD
XY NORMAL, REVERSE	NORMAL
POT SET	ZERO SET
PLOT SEQUENCE	A
NORMAL, NULL	NORMAL
PEN, ARM	ARM
AUTO, MANUAL	Neutral

Switch	Position
(3) BOARD	OPERATE
INCREMENT ADVANCE	RESET (will jump back to OFF)
PLOT SELECTOR	RESET
Rotate NULL SENSITIVITY	completely clockwise
Depress STORAGE CLEAR	
Depress STOP	

(4) Free locks on pen and arm helipots. By turning helipot knobs, set head above point of desired origin. Call this point (x_0, y_0) . Range of plotter is

$$-9,999 \leq x_i, y_i \leq 9,999$$

with the restriction

$$33\frac{1}{3} \leq \text{possible no. of points/inch} \leq 2353.$$

In effect the latter means that we can make a graph as small as $8\frac{1}{2}$ inches square. Since the plotting surface is 30 inches square, it also means that the minimum number of points along either side is 1,000.

Lock helipots. Depress RESET switch. Point (x_0, y_0) has just been plotted.

Of the points to be plotted, select one nearest limit of machine. Call this point (x_m, y_m) . Manually position head over location of this point.

Note: x_m, y_m must be > 0 .

POT SET _____ XSF

Depress STORAGE CLEAR

Type into keyboard: X, sign, $|x_m - x_0|$, PLOT

Note: On keyboard, $+$ = 0
 $-$ = 1

Wait until null indicator light goes out.

POT SET _____ YSF

Type into keyboard: Y, sign, $|y_m - y_0|$, PLOT

POT SET _____ ZERO SET

Depress STOP

RESET

Point (x_m, y_m) has just been plotted.

Depress STORAGE CLEAR

STOP

RESET

Point (x_0, y_0) has again been plotted.

(5) POT SET _____ XPX

Type into keyboard X-Parallax: X, sign, $|x_0|$, PLOT

Wait until null indicator light goes out.

POT SET _____ YPX

Type into keyboard Y-Parallax: Y, sign, $|y_0|$, PLOT

When light goes out,

POT SET _____ PLOT

The machine is now ready to accept points to plot.

Note: If the null indicator light fails to go out, set the BOARD switch to STANDBY, depress the STOP and RESET buttons, and return BOARD to OPERATE.

(6) There are three modes of input:

Keyboard Points to be entered into keyboard are of the following form:

X + _ _ _ _ , Y + _ _ _ _ , PLOT

Note: If, after plotting (x_3, y_3) , it is desired to plot (x_3, y_4) , only y_4 need be entered. In similar fashion if after this (x_4, y_4) is to be plotted, only x_4 need be entered.

Example: To plot the points $(-100, 300)$, $(200, 300)$, $(200, -6000)$, and $(5500, 6000)$, enter into keyboard the following:

X10100	Y00300	PLOT
X00200		PLOT
	Y16000	PLOT
X05500	Y06000	PLOT

The available point symbols are the following:

0	1	2	3	4	5	6	7	8	9	A	B
✦	⊙	□	◇	△	⊗	▽	⚡	⊕	⊖	⊘	.

To change point symbol, depress and keep down SYMBOL ADVANCE button, type into keyboard appropriate number, release button.

Note: To obtain symbol A, type (one at a time) into keyboard 8, 2. For symbol B, type 8, 3.

Paper Tape After step (5), depress STOP. Place the two switches on the reader in the ON position. Put tape in reader, depress RESET. Point (x_0, y_0) is plotted, reader will start reading, machine starts plotting.

IBM CARDS After step (5), depress STOP. Place TAPE KEYBOARD; CARDS switch in the CARDS position. On the IBM 523, turn the main power switch ON, place deck backed with two blanks in hopper, and depress START.

Place the AUTO, MANUAL switch in the AUTO position, and depress RESET. Point (x_0, y_0) is plotted, reader will start reading, machine starts plotting.

Notes: Card points may be plotted one at a time by depressing the AUTO, MANUAL switch from neutral to MANUAL successively, but the setting of this switch does not affect the automatic nature of plotting from paper tape. If the null indicator light fails to go out because an attempted plot is off-scale, press STORAGE CLEAR, STOP, and RESET.

(7) Incremental Advance Set YSF, YPX, and XSF in the normal fashion. XPX is normally set to zero but may be set otherwise if desired. After step (5), depress STORAGE CLEAR to set X and Y to zero and turn the INCREMENTAL ADVANCE switch ON. For negative incrementing insert minus zero into X-storage. Type (or read) in points of the form

Y + _ _ _ _ , PLOT.

The magnitude of X will now automatically be increased by 0010 units for each successive Y, or for each successive set of sequences if PLOT SEQUENCE is set to A-B, A-C, or etc. INCREMENTAL ADVANCE RESET reduces the accumulated increment to zero. Note that at least $3\frac{1}{3}$ points per inch must be plotted using INCREMENTAL ADVANCE.

(8) Plot Sequence This feature enables one to plot several different graphs or sequences of points at the same time, the number depending on the position of the PLOT SEQUENCE switch (step (2)):

A - B	2 graphs
A - C	3
.	
A - I	9

In step (4) each graph must have its own XSF, XPX, YSF, and YPX set, but semi-permanent wiring allocates one XSF, one XPX, six YSF, and two YPX pots for this purpose. Thus only YSF and YPX may change from graph to graph. MANUAL ADVANCE to the next sequence (or graph) may be made with the PLOT SELECTOR switch, which can also be used to RESET plotting to sequence A.

(9) XY Reverse The X and Y axes are interchanged by setting the XY switch to REVERSE.

b. Preparation of Paper Tape and IBM Card Data

(1) Paper Tape Seven-hole (GEORGE) paper tape initiates plotting by the following: an X character (tape code 21) must be followed by a PLUS (lower case 2C) or MINUS (2F) and four decimal digits - any more digits will simply be ignored; character Y (22) must likewise be followed by a PLUS or MINUS and four or more decimal digits; a PLOT character (3B) is then necessary for the plotting of each point (X,Y).

A SEQUENCE ADVANCE character (37) advances plotting to the next sequence and RESETS INCREMENTAL ADVANCE when the PLOT SEQUENCE switch is set to REMOTE. If PLOT SEQUENCE is set to A-B, A-C, or etc., each point plotted causes an advance to the next sequence; in these cases a SEQUENCE ADVANCE character will be ignored, except that INCREMENTAL ADVANCE will be RESET. Sequences are plotted cyclically, e.g., A-C plots A, B, C, A, B, C, A, B, Successive SEQUENCE ADVANCE tape characters must be separated by at least five irrelevant characters, such as DELETE (3F), or by one PLOT character (3B).

Let INCREMENTAL ADVANCE have been RESET (manually, or by a tape or card RESET character) and the INCREMENTAL ADVANCE switch be turned to ON. Then a + or - zero initial X-value (inserted manually or with the first plotted Y-value from tape or cards) will automatically be increased in magnitude by ten units for each plotted Y-value (if the PLOT SEQUENCE switch is set to A or REMOTE) or for each set of sequences (if PLOT

SEQUENCE reads A-B, A-C, or etc.). For example, four points are plotted for each X when PLOT SEQUENCE is set to A-D. Note that negative incrementing is done when the current X-value is negative. A non-zero initial X-value (not recommended) may or may not be incremented and reset properly, depending on the value of the initial X and the number of increments.

A RESET character (31) corresponds to turning the INCREMENTAL ADVANCE switch to RESET and the PLOT SELECTOR switch to RESET. This means that 1) the accumulated increment is always set to zero and 2) if PLOT SEQUENCE is set to REMOTE, plotting returns to sequence A. The RESET character has no effect on the sequence of plotting if PLOT SEQUENCE is on A, A-B, A-C, etc., although it does RESET INCREMENTAL ADVANCE.

The SYMBOL ADVANCE character (3A) corresponds to the SYMBOL ADVANCE button and the next tape character determines which of the twelve symbols will subsequently be plotted.

A STOP (3C) halts tape reading and point plotting, as does a parity error.

Other tape characters not among the first five following X or Y on the tape are ignored.

As with KEYBOARD points an X-coordinate (or Y) may be omitted if it is a repeat. This is not true with cards, however.

(2) IBM Cards (Decimal) Card format is controlled by board-wiring in the IBM-523 Summary Card Punch (card reader) as follows: number the first twelve horizontal entry points under COMP MAG. OR CTR. TOT. EXIT OR M.S. IN as 1, 2, . . . , 12. Then pulses to points 1-4 are interpreted as X-digits, 5-8 as Y-digits, 9 as the sign of X (PLUS: 12-punch or NO ZONE; MINUS: 11-punch), 10 as the sign of Y, 11 as SYMBOL ADVANCE, and 12 as a CONTROL CODE. Wire these holes from the desired card columns under PUNCH BRUSHES. Note that the characters X and Y need not appear on the card and cannot be interpreted as such even if they do appear. A MINUS sign may be overpunched in the same column with a decimal digit and the two characters interpreted properly by split-wiring. If two or more decimal digits are inadvertently punched in the same column, then the logical OR of the punched digits is the character interpreted.

There is no card PLOT symbol since each card-reading corresponds to one point plot. (If several points appear on each card, however, they may all be plotted by rewiring the board successively and rereading the cards.)

The CONTROL CODE characters are: FIVE (5-punch) for SEQUENCE ADVANCE; SIX (6-punch) for RESET; and SEVEN (7-punch) for STOP. These correspond exactly to the tape characters 37, 31, and 3C, respectively.

SYMBOL ADVANCE requires a decimal digit (zero through nine) to plot the current and subsequent points with the corresponding symbol. Symbols A(∞) and B(·) cannot be set by cards.

A repeated X-coordinate (or Y) must appear on the card and be wired properly or else the blanks will be interpreted as zeros. Blanks (or no wires) are ignored for SYMBOL ADVANCE and CONTROL CODE.

3. Paper-Tape-to-Card Converter

A paper-tape-to-card converter is available to translate 5, 7, or 8-hole paper tape characters to IBM card (Hollerith) characters and will also print these characters on the cards. A keyboard allows inserts to be made and a program control card permits tabbing if desired. A Parity Error stop and a Breakpoint stop switch have been built into the converter. Various translations of tape to card characters can be made by means of a plug board, but that of the chart entitled "GEORGE Paper Tape-to-IBM Card (TC) and to-Narrow Magnetic Tape (NT) Correspondence" is intended to be the primary one.

B. On-Line Equipment

The following auxiliary equipment is, or will be, connected to and controlled by GEORGE.

1. Wide Magnetic Tape (available in the near future)

This unit is described under "Components."

2. Console Typewriter

The slowest of all output devices, the console typewriter is used only when printed results are wanted immediately. Used in conjunction with the keyboard, it provides a record of manual operations. Operating speed is seven characters per second.

3. Console Keyboard

This standard sexadecimal keyboard is an input device which allows one to manually input information into the computer. Each character typed on the console keyboard is printed on the console typewriter. The following keyboard characters do not print on the console typewriter in the normal fashion, although they are read correctly by the computer: End 1 prints as space; End 2 prints as .; Delete causes a carriage return.

4. Two Readers

These are photoelectric devices that read seven-hole paper tape at the rate of 200 characters per second. The reader nearest the operator is known as reader 1 and the other as reader 2.

5. Punch

This unit punches seven-hole paper tape at the rate of sixty characters per second. A reel of tape to be punched can hold a maximum of 99,000 characters.

6. Cathode Ray Tube display (available in the near future)

This device converts digital information in core storage to analog information displayed in the form of spots on the face of a CRT. A 35-mm camera mounted over the tube then produces a permanent film record of the plotted data. A tube at the console will give a visual display.

7. Narrow Magnetic Tape Unit (available in the near future)

A modified IBM type 727 tape unit, with a 2400-foot tape having a capacity of about 800,000 words, reads and writes, with parity check, at the rate of over 6,000 characters per second and will provide added input-output facilities. One of these units will be used for high-speed input to the computer and one will be used for high-speed output. The output unit, when used with the IBM off-line printer, will be the principal means of outputting voluminous data.

GEORGE PAPER TAPE-TO-IBM CARD (TC) AND TO-NARROW MAGNETIC TAPE (NT) CORRESPONDENCES

GEORGE Char.	GEORGE Paper Tape Code (Sexd)	IBM Card (Hollerith) Rows Punched	IBM Mag. Tape BCD (Oct)	704 BCD (Oct)	IBM Char.	GEORGE Char.	GEORGE Paper Tape Code (Sexd)	IBM Card (Hollerith) Rows Punched	IBM Mag. Tape BCD (Oct)	704 BCD (Oct)	IBM Char.
0	00	0	12	00	0	Black Ribbon	24	Ignore			
1	01	1	01	01	1	Red Ribbon	25	Ignore			
2	02	2	02	02	2	* UC	26	11,8,4	54	54	*
3	03	3	03	03	3	(LC		0,8,4	34	74	(
4	04	4	04	04	4	# UC	27	No Punches	20	60	Blank
5	05	5	05	05	5	) LC		12,8,4	74	34	)
6	06	6	06	06	6	Δ UC	28	No Punches	20	60	Blank
7	07	7	07	07	7	Σ LC		No Punches	20	60	Blank
8	08	8	10	10	8	ε UC	29	No Punches	20	60	Blank
9	09	9	11	11	9	, LC		0,8,3	33	73	,
A	0A	12,1	61	21	A	UC	2A	No Punches	20	60	Blank
B	0B	12,2	62	22	B	. LC		12,8,3	73	33	.
C	0C	12,3	63	23	C	f UC	2B	11,8,3	53	53	↓
D	0D	12,4	64	24	D	/ LC		0,1	21	61	/
E	0E	12,5	65	25	E	= UC	2C	8,3	13	13	-
F	0F	12,6	66	26	F	+ LC		12	60	20	+
G	10	12,7	67	27	G	Backspace	2D	Ignore			
H	11	12,8	70	30	H	Space	2E	No Punches	20	60	Blank
I	12	12,9	71	31	I	- UC	2F	11	40	40	-
J	13	11,1	41	41	J	- LC					
K	14	11,2	42	42	K	End 1	30	No Punches	20	60	Blank
L	15	11,3	43	43	L	Reset	31	No Punches	20	60	Blank
M	16	11,4	44	44	M	Carriage Ret.	32	No Punches	20	60	Blank
N	17	11,5	45	45	N	End 2	33	Release (TC), End of Record (NT)			
O	18	11,6	46	46	O	Ignore	34	Ignore			
P	19	11,7	47	47	P	Lower Case	35	Ignore and Switch to L C			
Q	1A	11,8	50	50	Q	End 3	36	Ignore			
R	1B	11,9	51	51	R	Sequence Adv	37	No Punches	20	60	Blank
S	1C	0,2	22	62	S	End 4	38	Release and Stop (TC), End of File (NT)			
T	1D	0,3	23	63	T	Breakpt Stop	39	Breakpoint Stop (TC), Ignore (NT)			
U	1E	0,4	24	64	U	Symbol Adv.	3A	No Punches	20	60	Blank
V	1F	0,5	25	65	V	Plot	3B	No Punches	20	60	Blank
W	20	0,6	26	66	W	Stop	3C	Ignore			
X	21	0,7	27	67	X	Tab	3D	Blanksthru Field (TC), Blank (NT)			
Y	22	0,8	30	70	Y	Upper Case	3E	Ignore and Switch to U.C			
Z	23	0,9	31	71	Z	Delete	3F	Ignore			

V. BIBLIOGRAPHY OF GEORGE ROUTINES

Following is a summary of the subroutines available in the GEORGE Library, as of February 28, 1959.

<u>NUMBER</u>	<u>TITLE</u>	<u>REMARKS</u>
Classification A - Test Routines		
A-1-138	One Bit Multiply Test	
A-2-139	Zero Bit Multiply Test	
A-3-140	A and B Adder Test	
A-4-141	Multiply-Divide Test, Register Test IV	
A-5-142	One Word Leap Frog, Ac	
A-6-143	One Word Leap Frog, Qt	
A-7-144	Three Word through the Memory, Ac	
A-8-145	Three Word through the Memory, Qt	
A-9-146	Tadpole	
A-10-147	Shift End Connection 1	
A-11-148	Shift Gate Test	
A-12-149	Load and Copy Test 1	
A-13-150	F8, Record, STT Test	
A-14-151	β Test	
A-15-152	Jump Test	
A-16-153	Left Shift End Connections II	
A-17-154	Register Test	
A-18-155	1's and 0's Test	

<u>NUMBER</u>	<u>TITLE</u>	<u>REMARKS</u>
Classification A - Test Routines (Cont'd.)		
A-19-156	α Test	
A-20-157	Tag Test	
A-21-158	Pattern Test	
A-22-159	Break Point Test	
A-23-160	Sum Test I	
A-24-161	Sum Test II	
A-25-162	Leap Frog Mark I	
A-26-163	Short Form Input Output Test	
A-27-164	Shift Test I	
A-28-177	Leap Frog - Mark II	
Classification B - Input-Output		
B-1-116	Floating Decimal to Standard Binary Floating Point Input and Conversion	
B-2-117	Standard Binary Floating Point to Floating Decimal Conversion and Output	
B-3-165	Fixed Point Decimal to Binary Fraction Conversion	9 decimal places = max. allowed
B-4-166	Fixed Point Binary to Decimal Fraction Conversion	Gives 9 decimal places
B-5-184	Standard Binary Floating Point to Floating Decimal in Scientific Notation, Conversion and Output	
B-6-188	Fixed Point Binary to Decimal Conversion and Output, with Scaling and Format	
B-7-197	Floating Decimal (Scientific Notation) to Standard Binary Floating Point, Input and Conversion	Inputs B-5-184 output

<u>NUMBER</u>	<u>TITLE</u>	<u>REMARKS</u>
Classification B - Input-Output (Cont'd.)		
B-8-202	Fixed Point Decimal to Binary Integer Input and Conversion	
B-9-203	Point Plotter, Fixed Point	
B-10-208	General Decimal to Binary Input and Conversion, for Fixed or Floating Point Fractions or Integers, with Scaling	
B-11-209	Floating Decimal to Indexed Binary Floating Point Input and Conversion	Inputs data for D-3-131
B-12-210	Indexed Binary Floating Point to Floating Decimal Conversion and Output	Outputs D-3-131 data
B-13-213	Decimal to Binary Fraction Input and Conversion, Fixed Point	Allows up to 12 decimal places
B-14-214	Decimal Memory Print-Out, Fixed Point	Converts a block of data, and punches
B-15-228	Input and Conversion of a Sequence of Decimal Fractions to Binary, Fixed Point	Allows up to 12 decimal places
B-16-136	Output for Point Plotter	Punches B-4-166 type results for use on Data Plotter
B-17-234	Floating Point Decimal Dump	Floating point version of B-14-214
Classification D - Arithmetic		
D-1-115	Floating Point Arithmetic	Holds power and fraction in one word
D-2-110	Double Precision Arithmetic, Fixed Point	
D-3-131	Floating Point Arithmetic with Precision Index	Holds power, fraction and index in two words

<u>NUMBER</u>	<u>TITLE</u>	<u>REMARKS</u>
Classification E - Elementary Functions		
E-1-103	Exponential, Fixed Point	
E-2-104	Scaled Negative Exponential, Fixed Point	Computes $2^{-s} e^{-x}$ and s , where $x = 2^p x \geq 0$.
E-3-105	Square Foot, Fixed Point	
E-4-106	Square Root for Small Arguments, Fixed Point	Faster than E-3 for $x < 2^{-8}$
E-5-107	Negative Exponential, Fixed Point	Computes e^{-x} , where $x > 0$
E-6-108	Arcsine $x2^{-1}$, Fixed Point	
E-7-114	Exponential, Floating Point	
E-8-109	Square Root, Floating Point	
E-9-118	Natural Logarithm, Fixed Point	
E-10-127	Fast, Scaled General Exponential, Fixed Point	
E-11-168	$\sin(\pi x)$ or $\cos(\pi x)$, Floating Point	
E-12-169	$\sin(\pi x)$ <u>and</u> $\cos(\pi x)$, Fixed Point	
E-13-170	Natural Logarithm, Floating Point	
E-14-171	Arctangent, Fixed Point	$ \text{error} < 10^{-7}$
E-15-172	Arctangent, Floating Point	
E-16-173	Fixed Point Scaled Natural Logarithm	
E-17-174	Fixed Sin-Cos	
E-18-182	Fixed Point Hyperbolic Sin and Cosine	
E-19-207	Floating Point Cube-Root	
E-20-227	Arctangent, Fixed Point	$ \text{error} < 3 \cdot 10^{-11}$

<u>NUMBER</u>	<u>TITLE</u>	<u>REMARKS</u>
Classification F - Special Functions		
F-1-133	Gamma Function, Fixed Point	
F-2-199	Doppler Integrals, Fixed Point	
F-3-216	Bessel Function J_0 , Fixed Point	
F-4-229	Bessel Functions J_0 with Extended Range, Fixed Point	$-1 < J_0 \leq 1-2^{-32}$
F-5-232	Legendre Polynomials Generator, Floating Point	
Classification I - Integration		
I-1-112	Trapezoidal Integration, Fixed Point	
I-2-113	Simpson Integration, Fixed Point	
I-3-124	$\sin \pi x$, $\cos \pi x$, Error Function, Fixed Point	
I-4-125	Fifteen Point Gaussian Integration, Fixed Point	
I-5-134	Seven Point Gaussian Integration with Optional Controls, Fixed Point	
I-6-135	Fifteen Point Gaussian Integration with Optional Controls, Fixed Point	
I-7-183	$f(x) = e^{x^2} \int_x^\infty e^{-t^2} dt$, for $x \geq 0$, Fixed Point	
I-8-191	Laguerre-Gauss Quadrature, Fifteen Point, Floating Point,	
I-9-192	Laguerre-Gauss Quadrature, Modified, Floating Point	
I-10-193	Trapezoidal Integration, Floating Point	
I-11-200	Double Integration using Seven Point Gaussian Quadratures, Fixed Point	

<u>NUMBER</u>	<u>TITLE</u>	<u>REMARKS</u>
Classification I - Integration (Cont'd.)		
I-12-201	Double Integration using Fifteen Point Gaussian Quadrature, Fixed Point	
I-13-226	Two Word Floating Point Trapezoidal	
Classification K - Differential Equations		
K-1-194	Runge-Kutta Integration, Fixed Point	
Classification L - Linear Equations		
L-1-190	Linear Equation Solver Fixed Point General Routine	
L-2-219	Linear Equation Solver, Fixed Point	
Classification M - Matrix Algebra		
M-1-119	Matrix Inversion with Accompanying Solution of Linear Equations, Floating Point	Jordan Method
M-2-120	Matrix Product, Fixed Point	
M-3-121	Eigenvalues and Eigenvectors of a Real Symmetric Matrix, Fixed Point	
M-4-178	Square Matrix Transposition	
M-5-179	Matrix Product, Floating Point	
M-6-180	Matrix Inversion with Accompanying Solution of Linear Equations, Floating Point, General Routine	
M-7-181	Eigenvalues and Eigenvectors of a Real Symmetric Matrix, Fixed Point General Routine	
M-8-198	Matrix Inversion, with Accompanying Solution of Linear Equations, Floating Point	Enlarged version of M-1-119

NUMBER	TITLE	REMARKS
Classification M - Matrix Algebra (Cont'd.)		
M-9-205	Eigenvalues and Eigenvectors of a Real Symmetric Matrix, Fixed Point	E-vectors by rotational triangularization
M-10-206	Enlargement of a Triangular to a Square Symmetric Matrix	
M-11-211	Eigenvalues and Eigenvectors of a Real Symmetric Matrix, Fixed Point General Routine	E-vectors by rotational triangularization
M-12-224	Eigenvalues and Eigenvectors of the Product of Two Real Symmetric Matrices, Fixed Point	
M-13-231	Rectangular Matrix Transposition	Last bit of each element altered to save space
M-14-233	Rectangular Matrix Transposition	All bits of each element preserved
Classification N - Non-Linear Equations		
N-1-215	Solution of a System of Non-Linear Equations by the Newton-Raphson Method, Floating Point	
Classification P - Algebraic, Miscellaneous		
P-1-204	Polynomial Generator, Fixed Point, General Routine	
Classification Q - Probability Distribution		
Q-1-111	Error Function, Fixed Point	
Classification R - Roots of Equations		
R-1-101	650 Program	
R-2-102	650 Program	
R-3-123	*Zeros of a Polynomial, Newton's Method Fixed Point General Routine	

*Not Complete

<u>NUMBER</u>	<u>TITLE</u>	<u>REMARKS</u>
Classification S - Statistics		
S-1-129	Least Squares Linear Fit, Floating Point General Routine	
S-2-122	*Least Squares Fit of a Linear Combination of Exponentials, Floating Point General Routine	Fits Coefficients and parameter in exponent
S-3-128	Least Squares Fit to a Polynomial, Floating Point General Routine	
S-4-185	Fitting Linear Combinations of Legendre Polynomials, Fixed Point General Routine	
S-5-195	*Least Squares Linear Fit of Arbitrary Functions, Floating Point General Routine	
S-6-196	Least Squares Fit for Linear Combinations of Exponentials, Floating Point General Routine	Exponential Parameter known
Classification X - Service Routines		
X-1-126	Subroutine Loader	
X-2-000	Command Routine	
X-3-167	Subroutine Error Stop	
X-4-175	Store - Restore	
X-5-176	Register Print-Out	
X-6-187	GEORGE Code Listing Routine	
X-7-212	Address Search	
X-8-217	Flow Monitor	
X-9-218	Memory-Tape Comparison	
X-10-220	Checkpoint Monitor	
X-11-221	Tape Comparer, with or without Duplication	

*Not Complete

<u>NUMBER</u>	<u>TITLE</u>	<u>REMARKS</u>
Classification X - Service Routines (Cont'd.)		
X-12-222	Fixed Point Decimal Memory Print-out	
X-13-223	Program Trace	
Classification Z - Miscellaneous		
Z-1-130	Maximum or Minimum of a Function of Several Independent Variables, Floating Point General Routine Gradient Method	
Z-2-132	GEORGE - AVIDAC Decoder (GAD)	
Z-3-186	Floating to Fixed Point Conversion	
Z-4-189	Word Sorter	
Z-5-225	Cap-Punch	
Z-6-137	Breakpoint Punch	Punches all break-points set to jump in a program.

ACKNOWLEDGEMENT

Sincere appreciation is expressed to all the members of the Applied Mathematics Division who aided in the various phases of the preparation of this manual. In particular, thanks go to:

Dr. William F. Miller, Director of the Applied Mathematics Division, for all his helpful advice and counseling, and for contributing the Manual Introduction;

Dr. Richard F. King and Herbert L. Gray, for their many technical suggestions, and for contributing the sections on GAR and on the Auxiliary Equipment for GEORGE;

Burton S. Garbow for smoothing and improving the copy by his painstaking editing and his numerous ideas and constructive criticisms;

Stanley Zawadski for his help in providing needed technical information on the GEORGE Auxiliary Equipment;

Miss Jacqueline Anselmino and (Mrs.) Patricia Benson for their careful work in preparing the typewritten copy.

APPENDIX A

Boolean Algebras

by
Donald A. Flanders

.1 Definition

A Boolean algebra is any non-empty set M (whose members we shall denote by a, b, c , etc.) for which a one-valued monary operation (called bar) is defined for every $a \in M$ (notation: $\bar{a}$ denotes the result of operation bar on the member a of M) and a one-valued binary operation (called cap) is defined for every ordered binary subset (a, b) of M (notation: $a \cap b$ denotes the result of operation cap on the ordered binary subset (a, b) of M), where bar and cap satisfy the following axioms:

Ax. 1. Operation bar is an involution, i.e., if $a \in M$, then

$$(1) \quad \overline{\bar{a}} = a.$$

Ax. 2. Operation cap is (2.1) idempotent, (2.2) commutative, and (2.3) associative, i.e., if $a \in M$, $b \in M$ and $c \in M$, then

$$(2.1) \quad a \cap a = a,$$

$$(2.2) \quad a \cap b = b \cap a,$$

$$(2.3) \quad (a \cap b) \cap c = a \cap (b \cap c).$$

Ax. 3. If $a \in M$ and $b \in M$ then

$$(3.1) \quad b \cap (a \cap \bar{a}) = a \cap \bar{a}$$

and

$$(3.2) \quad b \cap a \cap \bar{\bar{a}} = b.$$

Ax. 4. If $a \in M$, $b \in M$ and $c \in M$, then

$$(4) \quad a \cap \overline{\bar{b} \cap \bar{c}} = \overline{\overline{a \cap b} \cap \overline{a \cap c}}.$$

Theo. and Def. 1. There is not more than one $a \in M$ such that $x \cap a = a$ for every $x \in M$. If such an $a \in M$ exists it is called the null element of M for operation cap and will be denoted by n .

Theo. and Def. 2. There is not more than one $a \in M$ such that $x \cap a = x$ for every $x \in M$. If such an $a \in M$ exists it is called the unit element of M for operation cap and will be denoted by u .

Theo. 3. Ax. 3 is equivalent to

Ax. 3'. For every $a \in M$

$$(3'.1) \quad a \cap \bar{a} = n;$$

and

$$(3'.2) \quad \bar{n} = u.$$

Cor. of Ax. 3 or 3'. $n \in M$ and $u \in M$.

Theo. and Def. 4. If $a \in M$ and $b \in M$ then there exists unique $c \in M$ such that $c = \bar{a} \cap \bar{b}$. The set of all such relations between members a, b, c of M is a one-valued binary operation (which we shall call cup) defined for every ordered binary subset (a, b) of M . We shall denote the result of operation cup on (a, b) by $a \cup b$.

Theo. 5. Ax. 4 is equivalent to

Ax. 4'. Operation cap is distributive with respect to operation cup, i.e.,

$$(4') \quad a \cap (b \cup c) = (a \cap b) \cup (a \cap c).$$

.2 Some consequences of the axioms.

Theo. 6. If $a \in M$ and $b \in M$ then

$$(6.1) \quad a \cap b = \overline{\bar{a} \cup \bar{b}} \text{ and } \overline{a \cap b} = \bar{a} \cup \bar{b},$$

$$(6.2) \quad a \cup b = \overline{\bar{a} \cap \bar{b}} \text{ and } \overline{a \cup b} = \bar{a} \cap \bar{b}.$$

Theo. 7. Operation cup is idempotent, commutative and associative, and is distributive with respect to operation cap, i.e.,

$$(7.1) \quad a \cup a = a,$$

$$(7.2) \quad a \cup b = b \cup a,$$

$$(7.3) \quad (a \cup b) \cup c = a \cup (b \cup c),$$

$$(7.4) \quad a \cup (b \cap c) = (a \cup b) \cap (a \cup c).$$

Theo. 8. For every $a \in M$

$$(8.1) \quad a \cap n = n, a \cap u = a, a \cap \bar{a} = n,$$

$$(8.2) \quad a \cup u = u, a \cup n = a, a \cup \bar{a} = u.$$

Note that in consequence of Theo. 8.2 we may regard u and n respectively as the null and unit elements of M for operation cup.

Theo. 9. In any Boolean algebra the statements resulting from replacing "cap" by "cup" and " $\cap$ " by " $\cup$ " in Axs. 2, 3, and 4 are theorems of that algebra.

Def. and Theo. 10. If n is an integer > 2 and $a_i \in M$ for $1 \leq i \leq n$, then we define $a_1 \cap \dots \cap a_n$, resp. $a_1 \cup \dots \cup a_n$, inductively by

$$(10.1) \quad a_1 \cap a_2 \cap a_3 = (a_1 \cap a_2) \cap a_3, \text{ resp. } a_1 \cup a_2 \cup a_3 = (a_1 \cup a_2) \cup a_3,$$

and

$$(10.2) \quad a_1 \cap \dots \cap a_i = (a_1 \cap \dots \cap a_{i-1}) \cap a_i, \text{ resp.}$$

$$\dots \quad a_1 \cup \dots \cup a_i = (a_1 \cup \dots \cup a_{i-1}) \cup a_i, \text{ for } 3 < i \leq n.$$

The value of $a_1 \cap \dots \cap a_n$, resp. $a_1 \cup \dots \cup a_n$, is independent of the order of the terms.

Corollary to Theorem 10.

$$\overline{a_1 \cap a_2 \cap \dots \cap a_p} = \bar{a}_1 \cup \bar{a}_2 \cup \dots \cup \bar{a}_p$$

$$\overline{a_1 \cup a_2 \cup \dots \cup a_p} = \bar{a}_1 \cap \bar{a}_2 \cap \dots \cap \bar{a}_p$$

Def. and Theo. 11. If $a \in M$, $b \in M$ and $a \cap b = n$, $a \cup b = u$, then a and b are said to be complementary, and each is called a complement of the other. Thus b is a complement of a if and only if $b = \bar{a}$

$$\text{Theo. 12. } a \cap (a \cup b) = a \cup (a \cap b) = a$$

$$\text{Theo. 13. } a \cap b = a \text{ if and only if } a \cup b = b.$$

.3 The relation $\leq$ in a Boolean algebra.

Def. 14. If $a \in M$, $b \in M$ and $a \cap b = a$ then we shall say that a precedes b in M and write $a \leq b$. We shall also say that b follows a in M and write $b \geq a$. If also $c \in M$ and $a \leq c \leq b$ we shall say that c lies between a and b .

Theo. 14. The relation "precedes" is reflexive, anti-symmetric and transitive, and every member of M lies between n and u , i.e., if $a \in M$, $b \in M$ and $c \in M$, then

$$(14.1) \quad a \leq a,$$

$$(14.2) \quad \text{If } a \leq b \text{ and } b \leq a \text{ then } a = b,$$

$$(14.3) \quad \text{if } a \leq b \text{ and } b \leq c \text{ then } a \leq c,$$

$$(14.4) \quad n \leq a \leq u.$$

.4 The operation $\oplus$.

Theo. and Def. 15. If $a \in M$ and $b \in M$ then there exists unique $c \in M$ such that $c = (a \cap \bar{b}) \cup (\bar{a} \cap b)$. The set of all such relations connecting members a, b, c of M forms a one-valued binary operation (which we shall call half-sum) defined for every ordered binary subset (a, b) of M . We shall denote the half-sum of a and b by $a \oplus b$.

Theorem 16. The operation half-sum is commutative and associative, and cap and cup are distributive with respect to half-sum, i.e., if $a \in M, b \in M, c \in M$, then

$$(16.1) \quad a \oplus b = b \oplus a,$$

$$(16.2) \quad a \oplus (b \oplus c) = (a \oplus b) \oplus c,$$

$$(16.3) \quad a \cap (b \oplus c) = (a \cap b) \oplus (a \cap c),$$

$$\text{Theo. 17. } \overline{a \oplus b} = (a \cap b) \cup (\bar{a} \cap \bar{b}).$$

.5 Boolean functions.

In this section B denotes any Boolean algebra with more than one member, and hence containing n and u , and B_0 is the special B consisting of n and u only.

Def. 18. If x is a Boolean variable (i.e., assumes only members of B as values), we shall call the complement of x , and shall denote by $\bar{x}$, that function of x such that $\bar{x} = \bar{a}$ if $x = a$. $x \cap y$ and $x \cup y$ are similarly defined and obey the laws of Boolean algebra, e.g., $x \cap \bar{x} = n$, $x \cup \bar{x} = u$.

Theo. 19. If $x_1, \dots, x_p$ are independent Boolean variables, so are $w_1, \dots, w_p$, where each w_i is either x_i or $\bar{x}_i$.

Def. 20. If $x_1, \dots, x_p$ are Boolean variables, $w_1 \cap \dots \cap w_q$, where $q \geq 1$, will be called a Boolean monomial in $x_1, \dots, x_p$ of virtual degree q if each w_i is either a member of B or one of the variables $x_1, \dots, x_p, \bar{x}_1, \dots, \bar{x}_p$.

Def. and Theo. 21. A Boolean monomial $w_1 \cap \dots \cap w_q$ in $x_1, \dots, x_p$ will be called irreducible if it has the form $a \cap t_{i_1} \cap \dots \cap t_{i_r}$, where $a \in B$, $t_{i_j} = x_{i_j}$ or $\bar{x}_{i_j}$, for each j and $i_1 < \dots < i_r$. $t_{i_1} \cap \dots \cap t_{i_r}$ will be called the body and a the coefficient of the irreducible monomial, and r will be called

its actual degree. (It is understood in this definition that r may be 0, in which case the monomial consists only of a , which may be n .) To each Boolean monomial in $x_1, \dots, x_p$ corresponds a unique irreducible monomial, to which the original may be reduced by successive applications of theorems of Boolean algebra.

Def. and Theo. 22. A Boolean monomial $w_0 \cap w_1 \cap \dots \cap w_q$ in $x_1, \dots, x_p$ will be called minimal if $q = p$, $w_0 \in B$ and $w_i = x_i$ or $\bar{x}_i$ for $1 \leq i \leq p$. If also $w_0 = u$, the monomial will be called basic. There are just 2^p distinct basic minimal monomials in $x_1, \dots, x_p$.

Theo. 23. If w_1 and w_2 are basic monomials in $x_1, \dots, x_p$, then $w_1 \cap w_2 = w_1$ or n according as w_2 is or is not identical with w_1 . (This may be expressed by saying that the basic monomials are orthogonal.)

Theo. 24. The cup of all basic monomials in p variables over B is identically u .

Theo. 25. The bar of a basic monomial in p variables over B is the cup of all basic monomials in these variables except the given basic monomial, i.e., if the variables are $x_1, \dots, x_p$ and the basic monomial w is $u \cap y_1 \cap \dots \cap y_p$, where for each i we have $y_i = x_i$ or $y_i = \bar{x}_i$, then $\bar{w} = (u \cap \bar{y}_1 \cap \dots \cap \bar{y}_p) \cup (u \cap \bar{y}_1 \cap \dots \cap \bar{y}_{p-1} \cap y_p) \cup \dots \cup (u \cap y_1 \cap \dots \cap y_{p-1} \cap \bar{y}_p) \cup (n \cap y_1 \cap \dots \cap y_p)$.

Proof. The theorem is true for $p = 1$ since $\overline{u \cap y_1} = n \cup \bar{y}_1 = \bar{y}_1 = (u \cap \bar{y}_1) \cup (n \cap y_1)$. Assume it true for $1 \leq p < q$. Then

$$\begin{aligned} \overline{u \cap y_1 \cap \dots \cap y_{q-1} \cap y_q} &= \overline{(u \cap y_1 \cap \dots \cap y_{q-1}) \cap y_q} = \\ &= \{[(u \cap \bar{y}_1 \cap \dots \cap \bar{y}_{q-1}) \cup \dots \cup (u \cap y_1 \cap \dots \cap y_{q-2} \cap \bar{y}_{q-1}) \cup \\ &\quad (n \cap y_1 \cap \dots \cap y_{q-1})] \cap (\bar{y}_q \cup y_q)\} \cup \{[(u \cap \bar{y}_1 \cap \dots \cap \bar{y}_{q-1}) \cup \dots \cup \\ &\quad (u \cap y_1 \cap \dots \cap y_{q-2} \cap \bar{y}_{q-1}) \cup (u \cap y_1 \cap \dots \cap y_{q-1})] \cap \bar{y}_q\} = \\ &= (u \cap \bar{y}_1 \cap \dots \cap \bar{y}_q) \cup \dots \cup (u \cap y_1 \cap \dots \cap y_{q-1} \cap \bar{y}_q) \cup (n \cap y_1 \cap \dots \cap y_q) \end{aligned}$$

Cor. 26. Using the notation of the theorem, the bar of a minimal monomial $a \cap y_1 \cap \dots \cap y_q$ is $(u \cap \bar{y}_1 \cap \dots \cap \bar{y}_q) \cup \dots \cup (u \cap y_1 \cap \dots \cap y_{q-1} \cap \bar{y}_q) \cup (\bar{a} \cap y_1 \cap \dots \cap y_q)$.

Def. 26. A Boolean polynomial (notation: B.p.) in $x_1, \dots, x_p$ over B of order r is defined inductively on r as follows:

A B.p. of order 1 is either an element a of B or one of the variables x_i .

A B.p. of order r ($r > 1$) is either the bar of a B.p. t of order $r-1$ or is the cap of two B.p.'s, t and s , of order $< r$, of which at least one is of order $r-1$.

We shall also say that any expression derivable from a B.p. of order r by successive substitutions of expressions involving the defined operations cup and half-sum for their equivalents in terms of bar and cap or by applying any of the theorems or axioms of Boolean algebra are also B.p.'s, but without specifying the order.

Theo. 27. Every Boolean polynomial in $x_1, \dots, x_p$ over B of order r can be expressed uniquely as a cup of minimal monomials in which each basic monomial appears just once.

Proof. The proof of the possibility of at least one such representation is by induction on r . Set $N = 2^p$.

For $r = 1$ the possible B.p.'s are (I) a , where a is any element of B and (II) x_i , where x_i is any one of the given variables. (I) $a = a \cap u = a \cap [(x_1 \cap \dots \cap x_p) \cup \dots \cup (\bar{x}_1 \cap \dots \cap \bar{x}_p)] = (a \cap x_1 \cap \dots \cap x_p) \cup \dots \cup (a \cap \bar{x}_1 \cap \dots \cap \bar{x}_p)$, which is of the desired form. (II) $x_i = [(u \cap x_i) \cup (n \cap \bar{x}_i)] \cap [(x_1 \cap \dots \cap x_{i-1} \cap x_{i+1} \cap \dots \cap x_p) \cup \dots \cup (\bar{x}_1 \cap \dots \cap \bar{x}_{i-1} \cap \bar{x}_{i+1} \cap \dots \cap \bar{x}_p)] = (u \cap x_1 \cap \dots \cap x_{i-1} \cap x_i \cap x_{i+1} \cap \dots \cap x_p) \cup \dots \cup (u \cap \bar{x}_1 \cap \dots \cap \bar{x}_{i-1} \cap x_i \cap \bar{x}_{i+1} \cap \dots \cap \bar{x}_p) \cup (n \cap x_1 \cap \dots \cap x_{i-1} \cap \bar{x}_i \cap x_{i+1} \cap \dots \cap x_p) \cup \dots \cup (n \cap \bar{x}_1 \cap \dots \cap \bar{x}_{i-1} \cap \bar{x}_i \cap \bar{x}_{i+1} \cap \dots \cap \bar{x}_p)$, which is of the desired form.

If $r > 1$, let $f_0(x_1, \dots, x_p), \dots, f_{N-1}(x_1, \dots, x_p)$ denote the basic monomials. First assume the polynomial is of the form $\bar{t}$, where t is of order $r-1$ and hence can be represented in the form $(a_0 \cap f_0) \cup \dots \cup (a_{N-1} \cap f_{N-1})$, by hypothesis. Then $\bar{t} = (\overline{a_0 \cap f_0}) \cap \dots \cap (\overline{a_{N-1} \cap f_{N-1}}) = [(\bar{a}_0 \cap f_0) \cup f_1 \cup \dots \cup f_{N-1}] \cap \dots \cap [f_0 \cup \dots \cup f_{N-2} \cup (a_{N-1} \cap f_{N-1})]$ (by cor. 26), which $= (\bar{a}_0 \cap f_0) \cup \dots \cup (\bar{a}_{N-1} \cap f_{N-1})$, by Theo. 23, and is of the desired form.

If $r > 1$ and the polynomial has the form $t \cap s$, let $t = (a_0 \cap f_0) \cup \dots \cup (a_{N-1} \cap f_{N-1})$, $s = (b_0 \cap f_0) \cup \dots \cup (b_{N-1} \cap f_{N-1})$. Then $t \cap s = (c_0 \cap f_0) \cup \dots \cup (c_{N-1} \cap f_{N-1})$, where $c_i = a_i \cap b_i$, and $t \cap s$ has the desired form.

To prove the uniqueness, assume there are two such representations, $(a_0 \cap f_0) \cup \dots \cup (a_{N-1} \cap f_{N-1}) = (b_0 \cap f_0) \cup \dots \cup (b_{N-1} \cap f_{N-1})$. If $a_i \neq b_i$ for some i , let $f_i = y_1 \cap y_2 \cap \dots \cap y_p$, where each $y_j = x_j$ or $\bar{x}_j$. If we select values (n or u) for the x_j 's such that $y_1 = y_2 = \dots = y_p = u$, then $f_i = u$ and every other basic monomial is n , so the value of the polynomial is a_i in one representation and b_i in the other, which is impossible.

.6 Construction of Boolean polynomials from tables.

Let $x_1, \dots, x_p$ and y be Boolean variables in B , and let $y = f(x_1, \dots, x_p)$, where f is a one-valued function given by a table of the value

of y for each possible set of values of $x_1, \dots, x_p$. We propose to show how one may construct the canonical representation of $f(x_1, \dots, x_p)$ if it is a Boolean polynomial without prior knowledge of this fact. When the putative canonical representation $g(x_1, \dots, x_p)$ has been set up, the decision as to whether it actually represents the function then depends upon whether it is feasible to find a particular set of values of $x_1, \dots, x_p, y$ from the table that fails to satisfy $y = g(x_1, \dots, x_p)$ or to verify that all sets given by the table do satisfy. If B contains only a finite set of members such a verification is theoretically possible in all cases.

We begin with that subset of the table which lists y for every possible selection of $x_1, \dots, x_p$ from n and u , a total of 2^p entries. Thus we have a table such as

x_1	x_2	$\dots$	x_{p-1}	x_p	y
n	n	$\dots$	n	n	b_0
n	n	$\dots$	n	u	b_1
n	n	$\dots$	u	n	b_2
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$a_{r,1}$	$a_{r,2}$	$\dots$	$a_{r,p-1}$	$a_{r,p}$	b_r
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
u	u	$\dots$	u	u	b_{2^p-1}

If we replace n by 0 and u by 1 in the set of values of $x_1, \dots, x_p$ and regard the resulting sequence of digits as the binary representation of an integer, then the integer thus corresponding to any line of the table may be taken as the value of the index r for that line. Corresponding to line r we form the monomial $t_r = b_r \cap w_1 \cap \dots \cap w_p$, where $w_i = x_i$ if $a_{r,i} = u$, $w_i = \bar{x}_i$ if $a_{r,i} = n$.

Then

$$\begin{aligned} t_r(x_1, \dots, x_p) &= b_r \text{ if } (x_1, \dots, x_p) = (a_{r,1}, \dots, a_{r,p}) \\ &= n \text{ if } (x_1, \dots, x_p) = (a_{s,1}, \dots, a_{s,p}), s \neq r. \end{aligned}$$

We now set

$$g(x_1, \dots, x_p) = t_0 \cup t_1 \cup \dots \cup t_{2^p-1}.$$

Clearly

$$g(x_1, \dots, x_p) = f(x_1, \dots, x_p)$$

for

$$(x_1, \dots, x_p) = (a_{r,1}, \dots, a_{r,p}) \text{ and } 0 \leq r < 2^p.$$

Hence $y = g(x_1, \dots, x_p)$ if y is a Boolean polynomial, since such a representation is unique. In particular, if $B = B_0$, $y = g(x_1, \dots, x_p)$ always.

.7 Arithmetization of B_0 .

It is possible and convenient to express the members and operations of B_0 by means of arithmetic. One obvious way to do this is to represent n by 0, u by 1, $\bar{a}$ by $1 + a$, mod 2 and $a \cap b$ by ab , mod 2. We then have the following lexicon for translation:

B_0	Arithmetic, mod 2
n	0
u	1
$\bar{a}$	$1 + a$
$a \cap b$	ab
$a \cup b$	$a + b + ab$
$a \oplus b$	$a + b$

Instead we may use ordinary arithmetic by setting $a = 1 - a$. We then have

B_0	Arithmetic
n	0
u	1
$\bar{a}$	$1 - a$
$a \cap b$	ab
$a \cup b$	$a + b - ab$
$a \oplus b$	$a + b - 2ab$

We shall usually use the latter system. Note that in this system we may have formal expressions, such as that for $a \oplus b$, which may involve quantities other than 0 or 1. If we keep in mind the fact that the algebra does not in fact produce elements outside of itself, then we can be sure that any number other than 0 or 1 produced by the arithmetic is purely transitory. The end result must be 0 or 1.

APPENDIX B.

Representations of and Notations for Numbers

by
Donald A. Flanders

.1 Introduction

.1.1 The use of mechanical (including electrical and electronic) equipment to carry out numerical computation has caused the development of a variety of mechanisms for carrying out the fundamental operations of arithmetic, and the theory of these mechanisms has in turn depended on a variety of number notations. While we shall not attempt to detail the theory of all the various mechanisms, we shall attempt to provide a basis for a considerable number of arithmetical schemes currently in use in high-speed electronic computers.

All of the systems we shall describe may be regarded as being based on generalizations of the decimal notation. Since most of us have acquired our knowledge of decimal notation gradually over early stages of our education and have had no occasion to systematize this knowledge since, it may seem that this account will emphasize unduly certain facts about arithmetic that we regard as too obvious to require explicit statement. Nevertheless, these obvious facts often form the point of departure for less obvious generalizations.

Let us consider some of the ways in which computing mechanisms affect number notation. (1) A digital computing machine always handles some fixed number of digits as the fundamental aggregate that is to be treated as a number. Hence, a number representable in writing by fewer than this number of digits must be represented in the machine by otherwise superfluous zeros in order that every digit place provided by the machine be occupied by some digit. Hence also numbers requiring more digits must be handled in some special way, e.g., by multiple precision or by floating point. (2) Generally speaking, every such machine operates on the assumption that the location of the (decimal) point is fixed. Hence, instead of moving the point with reference to the digits to effect multiplication by a power of the base, as in written arithmetic, the digits are shifted with reference to the point. This, combined with (1), motivates the procedure - presently to be introduced - of associating with any real number, even one ordinarily denoted by a finite set of digits, an infinite sequence of digits having neither beginning nor end. (3) Desire for speed and reliability have caused computing machine designers to use digital notations other than the decimal system. A digital "place" in a machine is occupied by a mechanism that can assume precisely as many distinct stable states as there are distinct

digits - e.g., ten in the decimal system, two in the binary system. Even at the apparent cost of using more places to represent a given number the simplicity of designs based on bi-stable mechanisms has recommended them. (4) In written arithmetic the procedure for adding and subtracting signed numbers is cumbersome if applied directly to mechanisms. In particular, if one wishes to add a and b , one must first decide whether the operation to be performed on $|a|$ and $|b|$ is actually addition or subtraction, according as the signs of a and b are like or unlike. Next, if the signs are unlike, say $a = +|a|$, $b = -|b|$, one must decide whether to subtract $|b|$ from $|a|$ or the reverse. This carries with it the implication of providing both an adding and a subtracting device. An almost universally used device to avoid these difficulties - more particularly the second - is to use a so-called "complement" notation, which permits subtraction to be treated as a minor modification of addition.

With these comments in mind we shall go ahead to develop the theoretical basis of various notational and operational schemes used in the design of computing machines. For convenience we shall distinguish between what we shall call "representations of" and "notations for" numbers. Every real number will be associated with at least one representation, so that in a given scheme there must be unaccountably infinitely many numbers to be represented. Notations will be reserved for an at most countably infinite subset of the real numbers - namely, those real numbers each of which requires the specification of a finite set of symbols to denote it unambiguously in a given scheme -, and in practise only a finite subset (though how large is not specified) of these is assumed actually to occur.

1.2 In introducing any mathematical topic it is impossible to specify completely the basic ideas assumed. At this point we shall list and in some cases describe some notations that will be used in the main body of the text without further elaboration.

1.2.1 The notion of entity (thing, etc) the notion of a set (class, aggregate, etc.) of entities, and very elementary relations, such as an entity belonging to a set of which it is a member, one set as a subset of another, 1 - 1 correspondence between sets, etc.

1.2.2 The notion of (linearly) ordered set as exemplified by various subsets of the real numbers ordered by $<$ (e.g., the continuum of all real numbers), discretely ordered sets such as the set of rational integers, the notions of first and last and of immediate predecessor and successor elements if they exist.

We shall use the term "sequence" to refer to a discretely ordered set, whether or not it possesses a first or last element or both or neither; and the term "two-way sequence" will be used to denote a sequence such as the set of all rational integers ordered by $<$, which has neither first nor last element.

.1.2.3 The operational and ordinal properties of the real number system.

.1.2.4 Mathematical induction as applied to discretely ordered sets (whether finite or infinite) having a first element, and as applied both to theorems and definitions. In particular, if it is known that a property holds for at least one integer, and that there is some integer N such that the property fails for every integer less than N , then there is a least integer for which the property holds.

.1.2.5 Very elementary notions of infinite series such as the meaning of convergence, the sum of a convergent series, and the interval of convergence of a real power series.

.1.2.6 Throughout this chapter "number" will be assumed to mean "real number." We shall use the term "magnitude" to denote a "non-negative real number."

.2 Digital Representation of Magnitudes

.2.1 Def. Any integer $b > 1$ may be chosen as a base of a digital number representation or notation. When so chosen, the integral magnitudes $< b$ are called the base- b digits.

Thus zero and one are base- b digits in every case and we shall use the conventional symbols 0 and 1 for them. If $b \leq 10$, we shall use such of the additional conventional symbols 2, 3, 4, 5, 6, 7, 8, 9, as are applicable. For $b > 10$ special symbols must be devised.

.2.1 Theo. If b is a base, N is an integer $\left(\begin{smallmatrix} < \\ > \end{smallmatrix} 0 \right)$ and x_i is a base- b digit for every $i > N$, then

$$(2.1.1) \quad \sum_{i=N+1}^{\infty} x_i b^{-i}$$

converges and its sum is a magnitude.

.2.2 Theo. If x is a magnitude and b is a base, then there exists an integer $N \left(\begin{smallmatrix} < \\ > \end{smallmatrix} 0 \right)$ and an infinite sequence of base- b digits x_i , for every $i > N$, such that

$$(2.2.1) \quad x = \sum_{i=N+1}^{\infty} x_i b^{-i} .$$

These theorems we accept without proof. They form the basis of the theorems and definitions in (2.3).

2.3.1 Theo. and Def. Let b be any arbitrary base. If $\{x_i\}$ is a two-way sequence of base- b digits such that there is an integer N such that $x_i = 0$ for $i \leq N$, then (2.3.1.1) there is a unique magnitude x such that

$$\sum_{i=N+1}^{\infty} x_i b^{-i} = x, \quad (2.3.1.1)$$

and (2.3.1.3) if M is any integer such that $x_i = 0$ for $i \leq M$, then

$$(2.3.1.4) \quad \sum_{i=M+1}^{\infty} x_i b^{-i} = x.$$

$\{x_i\}$ will be said to be a base- b (digital) representation of x , and we shall write

$$(2.3.1.5) \quad x = \{x_i\}_b.$$

2.3.2 Theo. Let b be a base and let x be a magnitude. Then x has at least one base- b representation.

The proofs of these theorems depend very directly on 2.1 and 2.2.

2.3.3 Theo. If $\{x_i\}_b$ and $\{y_i\}_b$ are distinct base- b digital representations of the magnitude x , then there exists an integer P such that (a) $x_i = y_i$ for $i < P$ and (b) by proper choice of notation, $x_P = y_P + 1$ and $x_i = 0$ and $y_i = b - 1$ for $i > P$.

Proof. Let N_x (resp. N_y) be an integer such that $x_i = 0$ for $i \leq N_x$ (resp. $y_i = 0$ for $i \leq N_y$). Let $N = \min(N_x, N_y)$, so that $x_i = y_i = 0$ for $i \leq N$ and

$$(2.3.3.1) \quad x = \sum_{i=N+1}^{\infty} x_i b^{-i} = \sum_{i=N+1}^{\infty} y_i b^{-i}$$

Since by hypothesis there is some value of i for which $x_i \neq y_i$, there must be a least integer P (which is also $> N$) such that $x_P \neq y_P$. If necessary change the notation so that $x_P > y_P$. Subtracting the series in (2.3.3.1) and using $x_i = y_i$ for $N < i < P$,

$$(2.3.3.2) \quad 0 = (x_P - y_P) b^{-P} + \sum_{i=P+1}^{\infty} (x_i - y_i) b^{-i};$$

whence, since $x_P > y_P$,

$$(2.3.3.3) \quad 0 < (x_P - y_P) b^{-P} = \sum_{i=P+1}^{\infty} (y_i - x_i) b^{-P}.$$

Since $0 \leq x_i \leq b - 1$ and $0 \leq y_i \leq b - 1$, then $y_i - x_i \leq b - 1$ for every i . Hence

$$(2.3.3.4) \quad \sum_{i=P+1}^{\infty} (y_i - x_i) b^{-P} \leq \sum_{i=P+1}^{\infty} (b - 1) b^{-P} = b^{-P},$$

with the sums equal only if $y_i - x_i = b - 1$ for every $i > P$. Combined with (2.3.3.3) this gives

$$0 < (x_P - y_P) b^{-P} \leq 1 \cdot b^{-P};$$

and since $x_P - y_P$ is an integer, $x_P - y_P = 1$. Hence, from (2.3.3.3,4)

$$(x_P - y_P) b^{-P} = 1 \cdot b^{-P} = \sum_{i=P+1}^{\infty} (y_i - x_i) b^{-P} \leq b^{-P},$$

which shows that the inequality is impossible. Hence for every $i > P$, $y_i = x_i + (b - 1)$, whence $x_i = 0$ and $y_i = b - 1$.

2.3.4 Def. and Theo. If $x = \{x_i\}_b$ then $\{x_i\}_b$ is called a proper base- b representation of x if and only if to every integer i there corresponds an integer j ($> i$) such that $x_j \neq b - 1$. The relation $x = \{x_i\}_b$ between magnitudes and their proper base- b representations is a one-to-one correspondence.

The property stated in the definition of proper representation means that the sequence cannot end in all $(b-1)$'s (e.g., all 9's in the decimal system). It could not be stated that every sequence must end in all 0's, since most numbers (e.g., $\sqrt{2}$) cannot be thus represented.

2.4 Associated with base- b digital representations are certain base- b digital notations applicable to numbers each of which can be specified in some way by only a finite set of digits. We assume some fixed base b throughout. 2.4.

2.4.1 Def. If $x = \{x_i\}_b$, if N and P are integers such that $N < P$ and $N \leq 0 \leq P$, and if $x_i = 0$ for $i \leq N$ and for $i > P$, then we shall say that

$$(2.4.1.1) \quad (x_{N+1} \cdots x_0 \cdot x_1 \cdots x_P)_b$$

denotes x , and we shall write $x = (x_{N+1} \cdots x_0, x_1 \cdots x_P)_b$. The period between digits x_0 and x_1 is called the basal (binal, ternal, etc., for $b = 2, 3, \dots$)

point. The ordinal position indicated by the subscript i , $N < i \leq P$, is called i th basal place. x is said to be denoted as a $(P - N)$ -place basal magnitude (or number).

If $P = 0$ the basal point is often omitted. In denoting a particular number - i.e., where actual digit symbols (0, 1, etc.) are used - the parentheses are omitted. If the base is prescribed in advance the symbol for it may be omitted. (To avoid ambiguity we shall assume every particular base b denoted by the decimal system without subscript.) Thus $0112.0_3 = 112_3 = 1 \cdot 3^2 + 1 \cdot 3^1 + 2 \cdot 3^0 = 14_{10}$.

2.4.2 Def. If $\{x_i\}_b$ has the property that there exist an integer $P (\geq 0)$ and a least positive integer Q such that if $i > P$, $j > P$ and $j \equiv i, \text{ mod } Q$, then $x_j = x_i$, then we shall denote $x = \{x_i\}_b$ by $(x_{N+1} \cdots x_0 \cdot x_1 \cdots x_P \dot{x}_{P+1} \cdots \dot{x}_{P+Q})_b$, where $x_i = 0$ for $i \leq N$. Magnitudes thus expressed are often called repeating or cyclic basal fractions. For example,

$$0.3_{10} = 1/3; 1.\dot{1}\dot{4}\dot{2}\dot{8}\dot{5}\dot{6}_{10} = 8/7; 1.\dot{0}\dot{1}_2 = 4/3, \text{ etc.}$$

Though it is outside the purpose of this chapter, the reader may be interested in proving that $\{x_i\}_b$ has this form if and only if it represents a rational magnitude x . Thus this extension of the notation in 2.4.1 permits us to denote every rational magnitude by a finite set of digits, choosing N and P suitably. But there is no finite cardinal number $P-N$ such that every rational number can be expressed in this way by $P-N$ digits, even allowing infinitely many choices of N .

3 Addition and Subtraction of Magnitudes, base b

3.1 We shall restrict ourselves in this section to magnitudes denoted as in 2.4.1, with b , N and P fixed. That is, the base of the notation, the number of places and the location of the basal point are all fixed. We shall write

$$x = (x_{N+1} \cdots x_0 \cdot x_1 \cdots x_P)_b$$

without further explicit statement of the hypotheses implicit in the definitions of b , N , P and the x_i 's for $N \leq i \leq P$.

3.2 One of the principal advantages of such a notation lies in the simplification of the actual performance of indicated arithmetical operations. For example, using decimal notation for the moment, the sum of 4952 and 1986 is known from the fundamental properties of the number system to be unique and to be expressible as $4952 + 1986$. But this does not tell us which number it is. The theory does tell us that it is the 4952nd integer after 1986 (or the 1986th integer after 4952), but this is clearly

impractical as a means of identifying the number. But when we add digitwise, as in ordinary arithmetic:

$$\begin{array}{r} 24952 \\ 35986 \\ \hline 60938 \end{array},$$

we find with much less effort a decimal notation for the sum that identifies the sum uniquely. (In the case of the decimal system we can also read the "name" of the number from the notation, if the number of places is not too large.)

The essence of the procedure lies in two ideas. First, the base b is small enough so that the sum of any two digits (originally found by a counting process) can be memorized (by a human) or built in (to a machine). Thus the sum of each pair of digits (one from each summand) in a given place can be determined independently for each place. The list of the sums of all possible pairs of base- b digits is the base- b addition table. Secondly, since some of the pairs of digits yield sums $\geq b$, we have the phenomenon of "carrying" from one place to its predecessor. These ideas will be made precise for an arbitrary base in the following definitions and theorems.

.3.2.1 Theo. and Defs. If x and y are digits, then $x + y$ is uniquely expressible in the form $c \cdot b + w$, where c and w are digits. Furthermore, either $c = 0$ and $0 \leq w \leq b - 1$, or $c = 1$ and $0 \leq w \leq b - 2$. w is called the sum digit and c the carry digit of $x + y$. If $c = 1$, $x + y$ is said to generate a carry.

Proof. Since $0 \leq x \leq b - 1$ and $0 \leq y \leq b - 1$, then $0 \leq x + y \leq 2(b - 1) = b + (b - 2)$. Hence, either $0 \leq x + y \leq b - 1$ and $x + y = 0 \cdot b + (x + y)$, so $c = 0$, $0 \leq w = x + y \leq b - 1$, or $b \leq x + y \leq b + (b - 2)$ and $x + y = 1 \cdot b + (x + y - b)$, so $c = 1$, $0 \leq w = x + y - b \leq b - 2$.

.3.2.2 Theo. and Defs. Let x , y and c_I be digits, with $c_I = 0$ or 1 . Then $x + y + c_I$ is uniquely expressible in the form $c_0 \cdot b + z$, where c_0 and z are digits. Furthermore $c_0 = 0$ or 1 and $0 \leq z \leq b - 1$. c_I is called the carry-in digit, c_0 the carry-out digit and z the sum digit for the sum $x + y + c_I$. If $x + y = b - 1$ and $c_I = 1$, then $c_0 = 1$ and $x + y + c_I$ is said to propagate a carry.

An immediate consequence of this theorem is that if $x + y$ generates a carry (and hence $w \leq b - 2$), then $x + y + c_I$ cannot propagate a carry. In using the terminology of .3.2.1 and .3.2.2 we shall usually refer to a carry as generated or propagated by the "place" at which the digits x and y are. Note that $c_0 = 0$ or 1 , and hence can serve as a carry-in digit.

.3.2.3 Theo. and Defs. Let $x = x_{N+1} \cdots x_0 \cdot x_1 \cdots x_P$ and $y = y_{N+1} \cdots y_0 \cdot y_1 \cdots y_P$. Define $c_P = 0$ and let it be the carry-in digit for the P^{th} place. Define c_{i-1} and z_i inductively for $N < i \leq P$ as follows: c_{P-1} and z_P are the carry-out and sum digits for the P^{th} place and $c_{P-1} = 0$ or 1 . For any i such that $N < i < P$ assume that c_{j-1} and z_j have been defined for all j such that $i < j \leq P$ and that $c_{j-1} = 0$ or 1 . Let c_i be the carry-in digit for the i^{th} place. Then c_{i-1} and z_i are the carry-out and sum digits for the i^{th} place and $c_{i-1} = 0$ or 1 . Then either $c_N = 0$ and $z = x + y = z_{N+1} \cdots z_0 \cdot z_1 \cdots z_P$ or $c_N = 1$ and $z = 1 z_{N+1} \cdots z_0 \cdot z_1 \cdots z_P$. In the latter case the sum is said to overflow since its notation requires more than the given fixed number of places.

Proof. $c_P = 0$. Hence

$$\begin{aligned}
 x + y = z &= \sum_{i=N+1}^P (x_i + y_i) b^{-i} = \sum_{i=N+1}^P (x_i + y_i + c_i) b^{-i} - \sum_{i=N+1}^P c_i b^{-i} \\
 &= \sum_{i=N+1}^P (z_i + c_{i-1} b) b^{-i} - \sum_{i=N+1}^{P-1} c_i b^{-i}, \\
 &= c_N b^{-N} + \sum_{i=N+1}^P z_i b^{-i} + \sum_{i=N+2}^P c_{i-1} b^{-(i-1)} + \sum_{i=N+1}^{P-1} c_i b^{-i} \\
 &= c_N b^{-N} + \sum_{i=N+1}^P z_i b^{-i} = \begin{cases} z_{N+1} \cdots z_0 \cdot z_1 \cdots z_P & \text{for } c_N = 0, \\ 1 z_{N+1} \cdots z_0 \cdot z_1 \cdots z_P & \text{for } c_N = 1, \end{cases}
 \end{aligned}$$

since

$$\begin{aligned}
 \sum_{i=N+2}^P c_{i-1} b^{-(i-1)} &= \sum_{j=N+1}^{P-1} c_j b^{-j}, \text{ by setting } j = i - 1, \\
 &= \sum_{i=N+1}^{P-1} c_i b^{-i}.
 \end{aligned}$$

.3.3 Subtraction may be handled similarly. By analogy we have the following definitions and theorems.

.3.3.1 Theo. and Defs. If x and y are digits, then $x - y$ is uniquely expressible in the form $-d \cdot b + w$, where d and w are digits. Furthermore, either $d = 0$ and $0 \leq w \leq b-1$, or $d = 1$ and $1 \leq w \leq b-1$. w is called the difference digit and d the borrow digit of $x - y$. If $d = 1$, $x - y$ is said to generate a borrow.

.3.3.2 Theo. and Defs. Let x , y and d_I be digits, with $d_I = 0$ or 1 . Then $x - y - d_I$ is uniquely expressible in the form $-d_0 \cdot b + z$, where d_0 and z are digits. Furthermore $d_0 = 0$ or 1 and $0 \leq z \leq b-1$. d_I is called the borrow-in digit, d_0 the borrow-out digit and z the difference digit for $x - y - d_I$. If $x - y = 0$ and $d_I = 1$, then $d_0 = 1$ and $x - y - d_I$ is said to propagate a borrow. If $x - y$ generates a borrow, $x - y - d_I$ cannot propagate a borrow.

.3.3.3 Theo. and Defs. Given x and y as in .3.2.3, define $d_P = 0$ and let it be the borrow-in digit for the P th place. Define d_{i-1} and z_i inductively for $N < i \leq P$ as follows: d_{P-1} and z_P are the borrow-out and difference digits for the P th place and $d_{P-1} = 0$ or 1 . For any i such that $N < i < P$ assume that d_{j-1} and z_j have been defined for j such that $i < j \leq P$, and that each $d_{j-1} = 0$ or 1 . Let d_i be the borrow-in digit for the i th place. Then d_{i-1} and z_i are the borrow-out and difference digits for the i th place and $d_{i-1} = 0$ or 1 . Then either $d_N = 0$ and $z = x - y = z_{N+1} \cdots z_0 \cdot z_1 \cdots z_P$, or $d_N = 1$, $x < y$ and $x - y$ is not a magnitude.

.4 Signed magnitude representation of real numbers.

From algebra we take the definition of $|x|$ as x if $x \geq 0$, $-x$ if $x < 0$ and the notation $x = \xi |x|$, where ξ is the symbol "+" if $x \geq 0$, the symbol "-" if $x < 0$. If x is a real number, $|x|$ is a magnitude. Hence we may give the following definition.

.4.1 Def. Let x be any real number, and let $x = \xi |x|$: Let $\{x_i\}_b$ be a base- b digital representation (proper or improper) of $|x|$. Then $\xi \{x_i\}_b$ will be called a (proper or improper) signed magnitude base- b digital representation of x .

Zero has just two such representations, both proper. If $x \neq 0$ it has just one proper representation, and if it has more than one representation it has two, of which the second is improper.

.4.2 We shall assume signed magnitude digital notations for real numbers defined in strict analogy with the digital notations for magnitudes; e.g., $+1.5_{10} = 3/2$; $-1.\dot{1}428\dot{5}7_{10} = -8/7$. Note that a sign place is necessary.

.4.3 Addition and subtraction. From algebra we know that if $x = \xi |x|$ and $y = \eta |y|$, then $z = x + y$ is similarly expressible as $\zeta |z|$, but that three cases can arise: (a) If $\xi = \eta$ then $\zeta = \xi$ and $|z| = |x| + |y|$. (b) If $\xi \neq \eta$ and $|x| \geq |y|$, then $\zeta = \xi$ and $|z| = |x| - |y|$. (c) If $\xi \neq \eta$ and $|x| \leq |y|$ then $\zeta = \eta$ and $|z| = |y| - |x|$. [The single case, $\xi \neq \eta$ and $|x| = |y|$, has purposely been subsumed under both (b) and (c).] This sort of procedure is poorly adapted to mechanisms since in cases (b) and (c) it would require that one of the possibilities, say $|x| - |y|$, be

attempted to find out the relative magnitude of $|x|$ and $|y|$ before it can be known whether $|x| - |y|$ or $|y| - |x|$ is actually to be performed. A less serious objection is that mechanisms for digit-wise addition and subtraction must be provided. The latter objection is more serious if these mechanisms must be provided for every one of the $P - N$ places (as in "parallel" machines) than if only one of each is used (as in "serial" machines).

The net result is that no automatic machine consistently uses signed magnitude notation in the actual performance of its arithmetic. For example, both UNIVAC I, which uses the decimal signed magnitude notation, and IBM 704, which uses the binal signed magnitude notation, use these in denoting numbers stored in the machine, but use a mixture of signed magnitude and (presently to be defined) complement notations in carrying out addition and subtraction of signed numbers.

.5 Base-b complement representation of real numbers.

.5.1 Let us return for a moment to subtraction of magnitudes in decimal notation, e.g., $24952 - 35986$, which cannot be expressed as a magnitude. However, by .3.3.3, $24952 - 35986 = -1 \cdot 10^5 + 88966 = -11034$. Here we have used 5 decimal places, with the decimal point after the last place. Suppose that we had used an additional place at the right. We would then have had $24952.0 - 35986 = -1 \cdot 10^5 + 88966.0 = -11034.0$. Adding further places to the right results in adding further 0's to each term. Note, however, that the magnitudes formed by the difference digits (88966, 88966.0, etc.) bear a simple relation in each case to the magnitudes of the corresponding absolute values of the difference (11034, 11034.0 etc.). The relation may be expressed as follows: given either one of a corresponding pair, the other may be formed by replacing each digit x_i by $9 - x_i$, and then adding 1 in the last place to the magnitude thus formed. Thus $88966 \rightarrow 11033 + 1$, $11034 \rightarrow 88965 + 1$, $88966.0 \rightarrow 11033.9 + 0.1$, $11034.0 \rightarrow 88965.9 + 0.1$, etc.

Now suppose we add places at the left. Then we find that $024952.0 \cdots 0 - 035986.0 \cdots 0 = -10^6 + 988966.0 \cdots 0 = -011033.9 \cdots 9 + 0.0 \cdots 1$, $0024952.0 \cdots 0 - 0035986.0 \cdots 0 = -10^7 + 9988966.0 \cdots 0 = -0011033.9 \cdots 9 + 0.0 \cdots 1$, etc. Note that if at least one extra place to the left is used, then the apparently positive number that corresponds to the actually negative result begins with 9. This gives the possibility of devising a representation of negative numbers that begins with an infinite sequence of base-b digits, all equal to 0 for positive numbers and all equal to $b-1$ for negative numbers. It will appear that if no distinction is made between proper and improper representations then there is essentially one system. However, proper representations of negative numbers can be defined in two ways which give rise to two different notations.

.5.2 Def. For given base b a two-way sequence $\{x_i\}$ of base- b digits will be called suitable if and only if there exists an integer N such that (a) $x_i = x_N$ for $i < N$ and (b) $x_N = 0$ or $x_N = b-1$.

.5.2.1 Theo. and Def. If b is any base and $\{x_i\}$ is any suitable two-way sequence of base- b digits, then

$$(.5.2.1.1) \quad -\frac{x_N}{b-1} b^{-N} + \sum_{i=N+1}^{\infty} x_i b^{-i}$$

converges to a real number x independently of N , provided $a_i = a_N$ for $i < N$. Furthermore $x \geq 0$ or ≤ 0 according as $x_N = 0$ or $x_N = b-1$. Then we shall call $\{x_i\}$ a base- b complement representation of x , and shall write $x = \{x_i\} b$.

Proof. That the series converges is obvious. Let $M < N$.

Then

$$\begin{aligned} & -\frac{x_M}{b-1} b^{-M} + \sum_{i=M+1}^N x_i b^{-i} = x_N \left(-\frac{b^{-M}}{b-1} + \sum_{i=M+1}^N b^{-i} \right) \\ & = x_N \left(-\frac{b^{-M}}{b-1} + \frac{b^{-M} - b^{-N}}{b-1} \right) = -\frac{x_N}{b-1} b^{-N}, \end{aligned}$$

so the sum is independent of N , provided $x_i = x_N$ for $i < N$. Also,

$$\sum_{i=N+1}^{\infty} x_i b^{-i} \leq \sum_{i=N+1}^{\infty} (b-1) b^{-i} = (b-1) \frac{b^{-N}}{b-1} = b^{-N}.$$

Hence, if $x_N = 0$, then $0 \leq x \leq b^{-N}$, while if $x_N = b-1$, then $x \leq -b^{-N} + b^{-N} = 0$.

Note that, since $\sum_{i=N+1}^{\infty} b^{-i} = -b^{-N}/(b-1)$, we may write (.5.2.1.1) in the form

$$(.5.2.1.2) \quad \sum_{i=N+1}^{\infty} (x_i - x_N) b^{-i}.$$

.5.2.2 Def. If x is a base- b digit, the base- b digit $b-1 - x$ is called the base- b complement of x and will be denoted by $\bar{x}$.

Clearly, $\overline{\bar{x}} = x$ and $\overline{\bar{x} - \bar{y}} = \overline{-(y - x)}$.

.5.2.3 Theo. If b is any base and x is any real number, then there exists at least one base- b complement representation of x .

Proof. Let $x = \xi |x|$, and let $\{x_i\}_b$ be a base- b digital representation of the magnitude $|x|$. If $\xi = +$, then $\{x_i\}_b$ is suitable (having $x_N = 0$) and $x = \{x_i\}_b$. If $\xi = -$, then $\{\bar{x}_i\}_b$ is suitable (having $\bar{x}_N = b - 1$). Then

$$\begin{aligned} -\frac{\bar{x}_N}{b-1} \cdot b^{-N} + \sum_{i=N+1}^{\infty} \bar{x}_i b^{-i} &= \sum_{i=N+1}^{\infty} (\bar{x}_i - \bar{x}_N) b^{-i} \\ &= \sum_{i=N+1}^{\infty} (x_i - x_N) b^{-i} = -|x| = x. \end{aligned}$$

.5.2.3.1 Cor. If $x = \{x_i\}_b$, then $-x = \{\bar{x}_i\}_b$.

Now let x be a magnitude >0 , let $+x = \{x_i\}_b$ and let there be an integer P such that $x_P \neq 0$ and $x_i = 0$ for $i > P$. Then $+x$ also has the representation $\{y_i\}_b$, where $y_i = x_i$ for $i < P$, $y_P = x_P - 1$ and $y_i = b - 1$ for $i > P$. It will be consistent with our previous definitions to define $\{x_i\}_b$ to be proper and $\{y_i\}_b$ to be improper. Further, $-x$ also has two representations, $\{\bar{x}_i\}_b$ and $\{\bar{y}_i\}_b$. If we define $\{\bar{y}_i\}_b$ to be the proper representation of $-x$, then this will have the advantage that

$$-x = -\frac{\bar{y}_N}{b-1} b^{-P} + \sum_{i=N+1}^P \bar{y}_i b^{-i} = \sum_{i=N+1}^P (\bar{y}_i - \bar{y}_N) b^{-i}, \quad \bar{y}_N = \bar{x}_N = b - 1,$$

but it will have the disadvantage that

$$\begin{aligned} -x &= \sum_{i=N+1}^{P-1} (\bar{x}_i - \bar{x}_N) b^{-i} + (\overline{x_P - 1} - \bar{x}_N) b^{-P} \\ &= \sum_{i=N+1}^P (\bar{x}_i - \bar{x}_N) + b^{-P}, \quad \text{since } \overline{x_P - 1} = b - 1 - x_P + 1 = \bar{x}_P + 1, \\ &= \sum_{i=N+1}^{\infty} (\bar{x}_i - \bar{x}_N) + b^{-P}, \quad \text{since } x_i = x_N = 0 \text{ for } i > P, \\ &= \{\bar{x}_i\}_b + b^{-P}. \end{aligned}$$

Thus, under this definition, the proper representation of $-x$ is equal to b^{-P} plus the complement of the proper representation of $+x$.

On the other hand, if we define $\{\bar{x}_i\}_b$ as the proper representation of $-x$, then this representation ends as well as begins with all $(b-1)$'s.

Both schemes are used. We shall treat each of them.

.6 b-complement digital representation.

.6.1 Def. We shall say that a base- b complement representation $\{x_i\}_b$ of x is a b -complement representation of x if $\{x_i\}_b$ is defined to be proper if and only if to every integer i there corresponds an integer j such that $x_j \neq b-1$.

.6.2 Def. If $\{x_i\}_b$ is a b -complement representation of a real number x , and if there exist integers N and P such that (a) $N < P$, (b) $N \leq 0 \leq P$, (c) $x_i = x_N$ for $i < N$ and (d) $x_i = 0$ for $i \geq P$, then we shall say that

$$(x_N \cdots x_0 \cdot x_1 \cdots x_P)_b$$

is the b-complement notation for x and shall write

$$x = (x_N \cdots x_0 \cdot x_1 \cdots x_P)_b .$$

(Assume remarks in .2.4.1 applied suitably.) x_N is called the sign digit.

In the remainder of .6 we assume b given and that $x = x_N \cdots x_P$ means that $(x_N \cdots x_0 \cdot x_1 \cdots x_P)_b$ is the b -complement notation for x .

.6.3 Theo. If $x = x_N \cdots x_P$, then

$$x = -\frac{x_N}{b-1} b^{-N} + \sum_{i=N+1}^P x_i b^{-i} .$$

Since $x_N = 0$ or $b-1$, we may set $\epsilon = 0$ or 1 accordingly and find

$$x = x_N \cdots x_P = -\epsilon b^{-N} + \sum_{i=N+1}^P x_i b^{-i} = b^{-P} \left[-\epsilon b^{P-N} + \sum_{j=1}^{P-N} u_j b^{P-N-j} \right],$$

which expresses x as a multiple of b^{-P} by an integer. (We have set $j = i - N$, $u_j = x_{j+N}$.) Then

$$0 \leq R = \sum_{j=1}^{P-N} u_j b^{P-N-j} \leq \sum_{j=1}^{P-N} (b-1) b^{P-N-j} = b^{P-N} - 1,$$

and R can assume all and only those values that are integers ≥ 0 and $< b^{P-N}$. Hence if $\epsilon = 0$, $0 \leq x = b^{-P} \cdot R \leq b^{-P} (b^{P-N} - 1) = b^{-N} - b^{-P}$. If $\epsilon = 1$, then $x = b^{-P} (-b^{P-N} + R)$ and

$$b^{-P} (-b^{P-N}) = -b^{-N} \leq x \leq b^{-P} (-b^{P-N} + b^{P-N} - 1) = -b^{-P}.$$

Hence if $x_N = b - 1$, x is negative and consists of all multiples of $-b^{-P}$ by integers from 1 through b^{P-N} , while if $\epsilon = 0$, x is ≥ 0 and consists of all multiples of $+b^{-P}$ by integers from 0 through $b^{P-N} - 1$. If x has one of these $2b^{P-N}$ values it is said to be in range for the given, b , N and P .

Note that 0 is denoted only by $z_N = \dots = z_P = 0$, and thus has 0 as its sign digit. Machines using this notation customarily are designed to treat a number in b -complement notation as positive or negative according as the sign digit is 0 or $b-1$, so in such a machine 0 is treated as though it were positive. To determine whether $x = 0$ or $x \neq 0$ one may test $-x$ to see whether it is positive or negative (in this sense), since if $x \neq 0$, $|x| > 0$ and $-|x|$ has $b-1$ as its sign digit, while if $x = 0$, $|x| = -|x| = 0$, so $-|x|$ has 0 as its sign digit.

.6.4 Theo. and Def. Let $x = x_N \dots x_P = x_{N-1} \cdot x_N \dots x_P$, $y = y_N \dots y_P = y_{N-1} y_N \dots y_P$, with $x_{N-1} = x_N$, $y_{N-1} = y_N$. Define $c_P, \dots, c_{N-2}, z_P, \dots, z_{N-1}$ as in .3.2.3. Then

$$z = x + y = z_{N-1} z_N \dots z_P,$$

where possibly $z_{N-1} \neq z_N$, in which case overflow is said to have occurred.

Proof. Since $x_{N-2} = x_N$ and $y_{N-2} = y_N$ we may write

$$x = -\frac{x_{N-2}}{b-1} b^{-(N-2)} + \sum_{i=N-1}^P x_i b^{-i},$$

with a similar expression for y . Then

$$\begin{aligned} z = x + y &= -\frac{(x_{N-2} + y_{N-2})}{b-1} b^{-(N-2)} + \sum_{i=N-1}^P (x_i + y_i) b^{-i} + c_P b^{-P} \\ &= -\frac{(x_{N-2} + y_{N-2})}{b-1} b^{-(N-2)} + \sum_{i=N-1}^P (x_i + y_i + c_i) b^{-i} - \sum_{i=N-1}^P c_i b^{-i} \\ &= -\frac{(x_{N-2} + y_{N-2})}{b-1} b^{-(N-2)} + \sum_{i=N-1}^P z_i b^{-i} + c_{N-2} b^{-(N-2)} + \sum_{i=N-2}^P c_{i-1} b^{-(i-1)} - \sum_{i=N-1}^{P-1} c_i b^{-i} \\ &= -\left[\frac{x_{N-2} + y_{N-2} - (b-1) c_{N-2}}{b-1} \right] b^{-(N-2)} + \sum_{i=N-1}^P z_i b^{-i} \end{aligned}$$

At stage N various cases can occur, depending on the values of x_N , y_N and c_N . These give rise to various values of z_{N-1} and c_{N-2} . Set $x_{N-2} + y_{N-2} - (b-1)c_{N-2} = c_{N-2}$. The accompanying table shows that in every case, $c_{N-2} = z_{N-1}$, so that $z = z_{N-1} z_N \cdots z_P$, which may be regarded as having been obtained by treating x_{N-1} and y_{N-1} as positive coefficients of $b^{-(N-1)}$ and disregarding the carryout of the $(N-1)^{\text{th}}$ place.

Case	x_N	y_N	c_N	z_N	c_{N-1}	z_{N-1}	c_{N-2}	c_{N-2}	
1	0	0	0	0	0	0	0	0	
2	0	0	1	1	0	0	0	0	
3	0	$b-1$	0	$b-1$	0	$b-1$	0	$b-1$	} similarly for $x_N = b-1, y_N = 0$.
4	0	$b-1$	1	0	1	0	1	0	
5	$b-1$	$b-1$	0	$b-2$	1	$b-1$	1	$b-1$	
6	$b-1$	$b-1$	1	$b-1$	1	$b-1$	1	$b-1$	

Note that overflow ($z_{N-1} \neq z_N$) occurs only when $x_N = y_N$, and that positive overflow is characterized by a carryout of place $N+1$, negative overflow by no carryout of place $N+1$. Also, note that whether or not overflow occurs, z_{N-1} contains the true sign digit of the sum, assuming that x and y were in range; and, since overflow can occur, the extra place is necessary to hold the true sign digit of the result, though if one merely wishes to determine whether overflow has occurred it is sufficient to determine whether or not one of the cases, $x_N = y_N = 0, c_{N-1} = 1$ or $x_N = y_N = b-1, c_{N-1} = 0$, has occurred.

.6.5 Theo. and Def. If $x = x_N \cdots x_P$, then

$$-x = \bar{x} + b^{-P},$$

where $\bar{x}$, the $(b-1)$'s-complement of x , is the number denoted by $\bar{x}_N \cdots \bar{x}_P$. $\bar{x} + b^{-P}$ and is called the b -complement of x .

To form the negative of a number in a machine it is sufficient to provide a mechanism for complementing each digit and adding "1" into the last place (the P^{th}). The second step can be accomplished by inserting a carry into the last position of the adder, or by providing some means of adding 1 to the last position of a register that is designed to propagate any carry that may be generated by the process. The first step (forming $\bar{x}$) requires a mechanism for the digit-wise complementation of a number.

While the procedure of .6.5 is the usual one in a machine, for written computation it is easy to see that it is sufficient to complement each digit from the N^{th} through the place containing the last non-zero digit, and to replace that digit by b minus it. (If all digits are 0, $x = -x = 0$ and no complementation is necessary.) Thus, with base 3, if $x = 010.200 = 11/3$, $-x = 212.100 = -3^2 + 3 + 2 + 3^{-1} = -9 + (16/3) = -(11/3)$.

.6.5.1 To find $x - y$ we can then add $x + \bar{y} + b^{-P}$, i.e., add $x + \bar{y}$ with a carry into place P .

.6.6 Theo. If $x = x_N \cdots x_P$ and $x_N = x_{N+1}$, then $bx = y = y_N \cdots y_P$, where $y_i = x_{i+1}$ for $N \leq i \leq P$, $y_P = 0$. Similarly, if $x = x_N \cdots x_P$ and $x_P = 0$, then $b^{-1}x = z = z_N \cdots z_P$, where $z_N = x_N$ and $z_i = x_{i-1}$ for $1 < i \leq P$.

Thus one can multiply numbers by a power of b by shifting the digits left and inserting 0's at the right end (for positive powers) and by shifting the digits right and inserting sign digits at the left end (for negative powers). Clearly these operations are exact only if the conditions stated in the theorem are met at the beginning of each shift. If they are not met, the result of a left shift will be meaningless as soon as the value of the digit inserted into the N^{th} place changes, while the result of a right shift will be a poorer and poorer approximation of the correct result every time a non-zero digit is shifted out of the P^{th} place.

.6.7 The interested reader might develop the theory of direct subtraction (i.e., digit-wise subtraction with borrowing) as the basic operation instead of addition.

.7 (b-1)'s -complement digital representation.

.7.1 Def. We shall say that a base- b complement representation $\{x_i\}_b$ of x is a (b-1)'s-complement representation of x if $\{x_i\}_b$ is defined to be proper if and only if there does not exist an integer i such that for every $j > i$, $x_j = \bar{x}_N$.

Thus an improper representation is one that begins with solid x_N 's and ends with solid $\bar{x}_N$'s.

.7.2 Def. If $\{x_i\}_b$ is a (b-1)'s-complement representation of a real number x and if there exist integers N and P such that (a) $N < P$, (b) $N \leq 0 \leq P$, and (c) $x_i = x_N$ for $i < N$ and for $i > P$, then we shall write

$$x = (x_N \cdots x_0 \cdot x_1 \cdots x_P)_b$$

and shall call $(x_N \cdots x_0 \cdot x_1 \cdots x_P)_b$ the (b-1)'s-complement notation for x ; x_N is called the sign digit. (Similar remarks to those in .6.2.)

.7.3 Theo. If $x = x_N \cdots x_P$, then

$$x = \sum_{i=N+1}^P (x_i - x_N) b^{-i} = -\frac{x_N}{b-1} \cdot b^{-N} + \sum_{i=N+1}^P x_i b^{-i} + \frac{x_N}{b-1} \cdot b^{-P}.$$

Proof. Since $x_i = x_N$ for $i > P$, all terms $(x_i - x_N) b^{-i}$ are 0 for $i > P$.

In this case also (cf. .6.3) we can set $\epsilon = x_N/(b-1) = 0$ or 1 according as $x_N = 0$ or $b-1$. Then

$$\begin{aligned} x &= -\epsilon(b^{-N} - b^{-P}) + \sum_{i=N+1}^P x_i b^{-i} = b^{-P} \left[-\epsilon(b^{P-N} - 1) + \sum_{j=1}^{P-N} u_j b^{P-N-j} \right] \\ &= b^{-P} \left[-\epsilon(b^{P-N} - 1) + R \right]. \end{aligned}$$

R has the same range as in .6.3, namely, integers such that $0 \leq R < b^{P-N}$. If $\epsilon = 0$, $x = b^{-P} R$, so

$$0 \leq x \leq b^{-P} (b^{P-N} - 1) = b^{-N} - b^{-P}$$

for $x_N = 0$. If $\epsilon = 1$, $x = -b^{-P} (b^{P-N} - 1 - R)$, so

$$-(b^{-N} - b^{-P}) \leq x \leq 0$$

for $x_N = b - 1$. Hence 0 has two notations: $0 \cdots 0$ and $(b-1) \cdots (b-1)$, which we may also denote by $+0$ and -0 , respectively. Every other number denoted is denoted uniquely and their whole set consists of those numbers whose absolute values are multiples of b^{-P} by an integer > 0 and $< b^{P-N}$. (Note that $-b^{-N}$ cannot be denoted.)

.7.4 Theo. and Def. Given x and y as in .6.4, but in $(b-1)$'s-complement notation. Define c_{i-1} and z_i for $N-1 \leq i \leq P$ as before, except that c_P is taken equal to c_{N-2} . (If $c_{N-1} = 1$, this is called an end-around carry.) Then

$$z = x + y = z_{N-1} \cdots z_P.$$

Proof. The proof in .3.2.3 must be modified to account for end-around carry. This is done by starting the induction at a place Q (presently to be defined) and proceeding through the places in cyclic order to the left, so that from place $N-1$ one proceeds to place P , then place $P-1$, etc. Q must be selected so that no carry is propagated through it. If no position can generate a carry, i.e., if for every i , $x_i + y_i < b$, take $Q = P$ and $c_P = 0$. The induction may then proceed to place $N-1$, giving $c_{i-1} = 0$ at every step, and hence yielding $c_P = c_{N-1} = 0$. Under the other alternative, let Q be a place at which a carry is generated. Set $x_Q + y_Q = c_{Q-1} b + w_Q$, where $c_{Q-1} = 1$ and $w_Q < b-1$. The induction may then proceed cyclically until place $Q+1$ is reached, where $x_{Q+1} + y_{Q+1} + c_{Q+1} = c_Q b + z_{Q+1}$. Then $x_Q + y_Q + c_Q = c_{Q-1} b + w_Q + c_Q$, i.e., $z_Q = w_Q + c_Q < b$, so the process terminates, since c_{Q-1} is not affected.

To prove that $z_{N-1} \cdots z_N$ gives the proper result, we proceed as in .6.4. Since now

$$x = -\frac{x_{N-2}}{b-1} (b^{-N+2} - b^{-P}) + \sum_{i=N-1}^P x_i b^{-i},$$

and similarly for y , and $c_P = c_{N-2}$, we find

$$\begin{aligned} z = x + y &= -\frac{x_{N-2} + y_{N-2}}{b-1} (b^{-N+2} - b^{-P}) + \sum_{i=N-1}^P (x_i + y_i + c_i) b^{-i} \\ &\quad - \sum_{i=N-1}^P c_i b^{-i} \\ &= -\frac{x_{N-2} + y_{N-2}}{b-1} (b^{-N+2} - b^{-P}) + \sum_{i=N-1}^P z_i b^{-i} + \sum_{i=N-1}^P c_{i-1} b^{-(i-1)} \\ &\quad - \sum_{i=N-1}^P c_i b^{-i} \\ &= -\frac{x_{N-2} + y_{N-2}}{b-1} (b^{-N+2} - b^{-P}) + z_{N-1} \cdots z_P + z_{N-2} b^{-N+2} \\ &\quad + c_{N-2} b^{-N+2} - c_P b^{-P} \\ &= -\left[\frac{x_{N-2} + y_{N-2} - (b-1) c_{N-2}}{b-1} \right] (b^{-N+2} - b^{-P}) + z_{N-1} \cdots z_P, \end{aligned}$$

since $c_{N-2} = c_P$. The rest of the proof is identical with the rest of the proof in .6.4.

Examination of the table in .6.3 shows that $c_{N-1} = c_{N-2}$ in every case, so if overflow detection only were desired, place $N-1$ could be omitted, with end-round carry from the N^{th} place.

It may be noted that the sum of x and $-x$ is always -0 . If subtraction, using borrows instead of carries, had been used as the fundamental operation, then $x + (-x)$ i.e., $x - x$, would result in $+0$. This idea is used in Remington-Rand's 1103A to prevent the occurrence of -0 as the result of addition or subtraction.

.7.5 Theo. If $x = x_N \dots x_P$ in $(b-1)$'s-complement notation, then $-x = \bar{x}$, where $\bar{x}$ is the $(b-1)$'s-complement of x .

.7.5.1 Theo. In $(b-1)$'s-complement notation, $x - y = x + \bar{y}$.

.7.6 Theo. If $x = x_N \dots x_P$ in $(b-1)$'s-complement notation and $x_{N+1} = x_N$, then $bx = y = y_N \dots y_P$, where $y_i = x_{i+1}$ for $N \leq i < P$, and $y_P = x_N$. Similarly, if $x_P = x_N$, then $b^{-1}x = z = z_N \dots z_P$, where $z_N = x_N$ and $z_i = x_{i-1}$ for $N < i \leq P$.

Proof. From the definition (.7.2) of the notation, $x_i = x_N$ for every $i > P$. Hence sign digits must be inserted at the right on left shifts, as well as at the left on right shifts.

If a number x is multiplied by a sufficiently high negative power of b , the result is $+0$ or -0 , according as x was positive or negative. Thus both forms of 0 may occur in $(b-1)$'s-complement notation.

.8 Miscellaneous

.8.1 From what has been said in the previous sections it is clear that the use of 2 as a base has an additional advantage to that of requiring only by-stable mechanisms. For if $b = 2$, $b-1 = 1$, so the factor $b-1$ may be omitted in the denominator of the first term of the series evaluating x in terms of its digits. Thus if $\{x_i\}_b$ is a binomial complement representation of x ,

$$x = -x_N 2^{-N} + \sum_{i=N+1}^{\infty} x_i 2^{-i};$$

and similarly for 2 -complement and 1 's-complement representations and notations, e.g.,

$$x = -x_N 2^{-N} + \sum_{i=N+1}^P x_i 2^{-i}$$

in 2 -complement notation;

$$x = -x_N (2^{-N} - 2^{-P}) + \sum_{i=N+1}^P x_i 2^{-i} = \sum_{i=N+1}^P (x_i - x_N) b^{-i}$$

in 1 's-complement notation.

.8.2 XS3 notation for decimal digits. In electronic machines that use decimal notation the difficulty of devising a mechanism that will assume just ten stable states is often surmounted by using ten of the sixteen stable states of a mechanism that contains four binomial places, and selecting the ten states to be used as follows: If x is a decimal digit it is denoted in any decimal place by $x + 3$ expressed in binomial form. (Thus there is an "excess 3" in the notation of each decimal digit.)

The following are some of the advantages of this scheme.

(1) If x is a decimal digit and $\xi = x + 3$, its XS3 notation, then $\bar{\xi} = 15 - \xi = (9 - x) + 3 = \bar{x} + 3$, where $\bar{\xi}$ is the complement of ξ in base 16 and $\bar{x}$ is the complement of x in base 10. Thus x is denoted by $\bar{\xi}$ in XS3. (2) If x and y are decimal digits and $\eta = y + 3$, then $x = y \geq 10$ if and only if $\xi = \eta = x + y + 6, \geq 16$. Thus in adding, a carry generated or propagated by $x = +y$ is similarly generated or propagated by $\xi + \eta$.

In carrying out addition of XS3 digits notice that if x, y, z are denoted by ξ, η, ζ , then (a) if $x + y < 10$, then $\xi + \eta = x + y + 6 = [(x + y) + 3] + 3$, so $\zeta = \xi + \eta - 3$, and (b) if $x + y \geq 10$, then $\xi + \eta = x + y + 6 = 1 \cdot 10 + z + 6 = 1 \cdot 16 + z$, so the base-10 sum digit z is also the base-16 sum digit, and hence must be increased by 3 to be denoted properly. Thus in performing an addition, the actual sum, $\xi + \eta + c_I$, is examined to see whether there is a carry-out or not and the base-16 sum digit is then increased or decreased by 3 accordingly. Note that c_I and c_0 have the values 0 or 1 as before.

With such a device decimal addition, complementation and shifting can be accomplished as though each place could be occupied by only 10 distinct digits instead of 16.

.8.3 Addition with signed-magnitude notation. To return to our earlier remark (.4.3) concerning the use of complement notation in performing some of the arithmetic when signed-magnitude notation is ostensibly used, let

$$x = \xi x_{N+1} \cdots x_P$$

in signed magnitude notation. Since ξ can have just two values, we may assume it denoted by 0 for "+" and 1 for "-." Then

$$|x| = 0 x_{N+1} \cdots x_P$$

where the notation is correct for either b -complement or $(b-1)$'s-complement notation, and

$$-x = \bar{\xi} x_{N+1} \cdots x_P,$$

where $\bar{\xi}$ is the complement of the binal digit ξ . Thus change of sign or production of absolute value is readily accomplished. Multiplication by a power of b is accomplished by shifting the magnitude digits, $x_{N+1}, \cdots, x_P$, left or right, and inserting 0's at the right or left (since $x_N = 0$), with the same limitations as for a complement notation; except that on a left shift the sign is unaffected but the magnitude will be as soon as a non-zero digit is shifted out.

We shall now assume that $|x|$ is expressed in $(b-1)$'s-complement notation, for reasons that will appear presently. Let $y = \eta y_{N+1} \cdots y_P$, and let us form the sum of x and y . Let $z = \zeta z_N \cdots z_P$, where z_N is included to permit access to the true result in case of overflow. If $\xi = \eta$, then $\zeta = \xi$ and $0 z_N \cdots z_P$, obtained as in .7.4, is the true value of $|z|$. Thus $z = \xi z_N \cdots z_P$. Note that, since $z_{N-1} = 0$ always, and since the value of ξ has been determined, we do not need to add in the $(N-1)$ th place.

Next let $\xi \neq \eta$. We can now form the $(b-1)$'s-complement of $|x|$ for insertion into the adder, thus forming $-x + y$. Assume that no addition in place $N-1$ is performed. We know that the summands, $-|x|$ and $|y|$, have opposite signs, so there can be no overflow and we have the equivalent of case 3 or 4 of the table in .6.4. The result is then negative or positive according as $c_{N-1} = 0$ or 1. If the result is negative this means that $|x| \geq |y|$, so $\zeta = \xi$, the adder contains $-|z|$, i.e., $1\bar{y}_{N+1} \cdots \bar{y}_P$, and its complement gives $|z|$. If the result is positive, then $|x| \leq |y|$, so $\zeta = \eta$ and the adder contains $|z|$.

The cases may be summed up in the following rules. Given $x = \xi |x|$, $y = \eta |y|$, to find $z = \zeta |z| = x + y$:

- (1) If $\xi = \eta$ then $\zeta = \xi$ and $|z| = |x| + |y|$. Overflow is possible.
- (2) If $\xi \neq \eta$ then form $u = -|x| + |y|$: No overflow is possible.
 - (2a) If $c_{N-1} = 0$ then $\zeta = \xi$ and $|z| = \bar{u}$.
 - (2b) If $c_{N-1} = 1$ then $\zeta = \eta$ and $|z| = u$.

$(b-1)$'s-complement notation was chosen because case 2a requires complementation of $|x|$ going into the adder and of $-|x| + |y|$ coming out, so it is convenient not to have to make the end correction, particularly on the second complementation.

.8.4 We have dealt with the operation of addition, complementation and shifting for the three principal notations. The latter operations provide for subtraction and for multiplication by positive or negative powers of the base. Multiplication and division can be performed by suitable sequences of the fundamental operations, so the operations described may be regarded as the basic ones in the design of any machine.

.9 A Binary Addition Algorithm

Let a and b be two real numbers expressible in 2-complement binary notation by digits in the same finite set of binary places, say $a = a_P a_{P+1} \cdots a_{P+N}$ and $b = b_P b_{P+1} \cdots b_{P+N}$. It has been shown that $-b$ is representable as $\bar{b}_P \bar{b}_{P+1} \cdots \bar{b}_{P+N} + 2^{-(P+N)}$, so that $a-b$ may be found by

the addition of a , $\bar{b}$ and $2^{-(P+N)}$. We may then state the requirements for an addition algorithm to be performed by the machine to be that it give the correct sum in 2-complement binary notation of a , b and $c_{P+N} 2^{-(P+N)}$, where c_{P+N} is a bigit.

Lemma. If a_i , b_i and c_i are bigits, then there exist unique bigits c_{i-1} and s_i such that $a_i + b_i + c_i = 2c_{i-1} + s_i$. Furthermore, if we regard the symbols a_i , etc., as being Boolean variables in B_0 , $c_{i-1} = a_i b_i \cup b_i c_i \cup c_i a_i$ and $s_i = a_i \oplus b_i \oplus c_i$.

Def. Given a , b and c_{P+N} as above, we define c_{P+i-1} and s_{P+i} for $N \geq i \geq 0$ inductively as follows: For $i = N$, c_{P+i-1} and s_{P+i} are the unique bigits satisfying the relation $a_{P+N} + b_{P+N} + c_{P+N} = 2c_{P+N-1} + s_{P+N}$ by the lemma. For $N > i \geq 0$ assume that bigits c_{P+j-1} and s_{P+j} satisfying $a_{P+j} + b_{P+j} + c_{P+j} = 2c_{P+j-1} + s_{P+j}$ have been defined for every j such that $N \geq j > i$. Then c_{P+i-1} and s_{P+i} are defined as the unique bigits satisfying $a_{P+i} + b_{P+i} + c_{P+i} = 2c_{P+i-1} + s_{P+i}$ by the lemma.

Theorem. Given a , b , c_{P+N} and c_{P+i-1} and s_{P+i} for $N \geq i \geq 0$ as in the preceding definition, then $a + b + c_{P+N} 2^{-(P+N)} = s$, where $s = s_{P-1} s_P \dots s_{P+N}$ and s_{P-1} is the bigit uniquely determined by $a_P + b_P + c_{P-1} = 2c_{P-2} + s_{P-1}$ by the lemma.

Proof. Since a may also be written as $a_{P-1} a_P \dots a_{P+N}$ with $a_{P-1} = a_P$, and similarly for b , we have

$$\begin{aligned}
 a + b + c_{P+N} 2^{-(P+N)} &= -a_{P-1} 2^{-(P-1)} + \sum_{i=0}^N a_{P+i} 2^{-(P+i)} \\
 &\quad - b_{P-1} 2^{-(P-1)} + \sum_{i=0}^N b_{P+i} 2^{-(P+i)} + c_{P+N} 2^{-(P+N)} \\
 &= -(a_{P-1} + b_{P-1}) 2^{-(P-1)} + \sum_{i=0}^N (a_{P+i} + b_{P+i}) 2^{-(P+i)} + c_{P+N} 2^{-(P+N)} \\
 &= -(a_{P-1} + b_{P-1}) 2^{-(P-1)} + \sum_{i=0}^N (2c_{P+i-1} + s_{P+i} - c_{P+i}) 2^{-(P+i)} + c_{P+N} 2^{-(P+N)} \\
 &= -(a_{P-1} + b_{P-1} - c_{P-1}) 2^{-(P-1)} + \sum_{i=1}^N c_{P+i-1} 2^{-(P+i-1)} \\
 &\quad + \sum_{i=0}^N s_{P+i} 2^{-(P+i)} - \sum_{i=0}^{N-1} c_{P+i} 2^{-(P+i)} \\
 &= -(c_{P-2} - c_{P-1}) 2^{-(P-2)} - s_{P-1} 2^{-(P-1)} + \sum_{i=0}^N s_{P+i} 2^{-(P+i)}
 \end{aligned}$$

Now $c_{p-2} - c_{p-1} = 0$. For $c_{p-1} = apbp \cup bpcp \cup cpap$ and
 $c_{p-2} = ap_{-1}bp_{-1} \cup bp_{-1}cp_{-1} \cup cp_{-1}ap_{-1} = apbp \cup bpcp_{-1} \cup cp_{-1}ap$
 $= apbp \cup bp(apbp \cup bpcp \cup cpap) \cup ap(apbp \cup bpcp \cup cpap)$
 $= apbp \cup bpcp \cup cpap = c_{p-1}$.

Hence $s = s_{p-1} s_p \cdots s_{p+N}$.

This indicates that if an adder is to give correctly the sum of two $(N+1)$ -bit numbers it should have $N+2$ identical stages, each with three inputs, a_j , b_j and c_j , and two outputs, c_{j+1} and s_j , as determined from the inputs by the lemma, though the output for the stage with lowest index may serve no function. Any sum thus formed from in-range numbers will be in range only if the s_j outputs of the two stages with lowest index agree.

.10 Binary Square Root Algorithm

Given x such that $0 \leq x < 1$, then for each integer (≥ 0) there exists a unique y_{n-1} such that

$$(\alpha) y_{n-1}^2 \leq x < [y_{n-1} + 2^{-(n-1)}]^2,$$

and

$$(\beta) y_{n-1} = h_{n-1} \cdot 2^{-(n-1)},$$

where h_{n-1} is a non-negative integer.

For each n define

$$r_{n-1} = 2^{n-1} (x - y_{n-1}^2),$$

$$s_n = 2r_{n-1}$$

$$t_n = 2y_{n-1} + 2^{-n},$$

$$d_n = s_n - t_n.$$

Note that

$$\begin{aligned} d_n &= 2^n (x - y_{n-1}^2) - 2y_{n-1} - 2^{-n} \\ &= 2^n [x - (y_{n-1}^2 + 2y_{n-1} \cdot 2^{-n} + 2^{-2n})] \\ &= 2^n [x - (y_{n-1} + 2^{-n})^2]. \end{aligned}$$

If

$$(1) d_n < 0,$$

then

$$(\alpha) \quad y_{n-1}^2 \leq x < (y_{n-1} + 2^{-n})^2$$

and

$$(\beta) \quad y_{n-1} = h_{n-1} \cdot 2^{-(n-1)} = 2h_{n-1} \cdot 2^{-n} = h_n \cdot 2^{-n} ,$$

where h_n is a non-negative integer. Hence

$$y_n = y_{n-1} .$$

Hence, also,

$$r_n = 2^n (x - y_n^2) = 2 \cdot 2^{n-1} (x - y_{n-1}^2) = 2r_{n-1} = s_n ,$$

$$\begin{aligned} t_{n+1} &= 2y_n + 2^{-(n+1)} = 2y_{n-1} + 2^{-n} - 2^{-n} + 2^{-(n+1)} \\ &= t_n - 2^{-n} + 2^{-(n+1)} . \end{aligned}$$

If

$$(2) \quad d_n \geq 0 ,$$

then

$$(\alpha) \quad (y_{n-1} + 2^{-n})^2 \leq x < [y_{n-1} + 2^{-(n-1)}]^2 = [(y_{n-1} + 2^{-n}) + 2^{-n}]^2$$

and

$$(\beta) \quad y_{n-1} + 2^{-n} = h_{n-1} \cdot 2^{-(n-1)} + 2^{-n} = (2h_{n-1} + 1) \cdot 2^{-n} = h_n \cdot 2^{-n} ,$$

where h_n is a non-negative integer. Hence

$$y_n = y_{n-1} + 2^{-n} .$$

Hence also

$$\begin{aligned} r_n &= 2^n (x - y_n^2) = 2^n [x - (y_{n-1} + 2^{-n})^2] = d_n , \\ t_{n+1} &= 2y_n + 2^{-(n+1)} = 2y_{n-1} + 2^{-(n-1)} + 2^{-(n+1)} \\ &= 2y_{n-1} + 2^{-n} + 2^{-(n-1)} - 2^{-n} + 2^{-(n+1)} \\ &= t_n + 2^{-(n-1)} - 2^{-n} + 2^{-(n+1)} . \end{aligned}$$

Thus we see that, according as (1) or (2) holds,

$$y_n = y_{n-1} \text{ or } y_{n-1} + 2^{-n} ,$$

$$r_n = s_n \text{ or } d_n$$

$$t_{n+1} = \left[t_n \text{ or } t_n + 2^{-(n-1)} \right] - 2^{-n} + 2^{-(n+1)} .$$

Always,

$$s_{n+1} = 2r_n ,$$

$$d_{n+1} = s_{n+1} - t_{n+1} .$$

Now for $n \leq 0$ we have $n-1 < 0$, $-(n-1) > 0$ and

$$0 \leq x < 1 < \left[2^{-(n-1)} \right]^2 ,$$

so

$$y_{n-1} = 0 \text{ for } n \leq 0.$$

Let $n = 0$. Then

$$y_{-1} = 0 ,$$

$$r_{-1} = 2^{-1} (x - 0^2) = 2^{-1} x ,$$

$$s_0 = 2r_{-1} = x ,$$

$$t_0 = 2y_{-1} + 2^0 = 2^0 = 1 ,$$

$$d_0 = s_0 - t_0 = x - 1 < 0 .$$

Hence

$$y_0 = y_{-1} = 0 ,$$

$$r_0 = s_0 = x ,$$

$$s_1 = 2r_0 = 2x ,$$

$$t_1 = t_0 - 2^{-0} + 2^{-1} = 2^{-1} .$$

$$d_1 = s_1 - t_1 = 2x - 2^{-1} .$$

Thus we may start with MR cleared to 1, = 01.0...0 = t_0 and $s_0 (= x = a_0 \cdot a_1 \cdots a_{39} q_0 \cdots q_{38})$ in (Ac, Qt). It will appear that q_{39} is irrelevant and that y_{-1} need not be stored since $y_{-1} = y_0$.

Now for $0 \leq n < 39$ let MR contain $t_n (= 2y_{n-1} + 2^{-n})$ and (Ac, Qt) contain $s_n + 2^{-[78-(n-1)]} y_{n-1}$, i.e., $m_x = b'_0$, $m_0 = b'_1$, $\dots m_{n-2} = b'_{n-1}$, $m_{n-1} = 0$, $m_n = 1$, where b'_i is the i th digit of y_{n-1} for $0 < i < n-1$; and $a_i = s'_i$ for $-1 < i < 39$, $q_{i-40} = s_i$ for $40 \leq i \leq 78-n$, $q_{39-n+j} = b'_j$ for $0 \leq j \leq n-1$, where s_i and b'_j are respectively the i th and j th digits of s_n and y_{n-1} . Thus for $n = 0$ we have $m_x = 0$, $m_0 = 1$, $q_{38} = a_{78}$, but q_{39} is irrelevant. Let $s_n = s_n^* + q_n^{**} \cdot 2^{-40}$, where s_n^* is the part of s_n in Ac, and let $d_n^* = s_n^* - t_n$. Then $d_n = s_n - t_n = s_n^* - t_n + s_n^{**} \cdot 2^{-40} = d_n^* + s_n^{**} \cdot 2^{-40}$, so $\text{sgn } d_n = \text{sgn } d_n^*$. Form d_n^* in P. According as $\text{sgn } d_n^*$ is 1 or 0 do Up A or In A, and do Up Q in any case, except that $Q^{39} = 0$ or 1 according as Up A or In A occurred. Follow with DL of A and Q. Hence, according as $d_n < 0$ or > 0 we have $(s_n \text{ or } d_n) + 2^{-[78-(n-1)]} y_n = r_n + 2^{-[78-(n-1)]} y_n$ in (A^U, Q^U) and $s_{n+1} + 2^{-[78-n]} y_n$ in (A_L, Q_L) , with Q_{39} irrelevant before and after. As the transfers into A^U and Q^U occur, set m_n to 0, m_{n+1} to 1 and set m_{n-1} to 0 or 1 according as $d_n < 0$ or ≥ 0 . We then have t_{n+1} in M.

For $n = 39$ the hypotheses and operations are as before except that Ac and Qt go straight down, so that (Ac, Qt) contains $r_{39} + 2^{-40} y_{39}$, so r_{39} lies in Ac and y_{39} in Qt. (Q_{39} now is the last bit of y_{39} .) Also we do not need to alter M, but any convenient operation on M may be performed, since the resulting contents are not used. We thus have the truncated value of y to 39 places in Qt, with the true remainder of $a - y_{39}^2$ in Ac.

APPENDIX C

GEORGE Logic

by

Donald A. Flanders

a. Overall Survey

.1 Types of information

An automatic digital computing machine such as GEORGE may be regarded as a machine for the processing of a certain type of material, alphabetical or numerical (hereafter called alpha numerical) information, from the raw material, the data of a problem, to the finished product, which may again be alpha numerical or graphical. To carry out the process in an automatic fashion there must also be given certain instructional information, since the machine is versatile and capable of producing a large variety of finished products from a given mass of raw material. This instructional information, together with functions of it formed by the machine, we shall call the control information.

.2 The flow of material information

We assume the raw material in one of the following four forms, which will usually have been produced by equipment not part of the machine itself: (1) punched holes on 7-channel teletype tape, which we shall refer to as paper tape, PA; (2) magnetized areas on 8-channel magnetic tape, which we shall refer to as narrow tape, N; (3) magnetized areas on 42-channel magnetic tape, which we shall refer to as wide tape, W; and (4) the successively depressed keys of a keyboard, KB. As we shall see, PA and N are used for holding the finished product, and W usually, and PA and N occasionally, will or may be used for storage. Somewhat arbitrarily we shall consider these tapes as containing basic raw material only if the material information they contain was placed on them by machines other than GEORGE itself. The finished product will be similarly recorded on PA or N (possibly also on W), as graphs or alpha numerical characters visible on the face of a cathode ray tube, VC, or caught on a photographic film from a similar cathode ray tube, PC, or as characters, printed on paper by a typewriter TW on the console.

The units of GEORGE that process material are the following: the paper tape reader PR, the narrow tape reader NR, the keyboard KB, the accumulator register Ac, the quotient register Qt, the adder P, the selector SE, the core storage CS, the memory register MR, the wide tape storage WT, the visual and photographic cathode ray tubes VC and PC

FIGURE I
MATERIAL PROCESSING AND STORAGE

(sometimes lumped simply as the cathode ray tube CRT), the narrow tape output NO, the paper tape punch PH and the console typewriter TW. In spite of the ambiguities noted above as to what is input, output or storage we shall consider that PR, NR and KB form the input equipment, MR, Ac, Qt, P, and SE form the arithmetic unit AU, CS and W form the storage, and CRT, NO, PH and TW the output equipment. All of these will be said to form the computer.

The accompanying figure is a block diagram indicating the possible channels through which material may flow among the various units. Clearly the material may not flow at will in these channels. To produce a desired finished product, the flow must be controlled. These controls have purposely been omitted from the diagram. The channels themselves are of three kinds: parallel (represented by solid lines), in which the material flows in parallel as 40-bit words along 40 separate channels; serio-parallel (represented by dot-dash lines), in which the material flows as a succession of 4-bit or 6-bit characters along six parallel channels; and serial (represented by dotted lines), in which the flow is bit by bit along a single channel.

.3 The flow of control information

Control information comes into the machine by the same channels as material information, as well as from manually operated switches on the console, CN. At various times some of it is treated as though it were material information and goes along the same channels through the various material processing units mentioned in .2. Mostly, however, it passes through various control units and through smaller control mechanisms, and enters the material processing units by separate channels to perform its control functions. Because of the complexity of the control mechanism and its interactions with the material information, it scarcely seems feasible to draw a diagram of the control channels that approaches the completeness of the diagram of the material channels. However, Fig. .2 gives a simplified version of the channels connecting the most important control mechanisms.

In the diagram are shown some of the channels connecting MR with the function register FR, the address adder AA, the dispatch register DR, the shift counter SC, the control counter CC, the static programmer SP, the memory and order sequence control MOSC, the dynamic programmer DP, the clear-transfer selector CTS, and the address recognition (AR) and break point (BP) stops, and finally with the material processing units of the computer proper.

Not indicated are such important parts as the input-output control IOC, the wide tape control WC, and the overflow control OFC. As in the previous diagram all gating has been omitted. Also we have omitted the many feedback paths from the computer proper to the various control mechanisms.

FIGURE 2
MAIN CONTROL CHANNELS

.4 Crude description of control functions in the execution of an order

Let us suppose that an order has reached FR and has received the signal to start its execution, either from the manual start button on the console or from a signal (continuous start) generated by the control itself. At this time not only is the order in FR, but O and T have gone into SP and MOSC, and from SP many gates have been set in various parts of the computer such as AC, Qt, SE, etc. DR holds the location in CS from which the order came, if it did not come from CN. The start signal generates a pair of signals, PCL1 and PT1, which perform the preliminary clear determined by S_5 of the tag and start the sequencing in MOSC of the different possible memory cycles. They also transfer the contents of DR, increased by 1 in the +1 adder, to CC, so that CC holds the presumptive address of the next order.

If the order requests a preliminary cycle PC (F_{25} or $F_{26} = 1$), then a chain of storage timing pulses, $P_0, \dots, P_7$ is started by PT1. B is sent to DR and to CS, and the number at that address is brought out from CS to MR. If $F_{25} = 1$, this number in MR is sent to P, added there to Ac or Qt as determined by F_{23} , and the sum returned to the same register. If $F_{26} = 1$, the first twelve bigits from MR are sent to FB and thence to AA, where $A + B$ serves as the address of the number on which the main part of the order will operate. The completion of PC, or PCL1 (if no PC occurs) generates pulses PCL2 and PT2.

We are now ready to perform the order proper. If this involves bringing a number from CS, as in Σ or π orders, or bringing A or $A + B$ to SC to determine the number of shifts in a shift order, then PT2 starts the operand cycle OC. (PCL2 will meanwhile have cleared SC.) Again PT2 starts the storage timing pulses, so either the number at address A or AA is sent from CS to MR (as in Σ or π orders), or A or AA goes to SC (in R or L orders). These pulses also start DP, which, through CTS, controls the sequencing of adding and shifting operations in AU necessary to perform the desired arithmetic. The completion of this part of the order then sends out a signal which will either initiate a result cycle (RC), if this is called for by F_{16} or F_{24} , or a fetch cycle (FC).

RC may be called for as just noted, or may be initiated by PT2 if no OC is involved. If RC occurs, we once more have the sequence of storage pulses, which control the writing of the result into CS.

The completion of RC, or of OC if it occurs but RC does not, or PT2 if neither OC or RC occurs (as in α orders) initiates FC. FC fetches the next order, whose address will be found at CC, FA, FB or $A + B$ according to the nature of the order. Again the chain of storage pulses is started, and at their conclusion the continuous start pulse is generated. Meanwhile the new order has been sent to FR and beyond, as described at the beginning of this section.

In this description we have assumed that each storage cycle, PC, OC, RC, FC, involves just one storage reference. This is true in all cases except for OC when the order involves input, output or wide tape. In the latter cases the execution of the order may involve many references to CS. In such cases OC initiates an auxiliary cycle, AC, which permits repeated storage cycles until the required number of words or characters has gone into or out of CS or W.

b. Basic Computer Components

.1 Logical diagrams and timing charts

To understand how a complex electronic mechanism such as GEORGE operates it is fortunately not necessary to understand the details of electronic engineering. Electronic engineers themselves have devised schematic representations of the basic electronic mechanisms and the logical and temporal relations connecting them. We shall follow their conventions with a few exceptions, of which one may be regarded as major. For the benefit of any engineers who happen to read this, the important exception is that the "0" and "1" states of any line will always be taken to correspond to the lower and higher potentials on that line. This may cause our logical diagrams to correspond less closely to the engineering drawings of the electronic circuits than the engineers customarily expect, but we believe that the elimination of the representation of polarities results in a justifiable simplification for the lay reader.

The two principal schematic aids are the logical diagram and the timing chart. The first of these represents the channels through which information flows from one mechanism to another, with established conventions as to the representation of the mechanisms themselves and their effect on the information passing through them. While every mechanism appearing in the logical diagram has some effect on the timing of the information, and these effects are either noted on the diagram or are implicitly known, it is frequently desirable to abstract from a logical diagram precise information as to the states of the various channels as a function of time. This is done in timing charts. The conventions to be used in the construction of these two schematic representations will be developed in the remainder of this chapter.

.2 Lines

We have noted in the overall survey that the material and control information introduced into the machine, whether from KB, PR, NR or CN, is binary in character. Although several channels may be used simultaneously, each channel conveys one of the two possible values of a bit. Each such channel conveys its information to some mechanism, which in turn

produces one or more bits of binary information to be carried on separate channels to still other mechanisms. Each mechanism alters the information received by it, putting out new binary information based in a determined way on the mechanism itself and the information received. Thus a channel leads from one mechanism to another, losing its identity in the receiving mechanism.

Such a channel corresponds in general to a wire in an electrical circuit in the machine. We shall find it possible to disregard its circuit character and concentrate on the single wire, which we shall call a line (and draw as a line) in our schematics. Every such wire is assumed capable of maintaining a voltage in either of two well separated ranges. When the voltage of a line is being maintained in one of these ranges the line will be said to be in state "0" or state "1" according as the voltage maintained is in the lower or higher of the two ranges. The length of time that the voltage of a line must lie in one of the two ranges before the line may be considered to be in the corresponding state depends in each case on the receiving mechanism since the line cannot be considered to have transmitted to the receiving mechanism the information represented by a given state unless the voltage has remained within the given state long enough for existence of this voltage to have the prescribed effect on the mechanism in question. It is, of course, a matter of engineering practice to determine these times and see that they contain safe margins. A line that is not in one or other of these two states will be said to be in transition. The "0" and "1" states will be called stable.

The fact that a change of state of a line corresponds to a change of potential and consequent flow of current in a wire means that it takes time for such a change to travel the length of the wire. Since this time is generally very brief in comparison to the time required to establish the change of potential at one end of the wire, we shall usually disregard this time lag except as it serves to indicate which end of the line is the source and which the destination of the information, thus establishing a sense for the flow of information in the line. Every line conveys information in just one sense, and this is indicated on the logical diagram by drawing an arrow-head in that sense at the destination end of the line and at any point where its presence may be presumed to add clarity to the drawing.

With such exceptions as we shall note when we come to them, every line originates at one mechanism, terminates at another, and at any instant is regarded as being in the same state throughout its length, thus transmitting its information without modification from its source to its destination. No two lines can merge except as they terminate (at separate terminals) in a common mechanism. A given line may, however, go to more than one mechanism. When this occurs the line may be indicated on the logical diagram as a set of lines branching out from a single line issuing from the originating mechanism, with each branching point emphasized by being covered by a dot; or the issuing line may be abruptly

terminated with an arrowhead and identifying label, and its entrance to each receiving mechanism represented by a correspondingly labelled line entering the receiving mechanism. The latter convention is useful to avoid crossing lines on a given diagram, and to indicate lines leading from one diagram to another. If two lines must be represented as crossing in a diagram, one of them is broken off near each side of the ostensible point of intersection and the two ends are joined by a small semi-circle.

The timing chart of a single line is a conventionalized graph of the state of that line as a function of time. Time will be assumed to increase from left to right and voltage to increase from below upwards. Time will usually be measured in microseconds (millionths of a second), though often relative rather than absolute time scales will be used. We shall use the following conventions, each illustrated in the correspondingly labelled line in the accompanying figure: (a) The "0" and "1" states will not be labelled, since the establishment of a line in one of these states will be denoted by a horizontal line segment at (respectively) the lower or upper of two levels at a uniform distance apart, without regard to the actual magnitudes of the voltages involved. (b) Whenever a transition may reasonably be supposed to correspond to a monotonic increase or decrease from one state to the other, the transition will be represented by a slanting line segment joining the termination of the first state and the initiation of the second. It is to be understood in any case that the termination of one state is to be indicated as occurring at a time when there is no doubt (well, hardly any) in the mind of the engineer that the line is still in that state, and similarly for the initiation of a state. (c) If there is a longer time between the termination of one state and the initiation of the next than is required by the nature of the initiating mechanism, due to ambiguity as to the states of some or all of the inputs to the mechanism, this will be indicated as in the figure. (d) If at any time the state of a line is not relevant to the states of the other lines in the chart and the actual state of the line at that time is not known, this will be indicated by drawing a dashed line midway between the "0" and "1" levels during this time. As soon as the state of

such a line is known it will be represented as in that state, whether or not it happens to be relevant at that time. (e) The principal use of a timing chart is to show the relation in time between the states of different lines. In the figure lines b, c and d are assumed independent of each other, while the state of line e is intended to represent the logical function $b \cap (c \cup d)$ of the states of the other three lines. Note that we here use the symbol designating the line as the symbol of a corresponding Boolean variable in B_0 . This we shall do systematically.

The state of a line represented in its timing chart by a solid line segment at the "1" state will be called a pulse. If the transition to or from this state is as described in (b) above, then the transition state will be called the beginning (b) or end (e) of the pulse. In case (c) above a pulse will not be considered to have a definite beginning or ending, as the case may be. In case of irrelevance a pulse may have a beginning and/or end, though it may not be shown on the particular timing chart. A pulse may be of long or short duration. In fact, certain lines are in state "1" whenever the machine is on (and these lines are operating properly), so such pulses may be regarded as of infinite duration.

We shall usually refer to a line as being On while in state "1" and Off while in state "0".

.3 Inverters

An inverter has one input and one output, the output having as its value the complement of the value of the input after the brief time required to effect the reversal. It has then the effect of operation bar on the input, so the output will be denoted by the bar of the input symbol. In logical diagrams the inverter is represented by a small circle. In most cases the inversion occurs at an input or output of another mechanism. In such

cases the inverter circle is represented as being exteriorly tangent to the other mechanism, with the line inverted terminating or starting at the point of the inverter circle diametrically opposite to the point of tangency. In those cases where the inversion is a separate operation the input and output lines are required merely to meet the circle at distinct points. These conventions will be illustrated in later diagrams. In timing charts an inversion will be indicated only by its effect as part of the mechanism to which it is attached unless the inverted line is separately established, in which case either or both lines may be gaphed

.4 And-gates and or-gates

These are combining mechanisms, each having two or more inputs and just one output. The value of the output of an and-gate is the cap of the values of the inputs - i.e., the output is On only while all the inputs are On and is Off only while some one or more of the inputs is Off. The value of the output of an or-gate is the cup of the values of the inputs - i.e., the output is On only when some one or more inputs are On, and is Off only when all of the inputs are Off. Under any other circumstances the output of either type of gate is in transition.

An and-gate is represented in logical diagrams by a square or rectangle with the output leaving one side and the inputs coming to any or all of the remaining sides. The or-gate is represented by a circle, generally larger than that used for an inverter, but in any case distinguished from the inverter circle by having more than one input. All points of entry and exit must be distinct. A given logical diagram will usually consist of a considerable number of mechanisms such as gates and inverters, with their connecting lines. In any such drawing we shall not ordinarily label those lines that originate and terminate completely on the given drawing (interior lines) unless the line is broken, as in Section .2 to avoid crossings. Occasionally an interior line of a logical drawing will be labelled for purposes of verbal reference. Lines, or branches of lines, that do not terminate on the drawing will be labelled with indication of the sheets on which they appear as inputs. Similarly, an input originating as an output of another diagram will be labelled on the current diagram at the point of entry, with indication of the diagram of origin.

Similarly, the timing chart associated with a given logical diagram will ordinarily list only those inputs that originate elsewhere and those outputs that go elsewhere, omitting the wholly interior lines. The

accompanying logical diagram and associated timing chart indicate the various conventions thus far mentioned for combinations of lines, gates and inverters.

.5 Differentiators

When a pulse has a beginning or end it may be sent to a differentiator, which produces a voltage proportional to the rate of change of the voltage in the incoming line. Thus the output of a differentiator stays nearly constant while the input remains On or Off, but rises or falls to a sharp peak during the beginning or end of a pulse; we shall indicate the two kinds by the symbols DB and DE, respectively. In accordance with our

general convention we shall assume that the output of a DE is inverted, so that each kind of differentiator in effect emits a brief pulse nearly coinciding with the entering b or e. Of course, the range and duration of the voltage issuing from a differentiator must be sufficient to enable it to act as a pulse in the mechanism to which the output of the differentiator goes. Since a differentiator is used as a "trigger" for some other mechanism it will usually appear to be fed directly into the triggered mechanism, and no special name will be attached to its output. In a logical drawing it will be represented by a square containing the appropriate label DB or DE, with just one input and one output. In a timing chart it will appear only implicitly in the fact that the mechanism it operates goes On at the beginning or end of a pulse on the line going into the differentiator. See the next section for drawings involving differentiators.

.6 Pulsers

A pulser is a device that produces a pulse of preassigned duration when triggered by a differentiator. Thus, in effect, a pulser originates a new line whose On state has its beginning at the beginning or end of a pulse on another line, and its relation to the other line is so indicated on timing charts, though on a logical diagram a differentiator is inserted between the pulser and the line whose beginning- or end-of-pulse is differentiated. The termination of the pulse is an end-of-pulse state which is independent of the trigger, once the output pulse has begun, except that if the differentiator operates again before the output pulse is complete, the effect on the pulser varies with the circumstances, so this should not be allowed to occur. In a logical diagram a pulser will be represented by a long rectangle containing the label PLR, with an input from a differentiator at one end and an output at the other end. The rectangle may also contain the length of the output pulse in microseconds.

.7 Delays

A delay is a device with one input and one output, the relation between them being that the state of the output is at any time identical with that of the input a fixed time earlier. Thus a delay serves to translate the input signal in time by the amount of the delay, which is fixed for the given delay mechanism. However, a delay should not be expected to handle two

pulses that begin or end within the delay time. (It should be noted that both pulsers and delays are adjustable by the engineers, so the times involved are to be regarded as arbitrary, within limits, but fixed.) A delay will be indicated by a square box containing the number of microseconds the delay lasts. It will usually be regarded as inserted in a line, though if the output is to be separately designated this will usually be done by attaching the symbol "-d" to the symbol of the input. Here are some typical uses of delays in logical diagrams, with their corresponding timing charts.

.8 Half-adders

A frequently occurring combination of gates and inverters is the half-adder, a mechanism with two inputs and one output that is the half-sum of the inputs. On a logical diagram it will be represented by square containing a + sign with the inputs attached to separate points on the same or different sides and the output attached to still a different side. The illustration shows the representation of the half-adder itself and possible representations of it and its complement in terms of gates and inverters.

.9 Toggles

The term toggle is used here to denote any of several electronic devices that is used to store a binary bit of information for an unlimited time. A toggle also has the property that consultation of the toggle (i.e., extraction of its information) does not destroy the information, since consultation does not affect the inputs and the mechanism is stable as long as its inputs are stable. Toggles also have the interesting property that the output (outputs) of a toggle is an (are) implicit function(s) of the input(s) and output(s). Three logically distinct types of toggles are used in GEORGE and we shall call them asymmetric, symmetric and counter toggles, respectively.

The accompanying figure gives a possible logical diagram of an asymmetric toggle, which has inputs $c\ell$ and s and output t as shown. From the figure we derive the relation

$$(.9.1) \quad t = f(c\ell, s, t) \equiv s \vee (\overline{c\ell} \cap t).$$

If we assign to the independent binary Boolean variables c and s all possible sets of values, then we may find f as a function of t for each set and then determine those values of t which make $f = t$. This gives us all solutions of Equation (.9.1), i.e., all sets of values of $c\ell$, s and t which can hold simultaneously. We thus get the table

	c	s	f	t
	0	0	t	t
	0	1	1	1
	1	s	s	s

(.9.2)

From this we see that while $c\ell = 1$, $t = s$, so that as long as $c\ell$ is fixed at 1, s may vary arbitrarily and t will follow it, whether s is in a stable state or in transition. Suppose now s is kept fixed while $c\ell$ changes from 1 to 0. If s is kept at 0, t , which started as 0, remains unchanged ($t = t$) when $c\ell$ goes to 0. If s is kept at 1, t , which was 1 before the transition of $c\ell$, becomes 1 after the transition, being momentarily unstable during the

transition. The effect of other possible transitions may be followed similarly. Note, however, that if both cl and s go from 1 to 0 simultaneously and are held at 0, the state of t is ambiguous.

The use made of this device is the following: Assume $s = a \cap tr$, where a is a line whose information it is desired to store and tr is a control line, here called a transfer line, since pulses on it are used to transfer information from a to the toggle, as we shall see. Initially we assume $cl = tr = 0$, so $cl = s = 0$ and t has some arbitrary value. If now we pulse cl only, t will become 0 - that is, we shall have cleared the toggle to 0. (cl is therefore called a clear line, more specifically a clear-to-zero line.) As the clear pulse ends (with tr held at 0), t remains at 0, as we noted before. In the absence of knowledge of the information contained in line a , this is the only specific value to which the toggle can be set. That is to say, we can always "clear" an asymmetric toggle to zero, regardless of the information currently in a or formerly in the toggle, but we have no way of setting the toggle to one without using $a = 1$.

To transfer a to the toggle we can proceed in either of two ways. (1) Having cleared the toggle to zero we can keep $cl = 0$ and pulse tr . During the pulse $s = a$, and t either preserves its former value 0, if $a = 0$, or becomes 1 if $a = 1$. In either case $t = a$ while $cl = 0$, $tr = 1$. When tr returns to 0 we have $cl = s = 0$, and t continues to have the value a . We have thus "transferred" a into the toggle. (2) A quicker way is to pulse cl and tr simultaneously, but allow tr to outlast cl . During $cl = tr = 1$ we have $cl = 1$, $s = a = t$. When $cl = 0$ and $tr = 1$ we have $cl = 0$, $s = a$, and t persists as a . Finally, when $cl = tr = 0$, we have $cl = s = 0$, and t persists as a . Here are the timing charts for (I) clearing to zero, (II) transferring $a = 0$, and (III) transferring $a = 1$. In any case the output t is available for use elsewhere as long as cl and tr are held at 0.

Since s comes into an or-gate it is possible to admit any number of lines, each gated by its own transfer line. Although the and-gates controlling the transfers are not, strictly speaking, part of the toggle, we shall assume them so in our abbreviated representation of asymmetric toggles. The diagrams below indicate the full and abbreviated logical

drawings of a toggle with two information inputs and one clear input. Note that the transfer lines are not drawn in the abbreviated representation but are specified in the notation.

The operation of a symmetric toggle may be understood from the next logical diagram. Again we assume inputs cl and s , but we now have two outputs, t and t' . From the diagram we have the relations

$$(.9\ 3) \quad t = s \cup \bar{t}', \quad t' = cl \cup \bar{t},$$

from which

$$(.9\ 4) \quad \bar{t}' = \overline{cl \cap t};$$

$$t = f(cl, s, t) = s \cup (\overline{cl \cap t}).$$

Thus t is the same function of cl and s as for the asymmetric toggle. Also, $t' = \bar{t}$, unless $cl = 1$ and $t = 1$, i.e., unless $cl = s = 1$, in which case $t = t' = 1$. Hence, when $cl = s = 0$ and the toggle is remembering, it puts out both t and $\bar{t}$, which is one of the uses of this type of toggle. The method of clearing the toggle to 0 is the same as for the asymmetric toggle. The methods of transfer are the same, but the first method given is usually used. (For engineering reasons little if any time is saved by the second method.)

The full symmetry of this toggle has not been shown. From the diagram it is clear that the roles of cl and s and of t and t' could be interchanged. Actually the symmetric toggle has four possible inputs, which for many purposes are paired as in the accompanying diagram. (Here we have replaced s by a $\cap tr$ in each case.) It is clear that $t_1 = a_1$

and $t_2 = \bar{a}_1$ after a transfer through G1, while $t_2 = a_2$ and $t_1 = \bar{a}_2$ after a transfer through G2. Also, if cl_1 is pulsed while tr_1 , tr_2 and cl_2 are held at 0, then we have $t_1 = \bar{t}_2 = 0$; and a pulse on cl_2 while $cl_1 = tr_1 = tr_2 = 0$ give $t_1 = \bar{t}_2 = 1$. Thus the toggle may be cleared to either $t_1 = 0$ or $t_1 = 1$. The toggle is completely symmetric. For engineering reasons no further inputs are available, and branching of either output is undesirable because it slows the toggle down.

In our logical diagrams the toggle is represented as completely symmetric, except that one side is selected as the one to which the name of the toggle (if any) is attached in capital letters. The output of that side is given the same name in small letters, and the output of the other side is denoted by the bar of the first output. The transfer gates are not part of the toggle, but must be drawn separately. In fact, in many cases no distinction is made between cl and s lines. Thus it is possible to represent the symmetric toggle as a horizontal rectangle bisected by a vertical line. The name is put in the preferred half. One or two inputs enter each side at the side or top, and one output leaves each side at the bottom

The counter toggle is a modification of the symmetric toggle. In the accompanying diagram it will be observed that two of the normal

inputs to a symmetric toggle (one on each side) have been brought from a common source p . We shall assume further that the information transmitted by p consists of a series of pulses, each of brief duration ϵ , and that the (variable) time from the initiation of one such pulse to the initiation of the next is η . It is seen that p is gated into the t and t' sides of the toggle by t' and t , respectively, each delayed by an amount δ as shown. The inputs $cl1$ and $cl0$ operate as normal clearing lines and will be disregarded in the present description of the operation of the toggle.

To define the states of t and t' they must be regarded as functions of time, so that δ may be taken into account. As in Section .9.7 we shall denote the delayed output of t by $t_{-\delta}$, etc. At any fixed time the value of $t_{-\delta}$ is that of t just δ microseconds earlier. Similarly, $t_{-2\delta}$ denotes the value of t just 2 microseconds earlier. Using these conventions we have the following Boolean equations:

$$(.9.5) \quad t = \bar{t}' \cup (t'_{-\delta} \cap p); \quad t' = \bar{t} \cup (t_{-\delta} \cap p).$$

Note that $p = 0$ implies $t' = \bar{t}$. From the second relation we find

$$t'_{-\delta} = \bar{t}_{-\delta} \cup (t_{-2\delta} \cap p_{-\delta}); \quad t'_{-\delta} \cap p = (\bar{t}_{-\delta} \cap p) \cup (t_{-2\delta} \cap p_{-\delta} \cap p);$$

$$\bar{t}' = (t \cap \bar{t}_{-\delta}) \cup (t \cap \bar{p}),$$

whence

$$(.9.6) \quad t = f(p, t, \delta) = (t \cap \bar{t}_{-\delta}) \cup (t \cap p) \cup (\bar{t}_{-\delta} \cap p) \cup (t_{-2\delta} \cap p_{-\delta} \cap p).$$

From this there may be obtained the accompanying table for f as a function of t :

(.9.7)

p	$p_{-\delta}$	f
0	0	t
1	0	$\bar{t}_{-\delta}$
1	1	$\bar{t}_{-\delta} \cup t_{-2\delta}$

The only ambiguity arises when $p = p_{-\delta} = 1$. This can be avoided by pulsing p so that the p and $p_{-\delta}$ pulses do not overlap, i.e., by having $\epsilon < \delta$ and $\eta > \epsilon + \delta$. The following timing chart shows what happens to t when p emits a series of pulses under these restrictions. It will be noted that $t'_{-\delta}$ and $t_{-\delta}$ have been shown also, since they admit p to G3 and G4 respectively through gates G1 and G2. The timing chart shows clearly how G1 and G2 operate to cause the sequence of pulses from p to store 0 and 1 alternately in T .

The manner in which a set of such toggles may be connected to form a counter, and the mode of operation of the cl_0 and cl_1 lines in such a counter will be described in the next chapter. The abbreviated representation of a counter toggle in a logical diagram is as follows:

.10 Relays

We come now to slower components, partly mechanical in their action. These will be found principally in the console or in conjunction with mechanical units, such as the paper tape punch or reader. The first of these, the relay, is logically equivalent to one or more and-gates operated by a single line r , each and-gate having just one other input and one output. Either inputs or output may be inverted. The accompanying diagram shows a single pulse operating a set of gates that

illustrate the different usages. Note the convention on inverting the pulse r . Here "1" represents a fixed On line.

.11 Switches

What we shall call the push-button switch is logically identical with the relay, except that r is replaced by the mechanical positioning of the push-button. Hence, we shall use the same representation as for relays, except that the input line r will be replaced by a stylized push-button labelled pb . On an output line pb will be interpreted as though it referred to the state of an electrical circuit, with the "in" and "out" positions of the push-button corresponding to the On and Off values of a hypothetical incoming line. Though there are many mechanically distinct forms of push-button switch, each is logically equivalent to one of the two types: the push-pull switch (which includes the mechanical toggle switch), which must be operated manually from either position to the other, and the spring-loaded button, which restores itself to the Off position when released, thus producing a finite pulse. The latter will be indicated by a stylized spring around the

stem of the button, as in the second of the two logical diagrams shown. In diagramming switches we shall omit reference to the switch in labelling the output.

The rotary gang switch has several positions at each of which it stays until manually set at a different position. At each position other than the common Off position it acts like a push-button switch in the On position for one or more lines simultaneously. Again we use the same sort of logical representation, with a separate rectangle for each line controlled. Each rectangle will contain the number of the position (or positions) of the switch at which the output line is On. A curved arrow replaces the push-button head, and the mechanical input is labelled gsw. We shall also extend the gsw "line" only part way into each rectangle, as shown.

.12 Non-logical elements

In order to correlate our logical diagrams and the circuit diagrams of engineers more closely, we shall sometimes include, in our logical drawings, representations of certain mechanisms that have no logical significance, though they may affect the timing slightly because of the time taken for them to operate. These are all of the one-input, one-output variety, so that we could regard a line as being unchanged by its passage through the mechanism. (If the line is inverted in the process this will be represented as usual by an inverter circle at the input or output of the figure representing the mechanism.) Whether the output is labelled as a new line or not will depend on the usage in the circuit drawings of the engineers.

The principal components of this sort that come to mind are amplifiers, discriminators and drivers. An amplifier increases the amplitude of the incoming signal, thus creating a larger gap between the ranges corresponding to the two states. A discriminator replaces a signal that may fluctuate considerably within the ranges corresponding to the "stable" states by a signal that is actually more nearly constant in these states.

A driver increases the current (i.e., strength) of a signal so that the output may be used as input to several mechanisms, or to a particular mechanism that requires a particularly strong input signal; a driver may be used also to bring up to standard the output of a mechanism that produces a weak signal for some reason. Generally speaking, amplifiers and discriminators will simply be inserted in lines without affecting their identity, while the outputs of drivers will often be given new names. All such non-logical elements will be represented on logical diagrams (if they appear at all) by suitably labelled hexagons. In general, such a component will be explicitly noted only if the chassis on which it lies is different from the chassis on which the input originates and that on which the output is used.

c. Complex Computer Components

So far we have dealt with what might be called atomic and molecular elements of the computer, such as gates and inverters on the one hand and half-adders and toggles on the other. While no sharp distinction is to be made (certainly decoders may be regarded as complex molecules), the mechanisms to be described now are perhaps more analogous to the simple machines of physics - the lever, pulley and inclined plane - in that, although more complex in structure, they may be elements of any computer, each providing its own function but not yet specialized to its particular place or places in a particular machine.

.1 Decoding matrices

Perhaps the conceptually simplest of these mechanisms is the decoding matrix, or decoder. Essentially a decoder is simply a collection of gates and inverters so connected as to produce some or all of the Boolean polynomials in the independent two-valued Boolean inputs. Particular decoders in particular machines may then have a wide variety of special forms. All, however, can be built up from two general types, which we shall call (complete) p-fold (basic) decoders and (complete) p x q-decoders. (The adjective "complete" is intended to indicate that all of the output functions specified in subsequent definitions are actually formed. Since we shall assume that this is always the case for decoders of these two fundamental types, the adjective will normally be omitted.)

A p-fold (basic) decoder is a mechanism with p input lines, $a_1, \dots, a_p$, and 2^p output lines whose values are precisely the 2^p possible distinct basic Boolean monomials in the p variables a_1 . The logic of such

a mechanism can be expressed solely in terms of inverters and and-gates, but even for p as small as 3 there are several ways of connecting them to produce the desired result. Two are given herewith. We shall not be concerned with the particular method chosen by the engineers. In our logical diagrams a p -fold decoder will appear as a rectangle thus labelled, with p inputs and 2^p outputs.

A $p \times q$ -decoder has p inputs $a_1, \dots, a_p$, q inputs $b_1, \dots, b_q$, and the pq outputs $a_i \cap b_j$. Again there are many different possible realizations, which we leave up to the engineers. Our representation of a $p \times q$ -decoder in logical drawings will be a rectangle so labelled, with p inputs on one side, q inputs on another and pq outputs from a third. Note that the second representation of a 3-fold decoder above is actually a 2-fold decoder, a 1-fold decoder and a 4×2 -decoder whose inputs are the outputs of the first two decoders. (A 1-fold decoder amounts merely to forming two branches of the single input and inverting one of the branches.)

Since all Boolean polynomials in p variables over B_0 are expressible as cups of basic monomials, more complex and specialized decoders may be regarded as forming the desired cups (by means of or-gates) of the outputs of some p -fold decoder. Actually it may be more economical to take outputs from $p \times q$ -decoders. Such a specialized decoder will not be represented on our logical drawings by a special symbol, even though it may be referred to as a separate entity in the text. Rather, it will be detailed by means of or-gates applied to decoders of the fundamental types defined in this section.

.2 Registers

A register is a device for storing a fixed number of bits of information for immediate access. By immediate access we mean that the

Triple Gates (p- way)

Double Gates (p x q)

information is in such form that it has only to be gated out when called for. In serial machines this may mean that the bits of the set are stored cyclically in some channel and are available only after waiting the access time - the time required for the first bit to appear after the call for information has been made. In such cases usually just one word is stored in such a channel.

In a parallel machine, such as GEORGE, a register is usually a set of toggles, which is what a register will be assumed to be here. A static register is just that, together with transfer gates, clear, transfer and information input lines, and information output lines. The diagram shows a 4-stage static register using

symmetrical toggles, as do all static registers in GEORGE that are not parts of shifting registers. Since the toggles are completely independent of each other, they need not be in proximity in the machine, though they usually are. We shall see that the toggles of the Function Register of GEORGE are grouped into four distinct sets in three different chassis. Only one set of clear, transfer and input lines has been shown. Another symmetrically arranged set could be added to allow transfers into the opposite side of the toggles.

A shifting register is one that permits the information held in it to be shifted in parallel, one position at a time. The shifting may be accomplished by providing a single static register with outputs containing delays gated from each stage into its neighboring stage or stages, the original information being in effect temporarily stored in the delay lines before being admitted to its new position. In AVIDAC and GEORGE,

however, this temporary storage is obtained by attaching a second static register into which the contents of the first static register are gated in parallel with unchanged relative position. The contents of this second static register are then gated into the first register in their new positions. In GEORGE the shifting registers contain asymmetric toggles.

There are two basically identical, 41-place shifting registers of this sort in GEORGE. The diagram shows the first two, the n^{th} and the last places of such a register without indicating the special connections that apply to the individual registers. The latter will be shown when the actual registers are taken up. This register provides for two information inputs, p_n and z_n , for each upper toggle. All the p_n inputs are gated into the toggles in parallel by a common transfer line, iz , and all the z_n inputs are similarly gated by a common transfer line uz . These common transfer lines and the corresponding gates are not drawn, but are implied in the labelling in the manner specified in Section .9. There is also a common clear line, clU , as shown. Each lower toggle has three information input lines controlled by three transfer lines, rz , dz , lz , and corresponding gates, and a common clear input, clL . The inverters on the information inputs are apparent rather than real, so far as the engineering is concerned. This is a case where the engineering definitions of "0" and "1" have been reversed, so the logical drawing reflects this by introducing fictitious inverters.

The mode of operation of such a register is as follows. The lower static register is assumed to hold any information stored in the register as a whole. Input to the whole register is to the upper static register; output is from the lower static register. When it is desired to introduce information $p_x, p_0, \dots, p_n, \dots, p_{39}$, into the register, lines $c\ell U$ and iz are pulsed, with iz outlasting $c\ell U$ (Section .9), so that Z^u reads $z^x = \overline{p_x}$, $z^0 = \overline{p_0}, \dots, \overline{z^n} = \overline{p_n}, \dots, \overline{z^{39}} = \overline{p_{39}}$. Then $c\ell L$ and dz are similarly pulsed, giving $z_x = z^x = p_x, z_0 = z^0 = p_0$, etc.

To shift the contents of the register one place to the right, $c\ell U$ and uz are pulsed to give $z^X = \overline{z}_X = \overline{p}_X$, etc., and $c\ell L$ and rz are pulsed to give $z_0 = \overline{z}^X = p_X, \dots, z_{39} = \overline{z}^{38} = p_{38}$. What goes into z_X depends on the connections special to the particular register; e.g., we could introduce z^{39} , which would make $z_X = p_{39}$, or we could leave z_X cleared to 0 by the $c\ell L$ pulse.

Left shifts are executed similarly, with a like ambiguity as to the input to z_{39} .

.3 Counters

We remarked in the previous chapter that counter toggles could be assembled to form a (binary) counter, a device that accumulates in binary form the total number of pulses received by it (up to its capacity) and at any time between pulses, if they are not too close together, makes available, in its character as a register, the current total count. Clear lines to each side of each toggle are provided to permit the pre-setting of the counter to any desired integer from 0 to 2^q-1 , where q is the number of toggles in the counter. We assume the input pulse to be counted formed by a differentiator, though other sources could be used. The differentiators on the lines from one toggle to the next must all differentiate either the beginning or end of the pulses received. We proceed to show that, according as they are all DB's or DE's, the binary number represented by the states of the labelled outputs will be increased or decreased by one every time an input pulse p is received, so that in effect the counter counts forwards or backwards accordingly. (As we shall see presently, the arrangement of clear and transfer lines in this drawing assumes that the differentiators shown are all DB's.)

Assume that with all inputs Off the set of toggles is stable and puts out an arbitrary but fixed set of output values: $c_0, c_1, \dots, c_{q-1}$, which we interpret as the positive integer $c = c_0 2^0 + c_1 2^1 + \dots + c_{q-1} 2^{q-1}$. Thus c is some (and may be any) integer in the range $0 \leq c \leq 2^q$. Assume also that all differentiators are DB's. Either $c_0 = 0$ or there is a maximum integer i with $0 < i \leq q$ such that $c_0 = c_1 = \dots = c_{i-1} = 1$. Referring to Section .9, let us observe the behavior of C_0 over a time interval of length η ($> \epsilon + \delta$) that begins at the beginning of the only pulse p occurring during this time interval η . The duration ϵ of this pulse must be $< \delta$. From the timing chart we see that if $c_0 = 0$ initially, then c_0 goes to 1, and hence c_0' goes to 0, and these values are held during the interval. Since c_0' was originally On, its change to Off produces no pulse in d_1 , so the inputs to C_1 , and hence all inputs to $C_2, \dots, C_{q-1}$ are 0. Hence, once C_0 has been established in the "1" state, all the toggles remain fixed for the remainder of the interval η , and the counter outputs denote $(c_0 + 1)2^0 + c_1 2^1 + \dots + c_{q-1} 2^{q-1}$, where $c_0, \dots, c_{q-1}$ denote the original outputs. Thus the counter has added one to its reading, and the new reading will hold until a new pulse p occurs.

If, on the other hand, we start with $c_0 = \dots = c_{i-1} = 1$, the pulse P will cause c_0 to go from 1 to 0 and c_0' to go from 0 to 1, thus initiating a pulse in line c_0' and producing a pulse in line d_1 . The time from the start of the pulse in p to the start of the pulse in d_1 under these conditions (i.e., $c\ell = tr = 0$, $c_0 = 1$ when p is pulsed) is negligible in comparison to ϵ and will be disregarded. If, now, originally $c_1 = 1$, the pulse in d_1 interchanges the values of c_1 and c_1' , producing a pulse in d_2 , which in turn will be assumed to start simultaneously with the pulse in d_1 and hence with the pulse p . Continuing in this way we find that p causes the toggles $C_0 \dots, C_{i-1}$ to be flipped simultaneously. If $i < q$, then the flipping of toggle C_{i-1} also flips toggle C_i , in which case c_i goes from 0 to 1 and c_i' from 1 to 0. Hence d_i is not pulsed and no toggles with index $> i$ are affected. Thus the counter reads $(c_0 - 1)2^0 + (c_1 - 1)2^1 + \dots + (c_{i-1} - 1)2^{i-1} + (c_i + 1)2^i + c_{i+1} 2^{i+1} + \dots + c_{q-1} 2^{q-1} = c + 2^i - (2^0 + \dots + 2^{i-1}) = c + 2^i - (2^i - 1) = c + 1$, so the count has been increased by one and the counter is maintained in this state for the rest of the interval η . If $i = q$, all the toggles are changed from 1 to 0, c was originally $1 \cdot 2^0 + 1 \cdot 2^1 + \dots + 1 \cdot 2^{q-1} = 2^q - 1$, and now reads $0 \equiv c + 1, \text{ mod } 2^2$. Hence the counter has indeed added 1 to the count, though the representation of the new count has been reduced to 0, mod 2^q , since the counter has exceeded its capacity. Here are timing charts for the different typical cases.

Suppose now that the differentiators are DE's. Then D_1 is activated as c_0' changes from 1 to 0, i.e., as c_0 changes from 0 to 1. If, initially, $c_0 = 1$, then p causes c_0 to go to 0 and c_0' to go from 0 to 1, so D_1 is not activated and the remaining toggles are not affected. Thus the new count is $(c_0 - 1)2^0 + c_1 2^1 + \dots + c_{q-1} 2^{q-1} = c - 1$. If $c_0 = \dots = c_{i-1} = 0$ and either $c_i = 1$ or $i = q$, then pulsing of C_0 by p changes c_0 to 1 and c_0' from 1 to 0, so D_1 is activated, which flips C_1 . This action goes on down the chain of toggles, with the result that $c_0, \dots, c_{i-1}$ are all changed to 1. If $i = q$, the original

count was 0 and the new count is $2^0 + 2^1 + \dots + 2^{q-1} = 2^2 - 1 \equiv -1, \text{mod } 2^{q_0}$, so the count has been decreased by 1 mod 2^q . If $i < q$, then D_i is activated, toggle C_i is flipped, beginning a pulse on c_i . Hence no pulse is emitted by D_{i+1} , so all toggles after C_i are unaffected. Thus the new count is $(c_0 + 1)2^0 + \dots + (c_{i-1} + 1)2^{i-1} + (c_i - 1)2^i + c_{i+1}2^{i+1} + \dots + c_{q-1}2^{q-1} = c + (2^0 + \dots + 2^{i-1}) - 2^i = c - 1$. Hence the counter counts backwards in every case. The timing is identical with that when the D 's are all DB's.

So far we have left $c\ell = tr = 0$ throughout. Suppose now that we wish to set the counter to the value given by the inputs a_i before the counting begins. We assume the $c\ell$, tr and a_i lines arranged as in the drawing, and that all D 's are DB's. The logical relations connecting the inputs and outputs of the successive counter toggles are:

$$\left. \begin{aligned} c_0 &= (a_0 \cap tr) \cup \overline{c_0'} \cup (p \cap c_0, -\delta), \quad c_i = (a_i \cap tr) \cup \overline{c_i'} \cup (d_i \cap c_i, -\delta) \\ c_0' &= c\ell \cup \overline{c_0} \cup (p \cap c_0, -\delta), \quad c_i' = c\ell \cup \overline{c_i} \cup (d_i \cap c_i, -\delta) \end{aligned} \right\} i > 0.$$

Assume $p = 0$ throughout. First assume $tr = 0$, $c\ell = 0$, and the C_i 's in any stable states. If we keep $tr = 0$ and make $c\ell = 1$ then $c_0' = 1$ and $c_0 = \overline{c_0'} = 0$, and these states persist upon the return of $c\ell$ to 0, regardless of the initial state of C_0 . However, if originally $c_0 = 1$ then c_0' goes from 0 to 1, so D_1 emits a pulse. For $i = 1$ we see that $c_i = 1$ and $c_i' = 0$ if d_i is not pulsed. If d_i is pulsed by the flipping of C_0 , the pulse in d_i begins with the $c\ell$ pulse and has duration ϵ . Hence if the $c\ell$ pulse is longer than ϵ the pulse in d_i will not affect c_i' and will cause ambiguity in c_i only for ϵ microseconds. During the remainder of the $c\ell$ pulse and until some outside pulse

d. The Internal Storage (CS)

.1 General description

Internal storage of material and instructional information is accomplished in a magnetic core device built by International Telemeter Corporation. It is designed to hold $4096 (=2^{12})$ words of 42 bigits each. Each bigit is stored in a separate toroidal ferrite permanent magnet, henceforth called a core, whose two possible stable states of magnetization correspond to the two possible values of a bigit. All 4096 cores corresponding to a given binary place in the 4096 possible words are mounted on a separate frame called a digit plane (DIP), and the DIP containing the bigit a_k of every word $a_0 \cdot a_1 \cdots a_k \cdots a_{41}$ stored will be designated by DIP_k . Since GEORGE normally handles only 40-bit words, there are thus two extra DIP's. One of these will be held as a spare in case of failure of one of the regularly used DIP's. The other may be used at a later time to hold a check bit. As of now (July, 1956) we shall assume that only DIP's 0 through 39 are in active use. Since the only distinction between the used and unused DIP's lies in the attachment of inputs from and outputs to the rest of the machine, the ensuing description may be taken to apply to all 42 DIP's, except for the assignment of these connections. As an extra precaution, each DIP holds at least 128 extra cores that may be used as replacements in that DIP, but their existence will be ignored in the ensuing description.

Information in the form of words is written into or read out of CS in parallel. Two microseconds are allowed for setting up the address (twelve-bit integer specifying the location) of the storage position to be consulted, and then 15 microseconds are occupied by a complete storage cycle, which must be finished before CS may be consulted again. This cycle always comprises reading the word stored at the location consulted and then either re-writing the word originally stored there or writing a new word in that position, depending on whether the particular consultation specifies "reading" or "writing." Writing requires the whole cycle for its accomplishment, but when a word is read the information is available for use by other units of the machine at the end of the fifth microsecond of the cycle, so that further operations called for by the instruction and not involving CS may be carried out while the cycle in CS is being completed.

The cores of each DIP are mounted on it in the form of a rectangular matrix of 32 rows (denoted by X_0 through X_{31}) and 128 columns (denoted by Y_0 through Y_{127}). Each row extends horizontally along one face of the DIP and back along the other face, so there are 64 columns on the front and 64 columns on the back of the frame, which is thus approximately 64 columns wide and 32 rows high. (There are actually at least two extra columns on each face of every DIP, which form the possible replacements mentioned in the first paragraph.) The address of a word in CS is specified

by a positive 12-bit integer, which we shall denote by $ad_{11}ad_{10}\cdots ad_1ad_0$, with the value $ad_{11}\cdot 2^{11} + ad_{10}\cdot 2^{10} + \cdots + ad_1\cdot 2 + ad_0$. (This notation has been chosen so as to conform more closely with the notation of engineers rather than that corresponding to the standard notation.) The symbol $ad_4\cdots ad_0$ may be interpreted as representing a positive integer i with $0 \leq i < 2^5$. Similarly, we may set $j = ad_{11}\cdots ad_5$, with $0 \leq j < 2^7$. Now i and j are interpreted by CS as referring to X_i and Y_j , respectively, so the k^{th} bit of the word at location $ad_{11}\cdots ad_0 (=2^5 j + i)$ will be found stored in the core that lies at the intersection of row X_i and column Y_j in DIP_k .

We have said that each core has two stable states of magnetization that are used to represent the two possible values of a bit. These states correspond to the two possible orientations of magnetization in the torus. To have a useful storage device one must be able to produce either state in a given core at will, and to determine that state at will. How this is done for an individual core will be described in the following Section. The succeeding Section will show how the whole set of cores can be manipulated to accept or transmit a word at any of the 4096 locations.

.2 The core as a storage device

Let us consider a single core with a current-carrying wire linking it as in the drawing. Such a wire is called a winding of the core, and the number of times the wire links the core is called the number of turns of the winding. In the drawing there are three turns. The current in the winding produces a magnetomotive force, H , that is proportional to the product of the strength of the current and the number of turns of the winding. The magnetomotive force produces in the core a magnetic flux density, B .

The behaviour of B as a function of H for such a material as that of which the cores in GEORGE are made is shown roughly in the accompanying graph. If we assume that H is initially large and negative,

then B is also negative and relatively large. As H increases toward zero, B slowly increases through point 1 of the graph toward point 2. Further increase of H causes B to rise slowly to point 3, when it suddenly increases very rapidly to point 4. For larger values of H , the value of B is nearly constant, rising slowly as H increases. If now H decreases to zero, B goes back along the same curve to point 4, but then continues its slow decrease along the upper branch of the curve to point 5. Further decrease of H through negative values carries

B along the upper branch of the curve through point 6 and then suddenly down to the common point 1, after which B goes along the portion of the curve to the left of point 1 along which it started up initially. The whole curve is symmetrical with respect to the origin.

The possibility of using the core as a storage device depends on the properties revealed by the portion of the curve from point 1 through point 6 and back to point 1, the hysteresis loop. Roughly speaking, if the current is turned off at a time when B lies between points 1 and 4 on the loop, then B returns to point 2 or point 5, depending on whether it lay on the lower or upper branch of the curve when the current was turned off. Thus there are two relatively stable states of the core that may be identified with the two values of a bit. The value "0" is arbitrarily assigned to the state represented by point 5 of the curve, the value "1" to that represented by point 2.

The absolute value, H_0 , of the amount of magnetomotive force corresponding to point 1 or point 4 on the curve is called the saturation value of H. Since for a given core and winding H is proportional to the current, which in turn for a given circuit may be regarded as proportional to the voltage, we shall speak of applying a saturation pulse to the line of which the winding is a part. (For the time being we shall allow a pulse to be either positive or negative, contrary to our general convention. Later this will be rectified.) If a core is initially in state "0" or "1" with no voltage applied in the winding, then the core returns to that state at the end of any pulse whose magnitude remains less than saturation. The fact that this statement is only approximately true need not concern us since (a) the individual disturbing (i.e., less than saturation) pulses are kept small enough to guarantee that B will not get near the steep portions of the curve, and (b) the points to which B returns after such excursions are actually very much closer to points 2 and 5 than to the origin. It will be convenient to idealize our original graph by representing the hysteresis loop as a rectangle with sides parallel to the axes

as a rectangle with sides parallel to the axes and assuming the extreme values of B to be constant. This does not do too much violence to the facts, provided we recognize that when we later refer to the rate of change of B with respect to H this rate is not infinite at $\pm H_0$, but merely very large in small neighborhoods of these values of H. Similarly, the rate of change of B for large values of H is small rather than actually zero.

We have shown that writing in a core can be achieved simply by applying a positive or negative saturation pulse to the winding of the core. Reading involves a different

phenomenon. Let there be two distinct windings of a core, of which the first, labelled W for "write," will be used as was the sole winding of the

earlier drawing to control the state of magnetization of the core, while the second, labelled R for "Read," will be used as will now be described. As B changes under the influence of the current in W, a current is induced in R, whose voltage (assuming R a portion of a fixed circuit) is proportional to the rate of change, dB/dt , of B with respect to time. Since $dB/dt = (dB/dH)(dH/dt)$ and dB/dH is very nearly zero except near $H = \pm H_0$, we may assume (even at the beginning

or end of a pulse on W, where dH/dt may be fairly large) that the voltage on R will suffer a pronounced change only when B is traversing one of the vertical sides of the hysteresis loop, i.e., when the core is changing state. For example, let the core be in state "1" with no current applied in W. Now let a saturating pulse be applied to W. Then a positive pulse will be produced in R as the core changes state. Likewise, if the core is in state "0" initially, a negative pulse in W produces a negative pulse in R. The accompanying timing chart indicates roughly the relation between these pulses in the two cases. Note that the response of R to a change of state of the core is relatively slow.

The practical application of the phenomena just described to the technique of reading and writing in GEORGE is as follows: Let the core to be consulted be in state "a," and let "b" be the desired final state if the operation is Write. (1) According as the operation is Read or Write, let a toggle be cleared to 0 or set to b. (2) Apply a positive saturating pulse to W. (3) Pass the resulting output of R through an amplifier and discriminator which will then put out a standard (positive) pulse or not according as the core does or does not change state, i.e., according as $a = 1$ or 0. (4) Let the output of the discriminator set the toggle to a or not according as the operation is Read or Write. (5) Send a negative saturating pulse through W or not according as the toggle (whose value is now a or b according as the operation is Read or Write) contains 1 or 0.

In order to re-establish our conventions of representing every pulse as positive we adopt the following expedient. A positive pulse on W will be called W_p , a negative pulse on W will be called W_N . In a timing

chart a positive (resp. negative) pulse on W will be represented by a positive pulse on W_P (resp. W_N). Since we record in the toggle only changes in the core from 1 to 0, which produce positive pulses in R , negative pulses in R are ignored by the discriminator.

A possible logical diagram and timing charts for the operation of a single core are shown.