NASA TECHNICAL NOTE

NASA TN D-3640

A STORED PROGRAM COMPUTER FOR SMALL SCIENTIFIC SPACECRAFT

by Rodger A. Cliff

Goddard Space Flight Center
Greenbelt, Md.

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION • WASHINGTON, D. C. • OCTOBER 1966

A STORED PROGRAM COMPUTER FOR SMALL SCIENTIFIC SPACECRAFT

By Rodger A. Cliff

Goddard Space Flight Center
Greenbelt, Md.

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

For sale by the Clearinghouse for Federal Scientific and Technical Information
Springfield, Virginia 22151 – Price $2.50

ABSTRACT

A computer design is presented for use on small scientific spacecraft, such as the Anchored Interplanetary Monitoring Platform (AIMP). To meet the requirements of reliability, low power drain, light weight, small size, problem solving power, and flexibility; the computer has 1024 words of program memory and 512 words of data memory (both randomly accessed). In order to protect the program, the program memory is of the non-destructive readout type. The data memory is of the conventional read/write variety. Power drain and hardware are minimized by restricting the amount of parallel gating of information, the number of registers, and the number and complexity of instructions. A set of arithmetic subroutines occupies nearly half of the program memory, and was utilized in exploring the capabilities of the computer.

CONTENTS

A STORED PROGRAM COMPUTER FOR SMALL SCIENTIFIC SPACECRAFT*

by
Rodger A. Cliff
Goddard Space Flight Center

INTRODUCTION

Problem Statement

It is the aim of this research to design a "minimum computer." The intention is to design a system which would be classified as a true computer by those conversant with computer technology, but which would be far smaller and simpler than the usual spaceborne computer. Such a minimum computer should be programmable and should be able to do both arithmetic and logical operations.

It must be recognized that the design of a computer is as much an art as it is a science because there are so many degrees of freedom and so few constraints. The design choices made in the course of this research reflect the author's engineering judgment, and alternate solutions are of course possible. The choices made are not "logically derivable," and indeed if they were the problem would not be interesting. It should be understood that the arguments explaining why certain methods were used in place of others are included in an attempt to exhibit the author's rationale rather than to prove that a computer for this application must be built as described.

When it has been demonstrated that an extremely small computer can be built, and that it is at least marginally useful, then the design can be expanded upon by others to suit specific situations. The aim is to open up a whole spectrum of possible computer applications which are inaccessible to presently available larger machines because of restrictions on power supply, weight, and size. It is expected that the results of this research will benefit fields other than spacecraft instrumentation.

The Small Scientific Spacecraft

The majority of scientific spacecraft are small, overall dimensions being of the order of 2 to 3 feet. Their attitude is stabilized by spinning them about the axis of highest moment of inertia.

*Thesis submitted to the Faculty of the Graduate School of the University of Maryland in partial fulfillment of the requirements for the degree of Master of Science, 1966.

Many of the sensors in the experiments are directional. Therefore, the spinning motion of the spacecraft causes these sensors to perform a circular scan of the sky, which in itself is desirable. However, the scanning motion does pose a problem: the experimental data is most meaningful when collected in synchronism with the spacecraft spin, whereas the time-division-multiplex telemetry system commutates experiments at a fixed rate which is not related to the spacecraft spin rate. Therefore, buffer memory must be provided. A second problem is that data from experiments are typically highly redundant. Data transmission capability of the telemetry link is limited; therefore it is desirable to compress redundant data before transmission. Hard-wired special-purpose data handling devices presently serve the functions of buffer memory and data compressor on small scientific spacecraft. However, the sort of simple computer which is the object of this research should be able to replace some of the special-purpose devices and provide an additional advantage: the flexibility which results from using a stored program.

In order to more clearly show the constraints imposed, some characteristics of a typical small scientific spacecraft will be given. The AIMP (Anchored Interplanetary Monitoring Platform) weighs about 120 pounds, of which 20 pounds is the range and range-rate transponder, telemetry and data system, and programmers. The power system weighs 50 pounds or close to one-half of the total. Of the approximately 50 watts of power produced, 20 watts are taken by the range and range-rate transponder, telemetry and data system, and programmers. The transmitter alone uses 16 of these 20 watts, or more than 30 percent of the entire spacecraft load.*

If a computer can compress the experimental data enough to make it possible to reduce the transmitter power by more than the computer power drain, then a net power saving can be gained by using a computer. When deciding whether or not one can save power by using a computer one must also take into account the power which would have been used by buffer memories and hard-wired special-purpose data compressors. Even if a net power gain cannot be obtained by using a computer, the flexibility provided by stored programs may still make it desirable to use computers in small spacecraft.

PRE-DESIGN CONSIDERATIONS

General Criteria for a Spacecraft Computer

Before the design of the computer can be undertaken it is necessary to establish guidelines and criteria. Listed below, in order of importance, are six desirable qualities for a spacecraft computer, followed by an explanation of each. The first four are the most important; for if they are not met, the system cannot be included in a small spacecraft.

1. Reliability
2. Low Power Drain
3. Light Weight
4. Small Size
5. Problem Solving Power
6. Flexibility

*J. J. Madden, "AIMP Summary Description," NASA Goddard Space Flight Center Document X-672-65-313, August 1965.

Reliability

Reliability must be of primary importance for space-borne equipment. Placing a spacecraft in its assigned trajectory is an expensive process; therefore, it is imperative that it perform its appointed task for the prescribed length of time without failure. Preventive or corrective maintenance is not possible in the small unmanned vehicles we are considering.

Low Power Drain

Low power drain is necessary because spacecraft power supplies consist of batteries and solar cell arrays both of which are heavy. Unfortunately, the cost of a launch vehicle increases steeply with increasing payload weight. High power dissipation can also present a temperature control problem on spacecraft, because ultimately all excess heat generated must be dissipated by radiation.

Light Weight

Light weight (of the computer itself) is less important than power drain because when microelectronics and modern spacecraft packaging techniques are used, a computer will have a high power dissipation to weight ratio. To reduce overall spacecraft weight it is more important to reduce the computer's power drain than its weight.

Small Size

Small spacecraft require small-size subsystems, but this is not as critical as weight. Available packaging techniques are quite good, and spacecraft usually have some empty space anyway. However, size is important enough to warrant the use of microelectronics and the best packaging available.

Problem Solving Power and Flexibility

Given that a computer meets the above criteria and is therefore capable of being put aboard a small spacecraft, one may worry about characteristics which affect its usefulness. Two such characteristics are problem solving power and flexibility, which are highly interrelated. If a decision must be made between them, however, problem solving power is most important because spacecraft are hand-built one at a time and sub-systems may be modified or redesigned for each flight if necessary. Caution must be exercised because many of the advantages of a computer are fully realized only if a system, sufficiently flexible to require little modification, is created.

Research Emphasis

Light weight and small size is a packaging problem, rather than an electronic one; therefore, these criteria will not be explicitly involved in the computer design. The computer will, of course,

be small and simple in order to meet the lower power requirement. The characteristics of Texas Instruments Series 51 integrated circuits are typical of circuits that might be used to construct the computer. Therefore, these characteristics will be kept in mind during the design process. Reliability can be enhanced by incorporating hardware redundancy into the design. Deciding the manner in which this redundancy should be applied is in itself a difficult problem. Therefore reliability will not be specifically emphasized in the research. If an opportunity to improve reliability presents itself, advantage will be taken of it. Otherwise, it will be the aim to design as simple a machine with as few components as possible. The remaining criteria, to which the most emphasis will be given, are low power drain, problem solving power, and flexibility.

Obtaining Low Power Drain

The first step in obtaining low power drain is to organize the computer in such a way that the number of logic gates is minimized. As a start, it is possible to eliminate the many handy but nonessential features of the ordinary computer. Indirect addressing, address modification by index registers, floating point hardware, special table look-up instructions, etc., can be eliminated if necessary. This simplifies instruction decoding and control logic. Programs must be longer and more complex if these features are not included. It is hoped that this will not be a serious handicap in the case at hand because the data rate both into and out of the computer will be low compared to those usually encountered in the computer field. Furthermore, difficulty of programming is not a serious drawback for a spacecraft computer because it will be programmed infrequently and by competent programmers. Under the conditions for which the spacecraft computer is being designed, it is acceptable if involved programs, which are difficult to write, must be used in order to minimize the amount of computer hardware.

To reduce the number of logic gates still further, arithmetic and logical operations should be done serially rather than in parallel. An N-bit serial adder, for instance, requires fewer gates than an N-bit parallel adder. Similarly, all data switching and transferring and all input and output should be done serially. Otherwise, additional power is required for each possible position to which data may be switched. Another way to reduce the number of logic gates is to minimize the number of registers. Reducing the word length shortens each register and eliminates more gates.

Desired Instructions

Before beginning the computer design, an enumeration should be made of the operations it is expected to perform. These operations may be grouped in several classes: arithmetic, logical, data transfer, and control.

Of the possible arithmetic instructions, fixed-point add and substract are easiest to implement in hardware. The other desirable operations (fixed-point multiplication and division, and floating-point addition, subtraction, multiplication and division) can be programmed using the fixed-point add and subtract instructions. This approach is in accord with the desire to hold the amount of hardware to a minimum even at the expense of increased program length and complexity.

The three logical instructions OR, AND, and COMPLEMENT are so useful and so easy to implement that they will probably all be included in the computer's instruction set. The Exclusive Or function is also readily available from the adder by constraining the carry to be zero. Of course if an absolute minimum computer is to be built, only one logical instruction is necessary; either NOR or NAND will suffice.

Quite a number of data transfer instructions are desirable. Two very important instructions are: LOAD, to read data from the memory; and STORE, to write data into the memory. Another necessary pair of instructions are INPUT and OUTPUT. They are used for communicating with the computer's environment. Data transfer between the various registers also must be provided. Finally, instructions must be provided to shift or rotate data within one register without affecting the other registers.

Control instructions are also necessary. Either conditional transfers or conditional skips are required. Sense instructions to produce program branching under external control are a useful type of conditional instruction. If conditional skips are used, unconditional transfers should also be included to assist in control of branching.

In the course of designing the computer and experimenting with programs other instructions may be found to be desirable. It is also possible that the hardware configuration produced will suggest that certain useful instructions could be included with a minimum of additional complexity. If so, these additional instructions will be appended to the computer's instruction set. In any case, the instructions enumerated in this section form a point of departure from which the computer design may be undertaken.

Interface

Interface with Spacecraft

Simplicity of the computer's interface with the spacecraft is desirable for two reasons. First, a simple interface minimizes the number of wires between the computer and the rest of the spacecraft. This increases reliability and decreases weight by eliminating as many unreliable, bulky interconnections as possible. Second, a conceptually simple interface (one with a minimum number of interactions between the computer and the spacecraft) facilitates computer and spacecraft testing. A simple interface also makes it easy to use the computer in a number of different applications without extensive modification. Since the internal organization of the computer is serial, and because one wire is simpler than many, data will enter and leave the computer in serial. To prevent timing difficulties, the computer will supply the spacecraft with strobe pulses for both input and output. Telemetry system and experiment status can be communicated to the computer through a small number of sense lines. A trapping feature may prove useful also. This will be determined when experimenting with programs.

Interface with Memory System

It is expected that the development of a low-power, flight-worthy memory will be undertaken as a separate project. Therefore, a well defined interface must exist between the memory and the

computer. Since it takes almost the same amount of energy to read or write one bit of a memory as it does to read or write an entire word, the memory access should be parallel rather than serial. To obtain the desired stored-program flexibility, the memory must be randomly accessed. Therefore, either parallel address information must be supplied to the memory or it must contain an address register which would duplicate registers already in the computer. Parallel addressing will be employed to prevent this unnecessary complexity and additional power drain. To complete the interface, control and timing signals are required. The simplest method has the computer supply the memory with either a READ or a WRITE pulse. Then the memory feeds the computer a MEMORY BUSY signal until it has finished cycling.

Now there are two basically different functions that the memory must perform. It must store data, which will repeatedly be modified, and it must also store the program. Somehow the program must be protected. Somewhere in the machine the program must be stored in a form which can be accessed non-destructively. There should be no possibility that a temporary malfunction could scramble the program. Possible configurations are: (1) a main memory, some locations of which are non-destructive; (2) a main destructive-type memory and an auxiliary non-destructive program store with which to refresh the main memory; and (3) two separate memories, a destructive one for data and a non-destructive one for program. If a non-destructive read-only memory is used, special provision must be made for instruction modification.

To complete the interface description; exact timing, voltage levels, etc. must still be specified. However, the interface just described is sufficient and work could begin on the computer, the memory system, and the spacecraft.

DESIGN OF A SPACECRAFT COMPUTER

Data Flow

The computer will perform operations serially for reasons previously discussed. Access to the memory will be parallel, however, because it takes little more energy to read a word out of a typical memory than it does to read out a single bit. The typical memory elements produce a pulse output. For this reason an instruction register must be provided to hold instructions while they are being decoded and executed. Furthermore, since the operations such as addition require two operands which must be available throughout execution, two more registers need to be provided. Therefore at least three registers of one word each must be provided in the computer. A memory and three registers will be used as a point of departure for beginning the computer design.

Data flow paths for the computer are indicated in Figure 1. The path which requires least discussion is the one from the Memory to the Instruction Register. This path is clearly necessary to the operation of the computer. Paths are shown between Data Register 1 and Data Register 2 to permit operands to be shuffled back and forth between them as required. Two more obviously necessary paths are the ones from the two data registers to the Arithmetic and Logic Unit.

A decision must be made as to which register should receive the output of the Arithmetic and Logic Unit. To minimize the number of possible paths, and thus to minimize the switching hardware required, only one path should be provided for this output. Similarly, only one path should be provided from the Memory to the data registers.

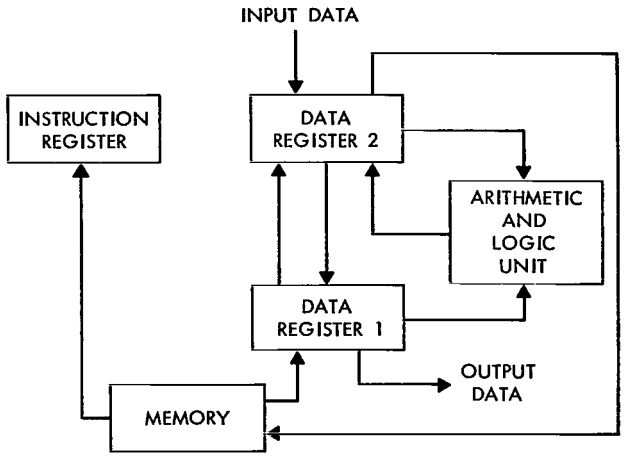

Figure 1—Data flow paths.

Assuming that the Memory feeds operands to Data Register 1, as shown in Figure 1, the output of the Arithmetic and Logic Unit should be connected to Data Register 2. This connection will save many instructions in programs which, for example, add a series of numbers together. If results were left in Data Register 1 instead, then they would have to be moved to Data Register 2 before each additional operand was read out of the memory. Furthermore, because results are found in Data Register 2 at the end of an operation, it is natural that the path from the data registers to the Memory be provided from Data Register 2. Otherwise the results would first have to be transferred from Data Register 2 to Data Register 1 before they could be stored.

The next items to be considered are the input data and output data. Input data should go to Data Register 2 because from there it may either be directly stored, or another operand can be read out of Memory into Data Register 1, and computation can be done on the input data. Output data is taken from Data Register 1 because it is expected that the computer will be serving a buffering function and that output data will be stored in the Memory until it is called for.

All of the paths shown in Figure 1, except two, are essential to the operation of the computer. The paths between the two data registers could be eliminated because there are parallel paths, either through the Arithmetic and Logic Unit or through the Memory. However, the utility of these two paths, which will become apparent when programming is discussed, and their ease of implementation make it advantageous that they be retained.

Program Protection

In order to have high reliability it is essential to protect a spacecraft computer program from transient malfunctions. The use of the two separate memories (one for program and one for data) seems to be the most promising approach because it provides additional advantages.

Figure 2 shows the memory system of a conventional computer. It stores both instructions and data. When the computer requires data the Address Switch connects the Memory address inputs to the Instruction Register and the Data Switch connects the Memory data output to the Data Register. On the other hand, when it is time to fetch the next instruction, the Address Switch

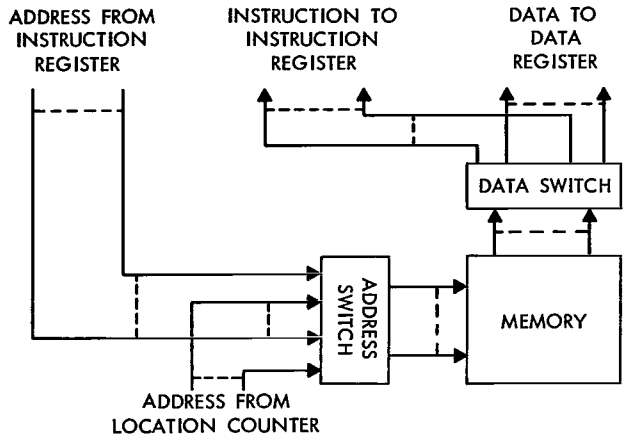

Figure 2—Conventional computer.

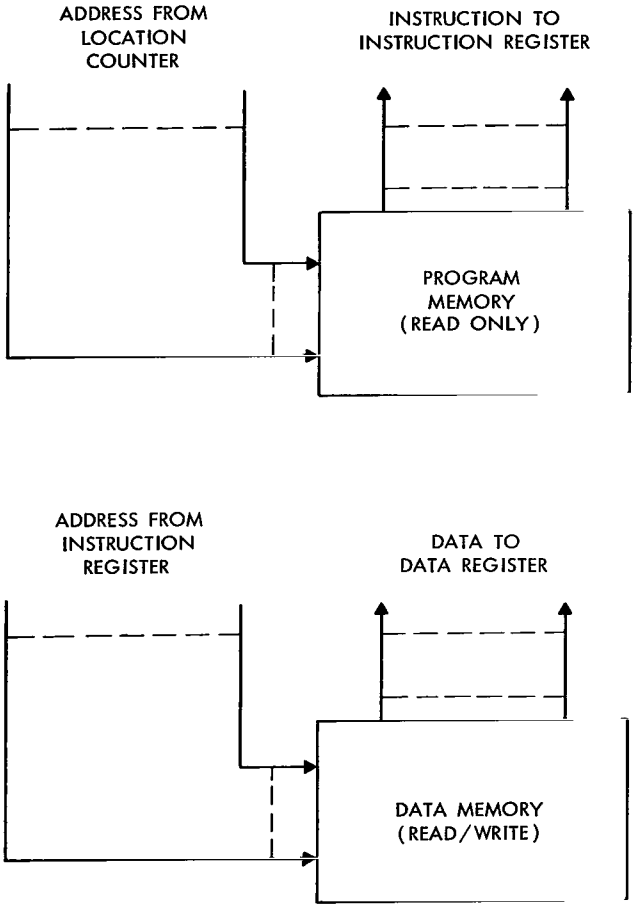

Figure 3—Spacecraft computer.

selects the Instruction Location Counter as the address source and the Data Switch sends the Memory output to the Instruction Register. Internal to the Memory there is no distinction between data and instructions and they are treated alike. This configuration has the advantage that only one memory system with all its associated electronics is required. Furthermore, it is not necessary when designing the computer to know how much of the memory capacity will be used for instructions and how much for data.

When two separate memories are used, as shown in Figure 3, both the Address Switch and the Data Switch are eliminated. This eliminates a large amount of parallel information switching and thereby saves much power.

Two memory systems require more components than one system of equivalent capacity; however, it was decided to pay that price to obtain protection for the program. A power saving is actually anticipated from the use of two separate memories, because spacecraft memories are designed to draw power only when they are reading or writing. In the first analysis, the number of memories does not affect the power drain since the number of accesses is not changed. In fact, a "read only" memory tends to require less power than a "read/write" memory, and over half of all memory accesses will be for instructions. Therefore, two separate memories will use less power than a combined memory in this application.

Comparison of Figure 4 with Figure 1 shows the changes resulting from the decision to use separate Program and Data memories. Notice that there are now a pair of data paths between the Instruction Register and Data Register 2. These paths allow the computer to operate on instructions as data when this is desirable. Although this ability is infrequently used in conventional computers, it is necessary for this computer because of the absence of address modification by index registers. To return from a subroutine, for instance, the program appends the return

address (which has been stored in the Data Memory) to a transfer instruction. The resulting transfer instruction is then transmitted from Data Register 2 to the Instruction Register and executed. A more detailed discussion of subroutine linkage will be found in the next section.

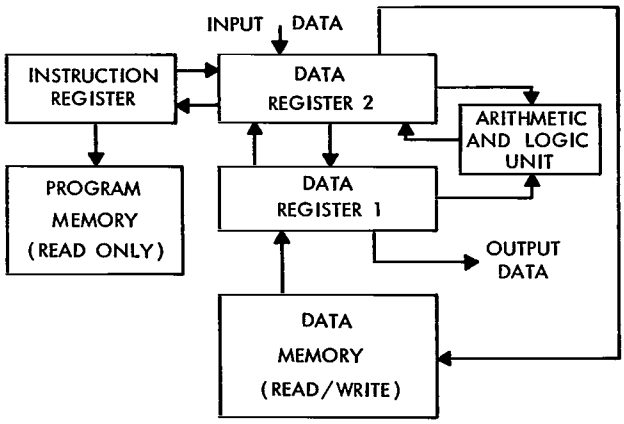

Figure 4—Data flow paths for computer with two memories.

Constants, as well as instructions, are stored in the Program Memory. All contents of the Data Memory must be considered volatile and subject to destruction at any moment. When a constant value, like +1, is required by a program it is first loaded into the Instruction Register from the Program Memory and then transferred to Data Register 2. Data Register 2 is chosen to communicate with the Instruction Register because this minimizes the number of instructions required both for instruction modification and introduction of constants.

One further feature is required if the programmed operation of the computer is to be effectively protected. Because subroutine return addresses are stored in the Data Memory during subroutine execution they may be accidentally damaged. Then, when the subroutine attempts to return, it will transfer to the wrong location, which might even contain a constant instead of an instruction. Another possibility is that the Instruction Location Counter could be disturbed by some transient phenomenon, producing similar results. In either case, the odds are that eventually the computer would settle down and resume following the program in a sensible fashion. However, there is no assurance of this. It is also possible that some unusual data could find an undiscovered program "bug" and cause the computer to "hang up."

To prevent these catastrophes, a trapping provision is included, whereby an external signal forces the instruction location counter to zero. Then, beginning at location zero the computer executes a program designed to put everything in its "ground state" and goes on its way as though the malfunction had not occurred. It is desirable that these trapping signals be supplied to the computer periodically from an external clock so that the computer cannot generate nonsensical data for too long a time before it is reset.

Word Length

The word length of the computer should be as short as possible to minimize hardware complexity and power dissipation. However, the word length must be long enough to accommodate memory addresses. Now certainly 500 to 1000 memory locations should be provided as a bare minimum. This requires 9 or 10 bits for addresses, unless the memory is arranged in sectors which are selected by additional instructions.

Another approach is to require two words for each instruction, or at least for certain classes of instructions. This complicates instruction decoding and lengthens programs. It also increases the number of memory accesses and thereby increases power drain. Therefore, as a minimum,

the word length should be, for instance, 10 bits plus a few bits for operation code. Likely candidates are in the 12 to 16 bit class.

Spacecraft telemetry systems usually transmit data in 3 or 4 bit bytes. For this reason it is desirable that the word length be a multiple of 3 or 4. Otherwise there will be difficulties in packing and unpacking data. Any word longer than 9 or 10 bits is probably sufficient for the data usually encountered. Reasonable choices, therefore, are 12, 15, or 16 bits. Because it is the object of this research to design as small a computer as practicable, 12 bits has been chosen as the word length. The length 12 also has the desirable property of being divisible by 1, 2, 3, 4, and 6. Considerable squeezing is required to fit all the necessary instructions into a 12-bit format, but this has been accomplished.

One possibility, which was discarded earlier is to choose two word lengths, one for instructions and one for data. Then only the Instruction Register need be lengthened to provide more address bits. Unfortunately, to do this complicates instruction modification and storage of constants. There is another reason why all the words should be of the same length. If they are, then all the registers can be identical modules.

Instruction Coding

The objective, in designing the instruction coding scheme, is to make decoding as simple as possible and still get all the required instructions into a 12-bit format. The decoding tree in Figure 5 shows how this is done.

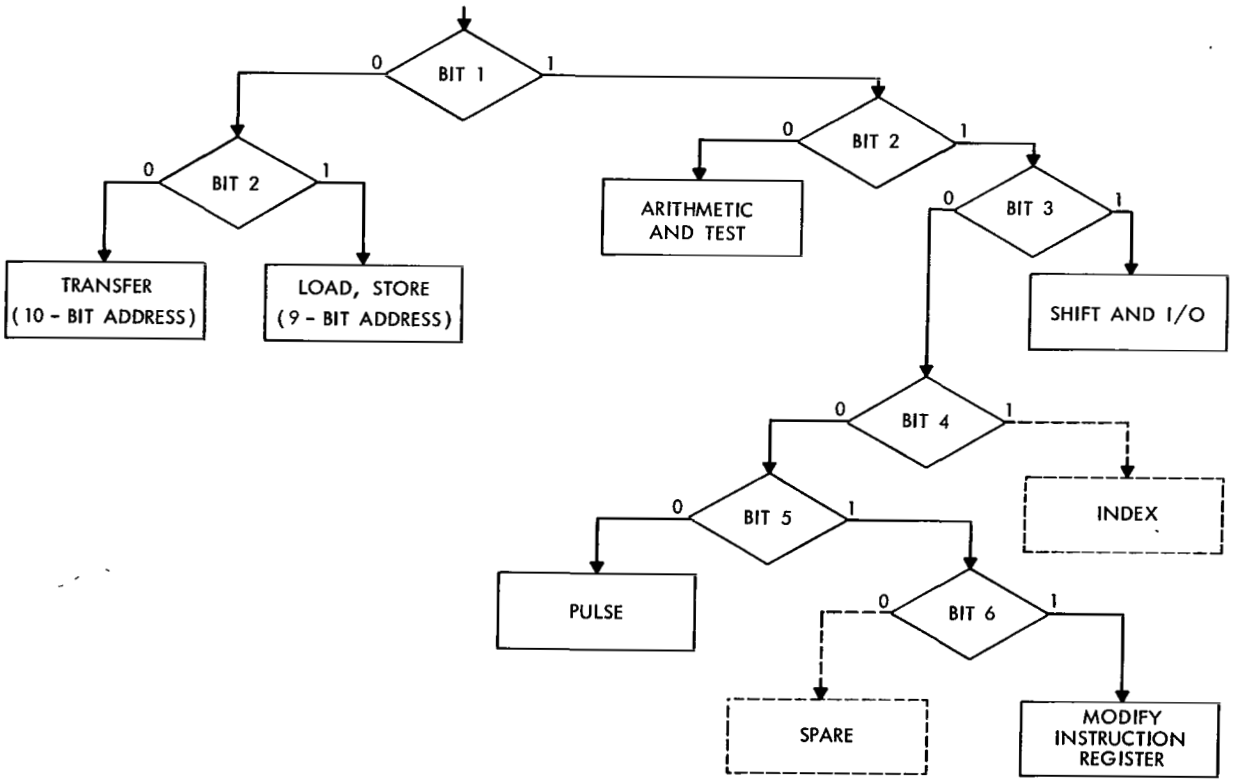

Figure 5—Decoding tree.

Transfer

The first instruction which must be accommodated is the unconditional transfer instruction. It is the only addressed instruction which refers to the program memory. This instruction takes the form

00 XXXXXXXXXX,

where 00 is the operation code and X. . .X is a 10-bit address. There is capability, therefore, for a program memory of 1024 words.

Load and Store

The next instructions to be considered are those to read and write the data memory. They are called LOAD and STORE and take the form

01 Y XXXXXXXXX,

where 01 determines that the instruction is of the data memory class, Y is the operation code (Y = 1 means LOAD and Y = 0 means STORE), and X. . .X is a 9-bit address. This allows 512 words of data memory.

Arithmetic and Test

Moving now to the right hand side of Figure 5, and starting down the tree, we find arithmetic and test instructions

10 YYY TTT CCCC.

The arithmetic and logic operations which have been chosen are listed in Table 1. "No Operation" is included so that the tests, which are explained in a subsequent paragraph, can be made without disturbing the contents of Data Register 1 and Data Register 2.

"Add" is self explanatory. It would not have been necessary to include "Subtract" but it is very easy to implement when one already has a serial adder. Subtraction is accomplished by taking the two's complement of the contents of Data Register 1 and adding it to the contents of Data Register 2. The two's complement is formed by the familiar procedure of inverting each bit and then adding 1. Inversion of the bits is accomplished by taking the output of Data Register 1 from the "false" side instead of from the "true" side which is normal. Adding 1 is accomplished by setting the carry flip-flop to 1 before beginning the operation. Additional discussion of two's complement arithmetic, including the reasons for choosing it, can be found in the next section.

Table 1

Arithmetic and Logic Operations.

YYY	Operation	Mnemonic
000	No Operation	NOP (or blank)
001	Add	ADD
010	Subtract	SUB
011	Or	OR
100	And	AND
101	Complement	COMP
110	Exclusive Or	EOR
111	Zero	ZERO

Table 2

Tests and Test Codes.

TTT	Test		Mnemonic
000	None		blank
001	Carry Flip-Flop	0	C
010	Data Register 2	0	Z
011	Data Register 2, Bit 1	0	B1
100	Interrupt Line 1	Off	I1
101	Interrupt Line 2	Off	I2
110	Interrupt Line 3	Off	I3
111	Interrupt Line 4	Off	I4

"Or," "And," and "Exclusive Or" perform the specified operation on a bit-by-bit basis using the contents of Data Registers 1 and 2. "Complement" operates successively, bit-by-bit, on the contents of Data Register 2 and "Zero" shifts zeroes into the left hand end of Data Register 2.

The next field of the arithmetic and test instruction specifies the test to be performed at the end of the specified operation. Test codes and their corresponding tests are listed in Table 2. If no test is specified, or if a test is specified but not satisfied, the operation proceeds to the next sequential instruction. If, however, a specified test is satisfied the computer skips the next sequential instruction and proceeds from there. This sort of conditional branch is easy to implement by merely sending an extra pulse to the instruction location counter whenever a test is satisfied.

The test of the carry flip-flop is used for sensing overflow. The test of Data Register 2, bit 1, senses the sign of a number. The zero test needs no explanation. The last four tests sense external conditions and are used for synchronizing the computer with external devices, such as a spacecraft telemetry system.

The last field of this type of instruction, CCCC, specifies the number of bits to be operated on. Normally, a count of 12 will be used in order to operate on a whole word. However, many examples will be found in the programs contained in the next section where it was advantageous to use a count other than 12. Any count from 0 to 15 may be specified. For counts less than 12, some high-order bits will not be operated upon and for counts greater than 12, some low-order bits will be operated upon twice. During Arithmetic and Test operations the output of Data Register 1 is connected back to its input so that the contents are rotated but otherwise not disturbed.

Shift and I/O

Proceeding further down the decoding tree again, the next class of instructions to be encountered is the "Shift and I/O" category. They contain

111 AA BB Z CCCC.

The bits in fields AA and BB control the setting of the switches at the inputs to the data registers (Figure 6). The possible combinations are shown in Table 3. The "Off" setting allows either register to be shifted without affecting the other register. This is particularly useful. Examples of uses for various combinations of these settings can be found in the following section.

The spare input to Switch 1 could serve any of a number of uses. It can be used as a second data input channel, as a source of 1's or 0's, or it could be connected to a tap on

some intermediate stage in either Data Register 1 or 2. No particular use will be specified at this time.

The Z field controls the output function. Data is always present on the output line, as shown in Figure 6, but no output data strobe pulses occur except when a "Shift and I/O" instruction, which has 1 in the Z field, is executed (Table 4). This class of instruction can do

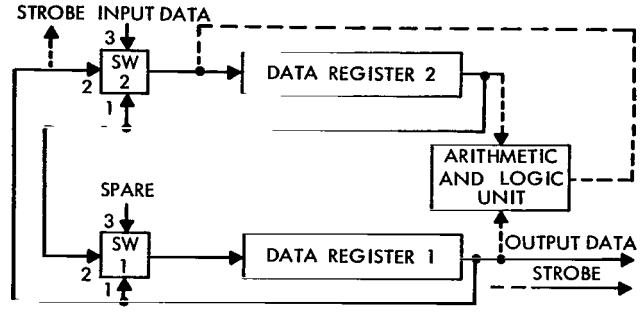

Figure 6—Data switches.

Table 3

Possible Combinations of Switch Settings.

AA	SW1 Setting	Mnemonic
00	Off (no shift pulses to Data Register 1)	OFF (or blank)
01	Data Register 1 output	DR1
10	Data Register 2 output	DR2
11	Spare	-
BB	SW2 Setting	Mnemonic
00	Off (no shift pulses to Data Register 2)	OFF (or blank)
01	Data Register 2 output	DR2
10	Data Register 1 output	DR1
11	Input (pulse Input Data Strobe line)	IN

input and output, or input and shifting, or shifting and output simultaneously. The number of input bits, shifts, or output bits is specified by the field CCCC. Again any count from 0 to 15 may be specified, but 12 will be most common.

Table 4

Z Field Output.

Z	Output	Mnemonic
0	Data, but no strobe pulses	blank
1	Data, and strobe pulses	OUT

Index

Room has been provided in the decoding tree for index register instructions. Figure 7 shows a way of reducing the hardware associated with an index register. Instead of adding the contents of the index register to the address, (which is out of the question because it would require an adder) the two are Or'ed together. This is not as general, but it can be done with simple diode gates which use few components and no power. The index register and the And gates will require power, however. Assuming a dissipation of 3 mw per flip-flop or gate and a 6-bit index register for each memory, then over 70 mw would be added to the computer's power drain, exclusive of the extra decoding hardware. Unfortunately no bits are available for tags in the addressed instructions, so a "tag next memory access" instruction would be required. It is concluded, therefore, that unless

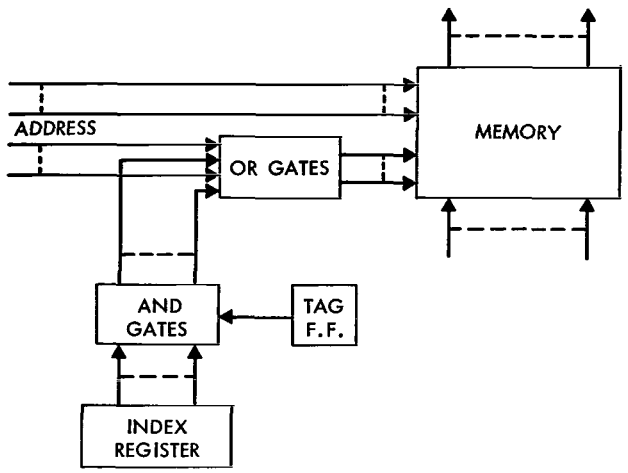

Figure 7—Index register.

Table 5

Operations for Bit N.

N	Operation	Mnemonic
0	Decrement XR by 1; skip if result 0	DXZ
1	Tag next memory access	TAG

Table 6

Operations for Bit X.

X	Operation	Mnemonic
0	Pulse specified combination of lines and halt	PUH
1	Pulse specified combination of lines and go on	PUG

a special need for some particular application presents itself the index register feature, and its associated decoding, will not be implemented.

The codes reserved for index registers are

1101 M 1 XXXXXX,

and 1101 M 0 N 00000.

The bit M specifies which memory is involved. If the 6th bit is a 1, the next 6 bits, X. . .X, are set into the specified index register. If the 6th bit is a 0, then bit N specifies one of the two operations shown in Table 5. The field 00000 has no effect and may contain any combination of bits.

Pulse

It is anticipated that the computer will be operated in conjunction with other devices. To aid in synchronization, a class of pulse instructions is included,

11000 X BBBBBB.

The field BBBBBB specifies what combination of 6 output lines will be pulsed. The bit X determines whether or not the computer halts (Table 6). When the computer halts, its power is shut off and the computer lies dormant until the start line is pulsed from an external source. Then it proceeds to the next instruction.

Instruction Register

There are two very valuable instructions which affect the contents of the instruction register. When either of these is decoded, a latch is set so that the operation can continue while the contents of the instruction register change. These instructions have the form

110011 XX 0000.

The field XX controls the operation and the field 0000 can be any combination of bits (Table 7).

Spare

The class of instructions beginning with the bits 110010 is a spare class. If the computer attempts to execute an instruction of this class, the result is a "pause." No further operation is

performed until the start line is pulsed from an external source. The power is left on and the contents of the data registers are not disturbed.

Table 7

Operations for Bit XX.

XX	Operation	Mnemonic
01	Read next instruction into Instruction Register and then shift it into Data Register 2. Next instruction to be executed is the one following the one shifted into Data Register 2.	MIN
10	Shift contents of Data Register 2 into Instruction Register and execute.	MIX

An Alternative

Programming experience shows that the number of instruction locations required, at least for certain types of programs, is much greater than the number of data locations required. For instance, the arithmetic subroutines described in the next section require 427 instruction locations and 10 data locations. If it becomes necessary, the instruction memory size can be doubled from 1024 words to 2048 words without resorting to a sectoring scheme. To do this, the load and store instructions are moved to the location in the decoding tree presently reserved for index instructions. Then the transfer instruction would have the form 0 XXXXXXXXXXX, with 11 bits for address. The LOAD and STORE instructions would be

1101 Y XXXXXXX.

There are 7 address bits, therefore, 128 data memory locations would be possible. The resulting ratio of instruction memory size to data memory size (about 40 to 1) closely matches the requirements of the arithmetic subroutines.

Summary of Operation Codes

Appendix A contains a list which summarizes the operations available in the computer in its most likely form. Those operations which are optional have been marked by an asterisk in this list. If an absolute-minimum machine is to be constructed, the optional instructions may be deleted. However, deletion of these instructions will significantly increase program length and complexity. It is the author's opinion that the complete instruction set as listed represents a reasonable compromise between excessive sophistication and extreme simplicity of the computer hardware.

Block Diagram

Now that the operation codes have been determined, and the functions they are to cause the computer to perform have been defined, the remainder of the design process is quite straightforward. For this reason the intimate details of a complete logic diagram will not be discussed. The novelty of this computer lies in the design features which were covered in the first four subsections. However, an explanation of an overall block diagram, which is found in Figure 8, is in order. Most of the parts of this diagram have been shown in previous figures but it is beneficial to consider how they fit together.

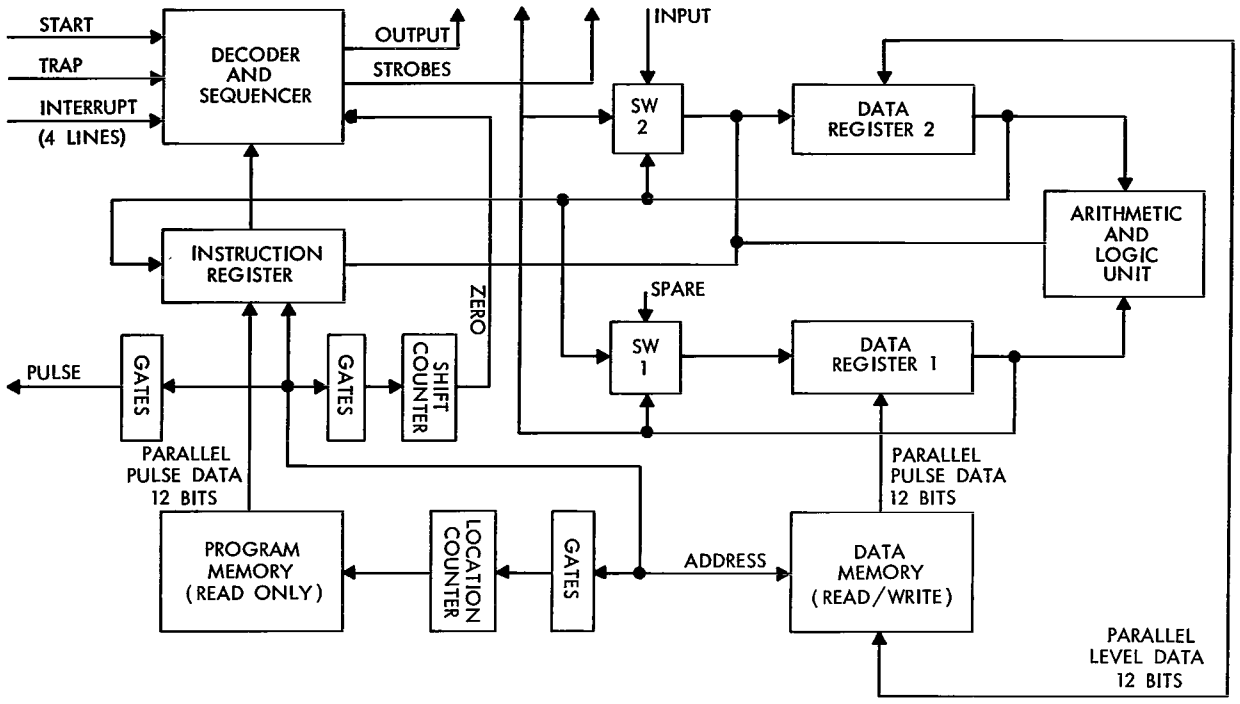

Figure 8—Block diagram.

In the upper left hand corner of Figure 8 there is a box called Decoder and Sequencer. Its function is to decode the instruction residing in the Instruction Register and to then send appropriate control signals to each of the other boxes in the diagram. In order to avoid undue congestion in the drawing, these control lines are not shown; however, a listing of them is contained in Table 8.

Table 8

Control Signals.

Block	Control Signals Required But Not Shown in Diagram
Instruction Register	Shift Pulses
Data Register 2	Shift Pulses
Data Register 1	Shift Pulses
Program Memory	Read Pulse
Data Memory	Read Pulse, Write Pulse
Output Pulse Gates	Output Pulse
Shift Counter Gates	Set Pulse
Location Counter Gates	Transfer Pulse
Shift Counter	Shift Pulses
Location Counter	Advance Pulse, Zero Pulse
SW2	Select Input
SW1	Select Input
Arithmetic and Logic Unit	Select Operation, Shift Pulses

The Instruction Register, Data Register 2, and Data Register 1 are identical 12-bit shift registers. As shown in the diagram, they shift from left to right. Serial input is shown at the left end and serial output at the right end. Data Register 1 requires parallel inputs and Data Register 2 requires parallel outputs. The Instruction Register requires both. These registers shift on the receipt of shift pulses from the Decoder and Sequencer.

The method of transferring data in parallel from one of the memories into one of these registers is of interest. When a memory has been accessed, it

first puts out a clear pulse which zeroes the register. Then the memory puts out a pulse on each bit line which is to transmit a 1, after which it turns itself off and remains dormant until accessed again. In this way there are no gates required and a considerable power saving and a reduction of complexity are achieved. No gating is required between Data Register 2 and the Data Memory either, because the memory samples its input data only when commanded to write. In fact, all necessity for parallel gating anywhere in the computer has been eliminated except for loading the Location Counter, loading the Shift Counter, and gating the Output Pulses.

At the beginning of each instruction which requires shifting, the Decoder and Sequencer opens the Shift Counter Gates and the Shift Counter is loaded from the count field in the Instruction Register. Then the Shift Counter counts down to zero. When it reaches zero, the Decoder and sequencer terminates that operation and again accesses the Program Memory for the next instruction.

The Location Counter is incremented each time the memory is accessed. In this way the instructions are executed in sequence. If a successful test instruction is executed, an extra pulse is added to the Location Counter and one instruction is skipped. To execute a transfer, the Decoder and Sequencer opens the Location Counter Gates and the Location Counter is loaded with address bits from the Instruction Register. When the Decoder and Sequencer receives an external TRAP command, it zeroes the Location Counter and the next instruction will be taken from location 0.

The Arithmetic and Logic Unit is quite simple. Its main constituent is a serial adder which is composed of one full adder and a carry flip-flop. Subtraction is accomplished by using the adder, as was previously explained. The Exclusive Or function is obtained from the adder by constraining the carry flip-flop to be in the zero state. The other functions, And, Or, and Complement are extremely easy to generate.

This completes the consideration of computer design. The remainder of the research is devoted to finding out how well this particular computer would be able to perform certain useful tasks on-board a spacecraft.

PROGRAMMING THE COMPUTER

Number Systems

Numerical Data

There are several ways that numerical data may be coded in an electronic system. Digital computers are restricted to binary coding by the available technology. There are, however, a number of possibilities within the binary regime. Figure 9 shows some of the choices which may be made. The unshaded boxes denote primary choices. It is expected that both fixed-point and floating-point formats will find use in this machine. Fixed-point will be used where speed is important and when there is little room in the memory for arithmetic subroutines. Floating-point operations will be slow and the programs will require a large amount of space, but they will find use where these factors are not of primary importance. Floating-point has the advantage of

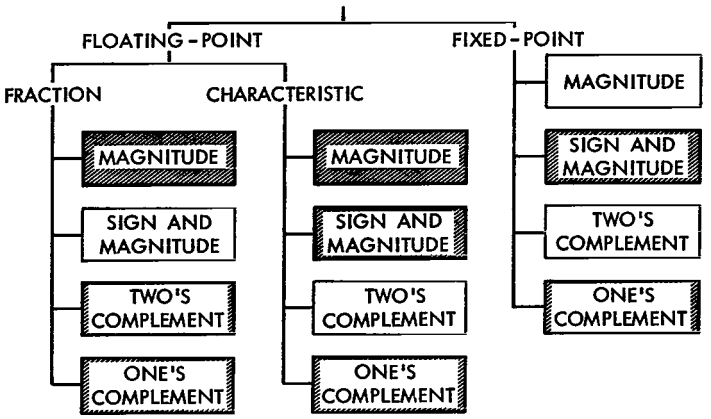

Figure 9—Numerical data.

accommodating wide dynamic range. Also, the programmer need not worry about keeping track of the location of the binary point.

Fixed-Point

Fixed-point coding will be considered first because it is simplest and because a floating-point number is composed of two fixed-point numbers. There are four fixed-point binary number systems which are commonly used in computers; they are "magnitude," "sign-and-magnitude," "two's-complement," and "one's-complement." These systems can be used to express integers, pure fractions, and mixed numbers depending on the location of the binary point. As an example, Table 9 gives all the possible 3-bit words and their interpretation as integers and fractions in the various systems.

Table 9

Number Systems.

Binary Code	Integers				Fractions			
	Mag.	Sign and Mag.	Two's Comp.	One's Comp.	Mag.	Sign and Mag.	Two's Comp.	One's Comp.
111	7				7/8			
110	6				3/4			
101	5				5/8			
100	4				1/2			
011	3	+3	+3	+3	3/8	+3/4	+3/4	+3/4
010	2	+2	+2	+2	1/4	+1/2	+1/2	+1/2
001	1	+1	+1	+1	1/8	+1/4	+1/4	+1/4
000	0	+0	+0	+0	0	+0	+0	+0
111		-3	-1	-0		-3/4	-1/4	-0
110		-2	-2	-1		-1/2	-1/2	-1/4
101		-1	-3	-2		-1/4	-3/4	-1/2
100		-0	-0	-3		-0	-0	-3/4

If the binary point is understood to be at the extreme right-hand end of the word, one obtains the integers listed in the first four columns of Table 9. The magnitude system is simplest, but it will not handle negative numbers. The other three systems use the left-most bit for a sign so that both positive and negative numbers may be represented. The remaining bits express the size of the number. At the other extreme, the binary point may be assumed to be at the left-hand end of the word (but not left of the sign bit). Then the binary words of Table 9 represent binary fractions,

which are listed in the rightmost four columns. To express a mixed number the binary point is placed in between the above extremes. In any case, the binary point is located at the discretion of the programmer since it does not affect the way the computer must operate. However, the design of the computer does determine which of the four number systems can be used efficiently.

It was stated in the previous section that subtraction was accomplished in the hardware of the computer by using the two's-complement. The two's-complement system is used because it is best suited to addition and subtraction in a serial machine. This is determined by examining the four systems and evaluating their performance.

Figure 10 outlines the logic required for addition in each of the systems. The magnitude and two's-complement systems use the same hardware and are therefore compatible. The magnitude system has two advantages; it provides the possibility of overflow detection, and it will accommodate numbers twice as large as the other systems for the same number of bits in a word. The great disadvantage of the magnitude system is that it will not handle negative numbers. Overflow is detected in the magnitude system by testing the carry flip-flop after an addition; if it is on, there was an overflow. This test is not meaningful when two's-complement numbers are used because the carry flip-flop is left on after an addition if the sum of two negative numbers does not overflow or if a positive and negative number produces a positive result.

One's-complement addition is accomplished by using the same hardware which would be used for magnitude or two's-complement addition, except for one important modification; if the carry

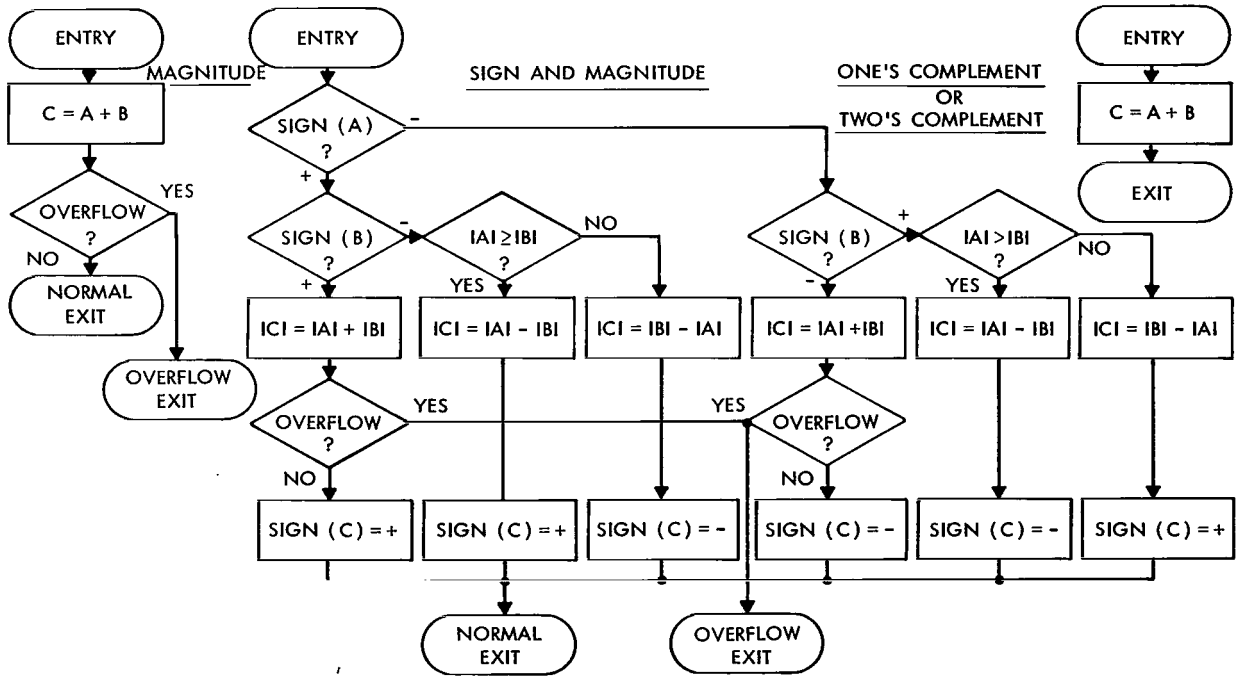

Figure 10—Fixed-point addition.

flip-flop is on after the addition, 1 is added to the result. It is easy to see that it is necessary to do this by looking at the examples in Table 9. Herein lies the disadvantage of one's-complement. To add 1 after the completion of the usual addition doubles the time required for the operation. In a serial machine the add time is large anyhow, and this additional increase is definitely undesirable. Furthermore, if one's-complement hardware were implemented, although it could be used for magnitude numbers also, there would no longer be the facility for detecting overflow.

Sign-and-magnitude coding of numerical data would be preferable, were it not for the undue complexity of the logic required. Figure 10 outlines this logic. Among the advantages are overflow detection, ease of interpretation of data by programmers, and desirability for multiplication and division. These advantages notwithstanding, sign-and-magnitude hardware will not be implemented because of its complexity.

Although two's-complement addition and subtraction are the only arithmetic operations which have been implemented in hardware in this computer, any operation in any number system can be programmed. Two's-complement arithmetic will be most frequently used of course. Later in this section, programs to add, subtract, multiply, and divide in sign-and-magnitude and in floating-point will be discussed.

Floating-Point

The two fixed-point numbers which make up a floating-point number are called the fraction and the characteristic, denoted by F and C respectively. A numerical quantity A is expressed in this system as

$$A = F \times 2^{C} , \tag{1}$$

where C is chosen such that

$$\frac{1}{2} \leq F < 1 . \tag{2}$$

Equation 2 insures the minimum possible percentage error. When condition 2 is satisfied, the quantity A is said to be expressed in normalized floating-point form.

Examination of the fractions in Table 9 quickly shows that it is difficult to determine whether or not condition 2 is satisfied for two's-complement fractions when the sign is negative. Furthermore, it is necessary to keep track of overflows, which are not errors in this case. The occurrence of an overflow during adding simply means that the resulting fraction, F, is greater than 1. To remedy this condition, F is shifted right one place, and 1 is added to the characteristic, C. It is not obvious, therefore, which number system should be used to express the fraction F.

In order to have sufficiently high precision that repeated calculations can be made without introducing unacceptable error, an entire 12-bit word will be used for the fractional part of a floating-point number. Putting the fraction and characteristic in separate words has the additional

advantage that no coding is needed to unpack them for the separate handling they require. Since there are 11 significant bits in the fraction, the maximum possible error is $\pm 2^{-12}$. The minimum value for a normalized fraction is 2^{-1} (condition 2), so the maximum relative error is

$$\text{R.E.} = \frac{\pm 2^{-12}}{2^{-1}} = \pm 2^{-11} . \tag{3}$$

In other words the precision is ±.05 percent, which should be sufficient for spaceborne applications.

The characteristic also contains a sign and 11 significant bits. It will be in two's-complement form because this is most efficient to program. The greatest number which can be expressed in the 2-word floating-point format is

$$\text{G.N.} = F_{max} \times 2^{C_{max}} \approx 1 \times 2^{+(2^{11}-1)} = 2^{2047} . \tag{4}$$

Likewise, the least number which can be expressed is

$$\text{L.N.} = F_{min} \times 2^{C_{min}} = 2^{-1} \times 2^{-(2^{11}-1)} = 2^{-2048} . \tag{5}$$

Using the relationship

$$\log_{10}(\chi) = \log_{10}(2) \cdot \log_2(\chi) \approx .3 \log_2(\chi) , \tag{6}$$

we see that

$$\text{G.N.} \approx 10^{614} \tag{7}$$

and

$$\text{L.N.} \approx 10^{-614} . \tag{8}$$

These numbers are unimaginably large and small respectively. Indeed the dynamic range is so large that tests for floating-point overflow and underflow need not be made because the possibility is so remote.

Figures 11, 12, and 13 are flow charts of the programming required to perform floating-point addition in the sign-and-magnitude, two's-complement, and one's-complement systems. It

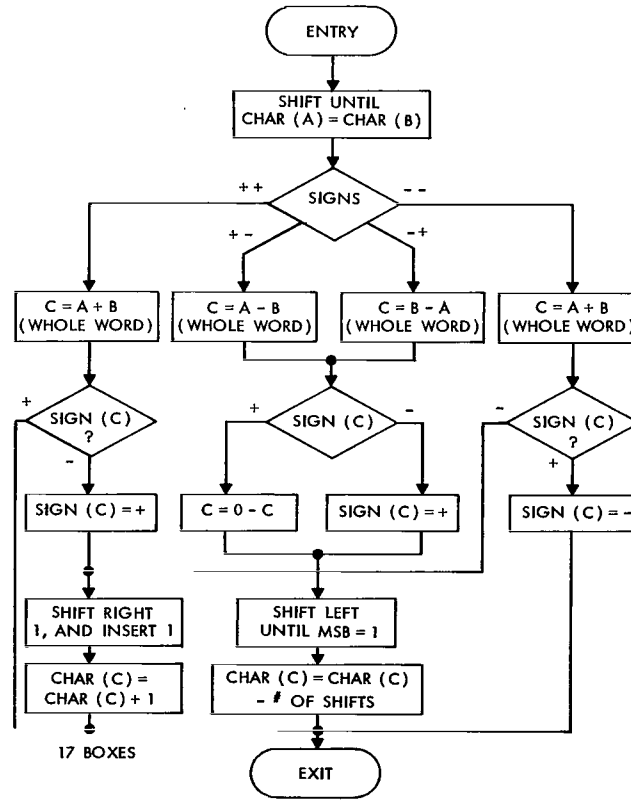

Figure 11—Floating-point add, sign-and-magnitude.

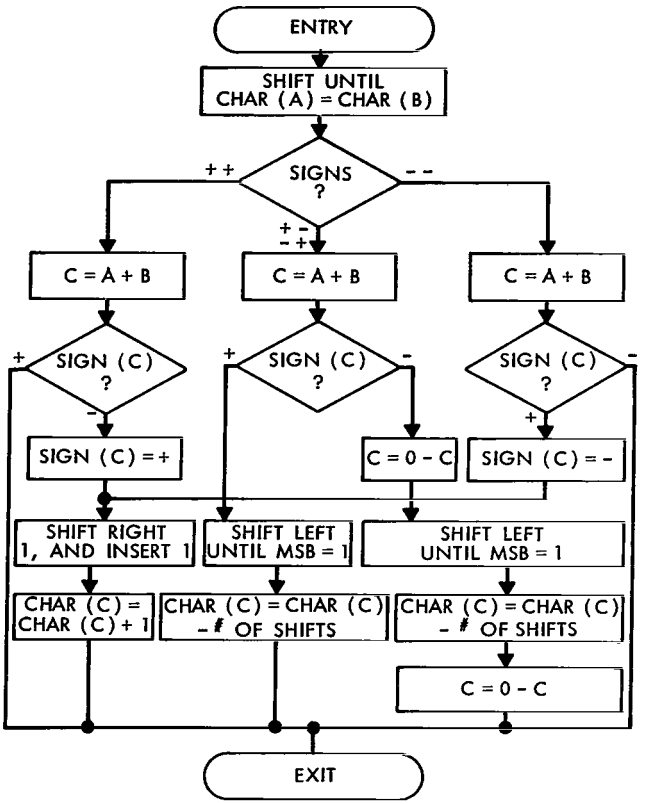

Figure 12—Floating-point add, two's-complement.

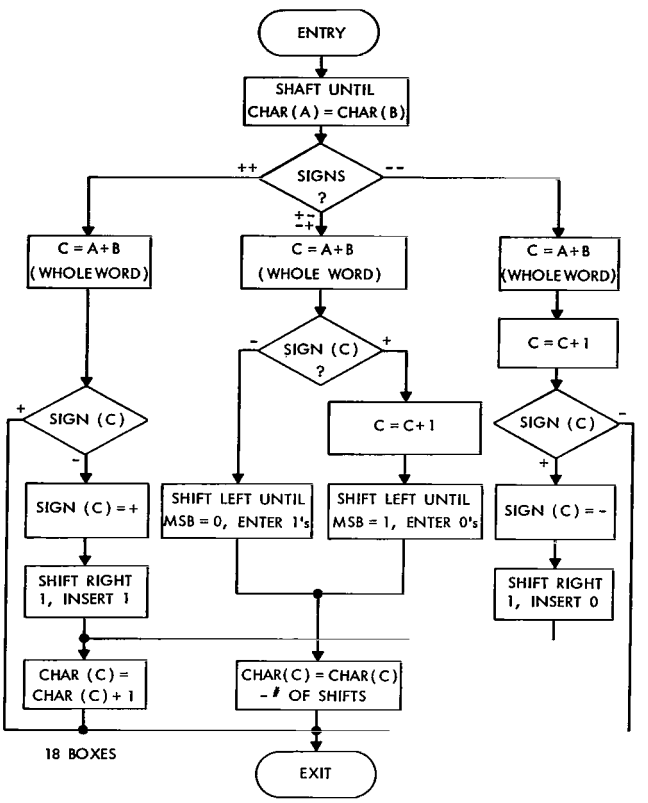

Figure 13—Floating-point add, one's-complement.

is evident that there is very little difference in the difficulty involved. Sign-and-magnitude is the most natural system to use because normalization is easiest; however, the addition of the two fractions is difficult. Using two's-complement notation, addition of the fractions is easy, but the result must be converted to sign-and-magnitude for normalization and extra coding is required to keep track of overflow. One's-complement requires extra coding both to keep track of overflows and to add 1 to the result when necessary. Since there is so little difference in the complexity of the programs required for these three systems, sign-and-magnitude will be used for the fraction.

Arithmetic Subroutines

A package of arithmetic subroutines (software) has been written to provide those operations which were not included in the hardware. Table 10 gives a summary of information about these subroutines.

Entry points are listed for two's-complement add and subtract subroutines which use the same argument cells that the other subroutines use. These subroutines are included in this discussion only as a basis against which the more involved subroutines may be compared. Because two's-complement add and subtract are implemented in hardware it is more efficient to use open subroutines for these functions.

Table 10

Arithmetic Subroutines.

Number System	Operation	Entry Point	Argument Cells A	Argument Cells B	Argument Cells C	Return Cell	Error Cond.	Error Result	Symbol Prefix
Two's Complement	ADD	TWCA	ARG1	ARG2	ARG2	TWCR	-	-	-
	SUB	TWCS	ARG1	ARG2	ARG2	TWCR	-	-	-
	MULT	TWCM	ARG1	ARG2	ARG2	TWCR	-	-	S
	DIV	TWCD	ARG1	ARG2	ARG2	TWCR	÷ by 0	divides by 1	T
Sign and Magnitude	ADD	FXPA	ARG1	ARG2	ARG2	FXPR	O'flow	modulo 2^{11}	Y
	SUB	FXPS	ARG1	ARG2	ARG2	FXPR	O'flow		Y
	MULT	FXPM	ARG1	ARG2	ARG1-ARG2	FXPR	-	-	U
	DIV	FXPD	ARG1-ARG2	ARG5	ARG1-ARG2	FXPR	O'flow	signs, ÷ by 1	V
Floating Point	ADD	FLPA	ARG1-ARG3	ARG2-ARG4	ARG2-ARG4	FLPR	-	-	Z
	SUB	FLPS	ARG1-ARG3	ARG2-ARG4	ARG2-ARG4	FLPR	-	-	Z
	MULT	FLPM	ARG1-ARG3	ARG2-ARG4	ARG2-ARG4	FLPR	-	-	W
	DIV	FLPD	ARG1-ARG3	ARG2-ARG4	ARG2-ARG4	FLPR	÷ by 0	divides by 1	X

Each of the operations has three arguments (A, B, and C). We shall adopt the following convention in regard to these arguments; C = A + B, C = A - B, or C = A/B, as appropriate. Several cells have been reserved in the data memory for arithmetic subroutine arguments. Table 10 shows how they have been assigned. For most of the fixed-point operations, ARG1 and ARG2 are used for the operands A and B and the result C is left in ARG2, ready to be used as an operand in further computation. Floating-point operations require two-word operands. ARG1-ARG3 and ARG2-ARG4 hold the fraction and characteristic of A and B respectively. The result C is left in ARG2-ARG4, ready for further computation.

The only unusual operations as far as argument assignment is concerned are sign-and-magnitude multiply and divide. Multiplication operates on two single-precision words to produce a double-precision product, and division acts on a double-precision dividend to produce a single precision quotient and a single precision remainder as shown in the table. The quotient is left in ARG2 and the remainder in ARG1 to minimize data shuffling preceding the next operation.

Three cells have been reserved in the data memory for arithmetic subroutine return addresses. They are TWCR for two's-complement subroutines, FXPR for sign-and-magnitude subroutines, and FLPR for floating-point subroutines. For a normal return, the arithmetic subroutines transfer control to the location specified by the appropriate return cell. Error conditions are also listed in Table 10 along with their affect on the arguments.

The last column of Table 10 lists symbol prefixes for the arithmetic subroutines. By giving each subroutine a unique prefix, they may all be assembled together without danger of multiply defined symbols.

The calling sequences for all of the arithmetic subroutines are similar. First, the arguments and the return address are stored in the proper cells of the data memory. Then control is transferred to the desired entry point. For example, to add two sign-and-magnitude numbers stored in NUM1 and NUM2 and place the result in ANSR, the following coding would be used:

```
        LOAD    NUM1
        X,X
        STO     ARG1        ARG1 = NUM1
        LOAD    NUM2
        X,X
        STO     ARG2        ARG2 = NUM2
        MIN                 GET RETURN ADDRESS
        TRA     RTRN
        STO     FXPR        STORE RETURN ADDRESS
        TRA     FXPA        TRANSFER TO ADD SUBROUTINES
*
        TRA     EROR        ERROR RETURN
RTRN    LOAD    ARG2        NORMAL RETURN
        X,X
        STO     ANSR        SAVE ANSWER
```

An instruction must be specified for the error return location. If overflows are to be ignored, the TRA EROR at the error return location can be replaced by NOP. The instruction must not be deleted, however, because if it were, an overflow would result in recalling the subroutine. A large number of recalls could be made before a normal return occurred. This would waste much time and the result would no longer be modulo 2^{11}. This particular arrangement for error returns has been chosen to minimize the number of machine cycles required for the normal returns.

Now that the arithmetic subroutine package has been introduced, the remaining subsections will describe and analyze the programs in detail.

Programming Techniques

Assembly Language

The computer will be programmed in assembly language. It could be programmed directly in machine code, but errors would be easy to make and hard to detect and correct. Compiler-type languages are not practical because the most efficient programs possible must be produced. As yet, no compiler can outperform the expert human programmer when tight coding is required. It is expected that an assembler for the language to be described will be written to run on a ground based computer. However, this is not necessary. The programs can be assembled by a human programmer if required. Even if the programs are assembled by hand, assembly language coding is preferable to direct machine language coding.

A detailed specification of the language is not necessary at this point. The only requirement is that reasonable notation be adopted in order that programs may be written and analyzed, but not necessarily assembled. For the present discussion, any feature of the language not specifically explained will be assumed to operate like 7090 FAP. The following fields will be used:

LOCATION OPERATION COUNT/ADDRESS/OCTAL COMMENT

The Location and Comment fields need no explanation. A blank Count/Address/Octal field will be assumed to contain 12 if a count is required and zero if an address or octal number is required.

The form of the Operation field varies depending on the type of instruction. Arithmetic and Test instructions have the form:

Operation Code Mnemonic, Test Code Mnemonic.

Shift and I/O instructions have the form:

SW1 Mnemonic, SW2 Mnemonic, Output Mnemonic.

A Pulse instruction consists of the operation code and a 2-character octal number which specifies the 6-bits which control the pulse output lines.

Instructions which modify the instruction register consist only of a mnemonic operation code. Transfer, Load, and Store consist of the mnemonic operation code and a symbolic or decimal address.

Arithmetic Subroutines

The arithmetic subroutine package was written to serve two purposes. The first is purely functional; to provide the operations it performs. The second was to gain experience programming the computer and insight into its capabilities. Therefore, each subroutine in the package will now be individually described, explained, and analyzed.

Sign-and-Magnitude Add and Subtract

The flow chart which has been developed for sign-and-magnitude addition and subtraction is shown in Figure 14. It is the most efficient of several methods which were experimented with. Subtraction is accomplished by an alternate entry point which changes the sign of the second argument before proceeding. One question which had to be answered was whether or not the sign should be separated from the magnitude before computation was done. As it turns out, operating on the whole word (sign included) is most efficient.

When both signs are plus (represented by zeroes), the entire words are simply added as shown in the left hand path of the flow chart (Figure 14). Since the sign bits add to zero, any overflow (a carry propagated past the most significant bit) will appear as a 1, or minus sign. Therefore,

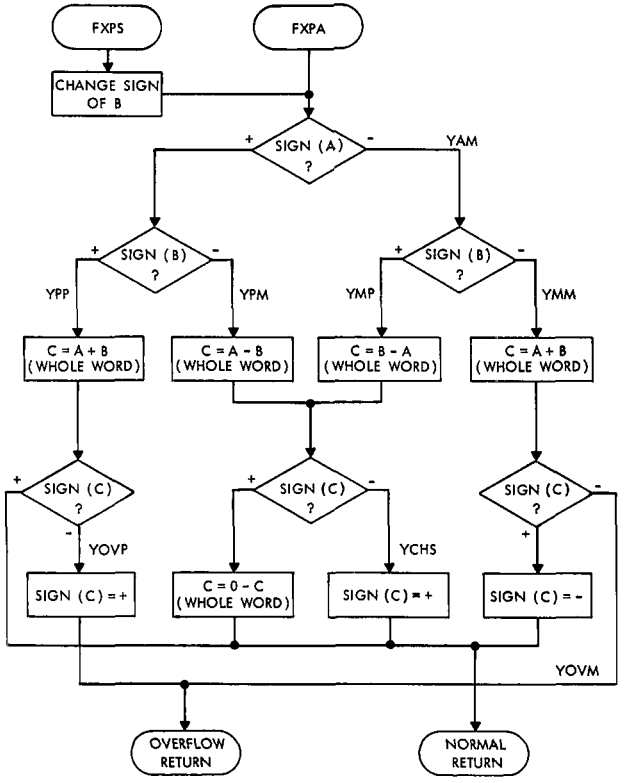

Figure 14—Sign-and-magnitude add and subtract.

```
                                                          CYCLES
FXPS    LOAD    ARG2    SUBTRACT ENTRY                      2
        X,X                                                13
        OFF,ROT 11                                         12
        COMP    1       CHANGE SIGN (B)                     2
        STO     ARG2                                        2
*
FXPA    LOAD    ARG1    ADD ENTRY                           2
        X,X                                                13
        ,B1     0       TEST SIGN (A)                       1
        TRA     YAM       -                                 1
        LOAD    ARG2      +                                 2
        X,X             B TO DR2, A TO DR1                 13
        ,B1     0       TEST SIGN (B)                       1
        TRA     YPM       -                                 1
YPP     ADD,B1            +, COMPUTE SUM                   13
        TRA     YOVP    OVERFLOW PLUS                       1
        TRA     YXIT    DONE                                1
*
YPM     X,X                                                13
YMP     SUB,B1          +- or -+                           13
        TRA     YCHS    -, MAKE +                           1
        X,X             +, MAKE -                          13
        ZERO                                               13
        SUB             C=0-C;                             13
        STO     ARG2    SAVE C                              2
        TRA     YXIT    DONE                                1
*
YAM     LOAD    ARG2                                        2
        X,X                                                13
        ,B1     0       TEST SIGN (B)                       1
        TRA     YMM       -                                 1
        TRA     YMP       +                                 1
*
YMM     ADD,B1          --, COMP. SUM                      13
        TRA     YOVM    OVERFLOW MINUS                      1
YCHS    OFF,ROT 11                                         12
        COMP    1       CHANGE SIGN                         2
        STO     ARG2    SAVE C                              2
        TRA     YXIT    DONE                                1
*
YOVP    OFF, ROT 11                                        12
        COMP    1       CHANGE SIGN                         2
YOVM    STO     ARG2    SAVE C                              2
        MIN             GET -1                             13
        OCT     7777    -1                                  1
        LOAD    FXPR                                        2
        ADD             FXPR=FXPR-1                        13
        MIX             OVERFLOW RETURN                    13
*
YXIT    LOAD    FXPR                                        2
        X,X                                                13
        MIX             NORMAL RETURN                      13
```

Figure 15—Program listing for sign-and-magnitude add and subtract.

if the sign of the result is plus, the answer is correct; and, if the sign if minus, there has been an overflow. If the answer is correct, the program executes a normal return; otherwise, the sign is set plus giving the result modulo 2^{11} and an overflow return is executed.

When both signs are minus (represented by ones), the entire words are also added as shown in the right hand path of the flow chart. Again the sign bits add to zero and any overflow will appear as a 1, or minus sign. If the sign of the result is plus, the sign is set minus (two negative numbers were added) and a normal return occurs. If the sign of the result is minus, an overflow occurred and an overflow return is executed.

On the other hand, if the signs of the two arguments are different, a subtraction is performed as shown in the two center paths of Figure 14. Again the entire words are used. The word with the negative sign is subtracted from the word with the positive sign. The magnitude of the result must be less than the greater of the argument magnitudes; therefore, the answer cannot overflow. The sign of the answer is found by testing the sign of the result of the subtraction. Since the subtract operation can be viewed as first taking the two's-complement and adding, we see that again the two sign bits will combine to produce a zero, or plus sign. Therefore, if the result of the subtraction has a minus sign, the equivalent two's-complement addition overflowed into the sign position. This means that the answer is positive. But, if the result of the subtraction has a plus sign, then the equivalent two's-complement addition did not overflow and the answer is negative, and in two's-complement form. In this case the answer is converted to sign-and-magnitude form by subtracting it from zero. This is equivalent to adding the two's-complement of the answer to zero.

The coding required to implement the flow chart (Figure 14) is shown in Figure 15. To aid the reader in correlating the flow chart and the program listing, many of the symbolic addresses have been included on the flow chart. The first part of the program tests the signs of the arguments. This part is straight-forward except for one thing. When the second argument is loaded from the memory, it enters Data Register 1. The subsequent exchange instruction leaves ARG1 in Data Register 1 and ARG2 in Data Register 2, therefore, it is not necessary to reload either argument after testing the signs.

Additional instructions are saved by testing the sign of the result of addition or subtraction with the add or subtract instruction itself. The computer was intentionally designed to make this possible. This can be observed in the add instruction at location YPP.

Further reduction in the number of instructions required is obtained by using the same coding to do either C = A - B or C = B - A. It has already been pointed out that at the end of the sign tests, both arguments are still in the data registers (A in Data Register 1 and B in Data Register 2). Thus, all is ready for subtracting A from B. To subtract B from A, it is only necessary to first exchange the contents of the two data registers with the single instruction at location YPM.

Another example will show the sort of savings which careful programming can produce. Consider the two boxes in the lower right-hand corner of the flow chart which set the sign of the answer. The most obvious way to code these is:

```
                                                        CYCLES
PL      X,X                                               13
        MIN             GET MASK                          13
        OCT     3777    + MASK                             1
        AND             SET SIGN +                        13
        TRA     GO                                         1
*
MI      X,X                                               13
        MIN             GET MASK                          13
        OCT     4000    - MASK                             1
        OR              SET SIGN -                        13
GO        .
          .
          .
```

This method requires 9 locations in the program memory. It also requires 4 or 5 memory accesses and 40 or 41 machine cycles per execution. A much better approach is

```
                                          CYCLES
PL (or MI)    OFF, ROT      11            12
              COMP           1             2
                .
                .
                .
```

This simplification is possible because careful scrutiny of the flow chart shows it is only necessary to change the sign, rather than setting it. The improved method uses 2 program memory locations instead of 9. Only two memory accesses are required instead of 4 or 5. It uses only 14 machine cycles instead of 40. The counting of machine cycles is discussed later in this section.

Overflow return is accomplished as follows:

```
OVFL    MIN             GET -1
        OCT     7777    -1
        LOAD    FXPR    GET RETURN ADDRESS
        ADD             SUBTRACT 1
        MIX             TRA TO C(FXPR) - 1.
```

Addition of -1 is used instead of subtraction of +1 to save instructions and machine cycles. The constant (+1 or -1) must be stored in the program memory; therefore, it enters Data Register 2. To subtract +1, it would have to be moved from Data Register 2 to Data Register 1, an unnecessary step. Normal return is very simple. The coding used is:

```
        LOAD    FXPR    GET RETURN ADDRESS
        X,X
        MIX             TRA C(FXPR)
```

Floating-Point Add and Subtract

Figure 16 is a flow chart of the floating-point add and subtract subroutine. Subtraction is accomplished by changing the sign of the second argument and then adding. To add two floating-point numbers, they must first be adjusted so that their characteristics are the same. For example:

$$\begin{array}{rl} & .10111 \times 2^5 \\ + & .10010 \times 2^3 \end{array} \Rightarrow \begin{array}{rl} & .10111 \times 2^5 \\ + & .00100 \times 2^5 \\ \hline & .11011 \times 2^5 \end{array}.$$

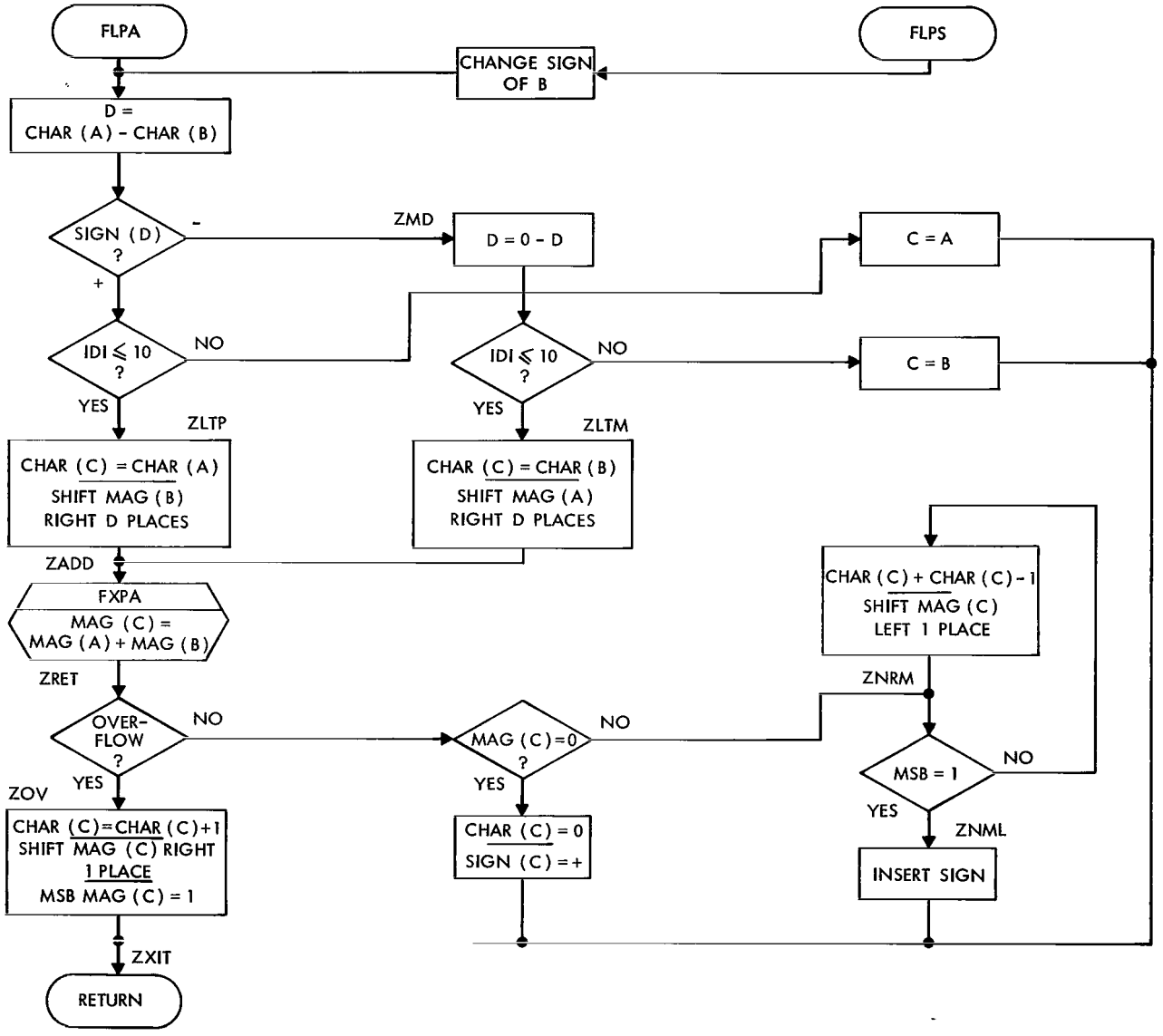

Figure 16—Floating-point add and subtract.

As shown, this adjustment consists of taking the difference (D) of the characteristics and shifting the fraction associated with the smaller characteristic right D places. If D is greater than N - 1, where N is the number of binary places in the fractions, the fraction being shifted is shifted out of existence and becomes all zeros.

After the characteristics and fractions have been properly adjusted, the fractions are added by the sign-and-magnitude subroutine. Three sorts of results are possible; overflow, a normal fraction, and a non-normal fraction. The worst possible overflow condition is:

$$\begin{array}{rr} & .11111 \\ + & .11111 \\ \hline & 1.11110 \end{array}$$

To remedy this, the result is shifted to the right one place and 1 is added to the characteristic. If, fortuitously, a normal fraction results from the addition, no further action is needed. When the argument fractions have different signs, it is possible to get a non-normal result ($.00101 \times 2^{-3}$ for example). Then it is necessary to shift the fraction to the left until the left most bit is a 1 and decrease the characteristic by the number of shifts. The example above would become $.10100 \times 2^{-5}$ by this process. If perchance the resulting fraction is zero, the characteristic should also be set to zero to produce the conventional "normal zero."

The coding for floating-point addition and subtraction appears in Figure 17. Several portions of this coding are of interest. Starting at FLPA the constant +10 is transferred from the program memory to the data memory. It happens that in this case it is more efficient to do this than to read the constant out of the program memory when it is needed. Then, 8 instructions after FLPA, advantage can be taken of one of the unique characteristics of the computer. Normally one would store the result of one computation before reading more data out of storage. The program originally read:

```
STO     ZD      SAVE D
LOAD    ZTEN    GET +10 CONSTANT.
```

Since the STO uses the contents of Data Register 2 and LOAD only affects Data Register 1, these two instructions may be interchanged:

```
LOAD    ZTEN    GET +10 CONSTANT
STO     ZD      SAVE D
```

Then the same location in the data memory can be used for both +10 and D, whereas before two locations were needed. Saving memory locations is important because the memory is so small.

The floating-point add and subtract subroutine requires the fractional magnitude to be shifted D places (where D is a variable). Therefore, we find in this program an interesting example of instruction modification. Starting 5 instructions past location ZLTM the following sequence occurs:

```
                                                    CYCLES
FLPS    LOAD    ARG2    SUBTRACT ENTRY      2
        X,X             .                   13
        ,ROT    11      .                   12
        COMP    1       CHANGE SIGN         2
        STO     ARG2    .                   2
*
FLPA    MIN             ADD ENTRY           13
        OCT     0012    + 10 CONSTANT       1
        STO     ZTEN    .                   2
*
        LOAD    ARG3    CHAR (A)            2
        X,X             .                   13
        LOAD    ARG4    CHAR (B)            2
        SUB, B1         D = CHAR (A) -      13
                          CHAR (B)
        TRA     ZMD       D IS -            1
        LOAD    ZTEN      D IS +            2
        STO     ZD      SAVE D              2
        SUB, B1         IS D LESS THAN 10   13
        TRA     ZLTP        YES             1
        LOAD    ARG3        NO, C = A       2
        X,X             .                   13
        STO     ARG4    .                   2
        LOAD    ARG1    .                   2
        X,X             .                   13
*       STO     ARG2    .                   2

ZXIT    LOAD    FLPR    .                   2
        X,X             .                   13
        MIX             RETURN              13
```

```
                                                    CYCLES
*
        MIN             GET X, ROT 0        13
        X, ROT  0       .                   1
        LOAD    ZD      .                   2
        OR              FORM X, ROT D       13
*
        LOAD    ZT      GET MAG (B)         2
        MIX             EXECUTE X, ROT D    13+D
        X,X     11      MAG (B) TO DR2      12
        LOAD    ARG2    GET SIGN (B)        2
        X,X     1       INSERT SIGN         2
        STO     ARG2    SAVE SHIFTED B      2

*
ZADD    MIN             CALL FXPA           13
        TRA     ZRET      .                 1
        STO     FXPR      .                 2
        TRA     FXPA      .                 1
*
        TRA     ZOV     OVERFLOW RETURN     1
ZRET    LOAD    ARG2    GET C               2
        ZERO    1       .                   2
        ROT, X  11      STRIP OFF SIGN      12
        ,Z      0       IS MAG (C) = 0      1
        TRA     ZNRM      NO                1
        STO     ARG2      YES               2
        STO     ARG4    C = 0               2
        TRA     ZXIT    .                   1
*
ZOV     LOAD    ARG2    OVERFLOW RETURN     2
        X,X     11      RENORMALIZE         12
        ZERO    1       .                   2
        X,X     13      .                   14
        MIN             .                   13
        OCT     2000    .                   1
        OR              .                   13
        STO     ARG2    .                   2
```

```
*
ZMD     X,X             MOVE D              13
        ZERO            .                   13
        SUB             D = 0 - D           13
        LOAD    ZTEN    GET + 10            2
        STO     ZD      SAVE D              2
        SUB,B1          IS D LESS THAN 10   13
        TRA     ZLTM      YES               1
        TRA     ZXIT      NO, C = B         1
*
ZLTM    LOAD    ARG1    SIGN (A) + MAG (A)  2
        X,X     11      STRIP SIGN (A)
                          FROM MAG (A)      12
        STO     ZT      SAVE MAG (A)        2
        X,X             .                   13
        STO     ARG1    SAVE SIGN (A)       2
*
        MIN             GET X, ROT 0        13
        X, ROT  0       .                   1
        LOAD    ZD      .                   2
        OR              FORM X, ROT D       13
*
        LOAD    ZT      GET MAG (A)         2
        MIX             EXECUTE X, ROT D    13 + D
        X,X     11      MAG (A) TO DR2      12
        LOAD    ARG1    GET SIGN (A)        2
        X,X     1       INSERT SIGN         2
        STO     ARG1    SAVE SHIFTED A      2
        TRA     ZADD    .                   1
*
ZLTP    LOAD    ARG3    CHAR (A)            2
        X,X             .                   13
        STO     ARG4    CHAR (C) = CHAR (A) 2
        LOAD    ARG2    SIGN (B) + MAG (B)  2
        X,X     11      STRIP SIGN (B)
                          FROM MAG (B)      12
        STO     ZT      SAVE MAG (B)        2
        X,X             .                   13
        STO     ARG2    SAVE SIGN (B)       2
                                            CYCLES
*
        MIN             .                   13
        OCT     0001    .                   1
        LOAD    ARG4    .                   2
        ADD             CHAR (C) =
                          CHAR (C) + 1      13
        STO     ARG4    .                   2
        TRA     ZXIT    .                   1
*
ZNRM    ,B1     0       IS IT NORMAL        1
        TRA     ZNML      YES               1
        STO     ZT        NO                2
        LOAD    ARG4    GET CHAR (C)        2
        MIN             GET - 1             13
        OCT     7777    - 1                 1
        ADD             CHAR (C) =
                          CHAR (C) - 1      13
        STO     ARG4    SAVE CHAR (C)       2
        LOAD    ZT      GET MAG (C)         2
        ZERO    1       .                   2
        ROT, X  11      SHIFT LEFT 1        12
        TRA     ZNRM    LOOP                1
*
ZNML    LOAD    ARG2    GET SIGN (C)        2
        ROT, OFF 11     .                   12
        ROT, X  1       INSERT SIGN         2
        STO     ARG2    SAVE C              2
        TRA     ZXIT    DONE                1
*
ZD      EQU     TS (1
ZTEN    EQU     TS (2
ZT      EQU     ARG3
```

Figure 17—Program listing for floating-point add and subtract.

```
MIN             GET X,ROT O INST.
X,ROT   O
LOAD    ZD      GET D
OR              FORM X,ROT D
LOAD    ZT      GET FRACTION
MIX             SHIFT RIGHT D PLACES.
```

The first 4 instructions are used to produce the instruction X,ROT D in Data Register 2. Then the fraction is loaded into Data Register 1. The MIX instruction causes the contents of Data Register 2 to be shifted into the instruction register and executed. Data Register 2 is filled with zeros during the shifting. Therefore execution of the contents of Data Register 2 (X,ROT D) has the desired effect of shifting the fraction to the right D places and inserting zeros in the vacated bit positions.

The count field is interpreted modulo 16; therefore, if D were to exceed 16, an incorrect number of shifts would occur. For that reason the number of shifts (D) is tested before instruction modification. If it is greater than 10, one of the arguments is zero (to the precision of the machine). The flow chart shows that in this case the other argument is taken for the answer.

After returning from the sign-and-magnitude subroutine without an overflow, the magnitude of the result is tested to see if it is zero. Not only is this required to produce a normal zero, but the subsequent normalizing loop would "hang up" if the magnitude were zero. This would occur because a 1 would never appear in the most significant bit of the fraction, no matter how many left shifts were made.

Sign-and-Magnitude Multiply

The sign-and-magnitude multiplication subroutine is used by both the floating-point and the two's-complement subroutines. Logically this program is quite simple since it operates in the conventional shift-and-add manner. It takes two arguments, each with a sign and an 11-bit magnitude, and produces a double precision product with a sign and a 22-bit magnitude. The sign and the most significant eleven bits are left in ARG1. The sign and the least significant 11 bits are left in ARG2. When integers or mixed numbers are multiplied, the programmer must take care to keep track of where, in these two words, the significant bits and the binary point will occur. Several examples using 4-bit words will show the problems. Table 11 contains 6 examples of multiplication. In example 1, two integers are multiplied and the entire result is in the least significant portion of the two-word answer. Again, in example 2, two integers are multiplied, but this time both of the answer words contain significant bits. In example 3 a mixed number has been used. Both answer words contain significant bits and the binary point location is different from the previous examples.

The easiest cases occur when the two arguments are normal fractions. Then ARG1 will always contain the significant part of the result and the answer will be a normal fraction, or will require at most one left shift to make it normal (examples 4, 5, and 6). This is discussed further in the section on floating-point multiplication.

Table 11

Multiplication Examples.

Example	Multiplicand (ARG1)		Multiplier (ARG2)		Product (ARG1/ARG2)		
1	+3		-2		-0×2^3		-6
	0011	X	1010	=	1000	+	1110
2	+6		+3		$+2 \times 2^3$		+2
	0110	X	0011	=	0010	+	0010
3	+3-1/2		+4		$+3 \times 2^2$		+2
	011.1	X	0100	=	0011	+	010.0
4	+1/2		+1/2		+1/4		0×2^{-6}
	0.100	X	0.100	=	0.010	+	0000
5	+1/2		+3/4		+3/8		0×2^{-6}
	0.100	X	0.110	=	0.011	+	0000
6	+7/8		+7/8		+3/4		1×2^{-6}
	0.111	X	0.111	=	0.110	+	0001

The flow chart for this program appears in Figure 18. Figure 19 shows how the multiplier, multiplicand, sign, and partial product are handled during multiplication. When the sign of the product is computed, it appears in the sign position of the multiplier. As the program was originally coded, this new sign was stripped off and stored temporarily until the multiplication was complete. It happens, however, that the sign can be left with the multiplier throughout the multiplication operation. This saves several instruction locations and one word in the data memory.

In step 1 of Figure 19 we see the state of the computer at the beginning of the multiplication process. This corresponds to location UGO in Figure 20. The multiplicand is stored in ARG1 preceded by a plus sign, Data Register 2 contains zero, and Data Register 1 contains the sign of the product and the multiplier. Then, in step 2, the two data registers are rotated to the right as shown. The least significant bit of the multiplier enters Data Register 2, where it may be tested. If this bit (denoted by M in the figure) is a 0, step 2 is repeated. Otherwise the sign and multiplier are temporarily stored, M is set to 0, and the multiplicand is brought into Data Register 1 and added to the contents of Data Register 2 to form the partial product (step 3). Then step 2 is repeated, shifting one bit of the partial product into Data Register 1. After all eleven bits of the multiplier have been tested, the data registers contain the product and sign as shown in step 4.

The following instructions adjust the words so that the signs are at the proper location in each and then store the results:

			CYCLES
X,X	1	SIGN TO DR2	2
STO	ARG1	SAVE MS PRODUCT	2

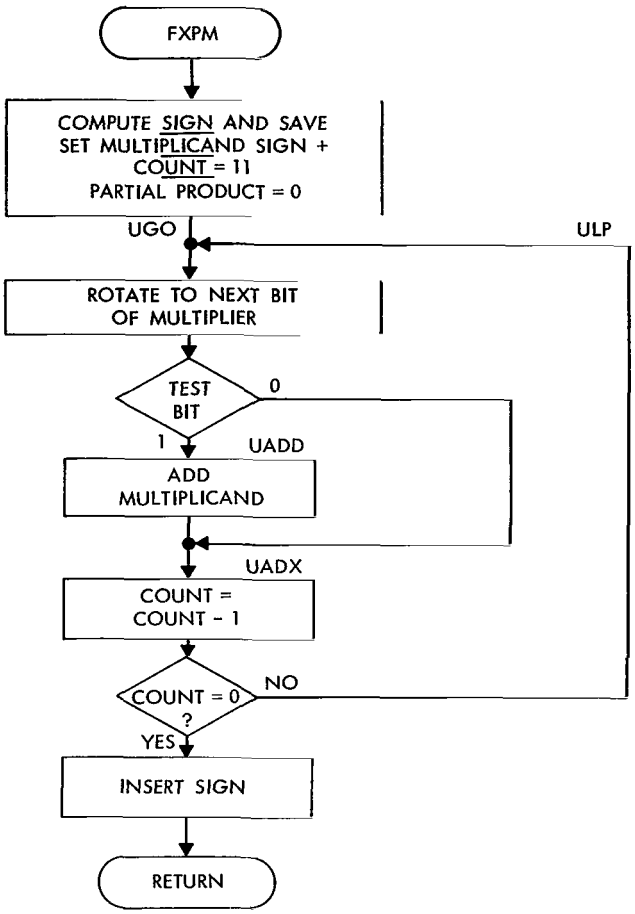

Figure 18—Sign-and-magnitude multiply.

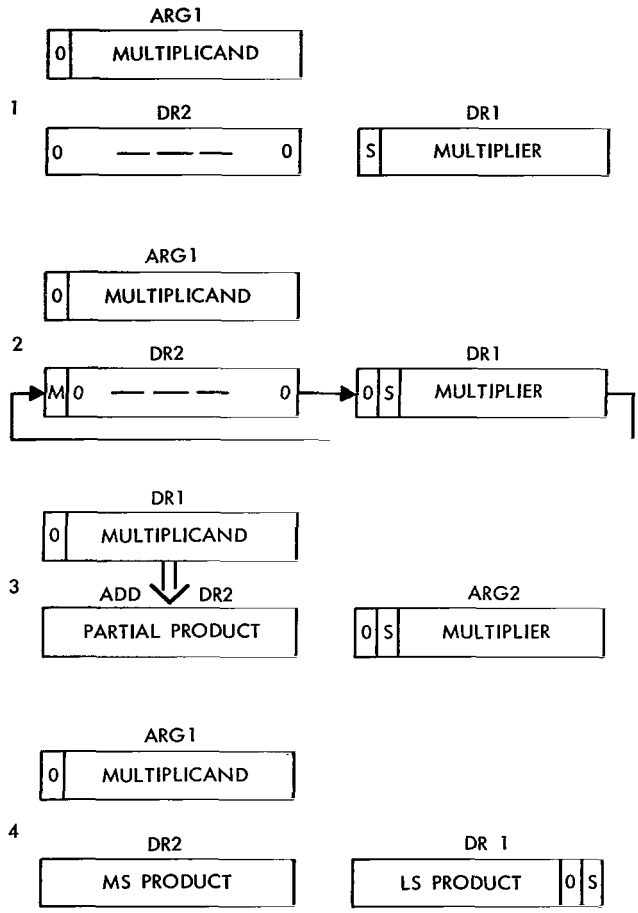

Figure 19—Multiplication.

```
                                                        CYCLES
FXPM    LOAD    ARG2    GET B (MULTIPLIER)              2
        X,X             (MULTIPLICAND)                  13
        LOAD    ARG1    GET A                           2
        ROT,ROT 11      PASS ALL BUT SIGN               12
        EOR     1       COMPUTE PRODUCT SIGN            2
        STO     ARG2    SAVE B AND NEW SIGN             2
        X,X     11                                      12
        ZERO    1       SET A +                         2
        STO     ARG1    SAVE A                          2
*
        MIN                                             13
        OCT     7776                                    1
        STO     UCNT    COUNT = 11                      2
        ZERO            MS PARTIAL PROD = 0             13
        TRA     UGO                                     1
*
ULP     STO     UCNT                                    2
        LOAD    UPP     GET MS PARTIAL PROD             2
        X,X                                             13
UGO     LOAD    ARG2    GET LS PARTIAL PROD             2
        X,X     1       LONG RIGHT ROTATE 1             2
        ,B1     0       TEST MULTIPLIER BIT             1
        TRA     UADD      1, ADD                        1
        STO     UPP       0, DON'T ADD                  2
        X,X                                             13
        STO     ARG2    SAVE PARTIAL PRODUCTS           2
*
UADX    LOAD    UCNT                                    2
        ZERO    1       COUNT = COUNT - 1               2
        X,X     11                                      12
        ,Z      0       TEST COUNT                      1
        TRA     ULP     NOT DONE                        1
*
        LOAD    UPP     GET MS PROD                     2
        X,X                                             13
        LOAD    ARG2    GET LS PROD                     2
        X,X     1       SIGN TO MS                      2
        STO     ARG1    SAVE MS PROD                    2
        ROT,OFF 1                                       2
        X,X     11      LS PROD TO SIGN                 12
        OFF,ROT 1       SIGN TO BIT 1                   2
        STO     ARG2    SAVE LS PROD                    2
*
        LOAD    FXPR                                    2
        X,X                                             13
        MIX             RETURN                          13
*
UADD    X,X                                             13
        STO     ARG2    SAVE LS PARTIAL PROD            2
        X,X     11                                      12
        ZERO    1       BIT 1 MS PP = 0                 2
        LOAD    ARG1    MULTIPLICAND                    2
        ADD                                             13
        STO     UPP     SAVE MS PARTIAL PROD            2
        TRA     UADX                                    1
*
UPP     EQU     TS(1
UCNT    EQU     TS(2
```

Figure 20—Program listing for sign-and-magnitude multiply.

			CYCLES
ROT,OFF	1	SHIFT OUT 0	2
X,X	11	11 BITS FOLLOW SIGN	12
OFF,ROT	1	SIGN TO BIT 1	2
STO	ARG2	SAVE LS PRODUCT	2
			22

Here again is an example of how careful coding can save execution time. This sequence of instructions was originally coded:

			CYCLES
X,X	1	SIGN TO DR2	2
OFF,ROT	11	SIGN TO BIT 12	12
X,ROT	1	SIGN TO LS PRODUCT	2
STO	ARG1	SAVE MS PRODUCT	2
X,X			13
STO	ARG2	SAVE LS PRODUCT	2
			33

Each of these sequences contain 6 instructions and each performs the same operation; however, the first sequence requires only 22 machine cycles while the second requires 33.

Another important improvement was made in the coding with a large resulting saving in execution time. The coding in question begins at location UADX (Figure 20). It performs the operations COUNT = COUNT - 1 and tests COUNT for zero. The obvious method of accomplishing this function requires 30 machine cycles:

				CYCLES
UADX	MIN			13
	OCT	7777	-1	1
	LOAD	UNCT	GET COUNT	2
	ADD,Z		COUNT = COUNT - 1	13
	TRA	ULP		1
				30

Since the maximum count is only 11, the same function can be performed by placing 11 ones in a word and repeatedly shifting zeros in until the entire word becomes zero. The coding shown below does the trick and requires only 18 machine cycles, a saving of 12 cycles:

				CYCLES
UADX	LOAD	UNCT	GET COUNT	2
	ZERO	1	INSERT 0	2
	X,X	11	SHIFT REST	12
	,Z	0	TEST	1
	TRA	ULP		1
				18

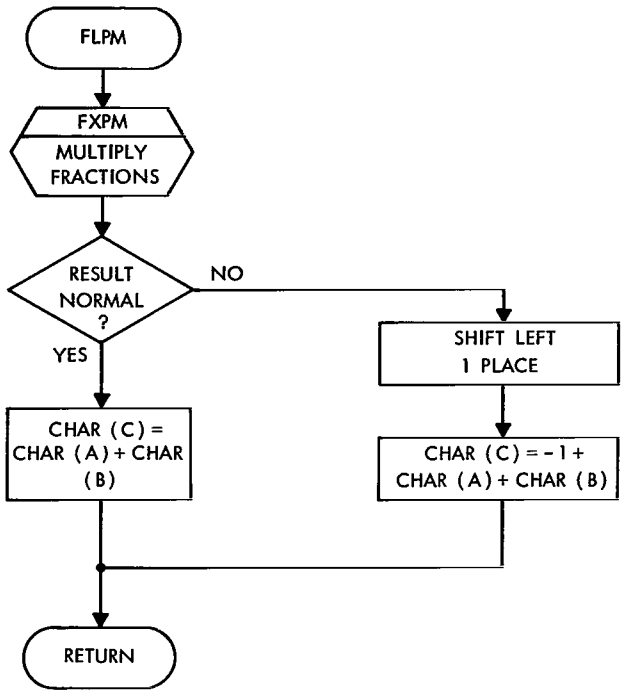

Figure 21—Floating-point multiply.

```
                                                          CYCLES
FLPM    MIN                    CALL FXPM                    13
        TRA     WRET           .                             1
        STO     FXPR           .                             2
        TRA     FXPR           .                             1
*
WRET    LOAD    ARG2           LS PRODUCT FRACTION           2
        X,X     11                                          12
        LOAD    ARG1           MS PRODUCT FRACTION           2
        ROT,X   11             12 SIG BITS IN DR2           12
        ,B1     0              TEST NORMALITY                1
        TRA     WNML               NORMAL                    1
        OFF,ROT 11                 NON-NORMAL               12
        X,X     1              NORMALIZE, INSERT SIGN        2
        STO     ARG2           SAVE PRODUCT FRACTION         2
        MIN                    GET -1                       13
        OCT     7777               -1                        1
        TRA     WCH                                          1
*
WNML    X,X     1              INSERT SIGN                   2
        STO     ARG2           SAVE PRODUCT FRACTION         2
        ZERO                                                13
WCH     LOAD    ARG3           CHAR (A)                      2
        ADD                                                 13
        LOAD    ARG4           CHAR (B)                      2
        ADD                    CHAR (C) = CHAR (A) + CHAR (B) 13
        STO     ARG4           SAVE CHAR (C)                 2
*
        LOAD    FLPR                                         2
        X,X                                                 13
        MIX                    RETURN                       13
```

Figure 22—Program listing for floating-point multiply.

This function is performed 11 times for each multiplication; therefore, a total of $11 \times 12 = 132$ machine cycles per multiplication are saved by using the new coding. Also, notice that a count of 0 has been specified for the TEST instruction Z. Any count could be specified, but at the expense of more machine cycles. If no count were specified it would have been assumed to be 12, and 13 machine cycles would be used instead of 1.

Floating-Point Multiply

To obtain the product of two floating-point numbers the characteristics are added and the fractions multiplied. The logic is shown in Figure 21. Adding the characteristics is simple because they are in two's-complement form. Multiplying the fractions is done by the sign-and-magnitude multiplication subroutine. The only problem which arises is the possibility of a non-normal fraction in the product. It was shown in Table 11 (Example 5) that if the resulting fraction is non-normal, one left shift is sufficient to restore normality. The characteristic of the product is then adjusted by subtracting one. Figure 22 contains the coding for floating-point multiplication. It will not be discussed because it is straight-forward and there are no tricks involved.

Two's-Complement Multiply

The easiest way to multiply two two's-complement numbers is to first convert them to sign-and-magnitude form, and then use the sign-and-magnitude multiplication subroutine. The product is subsequently converted back into two's-complement form. Since two's-complement notation will normally be used for fairly small integers, it is assumed that only the least significant half of the double-precision product contains significant data. No test is made to see if this condition is violated, just as no test

is made to detect overflow in two's-complement addition. The flow chart for this subroutine may be found in Figure 23 and the coding in Figure 24.

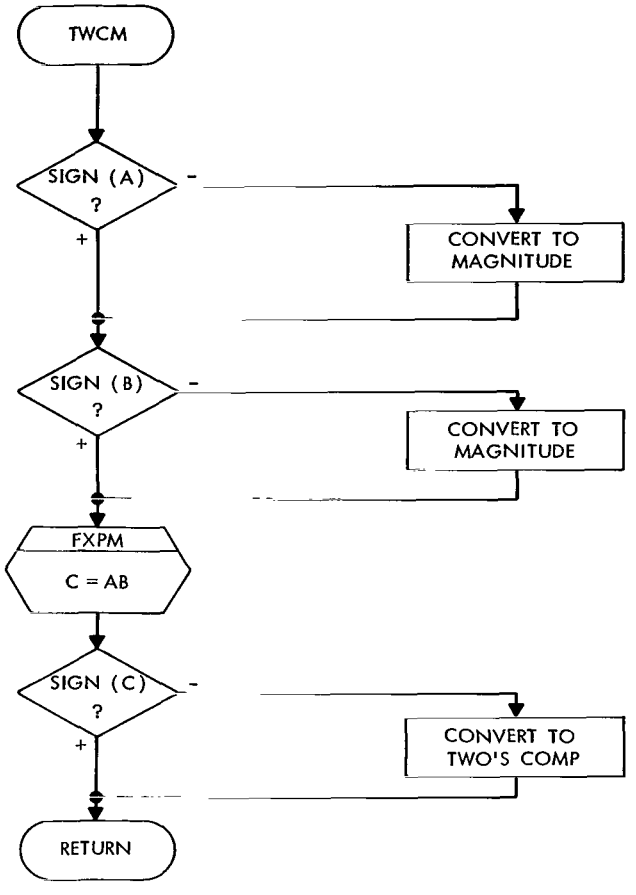

Figure 23 — Two's-complement multiply.

```
                                                              CYCLES
TWCM    LOAD    ARG1                                            2
        X,X                                                     13
        ,B1     0       TEST SIGN (A)                           1
        TRA     SAM     -                                       1
SAG     LOAD    ARG2    +                                       2
        X,X                                                     13
        ,B1     0       TEST SIGN (B)                           1
        TRA     SBM     -                                       1
SBG     MIN             +                                       13
        TRA     SRET                                            1
        STO     FXPR                                            2
        TRA     FXPM    CALL FXPM                               1
*
STRET   LOAD    ARG2                                            2
        X,X                                                     13
        ,B1     0       TEST SIGN (C)                           1
        TRA     SCM     -                                       1
SCG     LOAD    TWCR    +                                       2
        X,X                                                     13
        MIX             RETURN                                  13
*
SAM     X,X                                                     13
        ZERO                                                    13
        SUB     11      COMPLEMENT MAG (A)                      12
        X,X     1       INSERT SIGN                             2
        STO     ARG1    SAVE A                                  2
        TRA     SAG                                             1
*
SBM     X,X                                                     13
        ZERO                                                    13
        SUB     11      COMPLEMENT MAG (B)                      12
        X,X     1       INSERT SIGN                             2
        STO     ARG2    SAVE B                                  2
        TRA     SBG                                             1
*
SCM     X,X                                                     13
        ZERO                                                    13
        SUB     11      COMPLEMENT MAG (C)                      12
        X,X     1       INSERT SIGN                             2
        STO     AGR2    SAVE C                                  2
        TRA     SCG                                             1
```

Figure 24—Program listing for two's-complement multiply.

Sign-and-Magnitude Divide

This subroutine does binary division in a manner analogous to decimal long division. A single-precision divisor is divided into a double-precision dividend, which could very well be the double-precision result of a previous multiplication. The division subroutine produces a single-precision quotient and a single-precision remainder as answers. In basic principle, this program is like other binary division algorithms; however, considerable ingenuity must be applied to obtain an efficient program for this computer.

Figure 25 is the flow chart of the sign-and-magnitude division program. The coding is given in Figure 26. The program starts by first trying the highest possible power of 2 which could appear in the quotient, just as the highest power of 10 is tried first in decimal long division. Then successively lower powers are tried until the entire quotient has been developed. In decimal

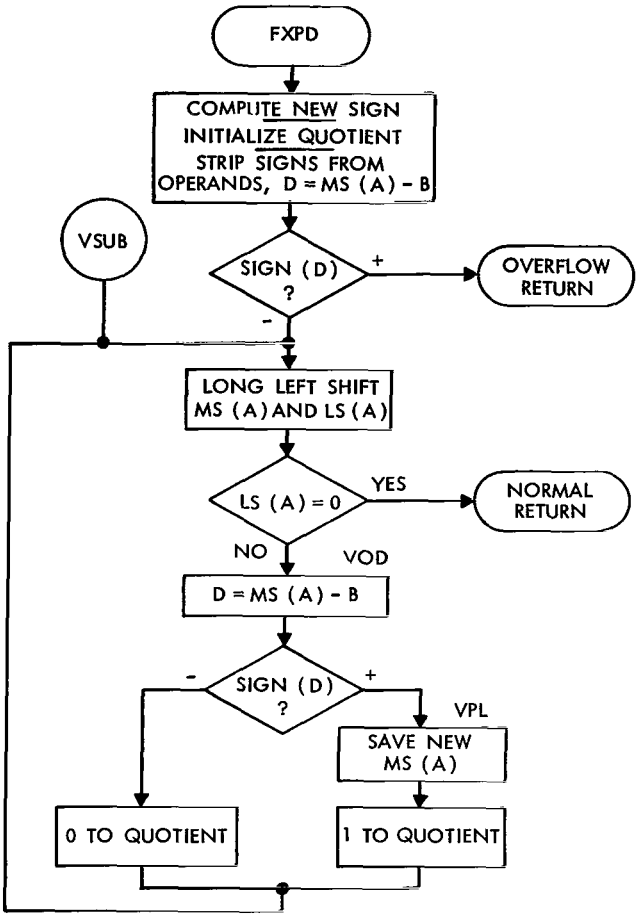

Figure 25—Sign-and-magnitude divide.

division, one must ascertain whether the multiplier of a particular power of 10 (a particular digit of the quotient) is 0, 1, 2, . . . , 8 or 9. In binary division the problem is much simpler because the only possibilities are 0 and 1. In decimal division, the divisor is multiplied by the quotient digit and power of 10 being considered, and then the resulting product is subtracted from the dividend. Binary division is easier because the only possible non-zero quotient bit is 1. Therefore, the divisor is simply multiplied by the power of 2 in question and subtracted from the dividend. If the subtraction produces a positive remainder, then a 1 is inserted in the bit of the quotient which corresponds to the power of 2 in question. Otherwise

```
                                                                  CYCLES
FXPD    LOAD    ARG1        GET MS (A)                              2
        X,X                                                        13
        LOAD    ARG2        GET LS (A)                              2
        OR                  OR SIGNS                               13
        LOAD    ARG5        GET B                                   2
        EOR                 COMPUTE NEW SIGN                       13
        ZERO    11          CLEAR ALL BUT SIGN                     12
        OFF, ROT 11         INITIALIZE AND                         12
        STO     VQUO          SAVE IN QUOTIENT                      2
        X,X     11                                                 12
        ZERO    1           SIGN (B) = +                            2
        STO     ARG5        SAVE B                                  2
*
        LOAD    ARG2        GET LS (A)                              2
        MIN                                                        13
        OCT     4000        FLAG                                    1
        OR                  FLAG LS (A),                           13
        OFF, ROT 11         INITIALIZE,                            12
        STO     ARG2        SAVE                                    2
*
        LOAD    ARG1        GET MS (A)                              2
        X,X     11                                                 12
        ZERO    1           SIGN = +                                2
        STO     ARG1        SAVE MS (A)                             2
        LOAD    ARG5        GET B                                   2
*
VSUB    SUB, B1             D = MS (A) - B                         13
        TRA     VLOP          D - , OK                              1
        MIN                   D + , OVFL                           13
        OCT     7777        -1                                      1
        LOAD    FXPR                                                2
        ADD                 FXPR = FXPR - 1                        13
        MIX                 OVERFLOW RETURN                        13
*
VLOP    LOAD    ARG2        GET LS (A)                              2
        X,X                                                        13
        LOAD    ARG1        GET MS (A)                              2
        ROT, ROT 11         LONG LEFT ROTATE 1                     12
        X,X     1           . . . . . .                             2
        ROT, ROT 11         . . . . . .                            12
        ,Z      0           IS LS (A) = 0                           1
        TRA     VOD           NO                                    1
        X,X     1             YES                                   2
        STO     ARG1        SAVE REMAINDER                          2
        LOAD    VQUO                                                2
        X,X                                                        13
        STO     ARG2        SAVE QUOTIENT                           2
*
        LOAD    FXPR                                                2
        X,X                                                        13
        MIX                 NORMAL RETURN                          13
*
VOD     STO     ARG2        SAVE LS (A)                             2
        X,X                                                        13
        STO     ARG1        SAVE MS (A)                             2
        LOAD    ARG5        GET B                                   2
        SUB, C              D = MS (A) - B                         13
        TRA     VPL           D +                                   1
        LOAD    VQUO          D -                                   2
        ZERO    1           PUT A 0 IN QUOT.                        2
        X,X     11          . . . . . .                            12
        STO     VQUO        . . . . . .                             2
        TRA     VLOP        LOOP                                    1
*
VPL     STO     AGR 1       SAVE NEW MS (A)                         2
        LOAD    VQUO        PUT A 1 IN QUOT.                        2
        X,X                 . . . . . .                            13
        COMP    1           . . . . . .                             2
        OFF, ROT 10         . . . . . .                            11
        STO     VQUO        . . . . . .                             2
        TRA     VLOP        LOOP                                    1
*
VQUO    EQU     TS (1)
```

Figure 26—Program listing for sign-and-magnitude divide.

a 0 is inserted in the quotient, the dividend is restored, and the next lower power of 2 is tried. In this way the quotient is built up, one bit at a time, starting with the most significant bit.

There is a problem which can occur. If the divisor is too small, or the dividend too large, the quotient will overflow. Therefore, the first subtraction which is made is programmed to correspond to the 11th power of 2. The most significant bit of the quotient corresponds to 2^{10}. If the result of the first subtraction is positive, an overflow condition obtains. An overflow return is provided to indicate that this eventuality has occurred.

Once it has been determined that overflow will not occur, the double-precision dividend is shifted to the left 1 place. This decreases by one the effective power of two by which the divisor is multiplied. The computer instruction set contains only right shift instructions. Therefore the coding of a long shift is somewhat more involved than would be desirable. A long left rotate will suffice, however. It is obtained by using the following three instructions:

```
ROT,ROT     11
X,X          1
ROT,ROT     11.
```

Figure 27 shows how these instructions produce a 1-bit long left rotation. In step 1 we see the desired operation and, in step 2, the initial contents of the data registers. The two bits which change registers during the operation are denoted A and B. The first instruction (ROT,ROT, 11) causes bits A and B to be positioned at the right-hand ends of their respective registers as shown in step 3. Then X,X 1 exchanges these bits between the registers (step 4). Finally the instruction ROT,ROT 11 repositions the bits A and B at the right-hand ends of their new registers and the operation is complete (step 5).

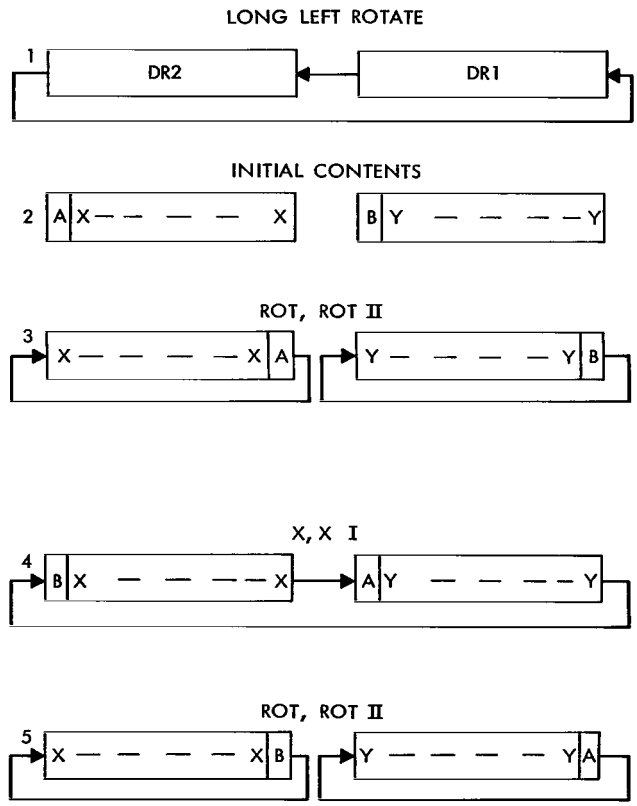

Figure 27—Long left rotate.

Returning again to Figure 25, we see that after the long left shift, the least significant part of the dividend is tested. If it is zero, the division is finished. It is not obvious that this condition should indicate that the division process is over; however, once it has been demonstrated that this is indeed a valid test, one may take advantage of it. The loop in the division program, like that in the multiplication program, is traversed 11 times. In the multiplication program 18 machine cycles per loop were

devoted to keeping track of when to stop. Five instructions were required for this purpose. The same technique could be used here, but testing the dividend requires only 2 machine cycles and 2 instructions. A saving of 176 machine cycles and 3 program memory locations is realized.

The validity of the dividend test depends on two conditions. First, that the least significant portion of the dividend must be non-zero until after the 12th long-left rotation. Second, that the least significant portion of the dividend must be zero after the 12th long-left rotation. The first condition is easy to satisfy. The least significant portion of the dividend consists originally of a sign and 11 data bits. To fulfill the first condition, the sign is stripped off, the 11 data bits are shifted to the left one place and the 12th bit (at the right-hand end of the word) is made a 1. This 1 will remain in the word until the 12th shift has taken place.

Condition 2 will be satisfied if each long-left rotation causes a zero to enter the right-hand end of the least significant part of the quotient. Figure 28 (step 1) shows the divisor and dividend just before division starts. Before the division loop is entered the most significant part of the dividend is tested to see that it is smaller than the divisor. If not, the division stops and an overflow return is made as previously explained. Let us denote the divisor by D and the most significant part of the dividend by M. We have

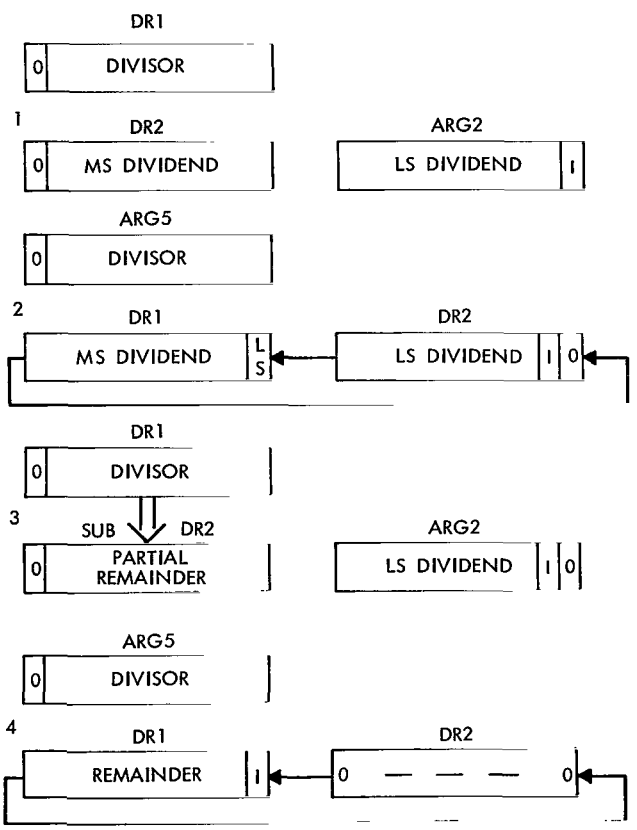

Figure 28—Division.

$$D > M , \tag{9}$$

and

$$D < 2^{11} . \tag{10}$$

The next step in the division process is a 1-bit long-left rotation; shown in step 2. The effect is to multiply M by 2 and possibly to add 1 (if a 1 was shifted in from the least significant part of the dividend). The largest possible contents of Data Register 1 at this step are $2M + 1$. Because M and D are integers (we may assume the binary point is at the right-hand end of the words), we know from expression 9 that

$$D \geq M + 1 . \tag{11}$$

Therefore,

$$2D \geq 2M + 2 , \tag{12}$$

which can be written

$$2D > 2M + 1 . \tag{13}$$

From expression 13, we deduce that when D is subtracted from $2M+1$, which is the next step in division, the following condition will hold:

$$D > (2M+1) - D \ . \tag{14}$$

Let us call $(2M+1)-D$ the remainder (R). Then using expression 10 we see that

$$R < 2^{11} \ . \tag{15}$$

Expression 15 says that the most significant bit of the remainder must be a zero, as shown in Figure 28 (step 3). The next long-left rotation will therefore insert another zero into the word containing the least significant part of the dividend.

The above argument is true for each step of the division process, so it can be shown by induction that zeroes are always shifted into the least significant portion of the dividend at each step. The only possible difficulty would occur if the result of subtraction of the divisor were negative. However, in this case, the dividend is restored and another shift is made, again inserting a zero as desired. It should be noted that if ones instead of zeros were being shifted out of the left-hand end of the remainder, significant bits would be lost and an incorrect answer would result. Therefore, it is shown that the register is long enough to be used for division by this method and that additional stages on the left hand end are not needed. Figure 28 (step 4) shows the condition of the computer at the end of the division process.

Considerable effort was also expended on this program in order to obtain an efficient method of handling the restoration of the dividend when a subtraction gave a negative result, and to obtain an efficient way of building up the quotient as the division progressed. It is most efficacious to store both the most significant and least significant parts of the dividend after each long-left shift. Then, as shown in the flow chart, if the subtraction of the divisor produces a positive result, the new dividend is stored in place of the old. If the subtraction produces a negative result the contents of the Data Register 2 are discarded instead of being stored, thus effectively restoring the dividend.

The word containing the quotient is initialized as shown in Figure 29 (step 1). The sign is placed in position 11 and all the other bits are zero. To insert a 1 in the quotient the five instructions used are:

```
LOAD    SQUO    GET QUOTIENT
X,X
COMP    1       INSERT 1
,ROT    10      POSITION QUOTIENT
STO     SQUO    SAVE QUOTIENT
```

They operate as follows: After the quotient has been moved to Data Register 2, a 1 is inserted with the instruction COMP 1 as shown in Figure 29 (step 2). Then the quotient is positioned to be ready for the next insertion by a 10-place rotation.

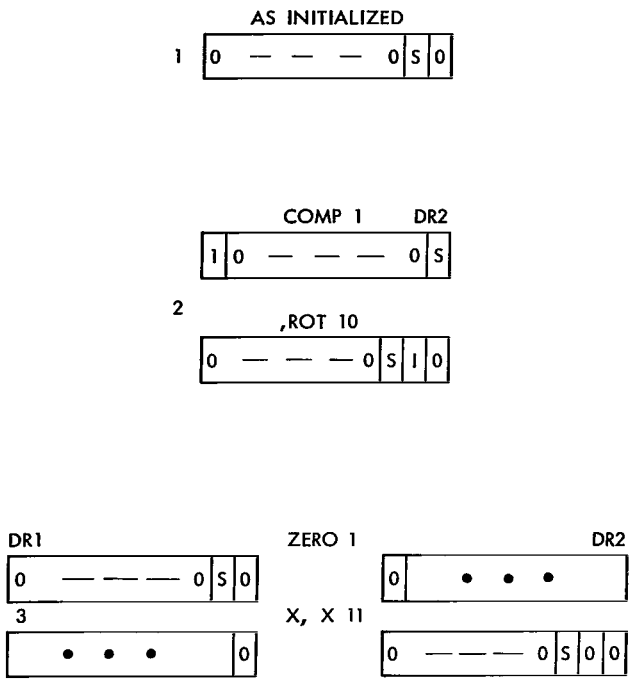

Figure 29—Formation of quotient.

To insert a zero in the quotient, four instructions are required:

LOAD	SQUO	GET QUOTIENT
ZERO	1	INSERT 0
X,X	11	TRANSFER PREVIOUS QUOT
STO	SQUO	SAVE QUOTIENT

The ZERO 1 instruction provides a zero and X,X 11 moves the rest of the quotient into Data Register 2 as shown in Figure 29 (step 3). These particular instruction sequences were chosen because they minimize the number of machine cycles needed for execution. They are also as compact in terms of required program memory locations as the other methods tried.

The sign-and-magnitude division subroutine contains a secondary entry point, VSUB. The use of this entry will be covered in the discussion of the floating-point division subroutine.

Floating-Point Divide

Floating-point division is accomplished by subtracting the characteristic of the divisor from the characteristic of the dividend and dividing the fractional part of the divisor into the fractional part of the dividend. Division of the fractions is performed by the sign-and-magnitude division subroutine. Floating-point division is similar to floating-point multiplication in that, if re-normalization of the fractions is necessary, one shift is the most ever needed.

The divisions for 4-bit normal fractions which produce the largest and the smallest possible quotients are:

$$.1000 \overline{)\ .1111 0000}\ \ 1.1110$$

and

$$.1111 \overline{)\ .1000 0000}\ \ .1001$$

The quotient will be normal unless the division overflows. The flow chart (Figure 30) shows that if an overflow return is made from the sign-and-magnitude division subroutine, the entire double precision dividend is shifted right one place, the characteristic of the quotient is increased by 1, and division of the fraction is re-initiated. The second time, the division subroutine will make a normal return.

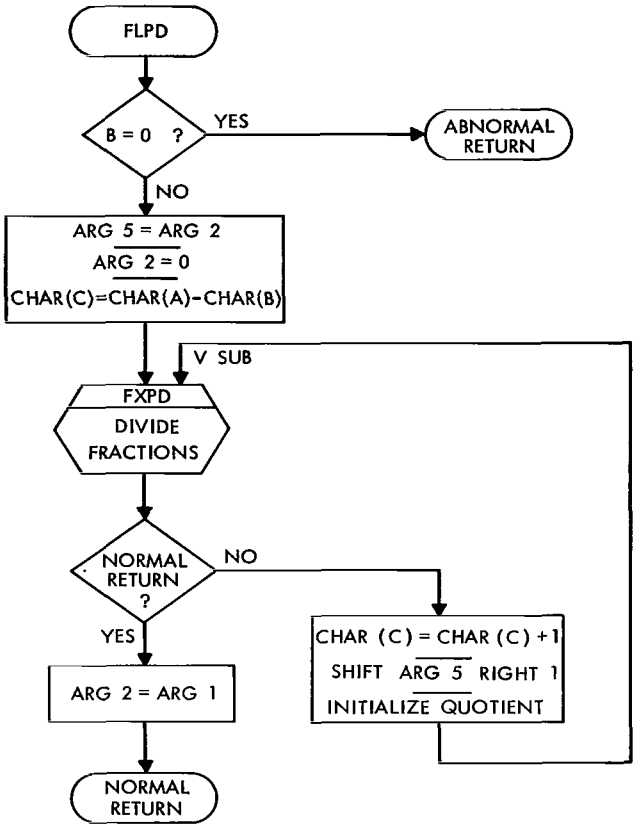

Figure 30—Floating-point divide.

```
                                                              CYCLES
FLPD    LOAD    ARG2                                            2
        X,X                                                    13
        ,Z      0       TEST FOR DIV. BY O                      1
        TRA     XGO           OK                                1
        MIN                   DIVISOR 0                        13
        OCT     7777    -1                                      1
        LOAD    FLPR                                            2
        ADD             FLPR = FLPR - 1                        13
        MIX             ABNORMAL RETURN                        13
*
XGO     STO     ARG5    ARG5 = ARG2                             2
        ZERO                                                   13
        STO     ARG2    ARG2 = 0                                2
*
        LOAD    ARG3    CHAR (A)                                2
        X,X                                                    13
        LOAD    ARG4    CHAR (B)                                2
        SUB             CHAR (C) = CHAR (A) - CHAR (B)         13
        STO     ARG4    CHAR (C)                                2
*
        MIN             CALL FXPD                              13
        TRA     XRET      .    .                                1
        STO     FXPR      .    .                                2
        TRA     FXPD      .    .                                1
*
        TRA     XABN    FXPD ABN. RET.                          1
XRET    LOAD    ARG1    NORMAL RETURN                           2
        X,X                                                    13
        STO     ARG2    ARG2 = ARG1                             2
*
        LOAD    FLPR                                            2
        X,X                                                    13
        MIX             NORMAL RETURN                          13
*
XABN    LOAD    ARG4    CHAR (C)                                2
        MIN                                                    13
        OCT     0001                                            1
        ADD             CHAR (C) = CHAR (C) + 1                13
        STO     ARG4                                            2
*
        LOAD    ARG1    MS (A)                                  2
        X,X                                                    13
        LOAD    ARG2    LS (A)                                  2
        X,OFF   1       1 BIT TO LS (A)                         2
        ZERO    1       0 TO MS (A)                             2
        STO     ARG1    SAVE SHIFTED MS (A)                     2
        MIN                                                    13
        OCT     0001                                            1
        OR                                                     13
        STO     ARG2    SAVE LS (A)                             2
*
        LOAD    ARG1    GET MS (A)                              2
        X,X                                                    13
        LOAD    ARG5    GET B                                   2
        TRA     VSUB    RE-ENTER FXPD                           1
```

Figure 31—Program listing for floating-point divide.

At the beginning of the floating-point division, a test is made to determine whether or not the divisor is zero. If it is, an error return is made. Besides being mathematically ill-defined, division by zero would allow a limitless number of overflow returns from the sign-and-magnitude division subroutine. This must be prevented because valuable data will be lost if the computer hangs up in an endless loop. The only other function performed by the floating-point division program is the moving of data to the correct argument cells before and after the division of the fractions.

Figure 31 contains the coding for floating-point division. It is all straight-forward except for the portion which shifts the dividend right, after an overflow return from the sign-and-magnitude division subroutine. This coding begins 5 instructions past location XABN. A long-right shift of one place is desired. That is, a 0 is to be shifted into Data Register 2 and the contents of Data Register 2 and Data Register 1, taken together, are to be shifted right one place.

First the data registers are loaded. Then, since the computer has no long-right shift instruction, a bit of trickery is used. The instruction X,OFF 1 shifts the least significant bit of Data Register 2 into Data Register 1. The trick is that it does this without altering the contents of Data Register 2 because when the OFF position of SW2 is specified, no strobe pulses are sent to Data Register 2. Then the instruction ZERO 1 shifts a zero into Data Register 2 and completes the operation. After the least significant bit of the least significant part of the dividend is set to 1 (for the end-of-division-detection scheme), the sign-and-magnitude division subroutine is re-entered at VSUB.

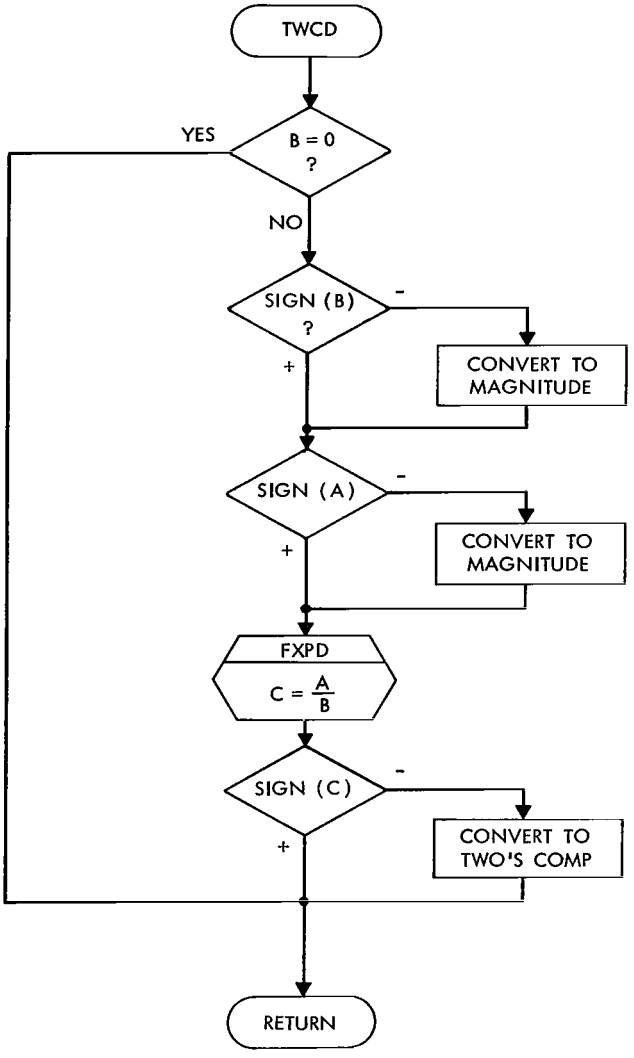

Figure 32—Two's-complement divide.

```
                                                          CYCLES
TWCD    LOAD    ARG2    GET B                              2
        X,X                                                13
        ,Z      0       TEST FOR DIV. BY 0                 1
        TRA     TGO         OK                             1
        TRA     TCG     DIVIDED BY 0                       1
*
TGO     ,B1     0       TEST SIGN (B)                      1
        TRA     TBM         -                              1
TBG     STO     ARG5        +                              2
*
        LOAD    ARG1                                       2
        X,X                                                13
        ,B1     0       TEST SIGN (A)                      1
        TRA     TAM         -                              1
TAG     STO     ARG2        +                              2
*
        ZERO                                               13
        STO     ARG1    ARG1 = 0                           2
*
        MIN             CALL FXPD                          13
        TRA     TRET    . . . . . .                        1
        STO     FXPR    . . . . . .                        2
        TRA     FXPD    . . . . . .                        1
*
TRET    LOAD    ARG2                                       2
        X,X                                                13
        ,B1     0       TEST SIGN (C)                      2
        TRA     TCM         -                              1
TCG     LOAD    TWCR        +                              2
        X,X                                                13
        MIX             RETURN                             13
*
TAM     X,X                                                13
        ZERO                                               13
        SUB     11      COMPLEMENT
                        MAGNITUDE                          12
        X,X     1       INSERT SIGN                        2
        TRA     TAG                                        1
*
TBM     X,X                                                13
        ZERO                                               13
        SUB     11      COMPLEMENT
                        MAGNITUDE                          12
        X,X     1       INSERT SIGN                        2
        TRA     TBG                                        1
*
TCM     X,X                                                13
        ZERO                                               13
        SUB     11      COMPLEMENT
                        MAGNITUDE                          12
        X,X     1       INSERT SIGN                        2
        STO     ARG2    SAVE C                             2
        TRA     TCG                                        1
```

Figure 33—Program listing for two's-complement divide.

Two's-Complement Divide

Figure 32 gives the flow chart for two's-complement division. Similar to two's-complement multiplication, it first converts to sign-and-magnitude form and then uses the sign-and-magnitude subroutine. After division has taken place, it converts the answer to the two's-complement form. The program assumes that the arguments are integers and produces an integer result. It truncates rather than rounding off. The coding appears in Figure 33.

Two's-Complement Add and Subtract

These two operations will usually be performed by open subroutines, which do not require that both arguments be stored, and then recalled from memory. However, to provide a basis for comparison, they are coded here as closed subroutines which find the arguments in the usual arithmetic subroutine argument cells and also leave the results there. Figures 34 and 35 show this coding.

				CYCLES
TWCA	LOAD	ARG 1	A	2
	X, X			13
	LOAD	ARG 2	B	2
	ADD		C=A+B	13
	STO	ARG 2	SAVE C	2
*				
	LOAD	TWCR		2
	X, X			13
	MIX		RETURN	13

Figure 34—Program listing for two's-complement add.

				CYCLES
TWCS	LOAD	ARG 1	A	2
	X, X			13
	LOAD	ARG 2	B	2
	SUB		C=A-B	13
	STO	ARG 2	SAVE C	2
*	LOAD	TWCR		2
	X, X			13
	MIX		RETURN	13

Figure 35—Program listing for two's-complement subtract.

Analysis of the Arithmetic Subroutine Package

Storage Requirements

Table 12 shows the number of program memory locations required for each of the arithmetic subroutines. Together they would occupy 427 locations, or nearly half of the program memory. There are almost 600 locations remaining, however. Also, it was pointed out in the previous section that the program memory could be readily expanded to 2048 locations if the data memory was reduced in size. This would leave 1600 locations for spacecraft data-handling programs. Without writing these programs, one cannot know for certain how many locations they will require. However, experience with the arithmetic subroutines, which are fairly complex, shows that one should be able to do a significant amount of data processing or compression with programs which will fit in the available memory.

The arithmetic subroutine package may be tailored for specific application by leaving out unused subroutines. For instance, if the data compression programs are designed to not require division, then 158 more memory locations are free for use. Eliminating the floating-point subroutines produces an even greater saving. The floating-point subroutines themselves occupy 190 locations. Furthermore, without floating-point operations, the fixed-point addition and subtraction

Table 12

Storage Requirements.

Subroutine	Program Memory Locations
TWCA/TWCS	-
FXPA/FXPS	46
FLPA/FLPS	112
TWCM	37
FXPM	47
FLPM	27
TWCD	42
FXPD	65
FLPD	51
Total	427

subroutine is not required. The total saving is 236 locations. If all operations are done in two's-complement form, only 191 locations are required for the arithmetic subroutines.

Execution Time

In order to estimate the execution times of the various subroutines, some assumptions will be made. A reasonable clock rate, for Series 51 intergrated circuits, is 200 kc. Therefore, it will be assumed that the basic machine cycle is 5μs. This is also a reasonable access time for a spacecraft memory. The number of machine cycles, of 5μs each, required for the execution of an instruction will be the sum of the number of memory accesses and the number of shifts.

All instructions except LOAD and STO make only 1 memory access. That access loads the instruction into the instruction register. LOAD and STO make two memory accesses, one to the program memory and one to the data memory. The number of machine cycles required for each instruction has been listed in the right-most column of each program listing. When the programs were coded, attention was given to minimize their execution times.

Each of the arithmetic subroutines was analyzed to determine the maximum and minimum execution times. The results of this analysis may be found in Table 13. Two's-complement addition and subtraction, and sign-and-magnitude addition and subtraction have modest execution times.

Table 13

Execution Times.

Subroutine	Minimum Machine Cycles	Minimum Time (ms)	Maximum Machine Cycles	Maximum Time (ms)
TWCA	60	.3	60	.3
TWCS	60	.3	60	.3
FXPA	74	.4	129	.6
FXPS	105	.5	160	.8
FLPA	199	1.0	904	4.5
FLPS	230	1.2	935	4.7
TWCM	857	4.3	1330	6.6
FXPM	764	3.8	1105	5.5
FLPM	888	4.4	1242	6.2
TWCD	45	.2	1729	8.6
FXPD	217	1.1	1489	7.4
FLPD	59	.3	1801	9.0

Floating-point addition and subtraction is about 2 to 10 times slower than sign-and-magnitude addition and subtraction because, not only must the fractions be added, but the characteristics must be handled too. Some time is also spent in the subroutine linkage to the sign-and-magnitude program, which is called by the floating-point program. The worse case occurs when the fractions need extensive renormalization after the addition is performed. As many as 10 trips through the normalizing loop can be required.

Multiplication is a lengthy process because the shift-and-add loop is traversed 11 times. Division is even slower because its main loop, which is also traversed 11 times, contains the instructions to produce a long-left shift. The difficulty of programming this operation was discussed earlier in this section. The division subroutines have short minimum execution times because the error conditions are detected early in the programs.

A Sample Computational Program

A sample program has been written to gain some knowledge of the computational power of the computer. This program computes the sum of the squares of a set of floating-point numbers. The program is written in subroutine form. The location of the numbers to be squared and the number of terms to be computed are specified in the calling sequence by the calling program. This makes the subroutine quite flexible.

The flow chart of the subroutine is shown in Figure 36, Table 14 gives storage requirements, and the coding appears as Figure 37. Because there are no index registers or hardware address modification features, the program must compute the address of each number which is to be squared. To do this, the program adds the base address of the data (supplied by the calling program) to the current value of the index. The resulting value is OR'ed to a LOAD instruction which is then used to fetch the data. The program starts with the

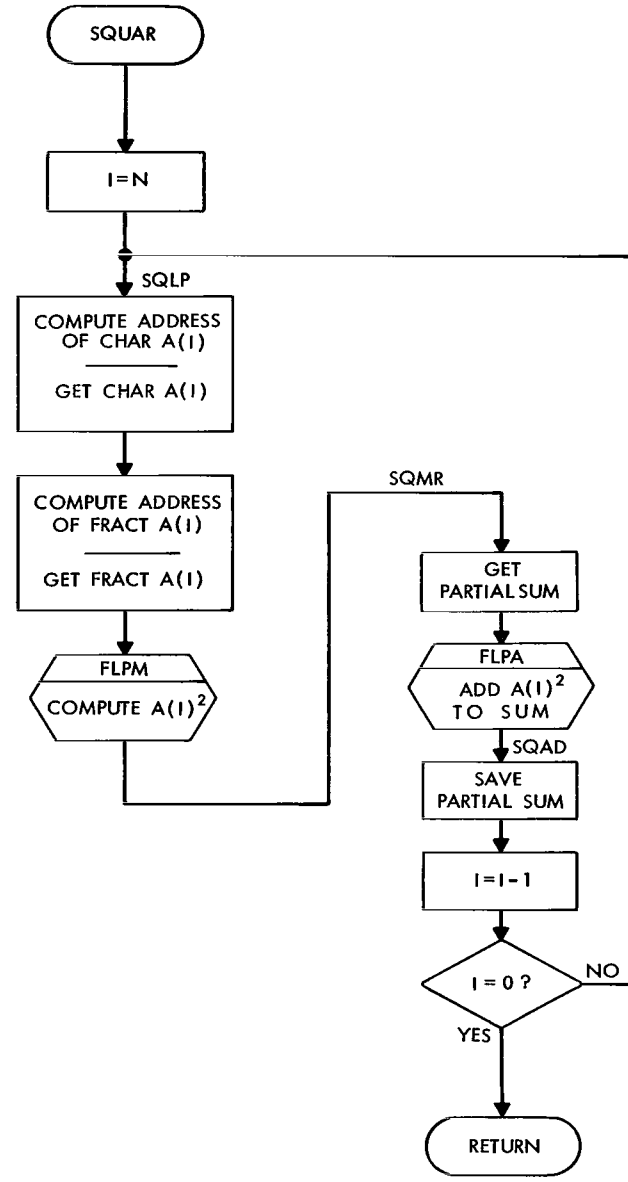

Figure 36—Computation of $\sum_{i=1}^{N} a_i^2$.

Table 14

Storage Requirements for SQUAR.

Data Memory

Location	Data Stored
SQQF	Fraction of Accumulated Sum of Squares
SQQC	Characteristic of Accumulated Sum of Squares
SQT	N (the number of terms in series)
SQAC	Base Address of Characteristics
SQAF	Base Address of Fractions
SQR	Return Address

In addition, two tables of N words each are required to store the fraction and characteristic of the numbers to be squared and added.

Program Memory

54 Locations Required

```
                                                          CYCLES
SQUAR   LOAD    SQT     GET N (I MAX)                       2
        X,X                                                 13
        TRA     SQLP+1                                      1
*
SQLP    STO     SQT                                         2
        LOAD    SQAC                                        2
        ADD             COMPUTE ADDRESS OF A(I) CHAR.       13
        X,X                                                 13
        MIN             MODIFY LOAD WITH ADDRESS OF         13
        LOAD              A(I) CHAR.                        1
        OR                                                  13
        MIX             GET CHAR. OF A(I)                   14
        X,X                                                 13
        STO     ARG3    CHAR A(I) FOR MULT.                 2
        STO     ARG4    . . . . .                           2
*
        LOAD    SQT     GET I                               2
        X,X                                                 13
        LOAD    SQAF                                        2
        ADD             COMPUTE ADDRESS OF A(I) FRACT.      13
        X,X                                                 13
        MIN             MODIFY LOAD WITH ADDRESS OF         13
        LOAD              A(I) FRACT.                       1
        OR                                                  13
        MIX             GET FRACT OF A(I)                   14
        X,X                                                 13
        STO     ARG1    FRACT A(I) FOR MULT                 2
        STO     ARG2    . . . . .                           2
*
        MIN             CALL FL. PT. MULT.                  13
        TRA     SQMR    . . . .                             1
        STO     FLPR    . . . .                             2
        TRA     FLPM    . . . .                             1
*
SQMR    LOAD    SQQC    GET CHAR OF PARTIAL SUM             2
        X,X                                                 13
        STO     ARG3                                        2
        LOAD    SQQF    GET FRACT. OF PARTIAL SUM           2
        X,X                                                 13
        STO     ARG1                                        2
*
        MIN             CALL FL. PT. ADD                    13
        TRA     SQAD    . . . .                             1
        STO     FLPR    . . . .                             2
        TRA     FLPA    . . . .                             1
*
SQAD    LOAD    ARG2                                        2
        X,X                                                 13
        STO     SQQF    SAVE PARTIAL SUM FRACT.             2
        LOAD    ARG4                                        2
        X,X                                                 13
        STO     SQQC    SAVE PARTIAL SUM CHAR.              2
*
        LOAD    SQT     GET I                               2
        MIN                                                 13
        OCT     7777                                        1
        ADD,Z           I=I-1, RESULT ZERO?                 13
        TRA     SQLP      NO, GO ON                         1
        LOAD    SQR       YES, DONE                         2
        X,X                                                 13
        MIX             RETURN                              13
```

Figure 37—Program listing for computation of $\sum_{i=1}^{N} a_i^2$.

highest index value (N) and works backwards through the data until the first number A_1 has been used. The base address which the calling program must specify is the address of the location immediately preceding the first word of the data table. The program requires 54 locations in the program memory.

Exclusive of the time spent in the floating-point subroutines, the program requires 321 machine cycles for each number to be squared and added to the sum. Counting the time in the floating-point subroutines, a total of 2467 cycles is required. Of this time, 87 percent is spent in floating-point multiplication and addition and only 13 percent in the calling sequences to the floating-point subroutines, computation of data addresses, data fetching and storing, etc. To execute 2467 machine cycles requires about 12.3 ms. Therefore, if N numbers are to be squared and added, the time required (in milliseconds) will be 12.3 X N. It would take 1.2 seconds to square and sum a set of 100 numbers.

This example shows that computational programs which use the floating-point subroutines are more likely to be limited by running time rather than program storage requirements. The

programmer should use fixed-point arithmetic where ever practical. In this case, the use of fixed-point arithmetic would reduce multiplication to 1105 cycles, addition to 129 cycles, and would reduce the time in the computational program to around 70 cycles. These total 1204 cycles, somewhat less than half that required when floating-point is used.

CONCLUSION

An extremely simple computer has been designed for use on-board spacecraft. Compared to the usual computer it has a small memory, few registers, few instructions, and short words. Programming, therefore, is difficult and great emphasis needs to be put on writing efficient programs. A package of arithmetic subroutines was written for the computer in order to gain insight into the actual capabilities of this computer. The only arithmetic instructions implemented in hardware are two's-complement add and two's-complement subtract. Addition, subtraction, multiplication, and division were programmed for both sign-and-magnitude and floating-point formats.

The execution times of the arithmetic subroutines appear to be quite large. Particularly those of the floating-point subroutines and all of the multiplication and division subroutines. These items should be put in perspective, however. The transmitted data rate for the AIMP spacecraft is about 25 bits per second. If all the data were to flow through the computer, it would only have to supply the telemetry system with a word twice a second, or once every 500 ms. The longest arithmetic operation is floating-point division (in the worst case it takes 9 ms). This operation could be performed 50 times during the inter-word interval.

The storage requirements for the arithmetic subroutines are quite large. These programs take almost half of the 1024 available locations. However, there is enough room left in the program memory that there should still be enough room for other useful programs. The arithmetic subroutine package uses 10 data locations, an insignificant portion of the data memory. If the 512 word data memory were to be used as a buffer, it would require over 1000 seconds for the telemetry system to empty it. That is almost 17 minutes of continuous data.

Unfortunately there has not been time to continue the research into programming this machine to cover input/output programming. Nor has synchronization of the computer and a telemetry system through the use of the interrupt, trap, and pulse provisions been investigated. It is doubtful that these programs will be any more difficult than, for instance, the sign-and-magnitude division subroutine.

The usefulness of a simple computer, such as the one which is the subject of this research, depends in large measure on the programming efforts applied. To obtain the sign-and-magnitude addition subroutine, for instance, required investigation of five different approaches and more than 30 hours of labor. To code and optimize a program such as the sign-and-magnitude division requires slightly more time.

Clearly, there is yet a large amount of work to be done in investigating the application of small simple computers. This research has shown, however, that a computer such as was designed is promising enough to warrant further investigation.

(Manuscript received May 31, 1966)

Appendix A

Mnemonic Operation Codes

Format A: Single Operation Codes

Mnemonic	Operation
TRA	Unconditional Transfer
MIN	Next instruction word to DR2
MIX	Execute contents of DR2

Format B: Single Operation Code with Operand Field

Mnemonic	Operation
LOAD	Load Contents of Specified Address into DR1
STO	Store Contents of DR2 at Specified Address
PUH*	Pulse Specified Output Lines and Halt
PUG*	Pulse Specified Output Lines and Go On

Note: * Denotes Optional Instructions. In an absolute-minimum computer they would be deleted.

Format C: A, B Count (Arithmetic and Test)

A Mnemonic	Operation
NOP*	No Operation
ADD	Add
SUB	Subtract
OR*	Logical Or
AND*	Logical And
COMP*	Logical Complement
EOR*	Logical Exclusive Or
ZERO*	Set Contents of DR2 to zero

Note: * Denotes optional instructions. In an absolute-minimum computer all of these instructions would be deleted and either NAND or NOR would be added.

B Mnemonic	Operation
Blank*	No Test
C	Test carry flip-flop for zero
Z*	Test contents of DR2 for zero
B1	Test bit 1 of DR2 for zero
I1	Test interrupt line 1
I2	Test interrupt line 2
I3*	Test interrupt line 3
I4*	Test interrupt line 4

Note: * Denotes optional instructions. In an absolute-minimum computer all of these instructions would be deleted. Two interrupt line tests would be retained; one would be used to synchronize data input and one would be used to synchronize data output.

Format D: A, B, C Count (Shift and I/O)

A Mnemonic	Operation
OFF (or blank)	No strobe pulses to DR1
DR1	Connect input of DR1 to output of DR1
DR2	Connect input of DR1 to output of DR2

B Mnemonic	Operation
OFF (or blank)	No strobe pulses to DR2
DR1	Connect input of DR2 to output of DR1
DR2	Connect input of DR2 to output of DR2
IN	Connect input of DR2 to Input Bus

C Mnemonic	Operation
Blank	No output strobe pulses
OUT	Pulse Output Strobe line

"The aeronautical and space activities of the United States shall be conducted so as to contribute . . . to the expansion of human knowledge of phenomena in the atmosphere and space. The Administration shall provide for the widest practicable and appropriate dissemination of information concerning its activities and the results thereof."

—NATIONAL AERONAUTICS AND SPACE ACT OF 1958

NASA SCIENTIFIC AND TECHNICAL PUBLICATIONS

TECHNICAL REPORTS: Scientific and technical information considered important, complete, and a lasting contribution to existing knowledge.

TECHNICAL NOTES: Information less broad in scope but nevertheless of importance as a contribution to existing knowledge.

TECHNICAL MEMORANDUMS: Information receiving limited distribution because of preliminary data, security classification, or other reasons.

CONTRACTOR REPORTS: Technical information generated in connection with a NASA contract or grant and released under NASA auspices.

TECHNICAL TRANSLATIONS: Information published in a foreign language considered to merit NASA distribution in English.

TECHNICAL REPRINTS: Information derived from NASA activities and initially published in the form of journal articles.

SPECIAL PUBLICATIONS: Information derived from or of value to NASA activities but not necessarily reporting the results of individual NASA-programmed scientific efforts. Publications include conference proceedings, monographs, data compilations, handbooks, sourcebooks, and special bibliographies.

Details on the availability of these publications may be obtained from:

SCIENTIFIC AND TECHNICAL INFORMATION DIVISION

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

Washington, D.C. 20546