Assembly Language Programming

for the

Control Data 6000 and Cyber Series

Ralph Grishman

Algorithmics

Assembly Language Programming

for the

Control Data 6000 and Cyber Series

Ralph Grishman

revised and enlarged in collaboration with

Kevin McAuliffe

Algorithmics

DEDICATION

This book is dedicated to A6 & A7,
without which none of the results
in this book could have been saved.

TABLE OF CONTENTS

INTRODUCTION

This text is intended to familiarize users with the design of the Control Data 6000 series, Cyber 70 series, and Cyber 170 series computer systems and to enable them to write central processor assembly language code. No knowledge of the hardware of this or any other computer is assumed; only a knowledge of the basics of FORTRAN, of running a FORTRAN program on a 6000 or Cyber machine, and of English are required.

In 1965, Control Data Corporation began delivery of its 6600 computer system, the most powerful computer delivered up to that time and still a very powerful machine by today's standards. They subsequently added four systems to the 6000 series: three smaller systems, the 6200, 6400, and 6500, and one larger system, the 6700. For the 1970's Control Data reintroduced these machines with a few small additions, as the Cyber 70 series, models 72, 73, and 74. In the mid 70's, using newer technology, they came out with the Cyber 170 series, models 171, 172, 173, 174, and 175. Most recently, to start off the 80's, they have added four models to the Cyber 170 series: the 720, 730, 750, and 760. All of the machines in these series are programmed identically. Thus, although I shall generally refer to the 6600, the information in this volume is applicable to all the above mentioned machines.

In 1969, Control Data regained the "most powerful computer delivered" title with its 7600 system, which is about five or six times faster than the 6600. To the general user the 7600 appears very similar to a 6000 series machine, although the overall system organization is quite different. Nearly all of the material to be presented here will also be applicable to the 7600, subsequently rechristened the Cyber 70 model 76, and to the Cyber 170 model 176, which has the same structure. We shall not consider any of the new features of the 7600 in any detail, however, unless they are connected with the development of the 6000 series or represent corrections of errors in the 6000 series design.

This text will try to be explanatory and not simply expository. That is, we won't simply tell you that the structure of the computer is so and so, and the instructions are (1)..., (2)..., etc., and "that's how things are." Undeniably, it is important to know the instruction set backwards and forwards to write really good code, but there is something else you should understand: why the machine was built the way it was. Clearly it is impossible to show that 6600 is the best machine

configuration (it isn't, as IBM, Univac, and CDC's other competitors will hasten to point out), but you should at least see why it is a reasonably good design, and why some of the choices were made as they were.

CHAPTER 1

THE BASIC DESIGN OF THE 6000 AND CYBER SERIES

1.1 COMPONENTS OF A DIGITAL COMPUTER SYSTEM

Any real electronic data processing system has four functions: the input of instructions and data, the storage of instructions and data, the actual calculation using the data, and the output of results. The input and output (I-O) devices, such as the card reader and printer, which interface the computer with the user and the external environment, are generally grouped together; some devices, in fact, such as magnetic tape drives, serve for both the input and output of information.

The storage of data for the actual computing unit of the system is done in a high speed memory. In some of the earliest computers, the program of instructions was kept on loops of perforated tape, separate from the data, which were then read by the computer. Present computer speeds dictate, however, that the instructions for the computing unit be much more rapidly accessible, so the program of instructions is also stored in the high speed memory. Such a system is thus called a stored program computer. (Using ordinary paper tape to supply its instructions, a 6600 would have to gobble tape at about 30 miles per second!)

1.2 DESIGN OBJECTIVES AND SPEED

The CDC 6000 and 7000 series were designed for large scale, extremely high speed scientific data processing. They are among the most expensive computers ever sold, and were not designed for installation in your local grocery store to help with the bills.

In discussing the speed of these machines, we will never be talking in terms of seconds or milliseconds (thousandths of a second); our basic units will be the microsecond, one millionth

of a second (abbreviated us) and the nanosecond, one billionth of a second (abbreviated ns). The 6600 is able to add two 18-digit integers in 300ns; to add two floating-point (i.e., FORTRAN type REAL) numbers in 400ns, with 14-place accuracy; and to multiply two floating-point numbers in 1000ns (=1 us). And, as if that weren't fast enough, the 6600 can do two multiplications, one integer addition, and one addition of floating numbers (and a few other things) simultaneously. Although just how it can keep all these things going concurrently is rather complicated, you should have some idea now of how fast the 6600's processing unit is. Going along at a typical 3 million calculations per second, a 6600, for example, should have no problem doing arithmetic accurately, faster than the entire population of the United States with paper and pencil. The 6600 obtains such speed from cleverly designed electronic circuitry, based upon electronic switches which can switch in 5ns (200,000,000 times a second).

The lower-numbered members of the 6000 series are of simpler design and correspondingly slower. The 6400, for example, takes 600ns to do an integer addition and 5.7us (5700ns) to do a floating-point multiply; also, it can only do one operation at a time. The 7600, using faster circuitry than the 6000 series, can do an integer add in 55ns and a floating-point multiply in 137.5ns.

The memory of a 6000 series machine is large and also quite fast. It has a maximum capacity of over 130,000 numbers. The time it takes the memory to deliver a particular number to the processing unit after the processing unit asks for it -- called the access time of the memory -- is 500ns. After a result is calculated in the processing unit and sent to the memory, it takes the memory 1000ns to store this result. To make things go even faster, the memory is divided up into 32 sections, called banks, and the memory unit is able to read numbers out of, or store numbers into, several banks, i.e., several parts of memory, at the same time. The memory of the 7600 is about four times faster: access time is 137.5ns, full cycle (store) time is 275ns.

The memory of the 6000 and 7000 series is made up of very large arrays of tiny ferrite (iron) magnetic cores (doughnut-shaped rings, a fraction of an inch in diameter), so it is often called core memory or core storage. Information is retained in core storage by magnetizing the individual cores; more about this later. Core storage is said to be random-access; that is, the time to access any information in the memory is the same, regardless of which piece of information is being accessed.

This is in contrast to magnetic tape, for example, where one can get at the information on the tape directly under the tape head (the part that reads and writes on the tape) right away, but one may have to rewind the tape for a minute or two to get what is on the beginning of the tape.

1.3 CENTRAL PROCESSOR DESIGN

Most of the machines of the early 1960's (the so-called second generation machines) were very much memory dependent. Typically, such a machine has a few transistor registers (electronic devices for holding numbers, as opposed to cores) for computation, but in an arithmetic operation one of the two operands comes from memory. For example, an add instruction would add a number stored in core memory to a number in one of the registers; a multiply instruction would multiply a number in memory by one in one of the registers. The advantage of this scheme is that you don't need many registers; in fact, if you want to be really cheap about it, you need only one. But there is one hitch: after the machine figures out that an instruction is an add instruction, it has to request the number from memory for the addition, and then wait until memory returns the needed number. The arithmetic unit can get the number out of the register very quickly, compared to memory speeds. The rest of the time, while the storage unit is getting out the information, the arithmetic section is waiting without anything to do.

In the "old days," when the arithmetic section was slow, this state of affairs wasn't too bad. It took about 1 to 2us to access information from storage, but it took about 10us to do a multiply and 20us to do a divide, so the extra 1 or 2us were not so bad. Nonetheless, to save these couple of microseconds, the computer designers used something called look-ahead. In the simplest form of look-ahead, the computer accesses the next instruction from memory at the same time as it is executing the present instruction; this is known as instruction overlap. This technique enables the processing unit to figure out what the next instruction is, and possibly even to request the operand (number) for the next operation before the previous calculation is finished.

Some machines, such as IBM's Stretch (7030), tried to do even better by an intricate look-ahead scheme, which examines the

subsequent instructions and tries to determine what data will be needed. This scheme, however, has some difficulties: for example, one instruction may determine from which place in core storage the next instruction will take its operand. Furthermore, as long as there are only a few registers in which arithmetic is done, the chances are quite good that one instruction will require the result of a previous instruction as an operand. So, even if the machine could handle more instructions at the same time, overlap is limited; one instruction using a register could not begin executing until a previous one delivering its results to this register is finished. It turned out, as a result, that the look-ahead did not make the machine so much faster as it did make it more expensive.

In the 6600, any computer organization that forced the computer to wait for an operand to come from memory would be very wasteful, since the arithmetic section is much faster (multiply 1us, other operations in 300ns). Knowing that one can get a number much faster out of a (transistor) register than from core storage, you might suggest that we build a machine with dozens or hundreds of registers. This solution, unfortunately, overlooks the shortage of that priceless ingredient, money: high-speed registers are about two orders of magnitude more expensive than core storage, at least. Aware of the rather limited market for billion-dollar computers, we have to limit ourselves to a few registers, and seek other solutions.

So the problem remains: how to avoid this situation without having a complicated look-ahead scheme which isn't very good anyway? The answer: have the programer do the looking ahead; i.e., shortly before the instruction to perform an add, multiply, etc., let the programmer put an instruction which tells the processing unit to load the operand from memory. In this way, the operand is already sitting in one of the registers, and can be fetched by the arithmetic unit very quickly.

This method imposes another requirement: we have to have several registers, so that the arithmetic unit can use two of them for operands, put the result in a third, while at the same time operands can be loaded from memory for the next instruction into a fourth and fifth, and the result of the previous operation kept in a sixth until it is stored in memory. The 6600 has 8 such high-speed registers, called X registers.

The place in memory where a memory word (number) is stored is called a location; like positions in a one-dimensional array, locations in core are designated by numbers. The number of the location where a particular variable or other information is stored is called its address.

Now let's say that, in order to figure out the address of a word (number) we want to load from memory into a register, we have to do a calculation such as adding two numbers. On a typical second generation machine, we would have to (1) do the calculation in an arithmetic register, (2) store the result in a special register, called an index register, and (3) execute the load instruction, with a "flag" in the instruction which tells the machine to use the address in one of the index registers. For example, if we want to load the contents of successive locations (as in a DO loop with subscripted variables), we would first have to increment the index register, and then execute the load. If you think about it for a while, you will see that this is clearly a waste of time; why not have a special "index register" which automatically loads the word into an arithmetic register when it is set to an address? And, similarly, have "index registers" which, when set, store an arithmetic register at the location in memory specified by the contents of the index register. This is precisely what is done on the 6600. These special registers are called A registers, and there are 8 of them, paired off with X registers. When some A registers are set to an address, the contents of that location is loaded into the associated X register; when other A registers are set, the contents of the X register is stored at that location.

So far we have A and X registers; do we need any others? We haven't considered yet the usual case with index registers, in which we have a counter (such as a DO loop index) which is not equal to an address which we want stored or loaded during the loop. We wouldn't want to keep this count in an A register, since it would be wasteful to have the computer load or store a location which we don't want. We could, of course, put the index in an X register. Loop indices, however, are usually quite small (since a good-sized loop takes a while to do even a million times) whereas the arithmetic operands of the problem being computed in X registers are often much larger than a million, or require more than 6 significant figures. Since registers which hold more digits (bigger numbers) cost more, reasonably enough, it would be wasteful to allocate additional large X registers for small counters. As a result, for reasons of economy, we have, for indices and similar purposes, a set of special registers, smaller than the X registers, called B registers (or index registers). Of course, if a counter is required which is larger than can be put into a B register, one can always use an X register for it.

1.4 PERIPHERAL PROCESSORS

Thus we have a processing unit for our computer with A, B, and X registers for holding data, and hardware for executing all the instructions -- doing the arithmetic operations. Now let's take the memory unit, this processing unit, and the required electronics for routing data back and forth, and wire them all together. What have we got? A most expensive pile of electronic junk (no reflection on the 6600).

What have we forgotten? Connections for the input and output of information. Unless the machine can input data and instructions and output results, it clearly will do us no good. And unless it can input and output fast enough, all the speed and power of the computer are for naught.

In the most elementary computers, input and output are taken care of by the same electronics that routes data between memory and the arithmetic unit. Thus, when an output instruction is executed, the computer requests a word (number) from memory as it would for a arithmetic operation, but instead of loading it into a register, it sends it out along one of the input-output lines, to a printer, for example. Alternately, it could take a word already in one of the registers and send it out on the same lines. The disadvantage of this scheme is thatno arithmetic processing can occur during input or output, since there is only one path to and from memory, and only one control section to direct the flow of data.

The logical solution is to create several additional paths to and from memory, each with its own control section. The control section, with internal connections to memory and external connections to input-output lines, is known as a data channel. The main control unit, wich executes the instructions in memory, can then instruct one data channel to read a certain section of memory and write it on magnetic tape, instruct another to transmit a second region of memory to the printer, and get a third to read information from punched cards and store it in memory. These commands are issued to the data channels when the main control section encounters an input or output instruction, just as it would instruct the arithmetic unit to do a multiply if a multiply instruction were encountered. Commands to the data channels may instruct them to do input-output, to pass on commands to an I-O device (e.g., tell a tape transport to rewind a tape), or to return to the main control section the status of any input-output requests previously made (successfully completed, error encountered, etc.).

The data channel concept was a very good one, and almost all second generation (early 1960's) computers used some variation of this scheme. In fact, with some further alterations, it is being used on most third generation (present-day) computers. To reduce the need for the program (i.e., the main control unit) to regularly interrogate the data channels for information, systems of interrupts have been devised for many computers. When an interrupt is on ("enabled") and a specified condition occurs (e.g., an error during input or output, the completion of input or output), the computer automatically transfers control to (starts executing) a program in memory designed to take care of the situation. For example, if a card gets stuck in the card reader, the computer might interrupt to a program which prints a line to the computer operator, "CARD READER JAMMED." Interrupt systems minimize the time that the computer has to spend checking up on the data channels.

This system is fine, especially for I-O devices that can transmit large quantities of information without much supervision, such as magnetic tape units. In contrast, suppose the computer was connected to 100 terminals -- units like typewriters, on which users can communicate with the computer. If "character-by-character response" were desired -- every time someone types in a character, the computer checks if any action by the computer is necessary -- the computer would be interrupted fairly often. Similarly, if a magnetic disk is used for storing programs to be executed, as on the 6600, and, as on the 6600, data is stored in small packets of 64 words, which can be read or written in 500us, there will be a need for frequent interrupts. Even if the data channel were capable of writing or reading several sectors (packets of 64 words) by itself it would be necessary to use the computer regularly to determine if the last needed sector had just been read, or where the next sector of a program is, or, if a search of part of the disk is being made, to examine the data as it is read in.

Thus it is clear that if several different I-O operations are going on simultaneously, the computer would have to be interrupted quite often. But many other computers, designed to handle considerable amounts of I-O, have adopted an interrupt scheme; why shouldn't the 6600? To answer this question we have to keep in mind several facts about the 6600: first, that it has a number of registers, and second, that it will usually be executing several instructions at the same time. Now, when an interrupt occurs, we would like to save the contents of all the registers, so that they will be unaltered when the interrupt program is over and control returns to the regular program. We can do this in two ways: first, we could use a different set of

registers for the interrupt program, and simply switch registers when an interrupt occurs; this is a very fast method, but a second large set of registers isn't cheap. As a result, a second, slower scheme was used: whenever the 6600 stops executing one program and starts executing the next, all the registers are automatically stored in memory and new values for the next program are loaded from memory. Since, at any moment, several instructions are executing, the computer has to wait until all the instructions are finished before it can begin storing the registers; as a result, it can take quite a few microseconds to change programs. If the 6600 were to interrupt every time some processing was needed for I-O operations, possibly 2,000 times a second or more if a lot of I-O is going on simultaneously, a sizeable fraction of the time of the main arithmetic unit would be spent simply exchanging back and forth between the regular and interrupt programs. So, no matter how fast the main processor could do the I-O chores, a significant amount of its time would be used up. In addition, the I-O processing is usually quite simple -- comparing numbers, searching for a particular item -- so during I-O processing the extraordinary power of the arithmetic unit would be largely wasted.

The logical solution is to provide each data channel with the ability to do some simple processing. The sophisticated data channels of the IBM 7000 series took a step in this direction: when a data channel finished one operation, it could read the next data channel commmand from memory without any intervention by the central processing unit (the main program). These data channels, however, still performed only I-O instructions.

The next step, as taken in the CDC 6000 series, is to give the "data channels" the ability to perform elementary arithmetic operations. These souped-up data channels can then take care of such processing as determining the next sector on the disk to read or write, searching for a certain item on the disk, or checking the input of 100 terminals letter-by-letter, and leave the central processor to do the work it was designed for: the more complex arithmetic processing for users' programs. These super data channels are called peripheral processors, because they act as an interface between the peripheral (input-output) eqipment. Warning: in the 6600, "data channel" refers to the lines to I-O equipment, not unlike the data channels of the very simple computers mentioned earlier.

For several reasons, the peripheral processors have been given their own individual memories, rather than share in the main memory of the central processor, the central memory. One reason is that the "ideal" word size for the PP's (peripheral

processors) is considerable smaller than that for the central processor; firstly, the "basic unit" in I-O is generally small, typically one number or character (e.g., one card column). Secondly, I-O devices are usually slow compared to computer speeds, so a PP can keep up with external equipment even if it works only a few digits at a time. And thirdly, they do not require a large word size for precision artithmetic, since they are designed to handle chiefly simple calculations. As a result, it would be wasteful to have a word size as large as that of the central processor. Furthermore, direct access of all PP's to central memory would require considerably more central memory electronics, and would complicate the control hardware of the PP's.

Thus we now have a machine configuration consisting of a high-speed central processor and a set of independent, less powerful peripheral processors for input and output. (Figure 1, page 11.) Because the PP's are all logically independent processors, running with their own programs, they can perform another function, in addition to doing I-O: they can act as system monitors. A system monitor supervises the running of the computer system: oversees input and output, determines which program will run on the central processor at any moment, and keeps records of all activities. In a 6600 system, there are normally 10 PP's; one or two can be assigned to supervisory functions, and the rest used for I-O operations.

1.5 VARIATIONS ON A THEME

The first machines in Control Data's 6000 series, the 6600 and the 6400, were originally offered only in the configuration shown in Figure 1. In order to expand and diversify its series, CDC soon offered two significant options: multiprocessors and extended core storage.

A multiprocessor configuration is simply an arrangement whereby two or more central processors are attached to central memory. Hooking together two 6400 processors yields a 6500, which ranks somewhere between a 6400 and a 6600 in computing power. Adding a 6400 central processor to a system with a 6600 yields a 6700, the top of the line. With either of these systems, the two CP's work on separate jobs in separate sections of central memory, so the

user need never be concerned that there is more than one CP in the system.

Extended core storage (ECS) is a larger, slower, and cheaper form of the core storage used for central memory; it is available in sizes up to 1 million words for the 6000 series and 1/2 million for the 7600. It provides a intermediate level of storage between high-speed central memory and the relatively slow-speed disk. It may be used to hold data or operating system tables which are not referenced often enough to justify a place in central memory, or programs which are executed very often (such as the FORTRAN compiler). Programs cannot be executed directly from extended core storage; they must first be moved to central memory.

ECS has been specifically designed to allow for the rapid transfer of large blocks of data from ECS to central memory and back. On 6000 series machines, the data rate is 10 million words per second; on the 7600, 36 million words per second. (To get some grasp of what 36 million words per second means, consider that the entire contents of the 2000-page Manhattan telephone directory could be transmitted in one-tenth of a second, or, more pertinently, that a FORTRAN compiler can be moved into central memory in about one-third of a millisecond.) A 6000 or 7000 series central processor can move a block of words to or from ECS with a single instruction; in addition, the 7600 CP can load a word into an X register from ECS and store into ECS from an X register. These particular instructions will not be discussed in detail in this volume; further information on them can be obtained from the reference manual for your machine.

In the usual ECS configuration, only the central processor, and not the peripheral processors, can access ECS. If the tables and principal programs of the operating system are kept in ECS, this means that a peripheral processor which wishes to change an entry in an operating system table (when a job starts or finishes executing, for example) has to request the central processor to make the change. This suggests, of course, that some system "bookkeeping" functions be performed by the central processor. Provisions have been made for this on the 6000 series machines through a special instruction (the "monitor exchange jump") which enables the central processor to switch back and forth between the user's program and the operating system's monitor program.

The 7600 system, which always includes ECS, was designed specifically to run with a central processor monitor. On one hand, a 7600 PP cannot do some things a 6600 PP could (such as start and stop the CP, or read any word in central memory) while,

on the other hand, the 7600 CP has some additional instructions to keep track of the PP's.

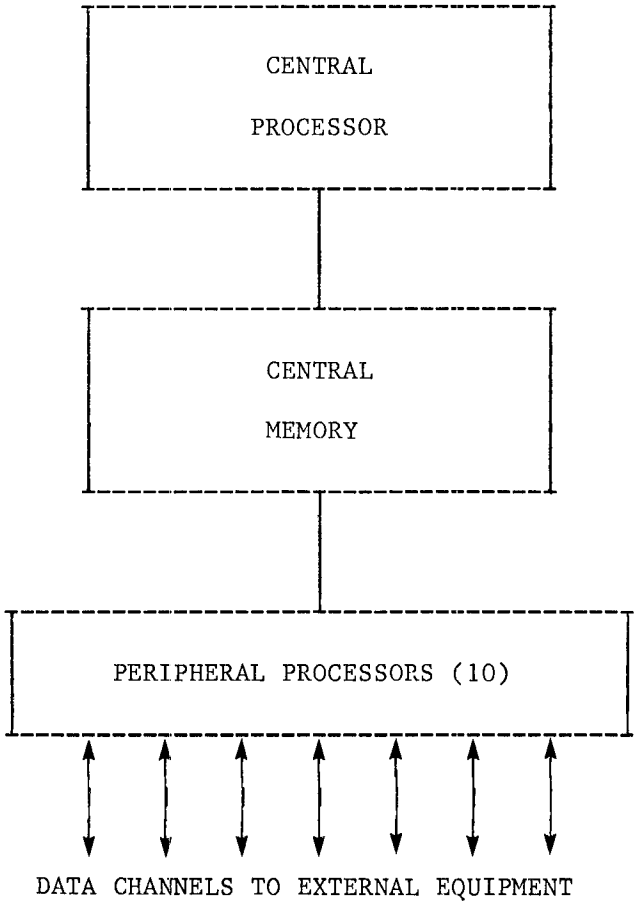


FIGURE 1

1.6 THE BOLD STEP FORWARD

As any marketing expert or homemaker will tell you, it just won't do to keep on selling the same product year after year under the same name. After a few years, good old SPQR becomes New Improved SPQR with Miracle Whitener. This is especially important in the computer business, where having the "newest and fastest" is a status and selling point. So, to meet the challenge of the 70's, CDC rechristened the 6000 and 7000 machines the Cyber 70 series. The 6200 became a Cyber 70 model 72, the 6400 a model 73, the 660 a model 74, and the 7600 a model 76. The dual processor systems, which formerly got the separate model designations 6500 and 6700, became options on the Cyber 70 models 72, 73, and 74.

The role of the New Miracle Whitener is played by the compare and Move Unit, or CMU. This small addition to the central processor was designed to make a computer originally intended for scientific calculations more efficient in commercial and text-processing applications. It is standard equipment on the Cyber 70 models 72 ad 73, and available as an option on the model 76. We shall consider the capabilities of the CMU in detail at the end of Chapter 3.

In the mid 70's, these machines were redesigned to take advantage of newer technology. In place of individual transistors, they used integrated circuits - small packages (less than an inch long) containing dozens of transistors. In place of core memory, they used integrated circuits that could store 1024 bits on each "chip" (package). Although quite different internally, these machines - the Cyber 170 models 171 through 175 - are programmed identically to the earlier series. Further advances in technology (such as memory circuits holding 4096 bits per chip) led CDC in 1979 to introduce four more models in the Cyber 170 line - the 720, 730, 750, and 760. These too were "program compatible" (identical from the programmer's viewpoint) with earlier models.

CHAPTER 2

NUMBER SYSTEMS AND COMPUTER ARITHMETIC

2.1 THE BINARY NATURE OF COMPUTERS

The question of the number system to be used by the computer in doing arithmetic is one of the basic questions in computer design. To those of you not familiar with the way arithmetic is done on computer, it may seem that "addition is addition, after all," regardless of the number system used. We shall presently see the error of such ideas.

It is a basic fact of computer design that virtually all digital electronic components built for computers are binary in nature. That is, every component can be in either of two states: a transistor, which is basically an electronic switch, can be either conducting (on) or nonconducting (off); a magnetic core, the tiny ring-shaped piece of ferrite used in core memories, can be magnetized in either a clockwise or counterclockwise direction; a light can be either on or off, etc. In each case, we associate one state with the digit 0 and one with the digit 1; the choice is usually quite arbitrary. Each digit in our representation may be either a 0 or a 1; hence it is called a binary digit, or bit. Thus each basic computer component represents one bit of information, a 0 or 1, yes or no, on or off datum. Larger amounts of information may be represented by using several bits together. Thus, a pair of binary digits may take on any of four possible values (00, 01, 10, and 11), a set of three bits may have any of 8 different values, and so forth. In general, n bits (which may be represented by n binary computer components) may have any of 2**n different values.

2.2 BINARY AND BINARY CODED DECIMAL REPRESENTATIONS

Using four bits, we can represent the first 2**4 = 16 numbers (0-15). There are many ways in which we can associate these 16 numbers with the 16 possible arrangements of four binary digits. One arrangement, however, is particularly desirable from both a logical and an electronic point of view; this arrangement is known as the binary number system. It uses the concept of place value in the same way as the decimal system, except that, instead of the place values going up in powers of ten, they go up in powers of two. Thus, just as

```
123 (base 10) = 1*10**2 + 2*10**1 + 3*10**0

              = 1*100   + 2*10    + 3
```

so

```
1101 (base 2) = 1*2**3  + 1*2**2  + 0*2**1 + 1*2**0

              = 1*8     + 1*4     + 0*2    + 1
```

Thus:

```
123 (base 10) = 1111011 (base 2)

              = 1*64 + 1*32 + 1*16 + 1*8 + 0*4 + 1*2 + 1
```

If one wants to store numbers, and not to do any arithmetic processing, the binary system has the one advantage that one need not memorize the value of each bit pattern, but, to convert binary to decimal, one need simply to add up the place values of all the "1" bits. Since, for everyday purposes, we would like to talk to the computer in decimal, it is important that we have a simple scheme for going back and forth between decimal and binary notation.

But there are other arrangements for storing numbers in memory which are even simpler to convert to decimal. Since four bits may take on any of 16 possible values, it is easy enough to use four bits to represent one decimal digit, 0-9. Again, there are many possible representations or codes for associating the decimal digits with the values of the four bits (16!/6!,which equals about 29,000,000,000 codes). However, there are again only a few desirable arrangements. One, in particular, codes each decimal digit in the binary system: 0 = 0000, 1 = 0001, 2 = 0010, 3 = 0011, ..., 9 = 1001.

In order to store a decimal number with more than one digit, we need merely set aside four bits for each decimal digit, and code

each decimal digit separately in its four bits. This technique is called binary coded decimal, or BCD. Thus in the computer memory we would store 123 as 0001 0010 0011, and 4978 as 0100 1001 0111 1000. This certainly is easier to convert to decimal than ordinary binary; in fact, after some practice, you could practically read the decimal when looking at the binary. Then why use the binary system at all? Simply because it is easier for the machine to do arithmetic in the binary system, as we shall see in the next section.

2.3 ARITHMETIC IN THE BINARY SYSTEM

To add two binary numbers is very simple; just as when adding in decimal, we do it bit by bit, starting at the right. Just as in decimal, 0+0 = 0, 1+0 = 1, 0+1 = 1; now 1+1 = 2 in decimal, but there is no 2 in binary. So in binary we write 1+1 = 10 or, if we are continuing the addition, 1+1 = 0 plus a 1 bit carry (since a 1 carried on to the next place is a 2 with respect to the present place). Now the only remaining problem is, what to do when adding up the next column, with a carry to take into account? It should be reasonably clear that, when there is a carry from the previous column, 0+0 = 1, 0+1 = 0 with a carry into the next column (i.e., = 10 (base 2) = 2), 1+0 = 0 with a carry into the next column, and 1+1 = 1 with a carry into the next column (i.e., = 11 (base2) = 3). In case this isn't absolutely clear, some examples:

```
        (carries -->          11  )

addend 1 =                 100110
addend 2 =               + 010011
                           ------
     sum =                 111001


        (carries -->       1111111 )

addend 1 =                 11111111
addend 2 =                        1
                           ---------
     sum =                 100000000
```

What makes this simple is that there are only eight possibilities, 0+0, 0+1, 1+0 and 1+1, with and without a carry, while in decimal arithmetic there are two hundred possibilities

(two possible carries, 0 or 1, ten possible digits in each place of each number). If we were to store the decimal digits as binary numbers, as suggested above, we could add the individual digits like binary numbers, but it would still be necessary to check whether, in doing the addition, we created an illegal digit (one greater than nine). For example, in

```
  23  =     0010      0011
+ 49  =   + 0100      1001
          ----------------
            0110      1100
```

the low-order digit is 12, so we have to change it to 2, and propagate a carry into the ten's digit, so that the result is

```
            0111      0010  =  72.
```

Regardless of what BCD code we used for the decimal digits, in fact, we would have to do some checking after the addition for illegal digits; as a result, it is simpler and faster to do binary addition (both for you and the computer) than to do BCD addition.

To learn binary multiplication is even easier: do the long multiplication just as you would decimal numbers, and then add the partial products up as you just learned to add binary numbers; for example,

```
      1000110
      * 10101
      -------
      1000110
    1000110
  1000110
  ------------
  10110111110
```

another example

```
      11101110
       * 11101
      --------
      11101110
    11101110
   11101110
  11101110
 ------------
 1101011110110
```

If you get confused keeping track of all the carries in binary, do as the computer does: as you multiply, keep a running total of the partial products so you only have to add two numbers together at a time. In fact, if you take a careful look at what you have done, you will see that the computer doesn't really have to multiply in the sense in which you had to learn to multiply in decimal. All it does is shift one of the numbers to the left, one binary place at a time, and check whether the corresponding bit in the other number (starting with the rightmost bit and going left) is 1; if it is, add the first number into the subtotal (in its present, shifted, position) before continuing with the shifting.

This makes binary multiplication really easy for the computer. In comparison, to do multiplication in decimal, it would have to learn a 10*10 multiplication table. Some machines which do decimal arithmetic, such as the IBM 1620 computer, store addition and multiplication tables in memory, and look them up every time they do an arithmetic operation; that is, to say the least, a slow process.

At this point we have significant information for a first decision concerning the number system for our computer: binary or decimal (BCD). It should be clear by now that if a lot of arithmetic will be done with the numbers in memory, it is simpler (in terms of computer electronics) and faster to use the binary system. On the other hand, for some business uses where an enormous amount of data is read in and out, many of the numbers are never used in arithmetic operations (who ever heard of multiplying a Social Security number?), and only some simple arithmetic is performed with the rest, decimal arithmetic may be best in order to save the time converting back and forth between decimal and binary. The CDC 6600, however, was made essentially for scientific uses, where a lot of arithmetic is done (in comparison with the amount of I-O), so the choice is clear: use the binary number representation.

Having covered addition and multiplication, let us complete the sequence with a discussion of subtraction and division. Just as for addition -- binary or decimal -- we need the concept of a carry, for subtraction we need the concept of a "borrow." The rules for doing a subtraction are quite simple: 0-0 = 0, 1-0 = 1, 1-1 = 0, and 0-1 = 1 with a borrow from the column on the left (in effect, borrowing 2, just as you borrow 10 in decimal). Borrowing from a 1 simply makes it a 0, while borrowing from a 0 makes it a 1 and requires still another borrow from the next column to the left (compare with the process involved in the subtraction, in decimal, 100 - 1). Some examples to make this

clear:

```
  1101111010          100000000
- 100110101         -        1
 ----------         ----------
  1001000101          11111111
```

In case you might have been confused by all those borrows, I have some good news for you: we are about to show that it is really unnecessary to learn to subtract at all. You may have heard this line somewhere before -- when you were told that you really didn't have to subtract, you just added a negative number. Well, this information really won't help you too much (unless you like to use nine's complements), since to add a negative you probably go through the motions of a subtraction anyway. But in a computer, where the electronics for doing a subtraction as we did above would require quite a few electronic components, we would much prefer to use some sly trick which enabled us to get away with having only an adder (and not a subtracter). After all, electronic circuits don't grow on trees.

So, we are going to investigate the possibility of adding negative numbers, instead of subtracting, in binary. Our first problem is how to get a negative number, but that isn't very difficult: simply subtract a number from a smaller number, say 24 - 32.

```
  011000
- 100000
 -------
 ?11000
```

What now? Well, there are at least two reasonable possibilities. Firstly, we could, if we were subtracting numbers with only six (or fewer) bits, agree that we are allowed a free borrow from the next column (sort of an imagined 1 bit in the next column of the upper number); in effect, we disregard any borrow into the last column. Then, where the "?" is, we would simply put a one:

```
24 - 32 =   011000
          - 100000
           -------
            111000  = -8
```

So -8 = 111000 for six-bit numbers. Alternately, we could agree to borrow from the bottom (rightmost) column; in this case things are a little bit more complicated because we have to continue borrowing up to the fourth column:

```
  011000
- 100000
  ------
  111000
-      1 borrow
  ------
  110111
```

Now, you might dare to ask, why in the world should we take the borrow from the low-order digit? Take a careful look at our second result: -8 = 110111 (for six-bit numbers), and recall that 8 = 001000; now do you notice anything? Wherever there is a one in the binary representation of the positive number, there is a zero in the binary of the negative number, and vice versa; this is called the one's complement of the number. The other representation of the negative number, the first one calculated, is called the two's complement; it is equal to the one's complement plus one (since to form the two's complement we don't borrow 1 from the low-order (rightmost) bit, as we did for the one's complement).

Let's assume for the moment that whenever we do a subtraction with end around borrow (borrow from the low bit) and get a negative number, it will be the one's complement of the positive number (we will prove this shortly). Then we have a simple rule for subtraction: just add the one's complement of the number. For this method to work, however, we have to keep one thing in mind: when we borrowed for the high-order (leftmost) bit, we took a 1 from the low-order bit, so when we have a carry from the high-order bit, we have to carry it around to the low-order bit. This, logically enough, is called an end around carry. Some examples:

```
32 - 24  =  00100000        00100000
         - 00011000  =  + 11100111
           --------       --------
                          00000111
                      +         1 <-- end around carry
                          --------
                          00001000  =  8!

28 - 1   =  00011100        00011100
         -  00000001  =  + 11111110
            --------       --------
                          00011010
                      +         1 <-- end around carry
                          --------
                          00011011  =  27
```

```
13 - 0  =   00001101        00001101
          - 00000000  =   + 11111111
            --------        --------
                            00001100
                          +        1 <-- end around carry
                            --------
                            00001101  =  13
```

Notice that in our third example we have found another zero (the so-called -0 = 11111111), in other words, another number which when added to any number N gives the same number N. Now, in order to find the negative of a number, one method is to subtract that number from zero; so, we can subtract the number from -0 just as well. We will thus get the one's complement of the number, since subtracting one from one gives zero, and zero from one gives one:

```
-0 - 27 =                   11111111
                          - 00011011
                            --------
                            11100100  = -27
```

Thus, we have "proven" that if we use end around borrows and carrys (one's complement arithmetic), the negative of a number is the one's complement of the number.

Now recall, if you will, why we started on this examination of negative numbers: we wanted to see if it is possible to perform subtraction on the computer using addition. We have found that one can do subtraction by adding the complement of the number, and that there are two convenient complements: one of them, the one's complement, can be calculated by simply putting ones where there are zeros and vice versa. The other, the two's complement, is found by adding one to the one's complement. The one's complement is clearly simpler to calculate (it requires only one operation) and largely for this reason one's complement arithmetic was selected for the 6600.

Having now reached this decision, some words of caution are in order. When we wrote down numbers and their complements above, we were usually careful to specify, "for eight-bit numbers," etc. Why we had to do this should be evident by now. If we have a five-bit number, say 27 (base 10) = 11011 (base 2), and want to represent it on a register which has six bits (six binary electronic components), we would write 011011; if we had an eight-bit register, we would write 00011011. In other words, if there are more bits than we need for the number, we do the

obvious thing, and fill the register with zeros. But consider now putting -27, the one's complement of 27, in the register; for the six-bit register, we have the complement of 011011, which is 100100, while for the eight-bit register we have the complement of 00011011, or 11100100. That is, when we put a negative number into a larger register we have to fill the register with ones, not zeros!

Complement arithmetic brings up one other problem. Let's say we have a six-bit register, and I tell you that it contains 101101. What number is in that register? You could say either 45 or -18 (in one's complement), but you couldn't tell me which of those two. So we have to make a convention about when we consider a register to contain a positive or negative number. Recall now that if the register is sufficiently large, a positive number will have the most significant (leftmost) bit equal to zero, while a negative number will have it equal to one (since for positive numbers we fill out the register with zeros, for negative numbers with ones). We therefore will adopt the convention that if the most significant bit is zero, the number is positive; if the most significant bit is one, the number is negative. As a consequence of this convention, we call the most significant bit the sign bit, and refer to the aforementioned process of filling a register with zeros (if sign bit = 0) or ones (if sign bit = 1) as sign extension.

A register of n bits can take on 2**n values, which we originally associated with the positive integers 0 through 2**(n-1). With our new convention, we are limited to n-1 bits (the high bit must be zero), and so to positive numbers 0 through 2**(n-1) - 1. In recompense for this loss, we obtain the ability to represent the negative numbers -0 through -(2**(n-1) - 1), so that all in all the register can still take on 2**n values. In doing integer arithmetic, one must always be careful to avoid exceeding the allowed range of positive or negative numbers; for example, in the addition in a six-bit register

```
24 + 24  =      011000
              + 011000
                ------
                110000 = -001111 = -15!
```

The sum is incorrect becuase we have exceeded the limit 2**5 - 1 = 31. Such an error is termed overflow because a carry spills over into the sign bit when the number exceeds the prescribed range.

In case you are confused by now about integer arithmetic, there is one additional point; we could either store a negative number in the computer as the complement of the positive number, or as the positive number (the magnitude) with a special bit set to indicate that it is a negative number. The latter is called a sign and magnitude representation, and the special bit is called the sign bit, since (like the sign bit discussed above) it is zero for positive numbers, one for negative numbers. For example, -27 would be

```
one's complement:      11100100
sign and magnitude:    1 0011011

sign and magnitude = 27
```

If we stored the numbers in one's complement, we could add them directly in the arithmetic unit; if we stored them in sign and magnitude, it would be necessary to convert the numbers to complement form before adding them. On the other hand, an advantage of storing numbers as sign and magnitude is that they are easier to read if you are looking directly at the 0's and 1's out of the machine (you don't have to complement the number in your head before figuring out what it is). But most people generally don't read numbers directly out of the machine's memory; they have the computer convert them back to decimal. As a result, the method simpler for the arithmetic section was selected for the 6600: storing numbers in one's complement.

We have finally finished addition, multiplication, and subtraction, and all the vagaries of one's complement. Division in binary is essentially the same process as in decimal, but is somewhat simplified because there are only two possibilities for each digit. The actual method for long division is identical to that for decimal; the only difference is that you have to remember to subtract in binary:

```
                        00000110 = 6
                    ----------
7/42 = 00000111 / 00101010
                      111
                    --------
                       111
                       111
```

You may notice that the only operations involved are shifting the divisor right one bit at a time and subtracting (or, alternately, adding the complement, 11111000). In the division process, the

divisor starts out shifted all the way to the left, and is repeatedly shifted right one binary place; each time the divisor is shifted, the arithmetic unit checks if the divisor is less than the dividend (as you do when performing the division by hand); if it is, the unit puts a one in the quotient and subtracts the divisor from the remaining dividend; in either case, it then continues shifting the divisor to the right.

Thus we have finally covered the four basic operations of binary arithmetic; if you don't have them quite straight yet, you will probably be heartened to know that you will probably never do a problem in binary arithmetic while you are programming (unless you feel a sudden urge to do so).

2.4 THE OCTAL NUMBER SYSTEM

The disadvantages of the binary system are obvious: it takes long enough just to write out a good-sized number (like 4100 (base 10) = 1000000000100), never mind doing a long multiplication or division. In designing the computer, we don't have to worry about this, but for ourselves we would like a more convenient notation. We could use decimal, of course, but this has one problem: if we want to examine the individual bits, we may have a hard time figuring out from the decimal value of a (binary) register whether a particular bit is a 0 or a 1 (quick, what is the thirteenth-from-low-order bit of 3,628,422,301?). The only way to insure that we can convert easily from the machine representation to our representation is to have one digit in our number system correspond to an integral number of bits. For example, in a machine using BCD for storing numbers, it is trivial to go from the machine representation to decimal, since one decimal digit corresponds to four bits. We can accomplish the same thing for a machine using the binary system by having a number system whose base is a power of two: base four (two bits to a digit in base four), base eight (three bits to a digit in base eight), base 16 (four bits to a digit in base 16), etc. Since we want to minimize the number of digits we have to write, bases eight and 16 are preferred to base four. You might think that, from this point of view, we should use base 32 or even 64. But for base 32, we need 32 different digits; knowing already 10 (i.e., 0 - 9), we would have to invent and memorize the values of 22 others, not a very pleasant thought at a time when we are trying to make work easier for ourselves. So the normal number

systems used are base eight -- octal -- and base 16 -- hexadecimal. For octal, we need eight digits: 0 - 7, logically enough; for hexadecimal, we take the ten decimal digits 0 - 9 and add six letters A - F to make 16.

On the 6600, the basic bit groupings are 6, 12, 15 and 30 bits -- all multiples of three -- so the octal system, which associates one digit with each three bits, was selected. To convert binary to octal and vice versa, you first have to learn the binary equivalents of the octal digits:

Octal		Binary
0	=	000
1	=	001
2	=	010
3	=	011
4	=	100
5	=	101
6	=	110
7	=	111

Then, to convert binary to octal, simply break up into groups of three and convert group by group; for example,

```
1  101  010 (base 2) =  1   5   2 (base 8)
```

In case the basis for this procedure isn't obvious, study the following:

```
= 1*2**6 + 1*2**5 + 0*2**4 + 1*2**3 + 0*2**2 + 1*2**1 + 0*2**0

=1*2**6 + (1*2**2 + 0*2**1 + 1)*2**3 + (0*2**2 + 1*2**1 + 0)*2**0

= 1*8**2 + (5)*8**1 + (2)*8**0

= 1   5   2  (base 8)
```

Converting from octal to binary is just as simple:

```
          3     7     4     2   (base 8)

      = 011   111   100   010  (base 2)
```

With a little practice this should become as automatic as breathing.

Converting from octal to decimal is a procedure analogous to converting binary to decimal.

```
  3        7        4        2    (base 8)

= 3*8**3 + 7*8**2 + 4*8**1 + 2

= 3*512  + 7*64   + 4*8    + 2

=2018
```

If you remember the powers of two (useful information in any case when you are programming) this is simple enough. But, as we shall see in the next section, there are techniques which absolve you from having to remember 2**39 if you want to convert 37421654513007(base8).

Doing arithmetic in octal is similar to decimal, except that you have to cut two fingers off before counting. When you add, you have to remember to propogate a carry and start counting again at eight, when subtracting, to borrow eight, not ten:

```
   70320
+ 165432
---------
  255752

   42731
 - 25616
 --------
   15113
```

Multiplying in octal can be messy unless you memorize the multiplication table, though if that gives you a hard time, you can multiply each pair of octal digits in decimal and then convert to octal (e.g., 7(base 8)*7(base 8) = 49 (base 10) = 61(base 8)). Since that is, to say the least, a slow procedure, you will be happy to hear that you won't often have to do a multiplication like this:

```
   1542
 * 317
 -----
  13656
  1542
 5046
-------
 536076
```

Once you have octal multiplication and subtraction down pat, octal division is simple -- the same procedure as long division in decimal:

```
                1542
              --------
317 / 536076
       317
       ----
       2170
       2013
       -----
        1557
        1474
        -----
          636
          636
          ---
          000
```

2.5 BASE CONVERSION ALGORITHMS

A base conversion algorithm is a technique for converting numbers in one base to numbers in another. We have already discussed a few; for example, from binary to decimal:

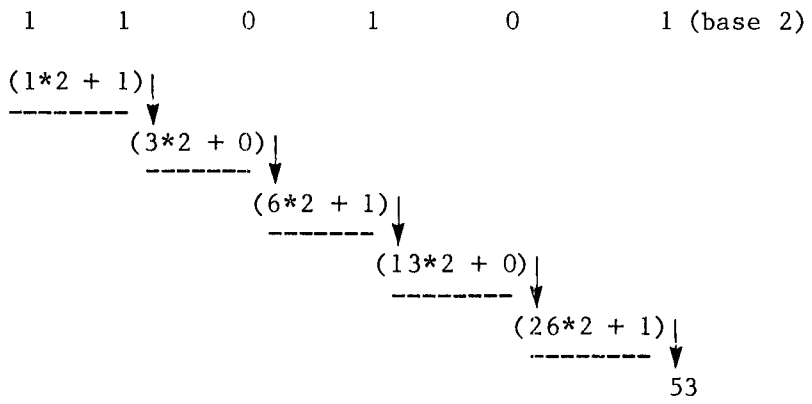
```
110101 (base 2)

= 1*2**5 + 1*2**4 + 0*2**3 + 1*2**2 + 0*2**1 + 1*2**0

= 1*32 + 1* 16 + 0*8 + 1*4 + 0*2 + 1

= 53 (base 10)
```

Now watch carefully

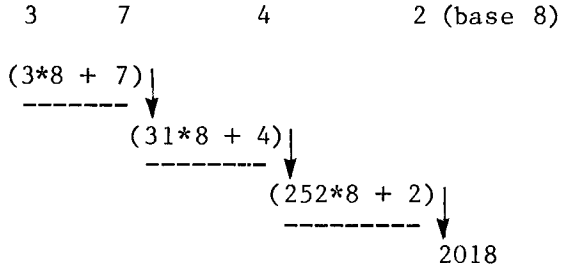
```
110101 (base 2)

= 1*2**5 + 1*2**4 +  0*2**3 + 1*2**2 + 0*2**1 + 1*2**0

= (1*2**4 + 1* 2**3 + 0*2**2 + 1*2**1 + 0 )*2 + 1
```

```
= ((1*2**3 + 1*2**2 + 0*2**1 + 1)*2 + 0)*2 + 1

= (((1*2**2 + 1*2**1 + 0)*2 + 1)*2 +0) *2 + 1

= ((((1*2 + 1)*2 + 0)*2 + 1)*2 + 0)*2 + 1
```

Thus converting binary to decimal can be reduced to multiplying by two and adding the next bit to the right, repeating this process till we run out of bits; this may be diagrammed:

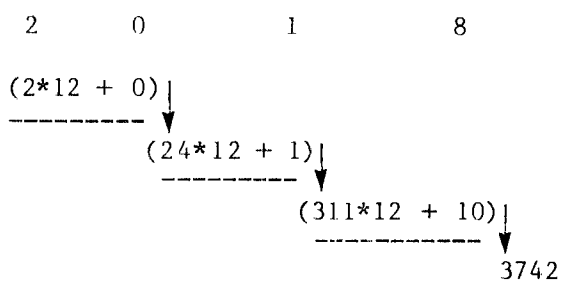


Converting from octal to decimal is the same, except we multiply by eight:



Thus, we have an algorithm for converting from binary and octal to decimal that doesn't require us to memorize tables of powers of two.

To go the other way -- to convert decimal to octal -- we can simply turn the tables and multiply by 12 (base 8),(= 10 (base 10)) and do all the arithmetic in octal:



But -- since most people prefer not to multiply in octal -- there is another conversion technique we can use, involving repeated division. Consider a three digit number, with digits d(3)d(2)d(1). Its value, in decimal, is

d(3)*8**2 + d(2)*8**1 + d(1)

If we now divide in decimal by 8, we get quotient d(3) * 8**1 + d(2), remainder d(1) (since d(1) is less than eight); if we divide the quotient again by eight, we get d(3), remainder d(2). Thus, by repeated division by 8, we will get the octal digits as a series of remainders. To convert 2018 to octal,

2018/8 = 252, remainder 2

252/8 = 31, remainder 4

31/8 = 3, remainder 7

3/8 = 0, remainder 3

so again, (reading the remainders from bottom to top), 2018 (base 10) = 3742 (base 8). For binary, we use the same procedure except we divide by two. To convert 53 to binary,

53/2 = 26, remainder 1

26/2 = 13, remainder 0

13/2 = 6, remainder 1

6/2 = 3, remainder 0

3/2 = 1, remainder 1

1/2 = 0, remainder 1

so (reading up) 53 (base 10) = 110101 (base 2). We will have more to say about conversion algorithms quite a bit later, when we will be writing programs to do the conversions.

2.6 FLOATING POINT NUMBERS

So far, in discussing the representation of numbers in the computer, we have considered only integers. But, as you know from FORTRAN, we have two basic modes for representing numbers: integer and real. Real numbers are needed for numbers much larger than or smaller than one, and numbers with fractional parts. In writing very large or very small numbers in FORTRAN we use scientific notation, E format, such as 3.E+30 or 7.1E-32 in order to avoid such space-consuming monstrosities as 3000 000 000 000 000 000 000 000 000. or 0.000 000 000 000 000 000 000 000 000 000 071. In storing the numbers in the machine, we use a similar technique: we convert the number to a reasonable-sized integer multiplied by a power of two; for example, the binary xumber 101 000 000 000 000 000 000 000 000 000. we can write as 101*2**11011.

Now what about numbers smaller than one? Just as we represent such numbers in decimal with decimal fractions, we can represent them in binary by binary fractions; as places to the right of a decimal point represent 1/10, 1/100, 1/1000, etc., the places to the right of a binary point (which looks just like a decimal point) represent 1/2, 1/4, 1/8, and so forth. Thus

```
  0.001 (base 2)
= 0*1/2 + 0*1/4 + 1*1/8
= 1/8
= 0.125 (base 10)
```

and

```
  1.0101 (base 2)
= 1 + 0*1/2 + 1*1/4 + 0*1/8 + 1*1/16
= 1 + 5/16
= 1.3125 (base 10)
```

Without describing the conversion algorithm--which is quite similar to those previously discussed--it is clear that given any decimal fraction we can convert to binary. Once this is done, it is trivial to get into our form of integer times exponent:

```
  1.3125 (base 10)
= 1.0101 (base 2)
= 10101*2**(-100)
```

When we store the number, of course, we will just store the integer coefficient and exponent, since the base of the exponent (2) is understood. In the 6600, memory words and X registers have 60 bits, to be allocated between the coefficient and exponent. The low 48 bits are used for the coefficient, the next 11 for the exponent, and the high-order bit has the sign of the number (i.e., of the coefficient):

SIGN	EXPONENT	COEFFICIENT
(1)	(11)	(48)

The 11 exponent bits contain a biased exponent; that is, 2000 (base 8) is added to the true exponent before it is put into the floating-point (real) number. Thus, a true exponent of 0 appears as 2000 (base 8), and a true exponent of 1777 (base 8) becomes 3777 (base 8), the largest number that will fit in all 11 bits.

However, an exponent of -1 becomes 1776 (base 8), and not 1777 (base 8) (the exponent 1777 (base 8) has a special significance, which will be discussed later). In other words, when we add the bias, 2000 (base 8), to -1 =3776 (base 8) (11-bit one's complement), we ignore the carry out and keep only the low 11 bits.

```
2000 (base 8) = 1 0 0 0 0 0 0 0 0 0 0
3776 (base 8) = 1 1 1 1 1 1 1 1 1 1 0
                ---------------------
              1 0 1 1 1 1 1 1 1 1 1 0

               keep 11 bits

              = 1  7  7  6  (base 8)
```

In this way, the smallest exponent possible, -1777 (base 8) becomes (+2000 (base 8) + 2000 (base 8)) 0000 (base 8).

At this point you should be able to figure out that

0.5 = 1776 00000000000000 01 (base 8)

16. = 2000 00000000000000 20 (base 8)

or should

16. = 2004 00000000000000 01 (base 8)

or perhaps

16. = 1724 40000000000000 00 (base 8) ?

Since all these representations have the same value (doubling the coefficient and subtracting one from the exponent leaves the values unchanged), which should we choose? The answer in general requires some understanding of the floating point arithmetic unit of the central procesor; however, in the cases where it is not possible to represent the number exacly in floating point format, the preferred form is clear. For example, 1/3 in binary is a repeating fraction,

0.010101010101010 ... (base 2)

= 0. 2 5 2 5 2 ... (base 8)

just as it is in decimal (0.333...), so we cannot represent it exactly using any combination of coefficient and exponent. However, the more digits we include in the coefficient, the more accurate will be our floating point number:

1774 0000000000000002 (base 8)

is only very approximate (1/4 = 0.25)

1771 0000000000000025 (base 8)

is better (= 0.328), and

1716 5252525252525252 (base 8)

is accurate to one part in 2**48 (since the coefficient has 48 bits); the pattern has changed to a leading 5 because we shifted 46 bits (not a multiple of 3) in order to get the most significant bit into the high order position of the coefficient. This form, in which the coefficient is shifted so that the most

significant bit occupies bit 47, the high order bit of the coefficient, is the standard form for floating point numbers in the 6600. The process of converting a number to this form is called normalization, and a number in this form is said to be normalized. When normalized, our earlier examples, 0.5 and 16, become

```
0.5 = 1717    4000000000000 (base 8)
16. = 1724    4000000000000 (base 8)
```

Our final problem with floating point numbers is how to represent negative numbers. The method used is very simple: the representation of a negative number is the one's complement of the positive number of the same magnitude. Everything is complemented, exponent and coefficient:

```
-0.5 = 6060   377777777777777 (base 8)
-16  = 6053   377777777777777 (base 8)
```

Note that, since we made sure bit 59 (the high order bit) is zero for positive numbers, it will be one for negative numbers; thus we have the same positive/negative rule we had for integers. Also, since the "normalized bit" (bit 47) is one for normalized positive numbers it will be zero for normalized negative numbers.

CHAPTER 3

CENTRAL PROCESSOR INSTRUCTION SET

3.1 A SUMMARY OF CENTRAL PROCESSOR HARDWARE

In the past two chapters we have gradually unfolded the basic facts concerning the central processor. Now, before we go on to the instruction set of the central processor, let us review and complete the information necessary for an understanding of these instructions.

The standard central memory for the 6600 has 2**17 = 131,072 words; 2**15 = 32,768 and 2**16 = 65,536 word memories are optionally available if you don't happen to have the extra half million dollars or so on hand for the larger memory (the 7600 comes only with a 32 or 65 thousand-word central memory). Having the number of words be a power of two simplifies central memory organization somewhat. As mentioned earlier, each word has 60 bits, relatively large as machine word sizes go. This size permits a floating point number with about 15 decimal places accuracy, sufficient for virtually all applications. A large word also permits several instructions to be put into one word (as we shall soon see), so that the number of memory accesses required to get out instructions is reduced. Finally, 60 is a multiple of 2, 3, 4, 5, and 6, so that several different subdivisions of the word may be conveniently made.

The central processor has three types of registers: A, B, and X registers; there are eight of each type, identified as A0 through A7, B0 through B7, X0 through X7. The X registers are used to hold the operands and results in arithmetic operations; since words are loaded from memory into X registers, and stored into memory from X registers, X registers also have 60 bits.

The A registers are special registers involved in the loading and storing of operands. When an address is put into one of A1-A5, the word at that address is loaded into the associated X register (X1-X5); when an address is put into A6 or A7, the contents of X6 or X7 are stored at that location. The A0 and X0 registers have

no connection with memory, and may be used for holding intermediate results.

The A registers are 18 bits, and thus can accomodate 2**18 addresses -- a 2**18 word memory had been envisioned for 6000 series computers. Eighteen is also a convenient size, being a multiple of 2, 3, and 6; eighteen-bit quantities appear in many of the central processor instructions.

The B registers are used for holding small numbers, such as loop indices. In order that the arithmetic hardware can be used for B and A registers, B registers also have 18 bits. Because the constant zero is needed so often, register B0 is permanently set to zero.

3.2 THE TYPES OF CENTRAL PROCESSOR INSTRUCTIONS

Every central processor program is a series of CP (central processor) instructions, each of which accomplishes a small step in the overall calculation. In selecting the instructions to be recognized by the CP, we seek a set which enables us to write efficient programs for a large variety of problems -- programs which run quickly and do not take up too much space. At the same time, we must resist the urge to include every useful instruction we can think of, lest we (after installing the hardware to recognize all these instructions) re-enter the billion-dollar computer market. In other words, we have to find a relatively small set of versatile instructions with which, in combination, we can perform all the different types of operations efficiently. So, as a first step, let us see what basic types of instructions we would want to have.

First we need the arithmetic operations: addition, subtraction, multiplication, and division. Although we could simulate subtraction, multiplication, and division with short programs using only addition and one's complementing, this would be very slow; hence, for high speed scientific computing, hardware subtraction, multiplication, and division are essential. The computer must be able to do all four operations on both integer and floating point operands. In all these instructions, the two operands will be taken from X registers and the result put in an X register.

In addition to the arithmetic operations, we will need the Boolean or logical operations: and, or, not. These are required for such purposes as masking and evaluating logical expressions (the .AND., .OR., .NOT., operations in FORTRAN). These instructions will also take operands from and put the result in an X register. In conjunction with the Boolean instructions, we will need shift instructions (for which there is no explicit analogue in FORTRAN), which enable the programmer to shift the position of a bit pattern in a register; for example, to change

63000 00000 00000 00000

to

00000 00000 63000 00000

and to

00000 00000 00000 00063

To get data to and from storage we will need instructions that set the A registers. Since the A registers only have 18 bits, these operations need only perform 18 bit arithmtetic, while the X register operations above perform 60 bit arithmetic. A similar group of instructions must be provided to set the B registers; here, too only 18 bit arithmetic is necessary. To make things "complete," a third set of 18 bit arithmetic instructions has been provided which leaves the result in an X register. This is necessary, for example, to get the contents of B and A registers into X registers when the contents are to be stored or are an operand in a multiplication or division (multiplication and division, after all, are only done among X registers).

Finally, we need branch instructions to transfer control within the program. Normally when the CP has finished executing instructions in one word, it takes the next instructions from the next location in memory; in other words, the instruction location counter is regularly incremented by one. When a branch is made, the CP instead takes its next instruction from a location specified in the branch instruction. There are both unconditional branches (corresponding to GO TO statements) and conditional branches (corresponding to IF statements).

3.3 INSTRUCTION FORMATS

Each computer instruction is nothing more than a set of bits occupying a word or part of a word. Particular groups of bits in the instruction give specific information about the operation to be performed. One group, known as the operation code or opcode specifies the particular operation to be carried out. Other groups specify the operands or the sources of operands (i.e., register numbers) and the destination of the result (another register number).

In 6600 instructions, the operation code is the leftmost 6 bits of an instruction. The six bits allow for 64 different CP instructions (well, actually 71, because one opcode specifies one of a group of eight similar instructions). Though 71 instructions isn't very many (most very large computers have several hundred), the 6600 instructions are sufficiently versatile and powerful and so fast that the 6600 can run circles around many other large computers with many more instructions. Often an entire program loop on a 6600 will be faster than a single instruction on another machine which performs the same calculation!

Consider a normal arithmetic instruction: how many bits are necessary to specify the entire instruction? We need the opcode (6 bits) plus the two operand registers and the result register. Since all registers involved in a arithmetic instruction are X registers, we need only give the register number, 0 - 7; hence each register specification requires 3 bits. Thus the instruction requires a total of 6 bits (opcode) + 6 bits (two operand registers) + 3 bits (result register) = 15 bits. Most of our instructions will be 15 bits; this size, conveniently enough, allows us to get 4 instructions in a single word.

If you think about it a while, however, you will see that inter-register instructions (those in which both operands and result are registers) aren't enough. Assume that at the beginning of your program you do not know what is in any of the registers; as the first step in your program, you would like to load a variable I, which you know to be at location 2167 (base 8) of your program. In other words, you want to set an A register to 2167. But now you realize that the only way to get the number 2167 is to load it out of memory (it's not in any of the registers), so you wisely put the number 2167 in your program, in a variable K, location 2166(base8). Now your problem is how to get 2166,...You suddenly realize that you're stuck, and can't even begin your problem (well, actually you could always keep the next needed addresses in loction 0, and set A register to B0 = 0; such a

programming restriction, however, verges on the absurd).

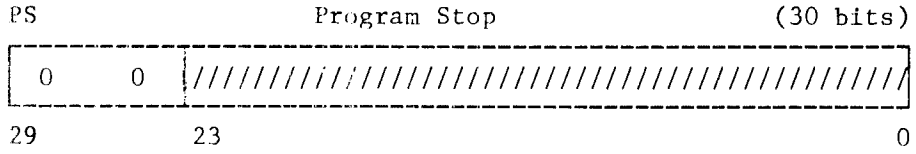
As a result, we have some instructions which include an 18 bit constant to be used as one operand in place of a register. In addition to the case cited, having a constant in the instruction is particularly useful in branch instructions, since the location to which we branch is generally a constant, and not the result of an arithmetic calculation. In a 30 bit instruction which sets a register (i.e., not a branch), the type of result (A, B, or X) and the type of register from which the second operand is taken are both specified by the opcode, along with the operation to be performed. Hence the total length of such an instruction is, conveniently, 6 bits (opcode) + 3 bits (result register) + 3 bits (one operand register) + 18 bits (constant for second operand) = 30 bits, so we can put two such "long instructions" in one word, or one long instruction and two "short instructions" (30 + 15 + 15 = 60). If this seems a bit confusing now, don't worry; it should become clearer after we study some specific cases later on.

Having 64 opcodes permits a convenient division into 8 sets of 8 instructions. In general the instructions within one set of 8 (00 - 07 (base 8), 10 (base 8) - 17 (base 8), etc.) are closely related. This organization of the opcodes will make it easier to remember the instruction set after we have discussed the individual instructions. And now, having waded through these preliminaries, we are ready to begin studying the central processor instructions.

3.4 BRANCH INSTRUCTIONS

We begin our study with branch instructions for several reasons, the least of which is the fact that they constitute the first set of eight opcodes, 00 - 07. The chief reason is that it is impossible to write any useful code without knowing at least two or three of the branch instructions. After all, if, in the middle of a FORTRAN program we want to execute some code we wrote in machine language, we have to branch to the machine-language-coded routine, and at the end of the routine branch back to the main program. As we shall soon see, this involves at least two different branch instructions.

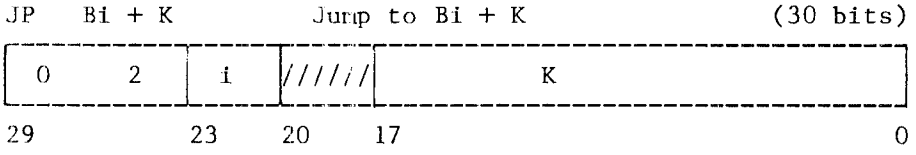
The simplest instruction of the group (not really a branch) is opcode 00, the program stop; the mnemonic we will use to designate the instruction when writing code is PS. It is a 30 bit instruction, but the remaining 24 bits are ignored; we diagram it thus



with the numbers below the figure designating the bits, counting the rightmost as bit 0. A program stop does exactly what it says: it stops the central processor at the current step in the program. After a program stop, the CP has no means of restarting itself; it can only be restarted by a PP. (On the 7600 the instruction is called "error exit" and causes the CP to start executing the monitor program.)

Quite a number of other computers, incidentally, use a zero opcode for program stop. The reason for this is quite simple: if, due to programming error, your program runs amuck, and, say, starts trying to execute your data, there is a good chance that it will run into a zero opcode (since positive integers less than 2**54 have the six high bits zero). Thus the program will come to a halt quickly, without wasting much time.

We digress now from numerical order to look at opcode 02, the (unconditional) jump instruction, mnemonic JP. This instruction transfers CP control to (branches to) a location specified by the low 24 bits of the 30 bit instruction. Bits 21 to 23 specify a B register, (these bits in an instruction are referred to a "i," and hence the specified register as "Bi"); bits 0 to 17 are treated as an 18 bit constant (these bits in an instruction are referred to as "K"). The 18 bit sum of this constant and the contents of the B register (i.e., Bi + K) is the location to which the branch is made. The instruction is diagrammed



Note that three bits of the instruction are ignored.

In most cases the address to which control is transferred is a constant which we can put in the K part of the instruction, so that we really don't want to add in any B register. In this case we let i = 0, so that the effective address is B0 + K = K (since B0 = 0). Using a B register, on the other hand, permits a simple computed GO TO statement. The FORTRAN statement

```
GO TO(100, 200, 300), I
```

may be coded, if the variable I is already in register B1, by

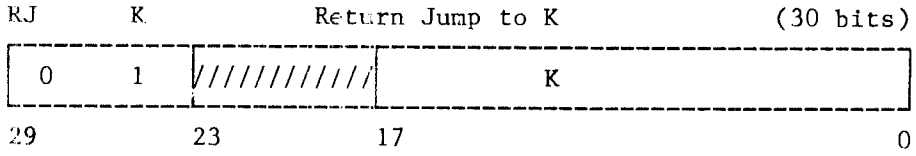
```
TABLE JP B1+TABLE
      JP S100
      JP S200
      JP S300
```

where S100, S200, and S300 are the address of the statements with numbers 100, 200, and 300, respectively. TABLE is the location of the first jump instruction; we designate this by writing the label TABLE to the left of the instruction. When a label appears to the right of the mnemonic (in the address field) without any B register, register B0 is implied; thus JP S200 means JP B0+S200. Each jump instruction occupies the upper 30 bits of a separate word. Thus the first jump instruction branches to the second, third, or fourth (at TABLE+1, TABLE+2, TABLE+3) depending on whether B1=I=1, 2, or 3; these latter jumps go in turn to statements 100, 200, and 300.

Having taken care of the GO TO and computed GO TO statements, let us now consider the implementation of the CALL statement. The magic of the CALL statement is that, when a RETURN occurs in the called routine, the program transfers control back to the next statement after the calling sequence. When the CALL is made, the computer must store somewhere the address to which the called program should return; otherwise, the called routine will be unable to tell from where it was called. Since the process of CALLing, i.e., of transferring control to a routine in such a way that the program can return to the original sequence, is so common, it has been implemented as a single hardware instruction.

This instruction is the return jump, opcode 01, mnemonic RJ. When a jump at a location HERE to a location THERE is executed, two things happen: (1) at location THERE is stored a jump to HERE + 1 and, (2) control transfers to location THERE + 1. When

the routine beginning at THERE + 1 is finished, a jump to THERE will get control back to HERE + 1. The return jump is a 30 bit instruction, like the jump instruction, but contains only an 18 bit constant for an address; no B register is specified (since a "computed call" is such a rare occurrence). The instruction is diagrammed:



With this information, we can give a specific numerical example: before execution of the RJ at location 325 to location 1732

location 325 (base 8)	01 00	001732	00000	00000
location 1732 (base 8)	00 00	000000	00000	00000

(the instructions of the called routine start at location 1733).

After execution of the RJ

location 325 (base 8)	01 00	001732	00000	00000
location 1732 (base 8)	04 00	000326	00000	00000

and the CP has begun executing the routine starting at 1733. You may notice that the instruction at 1732 is an 04 opcode, not an 02 jump; as we shall see directly, in this case it is also an unconditional jump instruction.

This completes our discussion of unconditional branch instructions, and we shall now proceed to study the remaining conditional branch instructions, opcodes 03 through 07. A conditional branch instruction is an instruction which causes a transfer of control only if a certain condition exists, such as an X register being zero or one B register being equal to another B register. If the condition is not met, the CP continues on to the next instruction, as usual. There are some branches which depend on the values of X registers, and some which depend on values of B registers; no branches involve tests of A registers, since A registers are generally not used for the results of arithmetic calculations.

There are a total of 8 different X register branches; because of the limited number of opcodes (64), all the branches have been put under one six-bit opcode, 03. The type of branch is specified by the next three bits, 21 to 23; thus for X register branches there is effectively a 9 bit operation code. The register number is given by the next three bits, 18 to 20 (termed the "j" part of the instruction), and the address to which the branch is made if the condition is met is given by the last 18 bits (K). We shall discuss the first four branches now; the other four are special tests for floating point numbers, and will be discussed later.

Opcode 030 is a zero jump (mnemonic ZR): control transfers to K if Xj is zero. Both plus zero, 000...000, and minus zero, 777...777 (base 8), meet this condition; any other value does not.

ZR Xj, K Jump to K if Xj = 0 (30 bits)

Opcode 031 is the exact opposite: a non-zero jump (mnemonic NZ). The branch to K occurs if Xj contains anything other than plus or minus zero

NZ Xj, K Jump to K if Xj ≠ 0 (30 bits)

The next two opcodes test the sign of an X register. Opcode 032 is a plus jump (mnemonic PL): a transfer is made to K if Xj is a positive. As you will recall, this has been defined to mean that the high order bit of the X register is 0.

PL Xj, K Jump to K if Xj positive (30 bits)

Finally, opcode 033 is a minus jump (mnemonic NG): the branch is made if Xj is negative, i.e., the high order bit of Xj is 1.

NG Xj, K Jump to K if Xj negative (30 bits)

By now you have probably recognized that the X register branches come in pairs of complementary (opposite) conditions. For example, if the condition for opcode 030 is met (=0) that for opcode 031 (does not equal 0) is not, and vice versa.

As an application of X register branches, consider the arithmetic IF statement

```
IF (NUMBER) 100, 200, 300
```

If NUMBER is in X1, this becomes simply

```
ZR    X1, S200
PL    X1, S300
JP    S100
```

where S100, S200, S300 are, as before, the locations corresponding to statement number 100, 200, and 300. Note that the order in which the tests are made is important; for example,

```
PL    X1, S300
ZR    X1, S200
JP    S100
```

will not work (do you see why?). Suppose X1 contains plus zero; we would then want to go to 200. The first code indeed does, but the second brings us instead to S300, since plus zero is positive. Thus one must check for zero before testing the number for sign. We may note, incidentally, that the instruction

```
NG    X1,S100
```

could be used instead of the unconditional jump; however, the unconditional jump is faster (since it does not have to make a test).

We now come to the last set of jump instructions, the B register branches. In contrast to the X register branches, the B register

branches depend on relational conditions: whether one B register is equal to another, greater than another, etc. B registers are typically used for indices, and these relational test instructions are particularly convenient for loop control (e.g., checking whether an index exceeds the upper limit, kept in another B register). Comparisons against B0 can always be used for tests like those in X register branches (zero, positive, etc.) B register branches occupy opcodes 04 to 07; the six bit opcode specifies the relation to be tested for.

Like all the other jump instructions, B register branches are 30 bit instructions. The numbers of the two registers to be compared are in bits 21 to 23 (i) and 18 to 20 (j). The address to which control is transferred if the branch is successful is in bits 0 to 17 (K).

Opcode 04 is the equal jump (mnemonic EQ); the branch is made if Bi = Bj.

EQ Bi, Bj, K Jump to K if Bi=Bj (30 bits)

A B register can be checked for a value of zero by setting one of the register numbers to zero:

ZR Bi, K Jump to K if Bi = 0 (30 bits)

Note that this instruction actually tests if Bi = B0; as a result, only plus zero (000000) and not minus zero (777777) causes a branch.

Opcode 05 is the complementary condition: not equal branch (mnemonic NE); the jump is performed if Bi does not equal Bj.

NE Bi, Bj, K Jump to K if Bi ≠ Bj (30 bits)

By setting j = 0 (or i = 0) this instruction can also be used for a non-zero test.

NZ Bi, K Jump to K if Bi ≠ 0 (30 bits)

0	5	i	0	K

Again, because a comparison is made against B0 (= plus zero), negative zero is treated as non-zero.

Opcode 06 is greater-than-or-equal jump (mnemonic GE); a transfer to K is made if Bi >= Bj.

GE Bi, Bj, K Jump to K if Bi ≥ Bj (30 bits)

0	6	i	j	K

In making the comparison, the B registers are treated as signed numbers. Thus a positive number, such as 012345 (base 8), is greater than a negative number, such as 765432 (base 8). With j = 0, the instruction becomes a positive test:

PL Bi, K Jump to K if Bi positive (30 bits)

0	6	i	0	K

Finally, opcode 07 is the complementary condition to 06, a less-than branch (mnemonic LT); the jump is made if Bi < Bj.

LT Bi, Bj, K Jump to K if Bi < Bj (30 bits)

0	7	i	j	K

By the rule that a negative number is less than a positive number, minus zero is considered less than plus zero (but don't ask how much less). By the same technique used above, this

instruction provides a negative B register test:

NG Bi, K Jump to K if Bi negative (30 bits)

After a first glance at the opcodes, you might remark that only four of the six basic relations have been provided -- less-than-or-equal and greater-than have been omitted. But a moment's reflection should make you realize that, by reversing the order of the registers, these other two conditions may be easily tested for. A LT Bi, Bj, K may also be considered a jump on Bj greater than Bi, and a GE Bi, Bj, K also used as a jump on Bj less-than-or-equal-to Bi. These relation-testing jumps thus constitute a powerful, versatile set of instructions. For example, to determine whether the contents of a B register is strictly positive (i.e., greater than zero), we require only one instruction,

```
LT    B0, Bi, K
```

Since we have not yet discussed the instructions for setting B registers, no real examples of the use of these instructions are possible at the moment. One item of note should, however, be mentioned: the instruction EQ B0,B0,K. This clearly is an unconditional jump to K, just like JP K (and just EQ Bi,Bi,K is, i=1,...,7). As you may recall, the EQ B0,B0,K instruction came up earlier in connection with the return jump instruction; it is the 0400 K which is stored by the RJ instruction. This EQ B0,B0 is also used by the FORTRAN compiler to implement the GO TO statement. Why? Because EQ B0,B0 under certain circumstances is faster than JP, and in the remaining cases takes the same time to execute (the reasons for this are too involved to explain at present). As a result, it is a common programming practice to use EQ B0,B0 in place of the JP when an unindexed jump is required. In accordance with the general rule that if no register is mentioned B0 is implied, we may write this unconditional branch as EQ K.

3.5 WRITING ASSEMBLY LANGUAGE CODE

Before we continue discussing the CP instructions, some practical information on how to write code at the machine-instruction level is in order. In the same way that a compiler is used to translate FORTRAN, a program called an assembler is used to translate the mnemonics we use in writing our program into the binary numbers accepted by the machine. Several assemblers have been written for the 6600; some are very simple, and can process little except the instruction mnemonics, while others are highly sophisticated -- almost as complicated as a compiler -- and offer the user great programming flexibility.

The original (non-operating) operating system for the 6600, SIPROS, (SImultaneous PRocessing Operating System), included a fancy assembler called ASCENT (Assembly System Central Processor). SIPROS ASCENT permitted the programmer to intermix FORTRAN statements and assembly language code (machine mnemonics), within the same program. Thus, if in the middle of his FORTRAN program the user had some processing which he wanted to code in assembly language, he could simply put his ASCENT code right there. This made things more complicated for the assembler-compiler writer, of course, because the combined program had to be able to recognize both FORTRAN and ASCENT instructions.

With the first operating operating system, the Chippewa Operating System, came several assembly programs. The Chippewa FORTRAN compiler alone accepted two different assembly languages (the same instruction had altogether different mnemonics in the two languages). The Chippewa assembly language (CLASS), the original 6600 assembly language, is very little used any longer. The other language, ASCENTF, was a subset of ASCENT, with the same mnemonics as ASCENT, though not all of its features. In addition, in the Chippewa operating system, a separate assembly program, similar to SIPROS ASCENT, was available under the name ASCENT.

More recent operating systems -- SCOPE, KRONOS, and NOS -- include the most sophisticated 6600 assembler, COMPASS. This assembler is considerably more versatile than even SIPROS ASCENT. It is not possible in COMPASS, however, to write a few lines of FORTRAN, then some assembly language, then some FORTRAN. The main program and each of the subprograms (subroutines or function subprograms) must be written entirely in one language. Thus one can mix a main program and FORTRAN subroutines with assembly language subroutines, but may not mix FORTRAN and assembly language within a subroutine. This, incidentally, is a general

restriction in assembler-compiler systems; the line-by-line mixing facility which SIPROS ASCENT would have had (if it had ever worked) is not generally available.

In this chapter we will try to cover the most basic information required for writing COMPASS subroutines. The next chapter will introduce some of the fancier features of COMPASS.

A line of source code representing an instruction consists of four fields: a location field, an opcode field, an address field, and a comments field. A field is a portion of a line given over to a particular purpose. For example, in a FORTRAN card the first five columns may be described as the statement number field. The location or label field permits a name to be associated with the location of the instruction (recall the example of the computed GO TO). The opcode and address fields specify the instruction; the opcode field has the mnemonic part, and the address field the remainder of the instruction. For example:

location field	opcode field	address field
HERE	EQ	B1,B3,THERE

Note that the address field will generate more than just an address (label). A blank indicates the end of each of these three fields; consequently, these fields must be separated by at least one blank and may contain no embedded blanks. The comments field, the last field on a card, allows the programmer to include informative remarks concerning the instruction; the comments field must be separated from the address field by at least one blank, and may contain blanks.

The label (symbol in the location field) must start in either column 1 or column 2. Symbols in COMPASS may have 1 to 8 characters. Although some special characters are allowed in symbols, we shall restrict ourselves to names beginning with a letter and containing only letters and digits (just like names in FORTRAN). The opcode may start in column 3 or after; general practice is to start in column 11. Although the address field may begin anywhere after the opcode (up to column 29), and the comments field anywhere after the address, easy readability of the written code dictates that standard starting columns be selected for these two fields. Representative conventions are: columns 18 (address field) and 30 (comments field), or columns 18 and 36. Thus a sample card would appear:

```
column:  2      11      18          30
         HERE   JP      B2+THERE    BRANCH TO TABLE
```

Any field except the opcode field may be omitted. If the location field is blank, no name is associated with the location of the instruction. Also, if there is a label in that field, the instruction automatically gets put at the beginning of a word (leftmost part); otherwise, the instruction would go into the next available space. As we have seen, this may mean that the instruction is put in the middle or end (right part) of a word. Thus the sequence of instructions

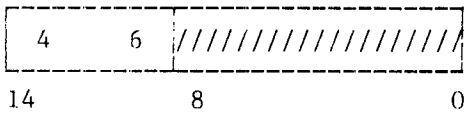
```
LABEL   EQ      B1,B2,TEST1
MABEL   ZR      X4,TEST2
```

will occupy two words (with the instructions left justified in those words), while

```
LABEL   EQ      B1,B2,TEST1
        ZR      X4,TEST2
```

puts both 30 bit instruction into one word. In the former case the question naturally arises, what is in the low 30 bits of the two words? We can't just leave them zero, because then (supposing B1 does not equal B2) the computer encounters a 00 opcode, a program stop. As a result, the computer has a special instruction used for "padding out" words of instructions like these. It is opcode 46, and is called a no operation instruction (more commonly, a "no-op"):

NO No operation (15 bits)



When the computer encounters a no-op instruction, the computer does absolutely nothing except pass on to the next instruction; hence the instruction is also called a "pass instruction." In our example, we have to pad out 30 bits, so we put two no-ops in a row: 4600046000. Thus, if TEST1 is location 242, and TEST2 location 272, the two words are

```
(LABEL)    0412000242  4600046000
(MABEL)    0304000272  4600046000
```

The effect of a label of putting the instruction on the card into the upper (leftmost) portion of a new word is called "forcing upper." One additional rule you should be aware of is that forcing upper is automatic after an unconditional branch instruction. This is reasonable enough, since in all probability you will never want to put another instruction in a word after an unconditional branch (it would never be executed). This explains why, in our computed GO TO table, each jump appears in a separate word: after each jump, forcing upper is automatic, so the next jump is put in a new word. Similarly, a RJ is unconditional, so

```
ME      RJ      EWE
        PL      XO, RAM
```

takes up two words. Now how about

```
THINK   EQ      HARD
        LT      B7,B6,RELAX         ?
```

You will be happy to hear that COMPASS realizes that this is really an unconditional branch, so the LT instruction is forced upper.

Comment cards are indicated by an asterisk (*) in column 1 (just as they are indicated by a C in column 1 in FORTRAN). Comments on an otherwise blank card (with no * in column 1) must begin in or after column 30, or an assembly error will result.

Like all other subprograms, assembly subroutines require header and end cards. The end card is simple enough: just the word END, starting in column 11. In COMPASS the form of the header card is

```
IDENT  SUBNAME
```

with IDENT beginning in column 11, followed by the subprogram name (SUBNAME in the example above.)

3.6 SUBPROGRAM LINKAGE AND PARAMETER TRANSMISSION

With what we have learned so far, we are ready to write our first assembly language subroutine. Since we don't know many instructions yet, our subroutine will be somewhat less than awe-inspiring; in fact, it will do absolutely nothing, just like the FORTRAN

```
      SUBROUTINE     DUMB
      RETURN
      END
```

The first line of our COMPASS routine will be

```
      IDENT     DUMB
```

Now, recall that we are going to enter this subroutine by means of a return jump instruction. The first thing the RJ is going to do is store a jump back to the calling routine, so we must set aside a word into which this will be stored. This word is called the entry line.

We could set aside a word by writing an unconditional jump instruction (so the next word is forced upper). There is, however, a special instruction which tells the assembler to set aside several words: BSS. A "BSS" allocates as many words as specified by the address field, for example,

```
      BSS       13
```

reserves 13 words of storage. You should be warned that words set aside by a BSS are not set to zero; the contents of these words is unpredictable (or, as we would say if we were writing a computer science text, undefined). BSS stands for "block started by symbol," so-called because if there is a symbol (label) in the location field, it is associated with the first word of the block. The BSS and similar instructions which do not generate machine instructions are called pseudo-instructions. Thus the next line of our subroutine is

```
DUMB  BSS       1
```

The instructions of the subroutine begin in the next word after the entry line. In our example, we have only a RETURN statement, i.e., a jump to the entry line:

```
      EQ        DUMB
```

Tack on an END card and put it all together and we're done -- well, almost. One small problem remains: ordinarily, symbols in one COMPASS subroutine, like variables and statement numbers in a FORTRAN subroutine, cannot be referred to in another routine. Since we want to be able to refer to the label DUMB in our calling routine, we have to declare it an entry point:

```
          ENTRY     DUMB
```

(Note: symbols which are declared to be entry points may have only 7 characters, not 8.) Then, if in the calling program we declare the symbol DUMB to be "external":

```
          EXT       DUMB
```

we are allowed to refer to it by name:

```
          RJ        DUMB
```

With this problem well in hand, we are ready to put together our first subroutine:

```
          IDENT     DUMB
          ENTRY     DUMB
DUMB      BSS       1
          EQ        DUMB
          END
```

Before we start discussing parameters, I ought to tell you what you can do with your DUMB subroutine.

If you wanted to assemble the routine separately, you could invoke the assembler directly (with a COMPASS control card). All the COMPASS routines we will write, however, are subroutines to be invoked by FORTRAN routines, so we will make use of a valuable feature of the 6600 series FORTRAN compilers: anywhere you can put a FORTRAN routine in your source deck, you can also put a COMPASS subroutine. The compiler automatically transfers control to COMPASS when it notices a card with IDENT beginning in column 11, and COMPASS returns control to the compiler when it encounters a FORTRAN header card.

Thus, a possible complete (and completely useless) source program is:

```
      PROGRAM DODO
      CALL DUMB
      STOP
      END
```

```
        IDENT   DUMB
        ENTRY   DUMB
DUMB    BSS     1
        EQ      DUMB
        END
```

Note that the FORTRAN statement

```
CALL DUMB
```

is effectively translated into

```
EXT     DUMB
RJ      DUMB
```

since DUMB is an entry point in the subroutine and hence must be declared external to the main program.

Now that you know what to do with your DUMB subroutine, we can proceed to the second topic of this section: the transmission of subroutine parameters. (The term parameter and argument are used interchangeably.)

When writing assembly code, of course, it won't do just to put the dummy parameter name in an instruction, as you would when writing FORTRAN, and have the assembler magically fetch the parameter for you. You are writing the code instruction-by-instruction, and so have to know, in real hardware terms, where the parameters are.

In the first place, FORTRAN always transmits the address of the parameter, and not the parameter itself. For example, if you write

```
CALL WXYZ(K)
```

it will transmit the address of K to subroutine WXYZ; likewise, if the statement is

```
CALL WXYZ(2)
```

it will transmit the address of a location containing the number 2. To try to transmit instead the value itself would cause quite a few difficulties; for example, if A were a 100 by 100 matrix, and subroutine OREZ set one element to zero, and the call were

```
CALL OREZ(A)
```

the FORTRAN routine would have to transmit all 10,000 elements, not knowing which would be changed. Or, consider transmitting a function name as a parameter:

```
EXTERNAL TAN
CALL SUN(TAN)
```

one can't very well transmit to SUN the value of TAN; it doesn't have one by itself. Hence, the only thing to do is to transmit the address of the TAN routine.

The calling sequence generated by the FORTRAN Extended compiler stores the addresses of the parameters in a series of consecutive words of memory, followed by a word of zeros. The compiler then sets A1 to the address of the first of these words. Recall from section 3.1 that setting A1 will cause the contents of that memory location to be put in X1; thus, in this case X1 will contain the address of the first parameter. For example, if we had a routine with two parameters, whose addresses were 2110 and 2120, the situation might look like this:

A1	4601	location	4601	2110
X1	2110	"	4602	2120
		"	4603	0000

It is usual that the calling sequences generated by compilers for different languages (such as FORTRAN, COBOL, and PL/1) are incompatible, so that a routine written in one language cannot directly call one written in another language. Control Data, however, always one step ahead of the other manufacturers, has arranged things so that the calling sequences generated by the two FORTRAN compilers -- RUN and FORTRAN Extended -- are also incompatible. Consequently, assembly language routines must be coded differently for use with the two compilers. RUN was the original CDC FORTRAN compiler, but it is no longer used very much, so all the examples presented in this volume will assume a FORTRAN Extended calling sequence. The RUN calling sequence (and some additional notes about parameter passing in FORTRAN Extended) are given in the section "More About Passing Parameters" toward the end of this volume.

One final item regarding argument transmission needs to be covered: returning the value of a function. Since we know that the result of a function is always a number, and not an array or

the address of a function, we needn't go through the trouble of storing the result at an address passed to the function. Instead, the result is returned directly in a register, X6. The calling sequence, after the RJ to the function, simply takes the result out of X6.

The assembler does not differentiate between subroutines and functions. It is up to you to remember, if you referenced the routine as a function (in a replacement statement, e.g., Y = TINY(X)), to write the routine as a function -- return the result in X6. Similarly, if you referenced the routine as a subroutine in a CALL statement, you have to write it as a subroutine -- return all results via parameters. That the differentiation between subroutines and functions is entirely your concern is emphasized by the fact that in COMPASS there is only one header card for both types of routines:

```
IDENT    TINY
```

In the next section, after introducing some more instructions, we will consider a simple assembly-language function. If this discussion of parameter transmission has gotten you somewhat confused, the examples in the next few sections should help to clarify matters.

3.7 SET INSTRUCTIONS

Together with the branch instructions, the set of instructions known as the set instructions (or, technically, "increment unit instructions") constitute the two most important groups of instructions. These two groups are both necessary and sufficient for writing useful assembly language routines.

The set instructions perform 18 bit (one's complement) addition and subtraction. They provide the only means of setting A registers, and, with the exception of a few instructions for manipulating floating point numbers, the only means of setting B registers. They are also required for putting an A or B register in an X register.

There are 24 set instructions, opcodes 50(base8) to 77(base8). They are organized as three groups of eight instructions (opcodes 50-57, 60-67, 70-77); these three groups are set-A-register, set-

B-register, and set-X-register instructions, respectively. Except for the type of result register, these three sets are entirely identical. We shall begin when with a consideration of the set-X-register group; when that is done, the set-A and set-B groups will require little further explanation.

The increment unit instructions leaving the result in an X-register (i.e., set X), opcodes 70-77, can be further divided into two groups, the long instructions, opcodes 70-72, and the short instructions, opcodes 73-77. The long instructions, as you may recall, are 30 bit instructions in which one of the two operands is an 18 bit constant appearing in the instruction. The three instructions permit calculation of A register + constant, B register + constant, and X register + constant.

SXi Aj + K Set Xi to Aj + K (30 bits)

SXi Bj+K Set Xi to Bj + K (30 bits)

SXi Xj+K Set Xi to Xj + K (30 bits)

Since you (hopefully) remember that X registers are 60 bit registers, you may be wondering how we can do 18 bit arithmetic with them. When an X register is an operand in a set instruction, only the low 18 bits are used -- the remainder are ignored. When the result of a set instruction is placed in an X register, the 18 bit result is put in the low 18 bits and the sign is extended. That is, if the result is positive, the high 42 bits become zero, while if it is negative the high 42 bits are set to one. In this way, the correct positive and negative numbers will be stored in X registers. So, for example, if A1 = 003215(base8), a

```
SX1     A1+2
```

will set X1 to 00000 00000 00000 03217 (base 8). If B4 = -4 (777773 (base 8)) then

```
SX7     B4+2
```

will set X7 to -2 (77777 77777 77777 77775 (base 8)). Note that in the last case the set instruction first creates the result 777 775, and the negative sign is then extended.

Results other than those desired may occur if the magnitude of an X register used as an operand exceeds 377777 (base 8). For example, if X0 = 1475231 (base 8), the instruction

```
SX0     X0+6
```

will set X0 to 77777 77777 77774 75237 (base 8) (surprise!). First X0 is truncated to 18 bits, 475231 (base 8), then 6 is added to give 475237 (base 8). This is a negative 18 bit number (bit 17 is a 1) so the negative sign is extended, setting the high 42 bits to 1.

The assembler allows instruction of the form SXi Aj-K, SXi Bj-K, and SXi Xj-K. There are not, however, any separate machine instructions for subtracting constants. Instead, the assembler stores the one's complement of K in the instruction. For example,

```
SX3     A1 - 10
```

becomes

```
70  3  1  777765
```

(since 10 is 12 (base8)); like FORTRAN, numbers are assumed decimal unless suffixed by a B). Also,

```
SX6     X1 - 777607B
```

becomes

```
72  6  1  000170
```

i.e.,

```
SX6     X1+170B
```

It is possible, as it was in the JP instruction, to omit specifying any register:

```
SX2     314
```

in this case B0 is understood, so a

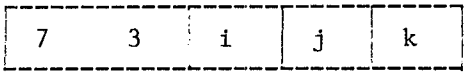
```
SX2     B0+314
```

is assembled, which indeed sets X2 to 314. Since a label is essentially a number (the number of the location it is associated with), a label can be used in place of a number in these instructions:

```
SX2     ABLE
SX5     B1+MABEL
```

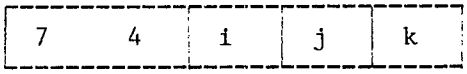
The short instructions, opcodes 73-77, are 15 bits long; they calculate the sum or difference of two registers. There are theoretically 15 possible combinations (SXi Aj+Ak, Aj+Bk, Aj+Xk, Bj+Bk, Bj+Xk, Xj+Xk, Aj-Ak, Aj-Bk, Aj-Xk, Bj-Bk, Bj-Xk, Xj-Xk, Bj-Ak, Xj-Ak, Xj-Bk) but the limited number of opcodes restricts us to five; which five? The hardware designers reasoned that, since indices would normally be kept in B registers, the most useful instructions would be those for adding and subtracting B registers from A, B, and X registers. This leaves six opcodes; deeming SXi Xj-Bk least useful, they implemented

SXi Xj+Bk Set Xi to Xj+Bk



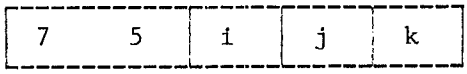
(15 bits)

SXi Aj+Bk Set Xi to Aj+Bk



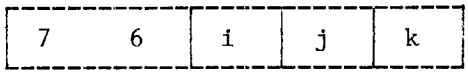
(15 bits)

SXi Aj-Bk Set Xi to Aj-Bk



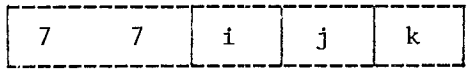
(15 bits)

SXi Bj+Bk Set Xi to Bj+Bk



(15 bits)

SXi Bj-Bk Set Xi to Bj-Bk



(15 bits)

As you must have realized by now, making a set-instruction mnemonic is very simple: the opcode field is the result register, prefixed by "S"; the address field has the two operands, separated by + or -. Unfortunately, this logical set of mnemonics tempts programmers to make up nonexistent instructions as necessary for their programs. You are urged to remember that it is possible to add or subtract a B register from another register, but not X register minus B register.

The mnemonics are self-explanatory, and the operations the same as those involved in the long instructions, so a few examples should suffice to make the instructions clear: if B1 = 2 and B7 = 3,

```
SX6     B1+B7
```

set X6 to 5, while

```
SX6     B1-B7
```

sets X6 to -1 (77777 77777 77777 777776 (base8)).

A set instruction may contain simply one register designation in the address field:

```
SX6     A0
```

In such a case, two different but equivalent instructions could be assembled:

```
SX6     A0+0 (opcode 70) or
SX6     A0+B0 (opcode 74).
```

Can you figure out which is preferable? Since the instructions are otherwise equivalent, the shorter one (opcode 74 -- 15 bits) is preferred, so the assembler will generate SX6 A0+B0. Although saving 15 bits may not seem very important, it adds up to a lot of words in a 20 or 30 thousand instruction program.

Now to write our first useful routine, LOCF. You may be acquainted with the LOCF function from FORTRAN. It has one argument and returns as its result the address of the argument: the value of

```
LOCF(A)
```

is the location of A in memory (the first location of A, if A is an array).

To write our routine, we have to answer: where do we get the operand (argument) from, what operation do we perform on the operand, and where do we leave the result? The input to the routine is the address of the argument, which is in X1; the expected output is precisely this address, in X6 (remember, a function always returns its value in X6). Thus all the routine has to do is take what it gets in X1 and put it into X6.

```
      IDENT     LOCF
      ENTRY     LOCF
LOCF  BSS       1          ENTRY LINE
      SX6       X1         PUT RESULT INTO X6
      EQ        LOCF       RETURN
      END
```

The set A register instructions, opcodes 50-57, are the same, execpt for result register type, as the set X instructions:

```
opcode  50  SAi  Aj+K  Set Ai to Aj+K    (30 bits)
opcode  51  SAi  Bj+K  Set Ai to Bj+K    (30 bits)
opcode  52  SAi  Xj+K  Set Ai to Xj+K    (30 bits)
opcode  53  SAi  Xj+Bk Set Ai to Xj+Bk   (15 bits)
opcode  54  SAi  Aj+Bk Set Ai to Aj+Bk   (15 bits)
opcode  55  SAi  Aj-Bk Set Ai to Aj-Bk   (15 bits)
opcode  56  SAi  Bj+Bk Set Ai to Bj+Bk   (15 bits)
opcode  57  SAi  Bj-Bk Set Ai to Bj-Bk   (15 bits)
```

Recall that setting A1 through A5 to an address loads the contents of that location into the associated X register, while setting A6 or A7 stores the contents of the X register at the specified location. Setting A0 causes no memory reference.

Your job is assigned a definite number of words in central memory in which to run, known as its field length (FL). If a program attempts to reference an address (load or store) beyond its field length, the job will be aborted with the day file message "CPU ERROR EXIT AT 012345, CM OUT OF RANGE".

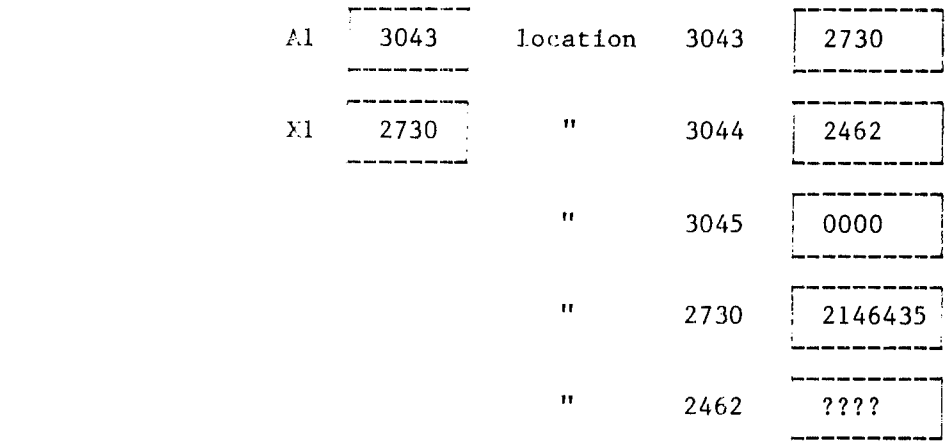
With this information, we are ready for our second subroutine. This routine has two arguments, and puts in the second argument the contents of the first, truncated to 18 bits, with the sign of the 18 bit number extended to the high 42 bits. This none-too-useful routine at least has the merit that it can be coded in terms of the instructions studied so far:

```
        IDENT     SET
        ENTRY     SET
SET     BSS       1          ENTRY LINE
        SA2       A1+1       X2=ADDRESS OF SECOND ARGUMENT
        SA1       X1         X1=FIRST ARGUMENT
        SX6       X1         X6=FIRST ARGUMENT, TRUNCATED
        SA6       X2         STORE INTO SECOND ARGUMENT ADDRESS
        EQ        SET
        END
```

Since this is our first procedure of some complexity, let's examine it closely, Suppose the routine is called as follows:

```
CALL SET (I,J)
```

where I and J's memory addresses are 2730 and 2462, respectively, and the value of I is 2146435 (octal). Thus, the calling sequence might look like this:



(the contents of location 2462 is of no interest to us).

The SA2 A1+1 adds one to the address in A1 (A1 itself is not modified) and set A2 to this new address, causing a fetch of the address of the second parameter to X2 (A2 = 3044; X2 = 2462). The SA1 X1 sets A1 to the address of the first parameter, hence fetches the first parameter to X1 (A1 = 2730; X1 = 2146435). The SX6 X1 performs the necessary truncation and sign extension, leaving the result in X6 (X2 = 2146435; X6 = 146435). SA6 X2 put the address of the second parameter in A6, storing the result into the second parameter (A6 = 2462; contains of location 2462 = 146435).

If the routine was called with a constant

```
CALL SET (1,J)
```

location 3043 would not contain the number one. The FORTRAN compiler would create a variable called ONE, for example, whose contents is the number one and set the contents of 3043 to the address of ONE.

The third group of set instructions, for B registers, is the same as the other two:

```
opcode  60  SBi  Aj+K  Set Bi to Aj+K    (30 bits)
opcode  61  SBi  Bj+K  Set Bi to Bj+K    (30 bits)
opcode  62  SBi  Xj+K  Set Bi to Xj+K    (30 bits)
opcode  63  SBi  Xj+Bk Set Bi to Xj+Bk   (15 bits)
opcode  64  SBi  Aj+Bk Set Bi to Aj+Bk   (15 bits)
opcode  65  SBi  Aj-Bk Set Bi to Aj-Bk   (15 bits)
opcode  66  SBi  Bj+Bk Set Bi to Bj+Bk   (15 bits)
opcode  67  SBi  Bj-Bk Set Bi to Bj-Bk   (15 bits)
```

The instructions SB0 B0+7 is perfectly acceptable, and will not cause the computer to blow a transistor. However, after the instruction has been executed, B0 will still be zero (it is permanently zero).

A common use of the set-B instructions is in program loops. For example the FORTRAN loop

```
      K = 0
      DO  20 I = 1,10
20    K = K+ 1
```

could be coded as follows, where X0 is used instead of K

```
      SB1       10             SET LOOP UPPER BOUND
      SB2       1              SET LOOP LOWER BOUND
LOOP  SX0       X0+B2          ADD IN LOOP COUNTER
      SB2       B2+1           INCREMENT LOOP COUNTER
      GE        B1,B2,LOOP     CHECK IF WE'RE DONE
```

Another, less useful, routine is a function to add two integers of magnitude less that 2**16 - 1. The function, called SUM, has as its arguments the two addends. Thus all the routine has to do is to load the two arguments and add them, leaving the result in in X6:

```
        SA1     B1
        SA2     B2
        SX6     X1+X2
```

All right? Not quite: there is no instruction SX6 X1+X2. What we have to do is put one operand into a B register, and then add them:

```
        IDENT   SUM
        ENTRY   SUM
SUM     BSS     1           ENTRY LINE
        SA2     A1+1        X2=ADDRESS OF SECOND ARGUMENT
        SA1     X1          X1=FIRST ARGUMENT
        SA2     X2          X2=SECOND ARGUMENT
        SB3     X2          B3=SECOND ARGUMENT
        SX6     X1+B3       X6=SUM
        EQ      SUM         RETURN
        END
```

We wouldn't want to use an A register (except A0) instead of B3, lest we produce a memory reference out of range or accidentally store into a needed location. Note that since the magnitude of the operands is less that 2**16 - 1, the sum is less than 2**17 - 1, avoiding the problem of truncation of the set instruction.

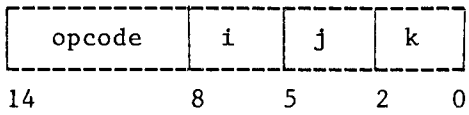
A small warning should be made at this point: the FORTRAN extended compiler assumed that functions and subroutines donot change the value of A0. If you write a routine which uses A0, you must save the value of A0 at the beginning of the routine and restore the original value before returning. To simplify matters, we shall avoid using A0 in the routines in this book.

3.8 BOOLEAN INSTRUCTIONS

Having dispatched the jump and increment unit (set) instructions, we shall now begin the discussion of the arithmetic operations among X registers (opcodes 10-47) with the Boolean instructions, the basic logical operations -- complement (.NOT.), logical sum (.OR.), logical product (.AND.), logical difference (exclusive or) -- and the moving of a number from one register to the other. Logical operations are done on a bit-by-bit basis; that is, the value of bit i of the result is determined by the value(s) of bit i in the operand(s).

The Boolean instructions are organized into two groups of four instructions. There are four basic operations: logical sum, product, difference and transmit. The transmit operation, the only unary operation (one operand) of the group, simply takes the contents of the one register and transfers it to another. In the first group of four, neither operand is complemented before the operation; in the second group, one of the operands is complemented before the operation.

With this introduction, the eight Boolean instructions will now be enumerated. All are 15 bit instructions, specifying two operand X registers and one result X register:



In the unary operations, one of the two operand designators (j or k) is ignored. Since these are bit-by-bit operations, they may be described precisely by giving the result for each possible bit combination in the operands. Sample operands and results (in binary), including all possible bit combinations, are given below for each instruction:

opcode 10: BXi Xj Transmit Xj to Xi

```
     Transfers a 60-bit word from Xj to Xi:
         if   Xj = 10 (base2),
         then Xi = 10 (base2)
```

opcode 11: BXi Xj*Xk Logical Product of Xj and Xk to Xi

```
     Bit in Xi is 1 when corresponding bitsin both Xj AND Xk are 1:
        if    Xj = 1010
              Xk = 1100
                   ----
        then  Xi = 1000
```

opcode 12: BXi Xj+Xk Logical Sum of Xj and Xk to Xi

```
     Bit in Xi is 1 when corresponding bit in either Xj OR Xk is 1:
        if    Xj = 1010
              Xk = 1100
                 = ----
        then  Xi = 1110
```

opcode 13: BXi Xj-Xk Logical Difference of Xj and Xk to Xi

Bit in Xi is 1 when corresponding bits in Xj and Xk are unlike (exclusive OR):

```
if   Xj = 1010
     Xk = 1100
          ----
then Xi = 0110
```

opcode 14: BXi -Xk Transmit the complement of Xk to Xi

Puts in Xi the complement of the contents of Xk:

```
if   Xk = 10
then Xi = 01
```

opcode 15: BXi -Xk*Xj Logical product of Xj and complement of Xk to Xi

Bit in Xi is 1 when corresponding bits in both Xj AND the complement of Xk are 1:

```
if   Xj = 1010                  Xj = 1010
     Xk = 1100      comp. of    Xk = 0011
                                ---------
then                            Xi = 0010
```

opcode 16: BXi -Xk+Xj Logical Sum of Xj and complement of Xk to Xi

Bit in Xi is 1 when corresponding bit in either Xj OR the complement of Xk is 1:

```
if   Xj = 1010                  Xj = 1010
     Xk = 1100      comp. of    Xk = 0011
                                ---------
then                            Xi = 1011
```

opcode 17: BXi -Xk-Xj Logical Difference of Xj and complement of Xk to Xi

Bit in Xi is 1 when corresponding bits in Xj and Xk are the same:

```
if  Xj = 1010                    Xj = 1010
    Xk = 1100       comp. of     Xk = 0011
                                 ---------
then                             Xi = 1001
```

The mnemonics are, as usual, straightforward: the letter "B" followed by the result register in the opcode field, the operands and operators in the address field. Note that the minus sign indicating complementation must appear first (BXi Xj*-Xk is not allowed) and that this minus sign signifies that only the immediately following register is complemented, not the entire quantity.

About the simplest possible subroutine is a transmit subroutine, which takes the contents of the first argument and puts it in the second:

```
      IDENT     XMIT
      ENTRY     XMIT
XMIT  BSS       1          ENTRY LINE
      SA2       A1+1       X2=ADDRESS OF SECOND ARGUMENT
      SA1       X1         X1=FIRST ARGUMENT
      BX6       X1         X6=FIRST ARGUMENT
      SA6       X2         STORE INTO SECOND ARGUMENT ADDRESS
      EQ        XMIT
      END
```

This routine is very similar to our earlier routine "SET", but this one transmits the entire 60 bit value, not just the low 18 bits. This routine is thus suitable for transferring any single-word datum, including integer and floating-point numbers.

A variation of our "SUM" function provides our second example. This function will take the 60 bit logical sum of the two arguments:

```
      IDENT     LSUM
      ENTRY     LSUM
LSUM  BSS       1
      SA2       A1+1       X2=ADDRESS OF SECOND ARGUMENT
      SA1       X1         X1=FIRST ARGUMENT
      SA2       X2         X2=SECOND ARGUMENT
      BX6       X1+X2      X6=ARG1 .OR. ARG2
      EQ        LSUM
      END
```

Then the statement

```
I = LSUM(J,77B)
```

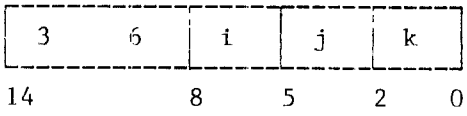
where J = 140B will set I to 177B.

Several additional examples will be forthcoming when we discuss bit and character manipulation.

3.9 INTEGER ARITHMETIC: ADDITION AND SUBTRACTION

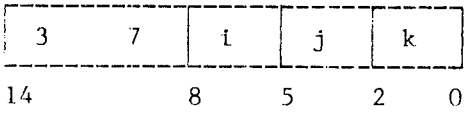
Integer addition and subtraction are about the simplest of the arithmetic instructions. These instructions form the 60 bit one's complement sum and difference of the contents of two X registers. Because these instructions do 60 bit arithmetic, they are termed long add instructions (in contrast to the floating operations, which we shall discuss shortly). The two instructions are:

IXI Xj+Xk Integer sum of Xj and Xk to Xi



(15 bits)

IXi Xj-Xk Integer difference of Xj and Xk to Xi



(15 bits)

As the characteristic mnemonic letter for set instructions is "S", and for Boolean is "B", for integer operations it is "I".

As a simple example, we may conjure up a routine which takes the (60 bit) difference of two integers. This function of two arguments is similar to several we have written before:

```
      IDENT     IDIF
      ENTRY     IDIF
IDIF  BSS       1           ENTRY LINE
      SA2       A1+1        X2=ADDRESS OF SECOND ARGUMENT
      SA1       X1          X1=FIRST ARGUMENT
      SA2       X2          X2=SECOND ARGUMENT
      IX6       X1-X2       X6=ARG1 - ARG2
      EQ        IDIF        RETURN
      END
```

We can now "jazz up" the routine so that, if A-B is negative, the routine will return zero instead of A-B; the result is a standard FORTRAN function, IDIM. All that is required is to add an instruction to set X6 to zero, preceded by a jump which returns directly (avoiding the set to zero) if the result is positive:

```
      IDENT     IDIM
      ENTRY     IDIM
IDIM  BSS       1           ENTRY LINE
      SA2       A1+1        X2=ADDRESS OF SECOND ARGUMENT
      SA1       X1          X1=FIRST ARGUMENTT
      SA2       X2          X2=SECOND ARGUMENT
      IX6       X1-X2       X6=ARG1 - ARG2
      PL        X6,IDIM     IF ARG1 - ARG2 POSITIVE, RETURN
      SX6       B0          ELSE SET RESULT=0
      EQ        IDIM        AND RETURN
      END
```

The original 6000 and 7000 series machines were not provided with any single instructions for doing integer multiplication and division. These operations were performed intead by a sequence of instructions using floating multiplication and division. The absence of hardware integer multiplication and division is attributable to the basic objective in 6600 design of high speed scientific calculation. Since scientific computation is characterized by a preponderance of floating point rather than integer arithmetic, it was decided to have the integer operations be somewhat slower and utilize the very-high-speed floating point capabilities. Recent machines have incorporated a slight modification of the arithmetic unit to permit integer multiplication in a single instruction; this change has by now also been added to most existing 6000 series machines. The sequences of instructions for integer multiplication and division on both old and new machines will be described in section 3.15.

3.10 FLOATING POINT ADDITION AND SUBTRACTION

Consider the problem of adding, using scientific notation,

```
  3.40 * 10**3
+ 1.77 * 10**2
```

The operation involves two steps: (1) changing one of the numbers (say, the one with the smaller exponent) so that both have the same exponent and (2) adding the coefficients, affixing to the result the common exponent of the addends:

```
3.40 * 10**3 = 3.40  * 10**3
1.77 * 10**2 =  .177 * 10**3
               ------------
                3.577 * 10**3
```

If we are retaining three digits of accuracy, we can either truncate the fraction, leaving 3.57*10**3, or, more accurately, round the fraction, yielding 3.58*10**3. To complicate the situation somewhat, let us consider

```
  1.21 * 10**3
- 7.82 * 10**2
```

By the same two operations as above, we obtain

```
 1.21 * 10**3 =  1.21 * 10**3
-7.82 * 10**2 = -.782 * 10**3
-------------------------------
                 .428 * 10**3
```

But .428*10**3 is no longer in normal form, i.e., does not have one significant digit to the left of the decimal point. Hence we must include one final step, normalization, to obtain our result in normal form, 4.28*10**2.

The process we have illustrated in decimal scientific notation is little different from that used for adding floating point numbers in the computer. First a floating add is done to add the two numbers (steps (1) and (2) above), and then a normalize is performed to get the result back into normal form. A special 98 bit register, known as an accumulator, is used to perform the shifting (to equalize coefficients), and the actual addition. A 98 bit register makes it possible to obtain a double precision result, with 96 bit accuracy. Just as our three digit operands may yield a sum contianing more than three digits, 48 bit addends may yield a more than 48 bit sum.

As a result of the addition of two 60 bit floating point numbers on the machine, three different numbers may be obtained: a floating sum, a round floating sum, and a floating double precision sum. The double precision sum gives the low order 48 bits out of the accumulator; as its name implies, it is used for double precision arithmetic. The other two sums both give the most significant 48 bits of the accumulator, and either can be used when only a single precision (48 bit) result is desired. The floating sum just chops out 48 bits from the accumulator, whereas the round sum includes a rounding procedure which yields a slightly better result.

To clarify this, consider the addition (all numbers in octal)

```
4710 0010 0000 0210.  * 2**3
6163 5050 0000 0421.  * 2**(-33)
```

These are essentially 6600 floating point numbers, with a 48 bit integer coefficient and binary exponent, except that here a true, rather than biased, exponent is given. Schematically, the addition is done on the 6600 as follows: first, the addend with the smaller exponent is put in the accumulator

```
0 6163 5050 0000 0421. 0000 0000 0000 0000 * 2**(-33)
```

where a decimal point has been inserted to indicate the assumed binary point between bits 47 and 48. The leftmost zero represents only two bits; the high-order bit, bit 97, is the sign bit, while the next bit is there to prevent possible error in the case of overflow. The coefficient is now shifted and the exponent correspondingly increased until it equals that of the other operand:

```
0 0000 0000 0061 6350. 5000 0004 2100 0000 * 2**3
```

The second operand is now added in:

```
 0   0000 0000 0061 6350. 5000  0004 2100 0000 * 2**3
+    4710 0010 0000 0210.
-------------------------------------------------------
 0   4710 0010 0061 6560. 5000  0004 2100 0000 * 2**3


    |<-- floating sum ->|   floating double
                          |<- precision sum ->|
```

The most significant part of the sum is given by the floating sum, 4710 0010 0061 6560*2**3. The double precision sum gives the least significant part, i.e., that number which, when added to the floating sum, gives the sum to 96 bit accuracy, 5000 0004 2100 0000 * 2**(-55). Note that, because in the accumulator the binary point is assumed at the left of these 48 bits, the exponent must be reduced by 48 (base 10) =60 (base 8) when the DP sum is stored as a separate floating point number, with the assumed binary point on the right.

The rounding procedure in the floating add unit is different from the usual method you know for rounding, though the effect is almost the same. The "usual method" for calculating a round floating sum would be to increment the integer part (bits 48-95) by one if the fractional part (bits 0-47) were greater than one-half, i.e., if bit 47 were a one. The add unit instead puts a one bit on the right end of the operands (if both are normalized); in our example the addition would become

```
                                          round bit
                                          ***  |
 0   0000 0000 0061 6350.  5000 0004 2140 0000 * 2**3
+    4710 0010 0000 0210.  4                   * 2**3
-------------------------------------------------------
                                  |
                                  └-round bit
 0   4710 0010 0061 6561.  1000 0004 2140 0000 * 2**3

    |<--- round sum --->|
```

Observe that the round sum, 4710 0010 0061 6561*2**3, is the same result we would have obtained by applying the "usual method" of rounding to our original 98 bit sum. This is because the effect of the actual rounding procedure is to add one-half to the sum. This is true even if the exponents of both operands are the same, since then the binary point must be shifted one bit left. For example, the round sum of 5100 0000 5200 0000*2**10 with itself,

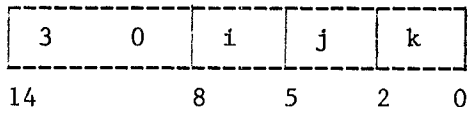
```
     5100 0000 5200 0000.  4000 0000 0000 0000 * 2**10
+    5100 0000 5200 0000.  4                   * 2**10
-------------------------------------------------------
 1   2200 0001 2400 0001.  0000 0000 0000 0000 * 2**10
=0   5100 0000 5200 0000.  4000 0000 0000 0000 * 2**11

    |<--- round sum --->|
```

Inserting bits before the addition is a simpler and faster procedure than adding one to the coefficient at the end, which may require several carries.

When calculating a round sum, only the upper 48 bits are usuable; the least significant 48 are meaningless. Conversely, when we wish to recover 96 bits, we must use the floating and DP sums, and not the round sum. Thus, the floating and DP sums are essential for double precision work, while the round sum is preferable for single precision, since it is slightly more accurate. (For reasons of tradition, however, most compilers, including those for the 6600, generate floating rather than round floating arithmetic instructions.)

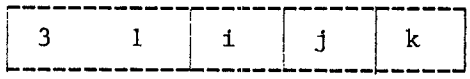
With this we are ready to enumerate the floating point addition and subtraction instructions.

FXi Xj+Xk Floating sum of Xj and Xk is Xi



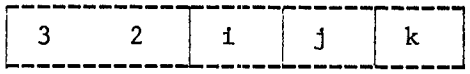
(15 bits)

FXi Xj-Xk Floating difference of Xj and Xk to Xi



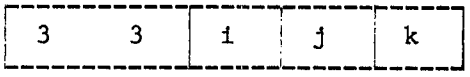
(15 bits)

DXi Xj+Xk Floating DP sum of Xj and Xk to Xi



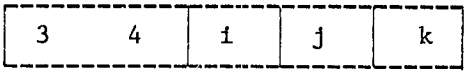
(15 bits)

DXi Xj-Xk Floating DP difference of Xj and Xk to Xi



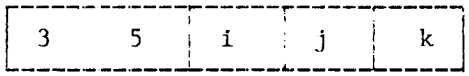
(15 bits)

RXi Xj+Xk Round floating sum of Xj and Xk to Xi



(15 bits)

RXi Xj-Xk Round floating difference of Xj and Xk to Xi

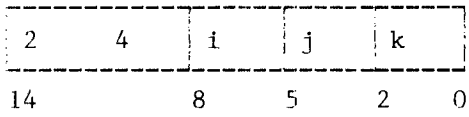


(15 bits)

As is evident, the characteristic letters for floating, DP, and round arithmetic are F, D, and R; from this you can probably already guess the mnemonics for the multiply and divide instructions.

As the preceding discussion suggested, we will require one additional instruction: the normalize instruction, which puts a number into normalized form. That is, the coefficient from the operand is shifted left bit-by-bit until the most significant bit is in bit 47; positions vacated on the right are filled with zeros (binary ones if the number is negative). For each bit that the coefficient is shifted, the exponent is decremented by one, so the value of the number is unchanged. The normalized number is put in the X result register; in addition, the number of shifts required for normalization is left in a B result register. Thus, this is one of the few (3) 6600 instructions with only one operand and two results. The normalize instruction is the first of the so-called "shift unit instructions" (opcodes 20-27) which we shall consider:

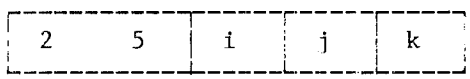
NXi Bj, Xk Normalize Xk into Xi and Bj



(15 bits)

For the sake of completeness, the round and normalize instruction will also be mentioned. This instruction adds 1/2 to the coefficient before normalizing; i.e., for positive numbers, a 1 bit is attached to the right of the binary point before shifing. It has the totally nonsensical mnemonic ZX:

ZXi Bj, Xk Round and normalize Xk into Xi and Bj



(15 bits)

For example, if

X2 = 1753 0005 7410 2121 6050

the instruction NX3 B3,X2 will set

B3 = 000011 X3 = 1742 5741 0212 1605 0000

while ZX3 B3,X2 results in

round bit →

B3 = 000011 X3 = 1742 5741 0212 1605 0400

Lest you fear that normalizing a zero coefficient puts the machine in some sort of infinite loop, let me inform you that the instruction ends with the shift count (Bj) = 48(base 10), and Xi cleared to zero. This is important in testing a floating point number for zero, using the ZR Xi,K instruction. In general, subtracting a number from itself will yield a result with a zero coefficient but not a zero exponent, so the result will fail the ZR test, which examines all sixty bits. After normalization, however, a result with a coefficient of zero will become all zero, so the branch will occur if a ZR test is made. To round out the picture, one may note that a ZX (rounded normalize) of a zero coefficient will reduce the exponent by 48 and leave a round bit in bit 47.

In the following examples we will use only the unrounded normalize, since we will be doing round floating arithmetic. Performing a rounded normalize on the result of a round floating operation would have the undesirable effect of adding a round bit in twice.

As a trivial example, we can modify the IDIM function coded earlier for floating point numbers (i.e., the FORTRAN ADIM function). The only change is in the subtraction:

```
      IDENT    ADIM
      ENTRY    ADIM
ADIM  BSS      1          ENTRY LINE
      SA2      A1+1       X2=ADDRESS OF SECOND ARGUMENT
      SA1      X1         X1=FIRST ARGUMENT
      SA2      X2         X2=SECOND ARGUMENT
      RX6      X1-X2      X6=ARG1-ARG2
      NX6      B0,X6
      PL       X6,ADIM    IF (ARG1-ARG2).GE.0,RETURN
      SX6      B0         ELSE SET RESULT=0
      EQ       ADIM       AND RETURN
      END
```

Note that we had no use for the shift count from the normalize instruction, so we designated B0 for the result (B0, however, is unaffected -- it is always zero); this could have been written NX6 X6, as B0 is assumed if no B register is specified. If the normalize were omitted, the routine would be unacceptable, since in our coding we shall always assume that floating point operands (which are results of previous operations) are normalized. Hence, every floating add or subtract must be followed by a normalize instruction.

As a slightly more complicated example, let us consider the function whose value is the trace -- the sum of the diagonal elements -- of a square matrix of arbitrary size. In FORTRAN this would be

```
      FUNCTION TRACE(ARRAY,N)
      DIMENSION ARRAY(N,N)
      TRACE = 0
      DO 10 I = 1,N
10    TRACE = TRACE+ARRAY(I,I)
      RETURN
      END
```

Now observe that the address of ARRAY(I,I) = address of the first word of ARRAY+(I-1)+N*(I-1)= address of the first word of ARRAY+(N+1)*(I-1). Since in memory the N*N array is stored as linear array of N*N elements (A(1,1), A(2,1),...,A(N,1), A(1,2),A(2,2),..., A(N,N)), it is simpler to manipulate ARRAY as a linear array when possible. In this case, it is particularly simple: the DO loop in effect goes through ARRAY in steps of N+1. Thus, to load the first element of ARRAY we set an A register to ARRAY (the value of an array tag is the first location of the array). To load subsequent array entries, we need merely increment the A register by N+1. In addition, we will have to keep a count so that the assembly language routine, like the DO loop, will know when its job is finished. Thus, in COMPASS, our function is

```
        IDENT   TRACE
        ENTRY   TRACE
TRACE   BSS     1
        SA2     A1+1
        SA2     X2              X2=DIMENSION OF ARRAY
        SB4     X2+1            B4=DIMENSION + 1
        SA1     X1              A1=STARTING ADDRESS OF ARRAY
        BX6     X1              X6=FIRST ARRAY ELEMENT
        SB3     1               INITIALIZE COUNTER
LOOP    SA1     A1+B4           GET NEXT ARRAY ELEMENT
        RX6     X1+X6
        NX6     X6              ADD INTO SUM
        SB3     B3+1            INCREMENT COUNTER
        LT      B3,B4,LOOP      IF NOT THROUGH, LOOP
        EQ      TRACE           ELSE RETURN
        END
```

It is unlikely that any but the best optimizing compilers would be able to produce such short code from the FORTRAN version of the routine. It is not so difficult to realize that it is more efficient to accumulate the sum in X6, since the result will have to be put in X6 before returning; on the other hand, it is a very shrewd compiler which sees the trick with an increment of N+1.

3.11 FLOATING POINT MULTIPLICATION

Multiplying two floating point numbers involves multiplying coefficients and addding exponents, in decimal:

```
    3.1  *  10**2
*   2.2  *  10**7
-----------------
    6.82 *  10**9
```

or in octal:

```
                4200  0000  0000  0000.  * 2**3
            *   6010  0000  0000  0000.  * 2**6
-------------------------------------------------
3144200000000000  0000  0000  0000  0000.  * 2**11
```

Multiplying together two-digit decimal coefficents yields a three or four digit result (3.1 * 2.2 = 6.82; 6.4 * 6.4 = 40.96); similarly, the product of two 48 bit coefficents is a 95 or 96 bit result. The multiply instructions on the 6600 automatically adjust a 95 bit result coefficent to get a normalized 96 bit quantity, shifting the coefficent left one bit and reducing the exponent by one; for example, the result above would become

```
6310400000000000  0000000000000000.  * 2**10
```

Thus if the two operands in a multiply instruction are both normalized, and so have 48 bit coefficents, the result will have a normalized 96 bit coefficent; no additional normalize instruction is required.

Just as there are three types of floating point addition and subtraction, there are three types of floating point multiplication: floating, round floating, and double precision floating. Since each instruction can only return one floating point number, with 48 bits of coefficent, two instructions, the floating and DP multiply, are needed to get out all 96 bits:

```
6310400000000000  0000000000000000.  * 2**10

floating product      DP product
```

So, in this example, the floating product is 63310400000000000 * 2**10; right? Wrong. At the end of a multiply, the binary point is all the way at the right of the 96 bits; when we put the most

significant 48 bits into the result register, the assumed binary point is at the right of those 48 bits -- in other words, it has been shifted 48 bits to the left. So, to have the floating product be the most significant half of the product, we have to increase the exponent put in the floating result by 48 (base 10) = 60 (base 8). On the other hand, no change is need to get the exponent of the DP result, since the assumed binary point starts out to the right of those 48 bits. Thus the two products are:

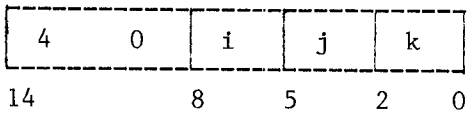
```
floating product = 6310400000000000 * 2**70
      DP product = 0000000000000000 * 2**10
```

their sum is the product with 96 bit precision. (Don't get the impression, incidentally, that the DP product is generally zero; this is only a consequence of the numbers chosen).

The round multiply gives a rounded version of the most significant half of the product -- a 48 bit coefficent "good to the last bit". As before, the rounded result is generally preferable for single precision, while the floating and DP instructions are needed to recover a double precision result.

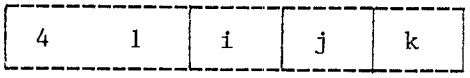
The three instructions occupy opcodes 40-42:

FXi Xj*Xk Floating product of Xj and Xk to Xi



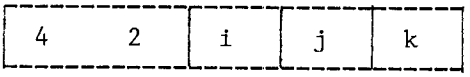
(15 bits)

RXi Xj*Xk Round floating product of Xj and Xk to Xi



(15 bits)

DXi Xj*Xk Floating DP product of Xj and Xk to Xi



(15 bits)

The example we shall consider to illustrate the multiply instructions is considerably more complicated than our previous subroutines. This routine will compute the product of two matrices; in FORTRAN it would be

```
                SUBROUTINE MATMU(A,B,C,L,M,N)
C
C               MULTIPLIES MATRIX A, DIMENSIONS L*M,
C               AND MATRIX B, DIMENSIONS M*N, LEAVES
C               RESULT IN MATRIX C, DIMENSIONS L*N
C
                DIMENSION A(L,M), B(M,N), C(L,N)
                DO 20  I = 1,L
                DO 20  K = 1,N
                SUM = 0.
                DO 10 J = 1,M
     10         SUM = SUM + A(I,J)*B(J,K)
     20         C(I,K) = SUM
                RETURN
                END
```

This example is important because it occurs often in practical calculations, and can consume a lot of computing time: the innermost DO loop is executed L*M*N times. For example, if the matrices were all 100*100 (quite large, but still easily within the capacity of memory) the inner loop would be executed 1,000,000 times.

We will try to write an optimized matrix multiply to the extent we have learned so far, i.e., minimize the number of instructions to be executed. Of course, since the inner loops are executed much more often than the initialization code (which is executed once), we shall be much more concerned about saving instructions in the loops. (To the ambitious student, may I suggest that, to get the most out of the following explanation, you code -- and perhaps even try to run -- a MATMU of your own in assembly language before reading on.)

As a first step let us consider what we have to do in the innermost loop. The biggest problem seems to be geting A(I,J) and B(J,K):

address of A(I,J) = address of A + (I-1) + L*(J-1)
address of B(J,K) = address of B + (J-1) + M*(K-1)

We can note that it would make more sense to keep indices I′ = I-1, J′ = J-1, and K′ = K-1; then

address of A(I,J) = address of A + I′ + J′*L
address of B(J,K) = address of B + J′ + K′*M

But we don't know how to do those integer multiplications yet, and anyway feel that there must be some better way than doing

these multiplications perhaps a million times. Taking a hint from our TRACE function, we suspect that we can simply increment the relevant A registers each time through the inner loop. And indeed, each iteration of the loop advances the A matrix address by L and the B matrix address by 1.

With this thought in mind, we can begin by constructing the inner loop. We may assume for the moment that the loop indices and limits will be kept in B registers, as usual; we can change this later if we get into difficulty. So suppose we have I' in B2, J' in B3, K' in B4, L in B5, M in B6, and N in B7. Let's also suppose that we are loading A and B matrix elements into X1 and X2 respectively, and accumulating the sum in X6. Then the innermost loop has to (1) advance A1 by L and A2 by 1, (2) multiply elements of matrices A and B and add into sum, and (3) increment J' and loop if not done:

```
LUP1  SA1       A1+B5        X1=A(I,J)
      SA2       A2+1         X2=B(J,K)
      RX1       X1*X2        X1=A(I,J)*B(J,K)
      RX6       X1+X6
      NX6       X6           ADD INTO SUM
      SB3       B3+1         J'=J'+1
      LT        B3,B6,LUP1   IF(J' .LT .M), LOOP
```

Note that the loop is done when J exceeds M, and so when J' is equal to M.

Having succeeded in coding the innermost loop in a minimum of instructions (each operation is essential to the loop), let us see whether we can do the same for the outer loops. Instructions are needed right before the inner loop to initialize the sum (X6) and after the loop to store the result. Now the question arises, should the I or the K loop be outer-most? Though the differences are small, I have selected the I loop to be in the middle, and the K loop to be on the outside. Consider the store instruction which follows the inner loop: previous experience indicates that we can compute the address for the store most simply if we find a constant increment by which to advance the store address throughout the loop. This is possible, clearly, only if the address is incremented by one each time; since address of C(I,K)= address of C + I' + K'*L, this means that we increment I' in the inner loop, and K' in the outer loop (thus obtaining the sequence C(1,1),..., C(L,1), C(1,2),..., (C(1,3),..., C(L,N)). Since each element of matrices A and B is accessed several times, it would not have been possible to reference eoither A or B simply by continuously incrementing an A register. The outcome of all this is that the store operation (C(I,K) = SUM) will become

```
          SA6       A6+1       STORE IN C(I,K)
```

The other tasks in the outer loops are to (1) initialize A1 and A2 for the inner loop and (2) initialize the sum (X6). From the FORTRAN routine we might suspect that the latter becomes SX6 B0. Some thought, however, should indicate that it would be faster to initialize X6 to A(I(1)*B(1,K), since it saves one iteration of the inner loop. (In exchange for this slight increase in efficiency, however, we obtain a routine which will not work for M=1.) And, at the same time, A(I,1) and B(1,K) are the correct initial addresses in A1 and A2 for starting the inner loop. So the only remaining problem is getting the addresses of A(I,1) and B(1,K); we decided before that we can't do this just by incrementing the previous contents of A1 or A2 (unless we are willing to increment/decrement more than once). But A(I,1) is simple, since address of A(I,1)= address of A+I' = B1+B2, if we keep the address of A in B1. Address of B(1,K) = address of B+K'*M is a bit more work, but we can manage by keeping the address of B in some register and adding M to the register each time through the K loop. Having run out of B registers, let's use, say, X5.

If you have been able to wade through all this, you will realize that the code before the inner loop is

```
LUP2 SA1       B1+B2      X1=A(I,1)
     SA2       X5         X2=B(1,K)
     RX6       X1*X2      INITIALIZE SUM
     SB3       1          J'=1
```

while at the end of the I loop we have to perform the store, increment and test I':

```
          SA6       A6+1          STORE SUM INTO C(I<K)
          SB2       B2+1          I'=I'+1
          LT        B2,B5,LUP2    IF I'.LT.L, LOOP
```

Our outermost (K) loop doesn't have to do very much: at the beginning, initialize I' for the middle loop, at the end increment and test K' -- and advance X5, which should have the address of B+K'*M:

```
LUP3 SB2       B0            I'=0
       .
       .
     SB4       B4+1          K'=K'+1
     SX5       X5+B6         X5=X5+M
     LT        B4,B7,LUP3    If K'.LT.N, LOOP
```

Since you are by now probably thoroughly confused as to how all these jig-saw puzzle pieces fit together, let me put all the loops together

```
LUP3 SB2      B0           I'=0
LUP2 SA1      B1+B2        X1=A(I,1)
     SA2      X5           X2=B(1,K)
     RX6      X1*X2        INITIALIZE SUM
     SB3      1            J' =1
LUP1 SA1      A1+B5        X1=A(I,J)
     SA2      A2+1         X2=B(J,K)
     RX1      X1*X2        X1=A(I,J)*B(J,K )
     RX6      X1+X6        ADD INTO SUM
     NX6      X6
     SB3      B3+1         J'=J'+1
     LT       B3,B6,LUP1   IF(J'.LT.M), LOOP1
     SA6      A6+1         C(I,K)=SUM
     SB2      B2+1         I'=I'+1
     LT       B2,B5,LUP2   IF(I'.LT.L), LOOP2
     SB4      B4+1         K'=K'+1
     SX5      X5+B6        X5=X5+M
     LT       B4,B7,LUP3   IF(K'.LT.N), LOOP3
```

All that's left now is the initialization of registers. L, M, and N go into B5, B6, and B7.

```
     SA2      A1+3         X2=ADDRESS OF L
     SA3      A1+4         X3=ADDRESS OF M
     SA4      A1+5         X4=ADDRESS OF N
     SA2      X2           X2=L
     SA3      X3           X3=M
     SA4      X4           X4=N
     SB5      X2           B5=L
     SB6      X3           B6=M
     SB7      X4           B7=N
```

the address of matrix A goes into B1 and the address of matrix B goes into X5

```
     SB1      X1           B1=ADDRESS OF A MATRIX
     SA5      A1+1         X5=ADDRESS OF B MATRIX
```

and K' is initialized to zero,

```
     SB4      B0           K' = 0
```

Finally, we have to set A6 to one less than the address of C

```
SA1     A1+2        X1=ADDRESS OF C MATRIX
SA6     X1-1
```

This unfortunately clobbers whatever was in that location before, so we need the sequence

```
SA1     A1+2        X1=ADDRESS OF C MATRIX
SA1     X1-1
BX6     X1
SA6     A1
```

to preserve whatever was in that location.

With a sigh of relief we can now tack on the usual beginning and ending for our routine (which appears on the next page).

All this gory detail has been presented to give you some idea of what is involved in coding an efficient routine. Of course, things don't usually go as smoothly as was indicated here-- a half dozen revisions were needed before I came upon the efficient code developed here. On the other hand, such extreme care in saving instructions is not generally required in assembly language coding.

It is worth remarking here that it rarely makes any sense to write really inefficent assembly language code. If one doesn't bother to write efficient machine code, one might as well program in FORTRAN or some other high-level language to begin with, and spare oneself the problems of debugging assembly language programs. Similarly, it doesn't pay to write program sections that will be executed only a few times in assembly language, since the time you could save by producing highly optimized code would be infintesimal. An efficent programmer will resort to machine language coding only when it is worthwhile -- when the potential time saving justifies the effort.

```
            IDENT     MATMU
*********************************************************
*                                                       *
*    MATRIX MULTIPLY ROUTINE                            *
*    CALLING SEQUENCE: CALL MATMU (A,B,C,L,M,N)         *
*    MULTIPLES MATRIX A, DIMENSIONS L*M                 *
*    AND MATRIX B, DIMENSIONS M*N, LEAVES RESULT        *
*    IN MATRIX C, DIMENSIONS L*N                        *
*                                                       *
*********************************************************
            ENTRY    MATMU
    MATMU   BSS      1
            SA2      A1+3          X2=ADDRESS OF L
            SA2      A1+4          X2=ADDRESS OF M
            SA3      A1+5          X4=ADDRESS OF N
            SA2      X2            X2=L
            SA3      X3            X3=M
            SA4      X4            X4=N
            SB5      X2            B5=L
            SB6      X3            B6=M
            SB7      X4            B7=N
            SB1      X1            B1=ADDRESS OF A MATRIX
            SA5      A1+1          X5=ADDRESS OF B MATRIX
            SB4      B0            K'=0
            SA1      A1+2          X1=ADDRESS OF C MATRIX
            SA1      X1-1          INITIALIZE A6
            BX6      X1            WITHOUT CLOBBERING
            SA6      A1            LOCATION
      LUP3  SB2      B0            I'=0
      LUP2  SA1      B1+B2         X1=A(I,1)
            SA2      X5            X2=B(1,K)
            RX6      X1*X2         INITIALIZE SUM
            SB3      1             J'=1
      LUP1  SA1      A1+B5         X1=A(I,J)1
            SA2      A2+1          X2=B(J,K)
            RX1      X1*X2         X1=A(I,J)*B(J,K)
            RX6      X1+X6         ADD INTO SUM
            NX6      X6
            SB3      B3+1          J'=J'+1
            LT       B3,B6,LUP1    IF(J'.LT.M), LOOP1
            SA6      A6+1          STORE SUM INTO C(I,K)
            SB2      B2+1          I'=I'+1
            LT       B2,B5,LUP2    IF(K'.LT.N), LOOP3
            SB4      B4+1          K''=K'+1
            SX5      X5+B6         S5=X5+M
            LT       B4,B7,LUP3    IF(K'.LT.N), LOOP3
            EQ       MATMU         RETURN
            END
```

3.12 FLOATING POINT DIVISION

This section, covering the two floating point divide instructions, completes our discussion of the arithmetic instructions. A floating point division involves dividing coefficients and subtracting exponents; in decimal:

```
4.5  *  10**8 =  45.0  *  10**7
9.0  *  10**2 =   9.0  *  10**2
-------------   ---------------
                  5.0  *  10**5
```

In octal it is similarly necessary to increase the coefficient and correspondingly reduce the exponent before dividing:

```
                    63104000  00000000.  *  2**70
                    60100000  00000000.  *  2**6
```

is changed to

```
631044000  00000000  00000000  00000000.  *  2**10
                    60100000  00000000.  *  2**6
-----------------------------------------------
                   104000000  00000000.  *  2**2
```

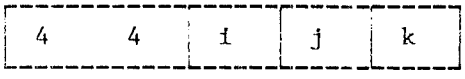
and, converting the coefficient to 48 bit form,

```
                  = 42000000  00000000.  *  2**3
```

If the operands are normalized, the exponent of the result = (exponent of dividend) - (exponent of divisor) - 57 (base 8) or 60 (base 8), depending on whether a one bit shift is required at the end of the operation, as it was above. Since the division process, as outlined above, yields a result with a 48 bit coefficient, there are only floating and round floating instructions. Double precision division is nonetheless possible, with a rather involved sequence of instructions. In the event of an inexact quotient (non-zero remainder), the floating divide gives a truncated quotient (the result of the remainder is ignored), while the round divide give an (approximately) rounded result.

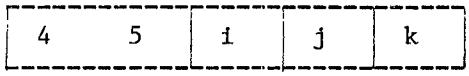
The two instructions are:

FXi XJ/Xk Floating Divide XJ by Xk to Xi



(15 bits)

RXi XJ/Xk Round floating divide Xj by Xk to Xi



(15 bits)

As the example above indicated, both instructions leave normalized results when both the dividend and divisor are normalized. On the other hand, if the divisor is not normalized the quotient conmputed may be wrong. In consequence, operands in these instructions should be normalized to insure correct results.

As an application of the floating divide, let us study a simple version of the Newton-Raphson square root:

```
      FUNCTION SQRT(X)
      IF (X.LT.0.) GO TO 2
      SQRT = X
      IF (X.EQ.0.) RETURN
1     SQRT = .5*(SQRT + X/SQRT)
      IF (ABS(SQRT**2 - X) .GT. X*1.E-12) GO TO 1
      RETURN
2     SQRT = 0.
      RETURN
      END
```

Our criterion for terminating the iteration is that SQRT**2 equals X to a precision of 1 part in 10**-12. We can rewrite the IF as:

```
      IF (SQRT**2 .LT. X - X*1.E-12) GO TO 1
      IF (SQRT**2 .GT. X + X*1.E-12) GO TO 1
```

This permits us to take the calculation of the limits out of the loop:

```
      FUNCTION SQRT(X)
      IF (X.LT.0) GO TO 2
      SQRT = X
      IF (X.EQ.0) RETURN
      SLIMIT1 = X - X*1.E-12
      SLIMIT2 = X + X*1.E-12
 1    SQRT = 0.5*(SQRT + X/SQRT)
      IF (SQRT**2.LT.SLIMIT1) GO TO 1
      IF (SQRT**2.GT.SLIMIT2) GO TO 1
      RETURN
 2    SQRT = 0.
      RETURN
      END
```

Converting this to assembly language is quite straightforward. First, check if the argument is negative:

```
        SA1     X1          X1=X
        NG      X1,NEG      SENSE ARGUMENT NEGATIVE
```

Second, set a first guess and compute limits:

```
        BX6     X1          X6=FIRST GUESS=X
        ZR      X1,SQRT     IF X=0,RETURN 0
        SA3     TINY        X3=10**(-12)
        RX3     X1*X3       X3=X*10**(-12)
        RX4     X1+X3       X4=UPPER LIMIT =X+X*10**(-12)
        NX4     X4
        RX5     X1-X3       X5=LOWER LIMIT=X-X*10**(-12)
        NX5     X5
        SA3     HALF        X3=0.5
```

SA3 HALF saves 0.5 so it will not have to be refetched each time in the loop.

```
   LUP  RX0     X1/X6       X0=X/LAST GUESS
        RX0     X0+X6       X0=X/LAST GUESS +LAST GUESS
        NX0     X0
        RX6     X0*X3       X6=NEW GUESS=0.5*X0
        RX0     X6*X6       X0=(NEW GUESS)**2
        RX2     X0-X5       X2=(NEW GUESS)**2-LOWER LIMIT
        NG      X2,LUP      IF VALUE TOO LOW,LOOP
        RX2     X4-X0       X2=UPPER LIMIT-(NEW GUESS)**2
        NG      X2,LUP      IF VALUE TOO HIGH, LOOP
        EQ      SQRT        EXIT
```

In comparing the result with the limits, we are only interested in the sign of the difference, and hence need not spend the time normalizing after the subtractions (normalization would be necessary if we were going to use the results or test whether they were zero). Two more instructions are needed to process the case where the argument is negative:

```
   NEG  SX6      B0        IF ARGUMENT NEGATIVE,
        EQ       SQRT       SET RESULT = 0 AND EXIT
```

Finally, we have to preset two words to 0.5 and 1.E-12. This is accomplished in COMPASS by the DATA pseudo-operation. This pseudo-op sets aside one word, and presets it to the constant in the address field. Thus, we would require:

```
    TINY DATA     1.E-12
    HALF DATA     0.5
```

Putting all this together, our routine would be:

```
        IDENT   SQRT
        ENTRY   SQRT
SQRT    BSS     1
        SA1     X1          X1 = X
        NG      X1,NEG      SENSE ARGUMENT NEGATIVE
        BX6     X1          X6 = FIRST GUESS = X
        ZR      X1,SQRT     IF X=0, RETURN
        SA3     TINY        X3=10**(-12))
        RX3     X1*X3       X3=X*10**(-12)
        RX4     X1+X3       X4=UPPER LIMIT=X+X*10**(-12)
        NX4     X4
        RX5     X1-X3       X5=LOWER LIMIT=X-X*10**(-12)
        NX5     X5
        SA3     HALF        X3=0.5
LUP     RX0     X1/X6       X0=X/LAST GUESS
        RX0     X0+X6       X0=X/LAST GUESS + LAST GUESS
        NX0     X0
        RX6     X0*X6       X6=NEW GUESS=0.5*X0
        RX0     X6*X6       X0=(NEW GUESS)**2
        RX2     X0-X5       X2=(NEW GUESS)**2-LOWER LIMIT
        NG      X2,LUP      IF VALUE TOO LOW,LOOP
        RX2     X4-X0       X2=UPPER LIMIT-(NEW GUESS)**2
        NG      X2,LUP      IF VALUE TOO HIGH, LOOP
        EQ      SQRT        EXIT
NEG     SX6     B0          IF  ARGUMENT  NEGATIVE, SET
        EQ      SQRT        RESULT=0 AND EXIT
TINY    DATA    1.E-12
HALF    DATA    0.5
        END
```

Let me conclude again with a comment on the importance of assembly language coding. Some functions, although in themselves very short, are used so often in so many programs that optimizing them becomes of prime importance. Chief among these are the FORTRAN library functions (SIN, COS, LOG, EXP, SQRT, etc.), where every attempt is made to squeeze the last microsecond out of the routine. Extensive studies have been made, for example, on the best starting value for Newton-Raphson square root iterations. In the case of the SQRT, a combination of an efficient algorithm and optimized machine language coding have resulted in a 6600 routine which computes a full-word precision (coefficient good to the last or next to the last bit) square root in 17-1/2 us. That is quite an accomplishment when one considers our effort above, which requires about that much time without any iterations!

3.13 ARITHMETIC EXIT

Let me begin by assuring you that trying to divide by zero does not cause the 6600 to blow a transistor. The divide unit senses that the divisor is zero, and, instead of going through the entire division process, puts out a special number as the quotient. For a positive number divided by minus zero, the result is:

3777 0000 0000 0000 0000,

a positive number with the largest possible exponent (real exponent = 1777 (base 8) = 1023 (base 10). A number with an exponent of 3777 (and any coefficient) is called "plus infinity", since we would ordinarily expect a non-zero number divided by zero to yeild infinity . A negative number divided by a plus zero or a positive number divided by minus zero produces

4000 0000 0000 0000 0000

the negative number formed by complementing the high 12 bits in plus infinity; note that the actual exponent is still 3777 (base 8). In case you haven't guessed, this number has been dubbed "minus infinity".

Dividing (plus or minus) zero by (plus or minus) zero is a special case yielding

1777 0000 0000 0000 0000

Considering its significance, this number is appropriately called "plus indefinite". There is also a "minus indefinite":

6000 0000 0000 0000 0000.

Indefinite has an actual exponent of minus zero, which cannot otherwise arise in normal computation. In contrast, it is possible to produce infinity without dividing by zero: for example, by multiplying two numbers whose product would have an actual exponent greater than 1777 . In such cases, known generally as "overflow", the 6600 automatically sets the result exponent to 3777 with the bias included.

The numbers infinity and indefinite are important because using either of them as an operand in a floating-point instruction causes the central processor to stop immediatelely (well, within a couple of microseconds, in any event). This feature, known as arithmetic error exit, is designed to avoid wasteful use of the

CPU, since infinite and indefinite results generally mean a program error. The operating system notices that your program has stopped, and puts out the dayfile message:

```
CPU ERROR EXIT AT 012345
ARITHMETIC INFINITE
```

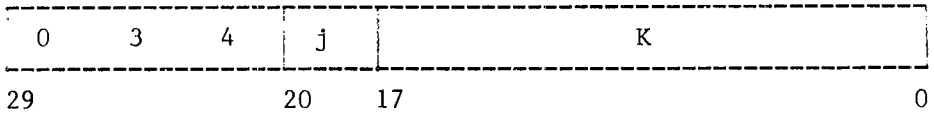
for using infinity as an operand, and

```
CPU ERROR EXIT AT 012345
ARITHMETIC INDEFINITE
```

for using indefinite as an operand. The address is the approximate location of the instruction which caused the error exit. Remember that "arith errors" are caused only by using these special numbers in floating point instructions.

To keep the accidental use of an illegal operand from throwing your program off the machine, there are four conditional branch instructions to test for infinity and indefinite. These are the four remaining X register branches, opcodes 034 through 037. Two test for infinity, also called "out of range" (i.e., outside of the range of legal exponents). These instructions only check whether the 12 high bits are 3777 or 4000; the coefficient of the number in the X register is ignored.

IR Xj, K Jump to K if Kj is in range (30 bits)



OR Xj, K Jump to K if Xj is out of range (30 bits)

Similarly, the definite/indefinite test only compares the high 12 bits against 1777 and 6000.

DF Xj, K Jump to K if Xj is definite (30 bits)

ID Xj,K Jump to K if Xj is indefinite (30 bits)

As an example of the use of these instructions, let us modify our square root routine to check for all illegal arguments: infinity, indefinite and negative numbers. For any of these arguments, the routine will return an indefinite (just as the FORTRAN library routine does). Returning indefinite rather than zero for negative arguments prevents the program from wasting CPU time if it tries to use the result of the illegal SQRT call in further computation. Thus, our improved square routine:

```
        IDENT     SQRT
        ENTRY     SQRT
SQRT    BSS       1
        SA1       X1          X1 = X
        NG        X1,ILL      SENSE ARGUMENT NEGATIVE
        OR        X1,ILL              INFINITE,
        ID        X1,ILL            OR INDEFINITE
        BX6       X1          X6 = FIRST GUESS = X
        ZR        X1,SQRT     IF X = 0, RETURN 0
        SA3       TINY        X3 = 10** (-12)
        RX3       X1*X3       X3 = X*10**(-12)
        RX4       X1+X3       X4 = UPPER LIMIT = X+X*10**(-12)
        NX4       X4
        NX5       X5
        SA3       HALF        X3 = 0.5
LUP     RX0       X1/X6       X0 = X/LAST GUESS
        RX0       X0+X6      X0 = X/LAST GUESS + LAST GUESS
        NX0       X0
        RX6       X0*X3       X6 = NEW GUESS = 0.5*X0
        RX0       X6*X6       X0 =(NEW GUESS)**2
        RX2       X0-X5      X2 =(NEW GUESS **2-LOWER LIMIT
        NG        X2,LUP      IF VALUE TOO LOW, LOOP
        RX2       X4-X0       X2 = UPPER LIMIT-(NEW GUESS)**2
        NG        X2, LUP     IF VALUE TOO HIGH, LOOP
        EQ        SQRT        EXIT
ILL     SA1       IND         IF ARGUMENT INVALID,
        BX6       X1          RETURN INDEFINITE RESULT
        EQ        SQRT        AND EXIT
TINY    DATA      1.E-12
HALF    DATA      0.5
IND     DATA      1777000000000000000B
        END
```

As the size of our routine increases, we note that the number of constants we have to keep track of and put at the end of the routine grows too. We might compare our current situation to writing FORTRAN programs under the restriction that constants may appear only in DATA statements. If we wanted to add 3.1 to X in this restricted FORTRAN, we would have to write

```
DATA CON3P1 /3.1/
X = X + CON3P1
```

Of course, FORTRAN does allow us to write

```
X = X + 3.1
```

the compiler will automatically set aside a word at the end of the routine and initialize it to 3.1.

The COMPASS assembler also provides a facility for automatically generating constants, called the literal. For example, if we include in our routine

```
SA3    =0.5
```

COMPASS will automatically generate a

```
DATA   0.5
```

at the end of the routine, and put the address of the DATA instruction in the K portion of the SA3 instruction. The assembler checks for duplicate literals; even if =0.5 is used in several instructions, only one DATA 0.5 will be generated.

Using literals, we can shave a few lines off our SQRT routine without changing the generated code one whit: (see next page.)

Before performing our own exit from this discussion of arith errors, it is worth mentioning that the approach taken on the 6600 to check for division by zero, exceeding the range of exponents, etc., is not the method used on most computers. Computer designers have generally favored terminating the program when an arithmetic fault (such as division by zero or exceeding the range of exponents) occurs, rather than waiting until the result of such an operation is used later as an operand. The 6600 scheme has a definite disadvantage in comparison: when an arith error occurs in a large program, it may be difficult to find the instruction which generated the infinite or indefinite. This has been rectified on the 7600, which provides for optional program termination when an infinite or indefinite is generated.

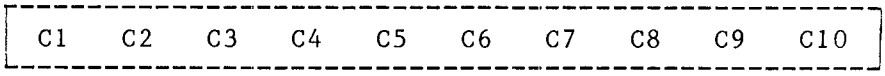
```
        IDENT     SQRT
        ENTRY     SQRT
SQRT    BSS       1
        SA1       X1        X1=X
        NG        X1,ILL    SENSE ARGUMENT NEGATIVE,
        OR        X1,ILL              INFINITE
        ID        X1,ILL            OR INDEFINITE
        BX6       X1        X6=FIRST  GUESS = X
        ZR        X1,SQRT   IF X=0,RETURN 0
        SA3       =1.E-12
        RX3       X1*X3     X3=X*10**(-012)
        RX4       X1+X3     X4=UPPER LIMIT=X+X*10**(-12)
        NX4       X4
        RX5       X1-X3     X5=LOWER LIMIT=X-X*10**(-12)
        NX5       X5
        SA3       =0.5
LUP     RX0       X1/X6     X0=X/LAST GUESS
        RX0       X0+X6
        NX0       X0        X0=X/LAST GUESS+LAST GUESS
        RX6       X0*X3     X6=NEW GUESS=0.5*X0
        RX0       X6*X6     X0=(NEW GUESS)**2
        RX2       X0-X5     X2=(NEW GUESS)**2-LOWER LIMIT
        NG        X2,LUP    IF VALUE TOO LOW, LOOP
        RX2       X4-X0     X2=UPPER LIMIT-(NEW GUESS)**2
        NG        X2,LUP    IF VALUE TOO HIGH, LOOP
        EQ        SQRT      ELSE DONE -- EXIT
ILL     SA1       =1777000000000000000000B
        BX6       X1        IF ARGUMENT INVALID,
        EQ        SQRT      RETURN INDEFINITE RESULT
        END
```

3.14 CHARACTER MANIPULATION

It is the task of every good computer manual to constantly reassure the student, in an effort to keep him from the realization that, by the time he has finished the twelfth volume and is an expert machine programmer, the machine is obsolete and he will have to start all over again with a new model. In accordance with this policy, I can reassure you that this section is not concerned with brain-washing.

A large portion of the data processed by the computer is text--strings of characters, rather than numeric data items. Source programs are text; in fact, anything coming in on Hollerith punched cards or going out onto the printer is at some point processed as text. Numeric data read in from cards must first be converted from a sequence of digits and decimal points to numbers in binary representation; in FORTRAN programs this is done automatically by the format-directed input processing routine (KRAKER).

Different computer applications require different sets of characters. The FORTRAN character set for the 6600 has 47 characters: 26 letters, 10 digits, and the "special characters" + - * / ( ) $ = , . and the blank. A set of six bits is required to represent these 47 characters; thus a 6600 word can hold 10 characters



For some applications a set with more than 47 characters is desirable. With six bits for each character, we can have up to 64 characters. The SCOPE system accepts cards punched in a 64 character set (adding the characters: ≡ [ ] : ≠ → < > ≤ ≥ ∧ ∨ ↑ ↓ % ¬ ; to those required for FORTRAN). These added characters are used, for example, for some of the special features of COMPASS. Some computer systems, such as the IBM System/360, allocate 8 bits for a character, thus permitting a very large character set, including upper and lower case letters.

The code used on the 6600 for associating a six-bit number with a character is called display code, because the computer display console, when it is being fed data by a PP, can display characters in accordance with this code. The letters are represented by the first 26 numbers, 01 (base 8) to 32 (base 8); the digits by the next 10, 33 (base 8) to 44 (base 8); the blank and special characters are matched up with 45 (base 8) to

77 (base 8) and 00 (the code is tabulated on the page following). The end of a card or a line is indicated by zeroes in bits 0 through 11 of a word. For example, the card

```
7
PROGRAM HOHUM(INPUT, OUTPUT)
```

would be stored in memory as

```
55 55 55 55 55 55 20 22 17 07          PROG
11 01 15 55 10 17 10 25 15 55      RAM HOHUM
51 11 16 20 25 24 56 17 25 24      (INPUT,OUT
20 25 24 52 55 55 55 55 00 00      PUT)
```

in four CM words.

Techniques for character manipulation are generally applicable to the wider class of problems involving the separate handling of portions of a computer word. The fundamental capability required in all these problems which we have not yet considered is that of shifting around the bits in a word. Two different kinds of shifts are available on the 6600: a left circular shift and an arithmetic right shift.

A left circular shift one bit takes the contents of each bit position and moves it to the next position on the left; the bit at the left end goes into the rightmost bit position. For example,

100110100000

shifted left circular one bit becomes

001101000001

A left shift n bits is simply the repetition of this process n times; for example, our original 12 bit number, shifted left circular four becomes

101000001001

shifted left twelve,

100110100000

we get back the same number (each bit has gone once all the way around). The 6600 instruction to shift an X register left circular is (the text continues following the table below)

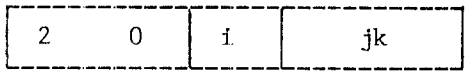
TABLE OF DISPLAY CODES

CHARACTER	CODE	CHARACTER	CODE
:	00 **		
A	01	0	33
B	02	1	34
C	03	2	35
D	04	3	36
E	05	4	37
F	06	5	40
G	07	6	41
H	10	7	42
I	11	8	43
J	12	9	44
K	13		
L	14	+	45
M	15	-	46
N	16	*	47
O	17	/	50
P	20	(	51
Q	21	)	52
R	22	$	53
S	23	=	54
T	24	(blank)	55
U	25	,	56
V	26	.	57
W	27	≡	60
X	30	[	61
Y	31	]	62
Z	32	%	63
		≠	64
		→	65
		∨	66
		∧	67
		↑	70
		↓	71
		<	72
		>	73
		≤	74
		≥	75
		¬	76
		;	77*

*Do not use the semicolon in COMPASS instructions

**Some installations may use a 63-character set, in which display code 63 is ":", and "00" is only used to indicate end-of-file.

LXi jk Left shift Xi, jk places



(15 bits)

The address field is a single number, the shift count, from 0 to 63 (shifting 61 places has the same effect as shifting left one place).

An arithmetic right shift one bit moves the contents of each bit position one place to the right. The rightmost bit, however, just "falls off" and is discarded; instead, the high order bit keeps its old value. In effect, as the number is shifted to the right, the bits vacated on the left end are filled up with the sign bit; thus, this process is called sign extension. Starting with the same 12 bit number, a right shift two yields

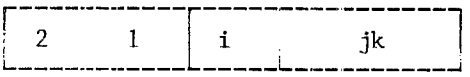
111001101000

and a right shift twelve leaves the sign bit in every position:

111111111111

The 6600 instruction for arithmetic right shifting an X register is

AXi jk Arithmetic right shift Xi, jk places



(15 bits)

Again, the shift count (bits 0 to 5) can be 0 to 63, a count of 60 or more leaving 60 copies of the original sign bit in Xi.

Observe that right shifting an integer one bit divides the integer by two. Similarly, a left circular shift one place multiplies the integer by two (unless the range of possible integers is exceeded); left shifting a negative number brings a one into the low order bit, just as the end around carry would if the number were added to itself. For example,

111001111010 = -605 (base 8) = -389 (base 10)

doubled:

110011110101 = -1412 (base 8) = -778 (base 10)

halved:

111100111101 = -302 (base 8) = -194 (base 10)

A devious absolute value function will illustrate use of the right shift. Basically, all that is necessary for an absolute value is to complement the number if it is negative. In straightforward code, to put the absolute value of X1 in X6

```
        BX6     X1
        PL      X6,NEXT
        BX6     -X6
NEXT    (next instruction)
```

Now consider

```
        BX2     X1
        AX2     60
        BX6     X1-X2
```

After the AX2 60, X2 contains in each bit the sign bit of the number; in other words, all zeros if X1 was positive, all ones if X1 was negative. So, if X1 was positive, the last instruction logically subtracts all zeros from the number, which leaves it unaltered, while, if it was negative, a logical difference of X1 with all ones is performed, which complements the number. Though both sequences require three instructions, the right shift takes much less time than the branch instruction, so the latter code is preferred.

An an example of the use of the left shift, consider the task of taking ten characters, stored in the low six bits of ten consecutive computer words (upper 54 bits zero), and packing them into one word. This may be accomplished by putting the first character into the low bits of a register, shifting it left six bits, entering the next character into the low six bits, shifting both left six, etc. until all ten characters have been packed. In machine code this is realized as

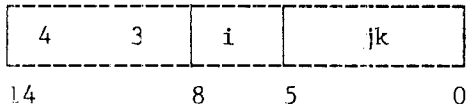
```
        SB1     B0          CHARACTER COUNT=0
        SB2     10          NUMBER OF CHARACTERS TO PACK
        SX6     B0
LUP     SA1     B1+CHAR     LOAD NEXT CHARACTER
        LX6     6           SHIFT PREVIOUSLY LOADED CHARS
        BX6     X6+X1       OR IN NEXT CHARACTER
        SB1     B1+1        INCREMENT CHARACTER COUNT
        LT      B1,B2,LUP   IF MORE TO PACK, LOOP
```

Then ten characters are expected to be in the array CHAR, and the packed result will be left in X6. If, say, the characters were A B C D E F G H I J, X6 will be, after successive iterations of the loop:

```
00000000000000000001,
00000000000000000102,
00000000000000010203, . . . and finally
01020304050607101112.
```

We can now reverse the process, and write a similar loop to unpack the characters. For the sake of variety, let us unpack them into the high order six bits of ten consecutive words. Each time through the loop, then, we want to "mask out" and store the leftmost character, and then shift the word left six bits, so the proper character is in position the next time around. Masking out the leftmost character involves taking the logical product of the packed word with an appropriate mask, in this case 77000000000000000000 (base 8). The high six bits of the product will be those bits from the packed word, since anding a bit with a one leaves the bit unchanged, while the low 54 bits will be zero, as the logical product of anything with zero is zero. So our last problem is to get 77000000000000000000 (base 8) into a register. We could, of course, load it from memory. The 6600 designers, however, have provided us with an instruction for just such occasions:

MXi jk Form mask in Xi, jk bits



(15 bits)

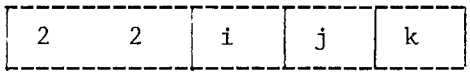
which sets the high order jk bits of Xi to one, and the rest to zero (if jk=0, the register is set to zero). In our example, we'll use a MX5 6 to generate our mask:

```
        SB1     B0          CHARACTER COUNT=0
        SB2     10          NUMBER OF CHARACTERS TO UNPACK
        MX5     6           FORM ONE-CHARACTER MASK
LUP     BX6     X1*X5       MASK OUT ONE CHARACTER
        SA6     B1+CHAR     STORE IN MEMORY ARRAY
        LX1     6           SHIFT NEXT 6 CHAR. TO HIGH BITS
        SB1     B1+1        INCREMENT CHARACTER COUNT
        IT      B1,B2,LUP   IF MORE TO UNPACK, LOOP
```

where the packed word is assumed to be in X1 at the start. If we had needed a low six bit mask, we could have simply used SX5 77B (a 30 bit instruction) or, more efficiently, set X5 to 7777777777777777700(base 8) with an MX5 54 (a 15 bit instruction), and then used a BX6 -X5*X1 instead of BX6 X1*X5.

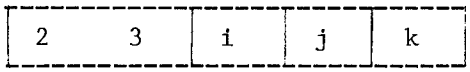
As a final complication, we shall modify our character packing routine to stop packing if it encounters a special character or a blank -- in other words, anything with a display code of 45 (base 8) or above (for simplicity, we will not stop packing at a colon, display code 00). At the same time, we shall insist that the characters be left justified; i.e., that the first character be in the leftmost six bits. Thus, whenever we are finished packing characters, we will have to figure out how many characters have been packed, and shift the word accordingly. The two shift instructions we have studied so far, however do not permit us in any simple way to use the contents of a register as the shift count. We shall rectify this presently by introducing the last two shifts, which take their shift counts from B registers:

LXi Bj, Xk Left shift Xk nominally Bj places to Xi



(15 bits)

AXi Bj, Xk Arithmetic right shift Xk nominally Bj places to Xi



(15 bits)

If Bj is positive, these instructions act just like the corresponding shifts described earlier, with the low six bits of Bj taken as the shift count. (In the realm of 6600 trivia we may note that if any of bits 6 through 10 of Bj are non-zero, the nominal right shift will, instead of performing the shift, set Xk to zero.) If Bj is negative, each instruction acts as the other would with the complement of Bj. That is, a nominal left shift with -5 in the B register causes an arithmetic right shift 5, while a nominal right shift with -5 produces a circular left shift 5 places. Hence the term "nominal": opcode 22 is not intrinsically any more a left shift than a right shift but, in order to distinguish the two shifts, we designate each by their effect with a positive B register.

To stop the packing process when a blank or special character is encountered requires two additional instructions:

```
        SB1         B0          CHARACTER COUNT
        SB2         10          NUMBER OF CHARS TO PACK
        SX6         B0
LUP     SA1         B1+CHAR     LOAD NEXT CHARACTER
        SX2         X1-45B
        PL          X2,DONE     IF BLANK OR SPECIAL, DONE
        LX6         6
        BX6         X6+X1       ELSE PACK CHAR INTO WORD
        SB1         B1+1        INCREMENT CHAR COUNT
        LT          B1,B2,LUP   IF MORE TO PACK, LOOP
DONE
```

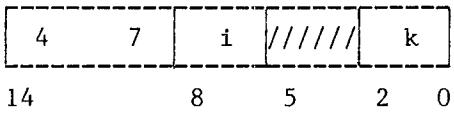
At DONE will go an instruction to shift X6 into position; if we are keeping the appropriate count in B3, we can use an LX6 B3,X6. B3, then, will have to be a count of the number of unfilled bits, starting at 60 and decreasing by 6 each time through the loop:

```
        SB1         B0          CHARACTER COUNT=0
        SB2         10          NUMBER OF CHARS TO PACK
        SB3         60          INITIALIZE SHIFT COUNT
LUP     SA1         B1+CHAR     LOAD NEXT CHARACTER
        SX2         X1-45B
        PL          X2,DONE     IF BLANK OR SPECIAL, DONE
        LX6         6
        BX6         X6+X1       ELSE PACK CHARACTER INTO WORD
        SB1         B1+1        INCREMENT CHARACTER COUNT
        SB3         B3-6        DECREMENT SHIFT COUNT
        LT          B1,B2,LUP   IF MORE TO PACK, LOOP
DONE    LX6         B3,X6       LEFT JUSTIFY PACKED WORD
```

The left shift is unnecessary in the event that ten characters are packed and we "fall through" the loop, but putting the left shift at the end of the loop makes the code simpler (if ten characters were packed, B3=0, so an LX6 0 is effectively performed).

Lest we omit discussing any of the instructions, let me make mention here of the "count ones" instruction. This rarely used instruction counts the number of one bits in an X register and places the number, between 0 and 60, in another X register:

CXi Xk Count of number of "1's" in Xk to Xi



(15 bits)

This instruction is of use when binary data, such as yes-no responses from questionaires, are stored one datum per bit rather than one datum per word (as they would be in a FORTRAN type LOGICAL array) so that sixty times as much informatioon can be stored in a given block of memory. A count ones instruction may then be used to determine the total number of yes responses (1 bits) in a word.

3.15 INTEGER MULTIPLICATION AND DIVISION

As we mentioned earlier, the original 6000 and 7000 series machines had no single instructions for integer multiplication and division. These operations were performed by converting the operands to floating point, executing a floating multiply or divide, respectively, and then converting the result back to an integer. Control Data subsequently realized that by making a fairly simple change in the floating multiply instruction it would be possible to perform integer multiplies without the conversions. This modified instruction is included in all recent 6000 and Cyber series, and has been installed as a "field change" in most earlier machines.

To begin this section, however, we will consider the situation before the field change, when integer multiplicatipon as well as division had to be done by conversion to and from floating point. Because of the frequency of these operations, two instructions have been included for conversion between integer and floating point.

Consider converting an integer to floating point: all that is necessary is to put a biased exponent of zero in the high twelve bits;

for example: 0000 0000 0123 0004 7621

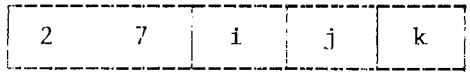
in floating point is: 2000 0000 0123 0004 7621

while its complement: 7777 7777 7654 7773 0156

is in floating point: 5777 7777 7654 7773 0156

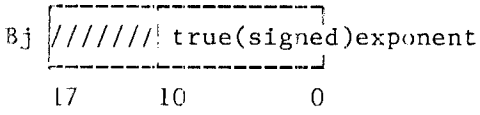
This operation can be performed in one instruction with the pack instruction:

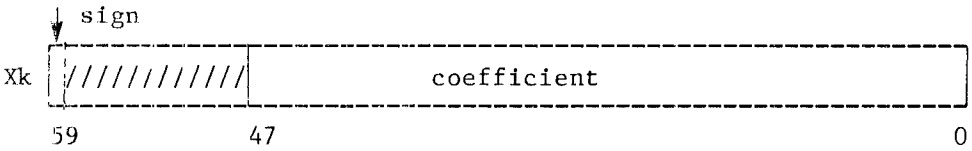
PXi Bj,Xk Pack Xi from Xk and Bj

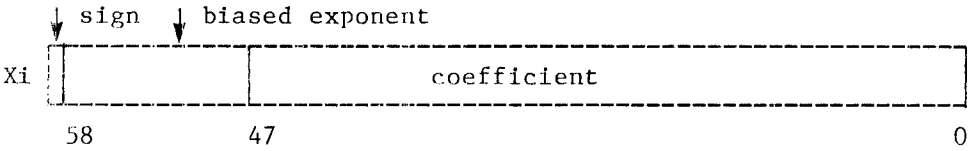


(15 bits)

This instruction takes the coefficient from Xk and the true exponent of the number from Bj and "packs" them together into a floating point number in Xi. In the usual case where a true exponent of zero is desired, one need merely specify B0: PXi Xk. The instruction may be diagrammed:







For example, if

Bj = 000010 and Xk = 0000 0000 4215 7123 6661

then Xi = 2010 0000 4214 7023 6661

If Bj = 777766 and Xk = 0635 0210 0011 0000 0003

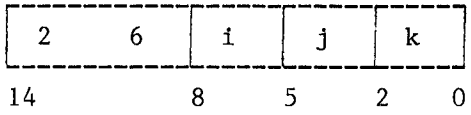
then Xi = 1766 0210 0011 0000 0003

Note, that in the last example that only the low 48 bits of Xk are used for the coefficient, so packing a number greater than 2**48 - 1 will give an incorrect floating point conversion.

If the resulting floating point number will be used in normal floating point calculations, it should be normalized before being used or stored. If, however, it is to be used in one of the special integer arithmetic sequences to be described below, it may be unnecessary or incorrect to normalize it.

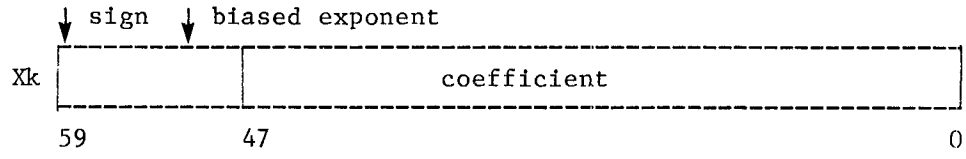
Floating to integer conversion is slightly more involved. The first step is performed by an unpack instruction, which does precisely the reverse of the pack instruction:

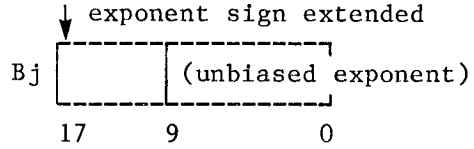
UXi Bj,Xk Unpack Xk to Xi and Bj

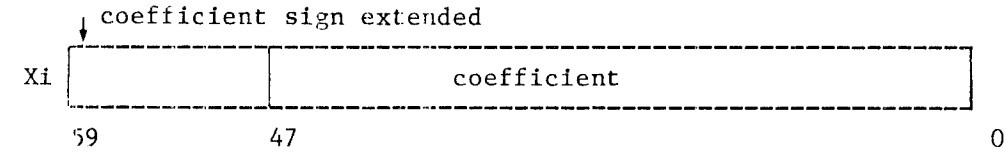


(15 bits)

This instruction unbiases the exponent from the floating point number in Xk and puts it in Bj, while sending the 48 bit 4oefficient to Xi:







For example, 1.5 =

Xk = 1720 6000 0000 0000 0000

is unpacked to

Bj = 777720 Xi = 0000 6000 0000 0000 0000

while an unnormalized 24. =

Xk = 2003 0000 0000 0000 0003

is unpacked into

Bj = 000003 Xi = 0000 0000 0000 0000 0003.

The second step is to convert this integer with exponent to a simple integer; for example, in the latter case we would want to multiply Xi=3 by 2**3 to get 24 as an integer. Now what is the fastest way to multiply an integer by 2**n? Shift it left n places. Thus we would want a sequence such as

```
UX1     B7,X1
LX1     B7,X1
```

to convert X1 from floating to integer. In the second example, X1 would be shifted left 3, leaving X1=30(base8)=24. In the first case, the shift count B7=-57(base 8), so the nominal left shift performs a right shift 57(base 8), yielding X1=1, the proper result of a floating to integer conversion of 1.5. If the floating point number is too large to be stored as an integer (magnitude>2**59 - 1) the left shift will rotate the coefficient around, causing totally erroneous results; since only the low six bits are used as a shift count, for example, an exponent of 101(base 8) is treated as an exponent of 1. On the other hand, if the floating point number is less than 1, the result of the integer conversion will always be zero. In particular, if the magnitude of a negative exponent (shift count) is more than 77(base 8), the result register will automatically be set to zero by the shift instruction (remember that bit of trivia concerning the nominal shift instruction?).

Now that we are thoroughly versed in integer-floating conversion, integer multiplication and division should be simple matters. Suppose we want to put into X6 the product of the integers in X1 and X2. We first convert the numbers to floating point:

```
          PX3       X1
          PX4       X2
```

Second, we multiply X3 and X4 together; to see which multiply instruction we require, consider the product of two sample numbers:

if X1 = 12(base 10) and X2 = 20(base 10), then

X3 = 2000 0000 0000 0000 0014 (=0000 0000 0000 0014 * 2**0)

X4 = 2000 0000 0000 0000 0024 (=0000 0000 0000 0024 * 2**0)

so the product is

```
                          0000 0000 0000 0014 * 2**0
                          0000 0000 0000 0024 * 2**0
      ---------------------------------------------
      0000 0000 0000 0000  0000 0000 0000 0360 * 2**0

     |<-floating product->|<---- DP product-->|
```

So clearly we want the DP product

```
          DX6       X3*X4
```

Finally, we have to convert the result back to integer; note, however, that the DP product of two numbers with true exponents of zero also has a true exponent of zero, so all we need is

```
          UX6       X6
```

"discarding" the exponent into B0. Thus, the sequence for an integer multiply is: pack both operands, DP multiply, unpack result.

Note that we do not normalize the operands before multiplying, and that the sequence of instructions described above would in fact not work if either of the operands were normalized prior to the multiplication. Also, the DP multiply will only recover the low 48 bits of the product; a more complicated sequence would be needed to recover the entire 96 bits. To save execution time, the FORTRAN compilers generate code to compute only the low 48

bits of an integer product, rather than compile the extra instructions necessary to get a 60 bit product (one clearly cannot store more than 60 bits into an integer variable).

If we now wished the integer quotient of X1 divided by X2 in X6, we would begin as before, packing the operands:

```
PX3     X1
PX4     X2
```

Before doing the division, however, we have to normalize the divisor, since, as was mentioned earlier, the divide instruction may otherwise produce an incorrect quotient:

```
NX4     X4
```

As division only generates a 48 bit quotient, there is no problem deciding which divide instruction to use; since we want the quotient to be truncated (rather than possibly rounded) we use a floating divide:

```
FX6     X3/X4
```

Lastly, we convert the result back to integer; inasmuch as we can't predict the exponent of the result, we have to unpack and shift the result:

```
UX6     B7,X6
LX6     B7,X6
```

To summarize the situation before the field change:

integer product	X1*X2 to X6	integer quotient	X1/X2 to X6
PX3	X1	PX3	X1
PX4	X2	PX4	X2
DX6	X3*X4	NX4	X4
UX6	X6	FX6	X3/X4
		UX6	B7,X6
		LX6	B7,X5

The change incorporated into recently manufactured machines, and made in the field to most earlier machines, modifies the operation of the double precision multiply instruction, DXi Xj*Xk. If an integer smaller than 2**48 is used as an operand to

a floating point instruction, it is treated as a floating point number with a biased exponent of 0 and hence a true exponent of -1777(base 8). Before the double precision multiply was changed, if both operands of the instruction were integers less than 2**48, the machine would compute a true exponent of -3776(base 8) for the product. Since a floating point number that small cannot be represented in 6600 floating point format, the instruction would return a zero result. The new, modified instruction checks for the condition where the high order 12 bits of both operands are all zeros or all ones (corresponding to a biased exponent of zero in a positive or negative number). In this case, the instruction returns the low 48 bits of the product of the coefficients in the low 48 bits of the result register, with the sign of the result extended into the high 12 bits. In short, if both operands and result are less than 2**48, the double precision multiply can be used as an integer multiply instruction, without packing and unpacking. Integer divides must still be performed by the six-instruction sequence given above.

To commemorate this change, a new instruction mnemonic has been added to COMPASS in version 3:

```
IXi     Xj*Xk
```

This line is assembled into the 15 bit instruction 42ijk, precisely like Dxi Xj*Xk.

Before using opcode 42 for doing integer multiplies, of course, you should check that your machine does have the integer multiply feature installed. In the example of integer multiply given below, we shall use the 4-instruction sequence for the multiply which, while less efficient, will work on all machines.

To illustrate the use of these integer operations we shall code a routine which converts an integer to the series of display code characters which is the decimal representation of the number. This is essentially the same task that is performed by the FORMAT-directed output encoding routine (KODER) when it is converting a number for output according to I format. The routine will be called with two arguments: the number to be converted, and the array into which the digits should be stored. We will agree only to process numbers whose magnitude is less than 2**48; any larger integer will be considered out of range, designated by the letter R in the character string generated by the routine. The algorithm we shall use is similar to those described earlier, in the "Base Conversion Algorithms": divide the absolute value of the number repeatedly by 10 until the quotient is zero; the remainders from the successive divisions

will be the decimal digits, the least signigicant digit coming out first. If the number was negative, a minus sign must be tacked on in front. We will assume that the character array, transmitted as a parameter to our routine, will be 20 words long; storing the digits one to a word, right justified, we can than be sure we will never run out of space (2**48 is a 15 digit decimal). However, we will not assume the array was blank to start, and so the routine's last job will be to fill out the unused portion of the array with blanks. The routine will have to

(1) compute the absolute value of the number
(2) check whether the number is in range
(3) generate the decimal digits by repeated division by 10
(4) add on a minus sign if the number was negative
(5) fill out the character array with blanks

Let us attack these tasks in order. In the previous section we saw an efficient technique for obtaining the absolute value of a number:

```
SA2     A1+1      X2=ADDRESS TO STORE RESULT
SB2     X2        B2=ADDRESS TO STORE RESULT
SA1     X1        X1=NUMBER TO BE CONVERTED
BX2     X1
AX2     60
BX2     X1-X2     X2=ABSOLUTE VALUE (NUMBER)
```

The number will be out of range if any of the high twelve bits in the absolute value of the number are non-zero. This can be checked for by shifting the number right 48 bits (so only the top twelve are left) and then testing for zero:

```
BX3     X2
AX3     48
NZ      X3,RANGE   SENSE OUT OF RANGE
```

Before diving into the loop, we should initialize a few registers. A pointer is needed to inform the routine where to store the next digit; since the first digit will be stored in the last array element, at B2+19, we preset a counter to 19;

```
SB3     19
```

We shall compute remainders as is done in FORTRAN with the expression remainder = NUMBER - (NUMBER/10*10). Since the number 10 is used twice in the loop (once to divide, once to multiply), we certainly will want to put it in a register outside the loop. Also, we need two forms of the number 10: an unnormalized floating point form for the multiply, and a normalized form for the divide. We shall keep one in X4, and the other in X5:

```
        SX4     10
        PX4     X4          X4=UNNORMALIZED  FL  PT  10
        NX5     X4          X5=NORMALIZED FL  PT  10
```

The loop begins by computing NUMBER=NUMBER/10*10:

```
NEXT    PX3     X2
        FX3     X3/X5
        UX3     B7,X3
        LX3     B7,X3       X3=NUMBER/10
        PX0     X3
        DX0     X0*X4
        UX0     X0          X0=NUMBER/10*10
        IX6     X2-X0       X6=NUMBER-NUMBER/10*10
```

Observe that we have been careful to preserve X3, since it will be the new value of "NUMBER" the next time through the loop. The decimal digit, now in X6, is converted to display code by adding 33(base 8), and is stored in the character string:

```
        SX6     X6+33B      CONVERT DIGIT TO DISPLAY CODE
        SA6     B2+B3       STORE CHARACTER
        SB3     B3-1        RESET CHARACTER POINTER
```

Each time a character is stored, the pointer is decremented by one, since the next digit goes into the previous array element. Finally, NUMBER/10 is put into X2 for the next loop iteration, and, if the quotient is not zero, the loop is repeated.

```
        BX2     X3          X2=NEW NUMBER=OLD NUMBER/10
        NZ      X2,NEXT     IF .NE.0, CONTINUE LOOPING
```

Next, we store a minus sign (display code 46 (base 8))in front of the number if it was negative:

```
        PL      X1,FILL
        SX6     46B         IF NUMBER NEGATIVE
STOR    SA6     B2+B3          STORE MINUS SIGN
        SB3     B3-1
FILL    ...
```

The label next to the store instruction anticipates a need we will see in a moment.

The final loop continues storing blanks (display code 55(base 8)) until B3 goes negative, i.e., the beginning of the array has been passed:

```
FILL  SX6      55B
DUN   NG       B3,CDC      IF ARRAY FULL, EXIT
      SA6      B2+B3       IF NOT, STORE A BLANK,
      SB3      B3-1          MOVE CHARACTER POINTER
      EQ       DUN               AND LOOP
```

Finally, for the out-of-range case, we want to store an R (display code 22(base 8), and then fill the array with blanks:

```
RANGE SX6      22B         IF OUT OF RANGE,
      SA6      B2+B3          STORE AN R
      SB3      B3-1
      EQ       FILL
```

If you are reasonably observant, you have noticed that we can save two instructions:

```
RANGE SX6      22B         IF OUT OF RANGE,
      EQ       STOR           STORE AN R
```

However, unless you are unusually perspicacious (more so than I was the first time I ran the program) you have not noticed that the SB3 19 came after the NZ X3,RANGE, so that, if the number was out of range and B3 was, say, 10 000, the routine could innocently wipe out your entire program. To rectify this error the SB3 19 has to be moved up a few instructions.

To add a crowning touch to this example, let us now dub it subroutine CDC (for convert to display code, of course), and show how such a routine might be usefully applied.

```
        IDENT     CDC
        ENTRY     CDC
CDC     BSS       1
        SA1       B1          X1=NUMBER
        BX2       X1
        AX2       60
        BX2       X1-X2       X2=ABSOLUTE VALUE(NUMBER)
        SB3       19
        BX3       X2
        AX3       48
        NZ        X3,RANGE    SENSE OUT OF RANGE
        SX4       10
        PX4       X4          X4=UNNORMALIZED FL  PT  10
        NX5       X4          X5=NORMALIZED FL PT 10
NEXT    PX3       X2
        FX3       X3/X5
        UX3       B7,X3
        LX3       B7,X3       X3=NUMBER/10
        PX0       X3
        DX0       X0*X4
        UX0       X0          X0=NUMBER/10*10
        IX6       X2-X0       X6=NUMBER-NUMBER/10*10
        SX6       X6+33B      CONVERT DIGIT TO DISPLAY CODE
        SA6       B2+B3       STORE CHARACTER
        SB3       B3-1        RESET CHRACTER POINTER
        BX2       X3          X2=NEW NUMBER=OLD NUMBER/10
        NZ        X2,NEXT     IF .NE.0 CONTINUE LOOPING
        PL        X1,FILL
        SX6       46B         IF NUNMBER NEGATIVE
STOR    SA6       B2+B3           STORE MINUS SIGN
        SB3       B3-1
FILL    SX6       55B
DUN     NG        B3,CDC      IF ARRAY FULL, EXIT
        SA6       B2+B3       IF NOT, STORE A BLANK
        SB3       B3-1        MOVE CHRACTER POINTER
        EQ        DUN                 AND LOOP
RANGE   SX6       22B          IF OUT OF RANGE,
        EQ        STOR            GO STORE AN R
        END
```

```
      PROGRAM TRUTH(OUTPUT)
      DIMENSION CHAR(20)
      CALL CDC (14710B,CHAR)
      PRINT 100,CHAR
100   FORMAT(18H1EVERYBODY LOVES A, 20R1)
      CALL EXIT
      END
```

which prints the line:

EVERYBODY LOVES A 6600

3.16 COMPARE AND MOVE

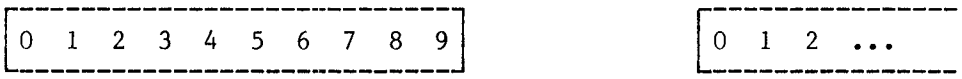
Operations on packed character strings are relatively cumbersome on the 6600. Ten instructions are required to extract the 7th through 14th characters of a 20-character (2-word) string and store them in some other word. Comparing two 20-character strings to determine which comes first in alphabetical order is a formidable task, for reasons to be explained later; perhaps 50 or 60 instructions are required on a 6600. These operations are so complex because the 6600 has no instructions for operating on units of data smaller than a 60-bit word. In contrast, most other large modern computers provide instructions for manipulating individual characters within a word. As long as the 6600 was intended primarily for running FORTRAN programs, this was not a major drawback; character manipulation generally plays a minor role in scientific calculation. When Control Data set its sights on capturing part of the large business market, however, the relatively slow character manipulation was a disadvantage, since commercial applications involve primarily the processing of large files of data, mostly in packed character format.

As a result, when Control Data decided to reincarnate the 6000 series under the name Cyber 70, they included in some of the models four new instructions for character manipulation. These four instructions are performed by a box added onto the old 6000 series central processors and dubbed the Compare and Move Unit, or CMU. The CMU is standard equipment on the Cyber 70 model 72 and 73 and the Cyber 170 models 172, 173, 174, 720, and 730; it is optionally available on the Cyber 170 model 171.

The four new instructions are radically different from the original 71 on the 6000 series: Three of the four instructions are 60 bits long. All work directly on operands in memory--data does not first have to be loaded into X registers. And all can directly address any character (6-bit field) in memory. Four new opcodes were created by dividing opcode 46 into 8 opcodes 460,461,...,467; 460 stays a no-op, 464 through 467 are used by

the CMU and 461 through 463 are left for the new instructions in the Cyber80.

The CMU treats memory not as a sequence of 60-bit words, but rather as a sequence of 6-bit fields or characters. Each character is designated by the address of the word in which it occurs, and its character position in that word, nunbered 0 to 9 from left to right:

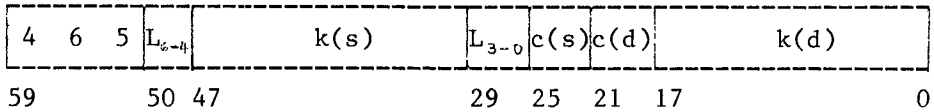


Character 9 of word 100 is thus followed by character 0 of word 101. A character string (which can be the operand or result of a CMU instruction) is specified by its starting character and a character count. For example, if we wanted to specify the 8th and 9th characters of word 100 and the 0th character of word 101, we would include in the CMU instruction the address 100, the character position, and the character count 3. Note that a character string may cross a word boundary with impunity.

The CMU instructions either move the contents of one character string into another character string or compare two character strings. In either case the instruction must specify the starting character of the two operands (or operand and result) and the length of the strings compared or moved. Specifying a character requires 22 bits (18 bit address + 4 bit character position); 2 *22 bits + 9 bit opcode = 53 bits. In a full-word instruction, this leaves 7 bits for a length field, so a string of up to 2**7 - 1 = 127 characters can be moved or compared.

The first of the quartet we shall present is the direct move, opcode 465. It is a 60-bit instruction, and like the other full word instructions to be presented later, cannot be split between words.

DM L, k(s), c(s), k(d), c(d) Move L characters, starting at (k(s),c(s)) to (k(d),c(d))



This instruction moves the L characters starting with character position c(s) of word k(s) to the L characters starting with character position k(d) of word c(d) (s and d stand for source and destination respectively). The source and destination fields should not overlap. The 7-bit length is divided between a 3-bit field, L(6-4), which contains bits 6,5 and 4 of the length, and a 4 bit field, L(3-0), which contains bits 3,2,1, and 0 of the length. This curious arrangement of the instruction was dictated by a desire to have the two address fields in bits 0-17 and 30-47, just as they would be if the word contained two 30 bit instructions.

This instruction can be used instead of a load-transmit-store sequence to compile a simple assignment statement. For example,

```
B = A
```

would move 10 characers:

```
DM 10,A,0,B,0
```

One instruction would even be enough to copy a small array:

```
      DO 1 I = 1,10
    1 B(I) = A(I)
```

could be compiled as a 100-character move:

```
DM 100,A,0,B,0
```

The problem with which we began this section is equally simply solved. If we want to extract the 7th through 14th characters from a 20-character (2-word) array STRING and place them in the variable NAME, left-justified, with the two rightmost characters filled with blanks, we need only code

```
DM      8,STRING,6,NAME,0

DM      2,BLANKS,0,NAME,8
```

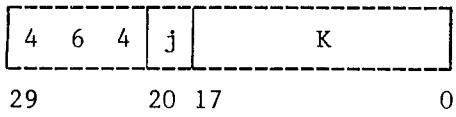
where BLANKS is word filled with BLANKS:

```
BLANKS  DATA  5555555555555555555B
```

On occasion we may want to move a string of more than 127 characters. We could, of course, code several DM's in a row, but control Data has been thoughtful enough to provide a second move

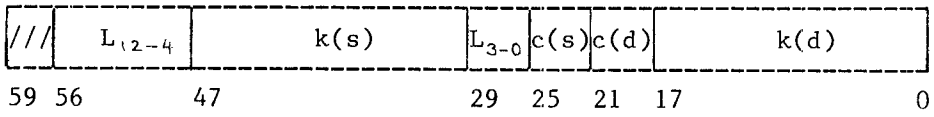
instruction, which can handle up to 8191 characters at a time. This is the indirect move:

IM Bj+K Move according to descriptor at address Bj+K (30 bits)



The word at address Bj+K should contain the length and the source and destination addresses and character positions in a special format called a move descriptor. COMPASS provides the pseudo-instruction MD for setting up a word in the proper format:

MD L,k(s),c(s),k(d),c(d) Move descriptor for moving L characters starting at (k(s),c(s))to (k(d),c(d))



All the fields in the move descriptor have the same meaning as those in the direct move instruction; all we have gained through this indirect addressing is a larger length field. The DO loop given above as an example of the direct move can also be translated as an indirect move:

```
        IM    DESCRIP
```

with the move descriptor

```
DESCRIP   MD    100,A,0,B,0
```

The IM instruction has the standard format of a 30-bit instruction with a Bj+K address field (it is the only CMU instruction in a familiar format) and so the usual rules for such address fields apply: in particular, if no B register is mentioned, B0 is implied. Note also that MD defines data, not an instruction; it belongs with the other DATA instructions of a routine.

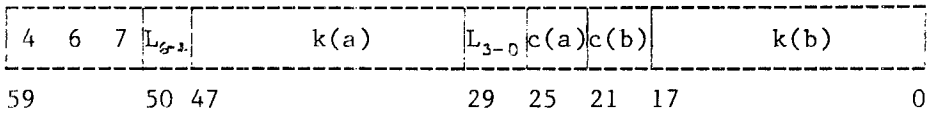
The CMU is very efficient for moving large blocks of data. After a couple of microseconds to start the ball rolling, the CMU can move one word every 300ns on the Cyber 70 series, one word every 500 ns on the 170 series. This is about ten times faster than

the most efficient loop using the other instructions on a Model 73.

One final note about the direct and indirect move: both use register X0 for temporary storage during execution of the instruction, and set X0 to +0 when the instruction is finished. All other registers are left intact by these instructions.

The two remaining instructions of the CMU compare two character strings; they return an indication of whether one string is less than, equal to, or greater than a second string. The comparison is done character by character, from left to right, until a difference in the two strings is found or the end of the strings is reached. In the comparison the characters are treated as 6-bit positive integers. We will begin with the 467 instruction, compare uncollated (just what uncollated means will be clear in a moment when we describe the compare collated instruction). This instruction is similar in format to the direct move:

CU L,k(a),c(a),k(b),c(b) Compare L characters starting at k(a),c(a) with L characters starting at k(b),c(b)



Here k(a), c(a) and k(b), c(b) give the starting addresses and character positions of the two string; L specifies the length of the strings, up to 127 characters. The result of the comparison is returned in X0 as follows:

string a = string b	X0=0
first n characters of strings are equal, (n+1)st character of a > (n+1)st character of b	X0=L-n>0
first n characters of strings are equal, (n+1)st character of a < (n+1)st character of b	X0=n-L<0

In effect, during the execution of the instruction, X0 contains the number of characters which remain to be compared. If the instruction terminates because X0 reaches 0 or string a > string b, X0 is left unchanged; if it terminates because string a < string b, X0 is complemented.

To appreciate the value of the compare instruction, let us consider how we might perform the same operation without the CMU instructions. To simplify matters, we won't be concerned with pinpointing the first character that is different in the two strings; we shall be satisfied if we can determine which string is larger or whether they are equal. The task of comparing two character string fields is then equivalent to treating the two fields as unsigned binary integers and comparing their values. To make the problem even simpler, the two strings we will compare are each 10 characters long and contained entirely in one word at locations FRITZ and SCHLITZ.

How do we compare two quantities? By subtracting and testing the difference for zero, plus, or minus. If we subtracted SCHLITZ from FRITZ, however, the machine would treat the two as assigned 60-bit quantities, which is not what we want (even comparing two signed 60 bit quantities is not so simple -- see exercise 22). For the machine to treat a quantity as a positive number, it has to be 59 bits or less, with the sign bit set to zero. As a result, we are going to have to do our 60-bit comparison in two parts; for example, first the high-order 59 bits, then the low order bit. So we begin by masking out the high-order 59 bits, shifting them right one bit, and subtracting them:

```
        SA1     FRITZ
        SA2     SCHLITZ
        MX5     59
        BX3     X1*X5           MASK OUT HIGH-ORDER
        BX4     X2*X5               59 BITS OF OPERANDS
        LX3     59              SHIFT RIGHT ONE, LEAVING
        LX4     59                  SIGN BIT = 0
        IX0     X3-X4           X0=DIFFERENCE OF HIGH 59 BITS
        NZ      X0,CDONE        IF HIGH BITS DIFFER,
                                    NEED NOT COMPARE LOW BIT
```

If the high 59 bits are equal, we have to compare the low order bits. Simply subtracting the full 60-bit words will give the difference of the low order bits, since the other bits are equal; there is no need to mask out the low order bits:

```
        IX0     X1-X2
CDONE   ...
```

This sequence of instructions leaves X0 equal to zero if FRITZ > SCHLITZ, and X0 < 0 if FRITZ < SCHLITZ. The same could be accomplished by the single instruction

```
          CU        10,FRITZ,0,SCHLITZ,0
```

If the character strings were longer than 10 characters or not contained entirely in one word, the non-CMU code would be even longer, but only one CU instruction would be required.

This example, however, does not fully indicate the power of the CU instruction. We have not yet made use of the ability of the CU instruction to pinpoint the first character at which the two strings differ. This feature makes it possible for a single CU instruction to replace certain searching loops in 6000-series code. For example, if we wanted to find the first non-zero element in a ten-word array TABLE, we could set up a second ten-word array, initialized to zero by

```
ZERO      BSSZ      10
```

and then execute

```
          CU        100,TABLE,0,ZERO,0
```

This will leave in X0 a number between 100 and 0 (since characters are treated as positive numbers, no character string can be smaller than ZERO, so X0 will always be positive). By computing (110-X0)/10, we obtain a number between 1 and 10 indicating the first non-zero entry, or 11 if all the elements of TABLE are zero. Because the CMU can compare a pair of words every 600 ns on the Cyber 70s, every 725 ns on the Cyber 170s (after a few microseconds start-up time), this code will in general be much faster than a search loop.

Of course, since the comparison is done character by character, the CU can scan individual characters as well as words. Suppose we have read an 80-character string into array CARD, and want to find the first non-blank character. If we set up another 8-word array BLANKS, initialized to all blanks (5555555555555555555B), all that is required is a compare.

```
          CU    80,BLANKS,0,CARD,0
```

After this instruction is executed, the magnitude of X0= 80 - n, where n is the number of leading blanks (the sign of X0 depends on whether the first non-blank has a display code below or above 55 (base 8)). Thus, 81-IABS (X0) will give the number of the first non-blank character, and 81 if CARD is all blank.

One thing the CU instruction is not very good for is deciding which of two strings comes first in alphabetical order. How's

that? Didn't we just see that the compare instruction compares strings from left to right according to the display code values of their characters? And doesn't the display code sequence correspond to alphabetical order? All quite right, but that overlooks one minor detail--blanks. In display code, a blank has a higher value than any letter, so

is longer than

which is contrary to the normal lexicographic convention -- SAM comes first in the dictionary. In order to have the comparison come out right, we would have to change all 55 characters (blanks) to 00 before we made the comparison.

In a practical application, the problem is further complicated by the appearance of non-alphabetic characters in the strings to be compared. If we are sorting addresses into alphabetical order, we will have to handle digits, hyphens, commas, and periods. Should digits rank higher or lower than letters in comparisons? And how about punctuation? We could insist that the ordering imposed by display code (first letters, then digits, then punctuation) be used, but some customers may not like that and not buy our computer. Businessmen often have definite ideas as to how things should be sorted (perhaps because their old machine did it that way), and, as long as they have a few megabucks to put up for a new machine, who are we to argue? So, as input to any sorting program we might write, we allow the customer to list the order in which he wants the characters ranked for comparison purposes. Such a list it called a character collating sequence.

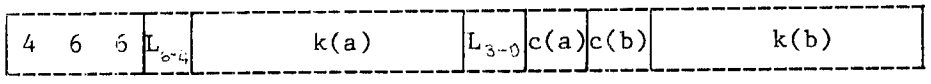
Suppose one user wanted his characters ranked as follows (in octal):

01	blank	(display code 55)
02	.	(" " 57)
03	'	(" " 56)
04	-	(" " 46)
05	0	(" " 33)
06	1	(" " 34)
	.	
	.	
	.	
16	9	(" " 44)
17	A	(" " 01)
20	B	(" " 02)
	.	
	.	
	.	
50	Z	(" " 32)

Before we could compare two character strings we would have to go through both strings, character by character, changing a 55 to a 01, a 57 to a 02, a 56 to a 03, etc., so that the comparison would come out as desired. Even using a table to do the conversion (with the 55th entry containing 01, the 57th, 02, etc.) this would take a long time. After this was all done, the compare instruction would take only a few microseconds, but this seems slim consolation to the man who realizes that he could have bought a yacht with what he paid for the CMU.

To keep the customers happy, CDC has included a fourth instruction in the CMU, compare collated. This instruction takes as input, in addition to the two strings to be compared, a table specifying the collating sequence. If a character in string A has the value m, and the corresponding character in string B has the value n, the compare collated will compare the values of the mth and nth entries in the table (this has the same effect as actually performing the substitution as described above). The format and significance of the fields is exactly the same as for the compare uncollated instruction:

CC L, k(a), c(a), k(b), c(b) Compare collated L characters starting at k(a), c(a), with characterssstarting atk(b),c(b)



The result is returned in X0, exactly as for the compare uncollated instruction. The collating table has 64 entries, one for each possible value of a single character; these entries are packed 8 per word, forming an 8-word table. A0 must be set to the address of the first word of the table before executing the instruction.

Each entry in the collating sequence table is 6 bits. The 8 entries in each word occupy the high order 48 bits; the low 12 bits of each word are ignored. The format of the table, with the entries labeled in octal, is thus:

word	59 53	47 41	35 29	23 17	11 0
A0	00 01 02 03	04 05 06 07			////////////
A0+1	10 11 12 13	14 15 16 17			////////////
A0+2	20 21 22 23	24 25 26 27			////////////
A0+3	30 31 32 33	34 35 36 37			////////////
A0+4	40 41 42 43	44 45 46 47			////////////
A0+5	50 51 52 53	54 55 56 57			////////////
A0+6	60 61 62 63	64 65 66 67			////////////
A0+7	70 71 72 73	74 75 76 77			////////////

Suppose we wanted to compare character strings using the collating sequence given earlier. Since we must assign a ranking to every possible character, we shall assign the remaining special characters values above Z, from 51 (base 8) to 77 (base 8), (except for the 00 character ":", which will will be assigned the lowest value, 00). The table required for the CC instructions would be:

```
CCTABLE     DATA    0017202122232425 0000B
            DATA    2627303132333435 0000B
            DATA    3637404142434445 0000B
            DATA    4647500506071011 0000B
            DATA    1213141516510452 0000B
            DATA    5354555657010302 0000B
            DATA    6061626364656667 0000B
            DATA    7071727374757677 0000B
```

All that is required to compare out friends SCHLITZ and FRITZ, according to this collating sequence, is then

```
SAO         CCTABLE
CC          10,FRITZ,0,SCHLITZ,0
```

It is possible to assign the same rank to two or more characters. Though this normally is not required in sorting applications, this possibility greatly increases the power of the compare instructions for scanning. Bear in mind in the following discussion that, since it is the ranks of two characters which are compared, two characters assigned to the same rank are considered equal.

With the uncollated compare it was possible only to scan a string for the first character which did not have some value "x", by comparing it with a string of all x's. With a collated compare, we can divide the character set into two classes, which we will call "interesting" and "uninteresting" characters, and then search a string for the first interesting character. We can do this by assigning all interesting characters rank 1 and all uninteresting characters rank 0 in our collating table, and then comparing our string with one consisting entirely of uninteresting characters.

For example, we might want to find the first arithmetic operator (+ - * or /) in the 80-character string card. To do this we create a table with rank 1 assigned to these four characters (45,46, 47, and 50) and rank 0 assigned to all the rest:

```
SCANTAB   DATA          0
          DATA          0
          DATA          0
          DATA          0
          DATA          0
          DATA          0
          DATA          0000000000001010100000B
          DATA          0100000000000000000B
          DATA          0
          DATA          0
```

We then compare CARD with a string of 80 blanks (any uninteresting character will do):

```
SA0             SCANTAB
CC              80,CARD,0,BLANKS,0
SX6             81
IX6             X6-X0
```

This sequence leaves in X6 the index of the first operator in CARD (81 if no operator was present).

CHAPTER 4

COMPASS

4.1 THE PSEUDO-INSTRUCTIONS

If we are to believe the reference manual, COMPASS stands for COMPrehensive ASSembly system. The operation codes we have used so far - the central processor instructions and a few of the most basic pseudo-operations - reflect only a small fraction of the capabilities of COMPASS. The eighty or so pseudo-instructions included in COMPASS offer the programmer powerful means for the control of code generation.

We shall not attempt to consider all these pseudo-instructions in this chapter; to do so in any detail would probably require several hundred pages and provide more information than most users will ever require. We shall rather restrict ourselves to a few of the most important COMPASS features -- MACROs, MICROs, and conditional assembly -- and then discuss primarily the principles involved, and not the detailed rules and restrictions. Control Data's COMPASS Reference Manual is generally understandable, if not totally lucid, and can provide the detailed information once the principles have been understood.

4.2 THE MACRO

What has been presented so far in this book, and indeed what is included in many books on programming, might leave the impression that, once the algorithm for a program has been determined, programming is essentially a task of translating this algorithm into an appropriate computer language. This is simply inaccurate. In an intelligently planned programming project, most of the time is spent in program design, and in program debugging and checkout; the actual coding represents only a small

fraction of the total effort. Needless to say, some projects, eager to produce results, skimp on program design and rush headlong into the coding phase; they soon pay in increased coding and debugging time, to remove errors that might never have arisen through proper design.

Good program design is in large part a task of identifying functions that must be performed repeatedly during the program, and of building larger functions out of more basic ones. Two fundamental programming constructs have been developed to assist the programmer once he has identified these functions: the subroutine and the macro. The idea behind the subroutine is that the code to implement the functions appears only once in the program. Each time the function is invoked, control is transferred to the subroutine; when the subroutine has completed its task, it returns control to the calling sequence. In contrast, the macro is based on the idea that the code to implement the function appear each time the function is required. When the statement invoking the function is encountered by COMPASS during assembly, it is replaced by the machine code necessary to perform the function.

A small example should help make this clear.

Suppose we have written a large program, and decide afterwards that we would like to keep count of the number of lines printed by our program. We would have to execute, after each section of code which generates a line of output, a sequence of instructions such as

```
        SA1     LCOUNT
        SX6     X1+1
        SA6     A1
```

There are 629 such sections of code in our program, and we aren't looking forward to adding 3*629 cards to our deck. We have two ways out: first, we could put these three lines of code in a subroutine

```
LNCTR   BSS     1           INCREMENT LINE COUNT
        SA1     LCOUNT
        SX6     X1+1
        SA6     A1
        EQ      LNCTR
```

and place after each of those 629 sections of code a call to our subroutine

```
        RJ      LNCTR
```

(note that, as long as our little subroutine appears in the same subprogram, i.e., between the same pair of IDENT and END cards, as the rest of our program, the label LNCTR is a perfectly ordinary symbol, and requires no ENTRY or EXTernal declarations). Alternatively, we could define these three lines of code as the macro LNCTR, by including at the beginning of our program

```
LNCTR   MACRO
        SA1     LCOUNT
        SX6     X1+1
        SA6     A1
        ENDM
```

Then, whenever we write, in the opcode field, the macro name

```
        LNCTR
```

COMPASS will expand the macro, generating

```
        SA1     LCOUNT
        SX6     X1+1
        SA6     A1
```

The macro approach is more space consuming (75 bits are required for the three instructions, while the RJ needs at most 60 bits, if it is the first instruction in a word) but it is much faster, since three jumps are necessary to get to and from the subroutine.

We shall close this section with a bit of terminology. Every macro definition, such as the one just above, consists of three parts:

1. Macro heading: the MACRO pseudo-instruction, with the macro name in the location field

2. Macro body: the instructions which constitute the macro code definition

3. Macro terminator: the ENDM pseudo-instruction, which marks the end of the macro definition

The macro definition must appear prior to any reference to the macro; it is generally a good idea to put all macro definitions at the beginning of a subprogram.

4.3 MACRO PARAMETERS

If each macro could perform only one specific function, the applications for macros would be severely limited. As a result, macros, like subroutines, include parameters which may be used to vary the function performed. There is a fundamental difference between subroutine and macro parameters, which must be kept in mind. Subroutine parameters are transmitted to the subroutine at execution time, and the subroutine must include code to test these parameters and take appropriate action. Macro parameters, on the other hand, determine the code that is generated by COMPASS when the macro is invoked. We can elucidate this distinction by pursuing the example from the previous section.

Suppose that three different types of lines are printed by our big program, and we want to keep a separate count for each type of line, in LCOUNT1, LCOUNT2 and LCOUNT3. We could modify our subroutine to accept as argument, passed in B1, the address of the count to be incremented:

```
LNCTR   BSS     1
        SA1     B1
        SX6     X1+1
        SA6     A1
        EQ      LNCTR
```

and then call it each time with the appropriate counter as argument, for example

```
        SB1     LCOUNT2
        RJ      LNCTR
```

Alternatively, we could define our macro with one parameter, the counter to be incremented:

```
LNCTR   MACRO   COUNTER
        SA1     COUNTER
        SX6     X1+1
        SA6     A1
        ENDM
```

Then, by invoking the macro with the name of one of the counters as argument (in the address field), we can generate the code to increment that counter; for example,

```
        LNCTR   LCOUNT2
```

would be expanded into

```
        SA1     LCOUNT2
        SX6     X1+1
        SA6     A1
```

Note that, now that the "increment counter" function has been parameterized, the advantage of the macro version in terms of execution time has increased, while the disadvantage in terms of space consumed has disappeared.

A macro may have up to 63 parameters. In the macro heading, these parameters are listed in the address field, separated by commas, thus:

```
MCNAM   MACRO   P1,P2,P3,P4
```

These parameter names may appear in any field of any line of the macro body. To be recognized as parameters, however, (and, therefore, replaced by the actual arguments when invoked) they must be bounded by one of the characters + - * / ( ) $ = . ,
or blank. Thus, P1 would be recognized as a parameter in

```
        SA1     P1-2
```

but not in

```
        SA1     P1CTR-2
```

A macro with several arguments is invoked by placing the macro name in the opcode field and listing the actual arguments in the address field, separated by commas. Macro MCNAM, for example, could be invoked by

```
        MCNAM   HE,HI,HO+2,B6+HUM
```

The actual arguments may be arbitrary character strings not including blanks or commas (a method for transmitting arguments with commas and blanks will be discussed later).

Our second example will confront, in rather simplified form, one of today's most important programming problems: transferring programs from one machine to another completely different machine. The usual solution to this problem has been the writing of programs in high-level, "machine independent" languages. Large computer systems now generally include FORTRAN, COBOL, and ALGOL compilers; the smaller systems at least a FORTRAN compiler. For the scientific and commercial applications for which these languages were designed, their use appears to offer the best solution to program transferability. There are, however, many applications, such as text processing, list processing, and

systems programming, for which these languages are less than ideal. The natural impulse under these circumstances is to design one's own programming language, and write a compiler to translate it; one soon realizes, though, that no else will be able to use your program (unless he has the same computer you do), since he doesn't have a compiler for your language. One way out of this morass is to design a small set of macros which perform all the basic functions you require, and write your program as a sequence of macro calls. Transferring the program to another machine would then involve recoding the macro bodies, certainly a much smaller job than recoding the entire program.

For this scheme we require a macro processor on each machine, but not necessarily a macro-assember. In other words, for one machine we might write our macro bodies in the language ALGOL, and have our macro processor expand our program (sequence of macro calls) into an ALGOL program. Some higher level languages do include a macro processor; PL/I is probably the most familiar example. However, since the only macro processor on most machines is the macro-assembler, and, as you may recall, we are supposed to be studying COMPASS in this chapter, we shall restrict ourselves to writing COMPASS versions of our macros.

The set of macros we shall prepare will not be in any sense complete; all we will be able to do with them is add and subtract integers and floating-point numbers. In constructing our macros, we shall imagine that we are writing code for a hypothetical machine with one register, which we shall call the AC (accumulator). This machine can load a word from memory into the AC, store a word in memory from the AC, and compute the integer or real sum or difference of a memory word and the AC, leaving the result in the AC. This makes a total of six hypothetical instructions, so we need six macros:

```
LOAD    MACRO   ADDR
        SA5     ADDR
        BX6     X5
        ENDM

STORE   MACRO   ADDR
        SA6     ADDR
        ENDM

IADD    MACRO   ADDR
        SA5     ADDR
        IX6     X6+X5
        ENDM
```

```
ISUB    MACRO   ADDR
        SA5     ADDR
        IX6     X6-X5
        ENDM

FADD    MACRO   ADDR
        SA5     ADDR
        RX6     X6+X5
        NX6     X6
        ENDM

FSUB    MACRO   ADDR
        SA5     ADDR
        RX6     X6-X5
        NX6     X6
        ENDM
```

Then, if we wanted to store the sum of the real variables A and B in C, and the difference of the integer variables K and L in M, we would code

```
        LOAD    A
        FADD    B
        STORE   C
        LOAD    K
        ISUB    L
        STORE   M
```

This would be expanded by COMPASS into

```
        SA5     A
        BX6     X5
        SA5     B
        RX6     X6+X5
        NX6     X6
        SA6     C
        SA5     K
        BX6     X5
        SA5     L
        IX6     X6-X5
        SA6     M
```

(If you wish to assemble these macros yourself, you must include

```
        LIST    M
```

immediately after the IDENT card, or the macro expansions will not be listed).

It should be a simple task to implement these macros on any computer. Of course, they are not likely to yield very efficient code (unless the real machine, like our hypothetical one, has only a single register). But we are often willing to sacrifice some efficiency in order to get an easily transferable program; once the program has been successfully transferred to a new machine, it is always possible to recode critical sections of the program in carefully optimized machine code.

Naturally, our little macro language has its problems too. One of them is indicated by the following code sequence, where A and B are floating point variables, K an integer variable:

```
LOAD    A
IADD    K
STORE   B
```

What's wrong? We are adding a floating point number (A) to an integer (K). If our macros are to be worth the cards they are punched on, we must either flag this as an error, or convert one of the operands to the mode (floating point or integer) of the other before performing the addition.

4.4 CONDITIONAL ASSEMBLY

In order to catch these "mixed-mode" operations (integer operand and floating point operand), we have to

1. keep track of the mode (integer or floating) of the quantity in the accumulator,

2. before every arithmetic operation, compare this mode with the mode of the operand in memory, and convert one of the two operands if necessary.

To do the first, we need a symbol to which we can assign the value 0 when the accumulator contains an integer quantity and 1 when it contains a floating point quantity.

Until now, the only way we had of assigning a value to a symbol was to place the symbol in the location field:

```
CMBAL   SBI     3
```

The value thus assigned is the address of the word containing the instruction. This value is permanently assigned to the symbol; anywhere in the program that this symbol is used, it will have this value. What we require now is a means of defining a symbol in such a way that it can take on different values at different points in the program. This ability is provided by the SET pseudo-instruction:

```
CMBAL     SET       1
```

The SET pseudo-operation assigns the value of the number, symbol, or expression* in the address field to the symbol in the location field. This assignment remains in effect until the next SET involving this symbol is encountered. So, for example, the sequence

```
Q         SET       A
          SB1       Q
Q         SET       Q+1
          SB2       Q
```

yields the same instructions as the sequence

```
          SB1       A
          SB2       A+1
```

These two methods of defining a symbol are incompatible; a symbol may be assigned a value either one way or the other, but not both. Thus, if you would like an assembly errror, code

```
OOPS      FX5       X1*X3
OOPS      SET       37
```

Armed with our SET instructions, we are now able to keep track of the mode of the quantity in the accumulator. If we use the symbol ACMDE to hold this information, any macro which leaves an integer quantity in the accumulator will have to include

```
ACMDE     SET  0
```

---

*Note: An expression in COMPASS, more precisely called an address expression, is a series of integers and symbols separated by the four operators + - * / ; for example, A+3, HUM-HO, C*2/D. Expressions are evaluated from left to right, with multiplication and division performed before addition and subtraction, as in FORTRAN. Wherever a number or symbol is expected in an address field, an address expression may appear instead.

---

while any macro which leaves a floating point number will have to include

```
ACMDE   SET     1
```

This brings us to our second problem: how can we use this information (the value of ACMDE) to control the code generated by the macros?

COMPASS provides for this purpose a set of IF pseudo-operations, which permit the conditional assembly of instructions. That is, an IF instruction specifies a relation between two quantities; if this relation holds, the following instructions are assembled by COMPASS as usual; if it does not hold, the following instructions (up to the next ENDIF instruction) are skipped and ignored. For example,

```
        IFEQ    A,B
        JP      THENCE
        ENDIF
```

will generate a jump to THENCE if the values of the symbols A and B are equal; if they are not equal, no code will be generated. It should be emphasized that the IF pseudo-operation causes a test to be made at the time the program is assembled; its effect is entirely different from that of the FORTRAN IF statement

```
IF (A.EQ.B) GO TO 10
```

which is compiled into code that compares the contents of locations A and B at execution time.

There are a total of ten conditional-assembly pseudo-operations in COMPASS. Six of them, with mnemonics IFEQ, IFNE, IFGT, IFGE, IFLE, and IFLT, compare the values of the two items in the address field; as should be evident, the mnemonics are formed from IF + name of corresponding FORTRAN relational operator. The items of the address field, separated by commas, can be numbers, symbols, or address expressions. The number of instructions to be assembled or skipped is indicated by an ENDIF instruction or by a line count as the third item in the address field:

```
        IFEQ    A,B,1
        JP      THENCE
```

We shall use the ENDIF in all our examples. (Note: lines skipped by an IF instruction are normally not listed on the assembler output.)

We now have all the information we need to write our improved macros. Let's start with the LOAD macro. We immediately encounter a problem: we can't tell from

```
        LOAD    VARB
```

whether VARB is an integer or floating point variable. What we will have to do is replace our old LOAD macro with two new macros, ILOAD, and FLOAD, for loading integer and floating quantities, respectively. The ILOAD macro will set ACMDE to 0, the FLOAD macro will set it to 1:

```
ILOAD   MACRO   ADDR
        SA5     ADDR
        BX6     X5
ACMDE   SET     0
        ENDM

FLOAD   MACRO   ADDR
        SA5     ADDR
        BX6     X5
ACMDE   SET     1
        ENDM
```

The addition and subtraction macros are more complicated, since they will have to include some conditionally assembled code. Let us begin by considering the integer add macro, IADD; a problem arose with this macro when the quantity already in the accumulator was floating-point. Taking our cue from the mixed-mode rules in FORTRAN, we shall, in this case, convert the integer to floating-point and perform a floating point addition. Thus, our new IADD macro will be

```
IADD    MACRO   ADDR
        SA5     ADDR
        IFEQ    ACMDE,0
        IX6     X6+X5
        ENDIF
        IFEQ    ACMDE,1
        PX5     X5
        NX5     X5
        RX6     X6+X5
        NX6     X6
        ENDIF
        ENDM
```

If ACMDE=0 (the accumulator contains an integer)

```
            SA5         ADDR
            IX6         X6+X5
```

is generated as before; if ACMDE=1 (the accumulator contains a floating point number) the operand from memory is first converted to floating point, and then the floating addition is performed:

```
            SA5         ADDR
            PX5         X5
            NX5         X5
            RX6         X6+X5
            NX6         X6
```

Note that, since the mode of the quantity in the accumulator is not changed by either operation, ACMDE need not be reset.

For the floating point add macro, FADD, the situation is reversed: if the accumulator contains an integter, we must convert it to floating point before performing the addition. Thus, this macro becomes

```
FADD        MACRO       ADDR
            SA5         ADDR
            IFEQ        ACMDE,0
            PX6         X6
            NX6         X6
ACMDE       SET         1
            ENDIF
            RX6         X6+X5
            NX6         X6
            ENDM
```

If ACMDE=1 we get back the original FADD macro:

```
            SA5         ADDR
            RX6         X6+X5
            NX6         X6
```

however, if ACMDE=0, the accumulator is first converted from integer to floating point:

```
            SA5         ADDR
            PX6         X6
            NX6         X6
 ACMDE      SET         1
            RX6         X6+X5
            NX6         X6
```

Since this changes the mode of the accumulator, ACMDE must be reset.

Replace addition by subtraction, and we have ISUB and FSUB:

```
ISUB    MACRO   ADDR
        SA5     ADDR
        IFEQ    ACMDE,0
        IX6     X6-X5
        ENDIF
        IFEQ    ACMDE,1
        PX5     X5
        NX5     X5
        RX6     X6-X5
        NX6     X6
        ENDIF
        ENDM

FSUB    MACRO   ADDR
        SA5     ADDR
        IFEQ    ACMDE,0
        PX6     X6
        NX6     X6
ACMDE   SET     1
        ENDIF
        RX6     X6-X5
        NX6     X6
        ENDM
```

This brings us finally to the STORE macro. As we did for the LOAD macro, we shall introduce two macros, ISTORE and FSTORE, for storing into integer and floating point variables respectively. The ISTORE macro will have to first convert the accumulator to an integer if it contains a floating point number:

```
ISTORE  MACRO   ADDR
        IFEQ    ACMDE,1
        UX6     B1,X6
        LX6     B1,X6
ACMDE   SET     0
        ENDIF
        SA6     ADDR
        ENDM
```

Similarly, the FSTORE macro will have to convert the accumulator to floating point if it contains an integer:

```
FSTORE    MACRO     ADDR
          IFEQ      ACMDE,0
          PX6       X6
          NX6       X6
ACMDE     SET       1
          ENDIF
          SA6       ADDR
          ENDM
```

Before we set aside our mixed-mode problems for a while, let us add two small refinements. In these eight macros, the pack and normalize sequence required to convert an integer to floating point occurs five times, so it seems reasonable to make this sequence into a macro itself. Something like

```
FLOAT     MACRO     X1
          PX1       X1
          NX1       X1
          ENDM
```

seems appropriate. Then, if we wanted to generate the code to "float" X6, we would write

```
          FLOAT     X6
```

Just to be sure that this works (although this is such a trivial macro it is hard to imagine we could have made a mistake) we look at the macro expansion:

```
          PX1       X6
          NX1       X6
```

Surprised? If you are, you forgot the rule concerning the recognition of parameters in the macro body. Parameters are recognized only if they are delimited by one of the characters + - * / ( ) $ = . , → ≠ or blank. The X1's in the address fields meet this requirement, since they are surrounded by blanks; however, the X1's in the opcode field are immediately preceded by a letter, so they fail this test. This predicament was anticipated by the assembler designers, and a special character " ", was introduced to circumvent the problem. The magic of this character is that, as soon as actual parameters have been substituted into the macro body, all the 's disappear. This character is referred to as the catenation (or concatenation) mark, and the procedure of removing these blanks is called catenation. To see how this works, let us fix up the definition of FLOAT:

```
FLOAT     MACRO     X1
          P→X1      X1
          N→X1      X1
          ENDM
```

When the macro is invoked by

```
          FLOAT     X6
```

COMPASS first substitutes X6 wherever X1 occurred as a parameter in the macro body:

```
          P→X6      X6
          N→X6      X6
```

then it deletes the catenation marks and squeezestogether the characters surrounding the catenation marks (so no blank is left where a catenation mark was deleted):

```
          PX6       X6
          NX6       X6
```

(Note: if only LIST M is requested, COMPASS will print these lines with the catenation marks still in; to see the lines with these marks removed, a "list" card must be included; i.e.:)

```
          LIST      A
```

Now that our FLOAT macro is in working order, we can use it in our earlier macros. For example, IADD would become

```
IADD      MACRO     ADDR
          SA5       ADDR
          IFEQ      ACMDE,0
          IX6       X6+X5
          ENDIF
          IFEQ      ACMDE,1
          FLOAT     X5
          RX6       X6+X5
          NX6       X6
          ENDIF
          ENDM
```

This macro is a simple illustration of the property that makes macros such a powerful tool: the ability to nest macros, to invoke one macro within the body of another. The inner macro (FLOAT) need not be defined before it appears in the definition of the outer macro (IADD), but only before the outer macro is invoked.

We can pare one more line from the IADD and ISUB macros by using the ELSE pseudo-operation. An ELSE appearing in the range of an IF (between the IF and the ENDIF) reverses the effect of the IF instruction. That is, if the condition in the IF instruction is true, the lines between the IF and the ELSE are assembled while the lines between the ELSE and the ENDIF are skipped; if the condition is false, the lines before the ELSE are skipped and the lines after it are assembled. For example,

```
        IFEQ    DESPERAT,0
        EQ      EXIT
        ELSE
        EQ      HELP
        ENDIF
```

generates a jump to EXIT if DESPERAT is zero, and a jump to HELP if it isn't. In the case of our IADD macro, since ACMDE will always be 0 or 1, we can code

```
IADD    MACRO   ADDR
        SA5     ADDR
        IFEQ    ACMDE,0
        IX6     X6+X5
        ELSE
        FLOAT   X5
        RX6     X6+X5
        NX6     X6
        ENDIF
        ENDM
```

We leave to the reader the straightforward task of modifying all the earlier macros to use FLOAT, and then trying these macros out on a few sequences of mixed-mode arithmetic. Having now whetted the reader's appetite for the excitement of mixed-mode arithmetic, rest assured that we shall not abandon the topic; after a brief respite, we shall return with yet finer and fancier macros.

4.5 DEBUGGING WITH MACROS

Although any programmer who gets anything accomplished spends at least 90% of his time debugging, most programming texts give

short schrift to the topic. This book will not be different. This is hardly surprising; a thorough exposition of coding requires a knowledge of the rules of the programmming language, which are usually readily available. A proper exposition of debugging, on the other hand, would require some knowledge of the kind of mistakes people make in writing programs, and this complex and mysterious area has been the subject of relatively little study (perhaps because people are reluctant to admit the mistakes they make).

Of course, the best solution would be to not make mistakes in the first place; some of the philosophical interludes in this volume have tried to suggest how using a higher-level language where appropriate and organizing the program carefully may help to reduce the frequency of bugs. Still, the best of us will make mistakes, so we may reasonably ask what tools are available to help us track down these errors.

The standard tool used in debugging programs on the CDC 6000 series (and many other machines) may best be characterized by one word: medieval. It is the absolute octal core dump: a print-out of the contents of core, in octal, near the point where the program stopped, or got an arith error, or whatever (in chapter 5 we shall briefly describe how to read a dump). There is a certain irony in providing the user with an array of powerful machine-independent languages and then, when the program "bombs out," telling him that he has to learn machine language in order to interpret the dump that is produced. In fairness to Control Data, some additional debugging tools are provided with the system, although also oriented towards assembly-language programmers. However, because these tools are more complicated than many programmers would like, and because they did not work when first introduced, they have not become very popular. (The author, having developed a much more powerful debugging system, which had these problems to a correspondingly greater degree, is painfully aware of the lack of enthusiasm which ensues.) As an alternative to these tools, we shall examine how macros can be used to build simple yet powerful debugging aids.

We shall begin with a debugging aid called a store trace. This is a print-out, each time a store occurs, of the name of the variable stored into and the value stored. To make life simple, we shall assume that we have only integer variables (no mixed-mode). Also, so that we need not concern ouselves with calling output routines in assembly language, we shall use a small FORTRAN routine to print the trace:

```
      SUBROUTINE TRACE(NAME, IVALUE)
      PRINT 1, NAME, IVALUE
1     FORMAT(* STORE TO *,A8 ,*=*, I17)
      RETURN
      END
```

All that is now required is to place a call to TRACE after every store instruction in the program. If the program were coded instruction by instruction, this might be a sizeable task; for a program using our macro package, however, this involves merely putting a call to TRACE in the STORE macro.

The STORE macro will begin, as before, with the actual store operation:

```
STORE   MACRO   ADDR
        SA6     ADDR
```

Next, we must set up the arguments for the call to TRACE. Somewhere in our COMPASS program we set aside space for the argument list:

```
TRCARGS BSS     2
        DATA    0
```

The macro must store the addresses of the two arguments in TRCARGS and TRCARGS+1. The second argument is easy, since the value stored is now in location ADDR:

```
        SX7     A6
        SA7     TRCARGS+1
```

The first argument, the name of the variable, is a bit more complicated. If we were invoking TRACE from a FORTRAN program, the call would have the general form

```
CALL TRACE(4HIVAR,IVAR)
```

Character string constants valid in FORTRAN, such as 1H+ or 3REOF, are also allowed in COMPASS DATA instructions, so for our macro the DATA instruction would be

```
        DATA    8H→ADDR
```

If we included this DATA instruction in the macro, it would be necessary to jump around it; this seems neither elegant nor efficient. Fortunately, we can circumvent this problem without circumventing the DATA instruction by using a COMPASS feature

introduced in the last chapter (3.13): the literal. If we code

```
        SX7     =8H→ADDR
        SA7     TRCARGS
```

COMPASS will generate the DATA instruction at the end of the program, and put its address in the K portion of the SX7 instruction. This takes care of the arguments, so we can call our FORTRAN routine

```
        SA1     TRCARGS
        RJ      TRACE
```

On return from TRACE, X6 will no longer contain the value of the accumulator (routines generally do not save and restore registers), so we must reload X6:

```
        SA5     ADDR
        BX6     X5
        ENDM
```

If we glue all the parts together we have

```
STORE   MACRO   ADDR
        SA6     ADDR
        SX7     =8H→ADDR
        SA7     TRCARGS
        SX7     A6
        SA7     TRCARGS+1
        SA1     TRCARGS
        RJ      TRACE
        SA5     ADDR
        BX6     X5
        ENDM
```

Starting with this basic pattern, the variations which can be made are endless. Using conditional assembly, we can trace stores to one or a few variables; through appropriate tests in the TRACE routine, we can start printing the trace when a particular variable is assigned a particular value. We shall exercise some restraint, however, and consider only two small improvements.

Debugging aids like the store trace are best designed into the program from the start, and left in throughout program development. This is possible if we make assembly of the code for tracing stores conditional on the value of a symbol:

```
STORE     MACRO     ADDR
          SA6       ADDR
          IFNE      STORTRAC, 0
          SX7       =8H=ADDR
          SA7       TRCARGS
          SX7       A6
          SA7       TRCARGS+1
          SA1       TRCARS
          RJ        TRACE
          SA5       ADDR
          BX6       X5
          ENDIF
          ENDM
```

In this way the debugging code will be assembled only if STORTRAC is non-zero. We may also make the external declaration which is required for TRACE conditional:

```
          IFNE      STORTRAC,0
          EXT       TRACE
          ENDIF
```

As well as the space for the argument list

```
          IFNE      STORTRAC,0
TRCARGS   BSS       2
          DATA      0
          ENDIF
```

Finally, we must set STORTRAC at the beginning of the program, with appropriate flourish:

```
*
*         STORTRAC : NON-ZERO TO TRACE STORES
*
STORTRAC  SET       1
```

Our second improvement is concerned with the traceback features of FORTRAN and other high-level languages. When a system routine detects an error, it traces backwards the series of subroutine calls which brought control to this routine, starting with the subroutine which invoked the current routine, then the subroutine that invoked that routine, and so forth until the main program is reached. A traceback is made possible by the convention that a RJ always be placed in the high-order 30 bits of an instruction word, and the low-order 30 bits contain

```
nnnn      TRACEBAK
```

where nnnn (12 bits) is the line number of the statement which caused the RJ to be generated (0 for RJ in COMPASS programs) and TRACEBAK is the (18 bit) address of a word in the calling routine. This word contains the name of the calling routine, and in the low-order 18 bits the address of the entry line of the calling routine.

In order to set up this word and the pointer to it, we must introduce a new pseudo-operation, VFD (variable field definition). In contrast to the DATA operation, the VFD permits us to specify separately the contents of portions (fields) of a word. The address field of the VFD consists of one or more subfields, separated by commas; each subfield has the form

bit-count/ address-expression

and specifies that the value of the address expression is to be placed into the next "bit count" bits of the word. For example,

```
VFD       12/1777B,48/0
```

generates a plus indefinite, and

```
VFD       60/TRACEBAK
```

generates a word containing the address TRACEBAK. If the expression includes a symbol (is not composed entirely of constants*) it must be placed in a field at least 18 bits long, ending at bit 0, 15, or 30 (in a position where the K portion of a 30 bit instruction could occur); thus

```
VFD       59/TRACEBAK,1/0
```

is illegal. Note that although the previous VFD specifies a full word field, a DATA instruction could not be used; the address field of a DATA instruction can contain only constants.

The STORE macro would be modified as follows:

*Note: More precisely, as shall be explained in Section 4.7, if it is not an absolute expression.

```
STORE     MACRO     ADDR
          SA6       ADDR
          IFNE      STORTRAC,0
          SX7       =8H←ADDR
          SA7       TRCARGS
          SX7       A6
          SA7       TRCARGS+1
          SA1       TRCARGS
+         RJ        TRACE
-         VFD       12/0,18/TRACEBAK
          SA5       ADDR
          BX6       X5
          ENDIF
          ENDM
```

The + in the location field (column 1 or 2) of the RJ instruction forces upper, i.e., places the RJ in the high-order part of a new instruction word. Contrarily, the - in the location field of the VFD overrides the force upper which is automatic after the RJ, and places the VFD in the next available 30 bits.

If the routine written in our macro package is called LINDA , it must begin essentially thus:

```
          IDENT     LINDA
TRACEBAK  VFD       42/7LLINDA    ,18/LINDA
          ENTRY     LINDA
LINDA     BSS       1
```

If we adhere to these conventions, it is a simple matter to generate a traceback. To indicate how this is done, we shall consider the example shown in Figure 2. The main program, MAIN, calls a user-written subroutine, MYSUB; MYSUB detects an error and calls the system subroutine SYSTEM to generate a traceback. SYSTEM does the following:

1. extracts from the entry line of SYSTEM the address portion of the EQ instruction, and subtracts one to get the address of the call to SYSTEM, 743

2. the low 18 bits of location 743 point to word 700, which contains the name of the invoking routine ("MYSUB"); the first lines of the traceback can now be printed:

```
SYSTEM
CALLED FROM MYSUB AT 000743
```

3. the low 18 bits of word 700 point to the entry line of MYSUB; the entry line contains an EQ instruction, so the procedure of steps 1 and 2 can be repeated: the address from which MYSUB was called is determined (120), the name of the calling routine is found ("MAIN"), and the next line is printed:

```
CALLED FROM MAIN AT 000120
```

4. The low 18 bits of the word containing "MAIN" point to a word which does not contain an EQ instruction; we have reached the main program, so the traceback is complete.

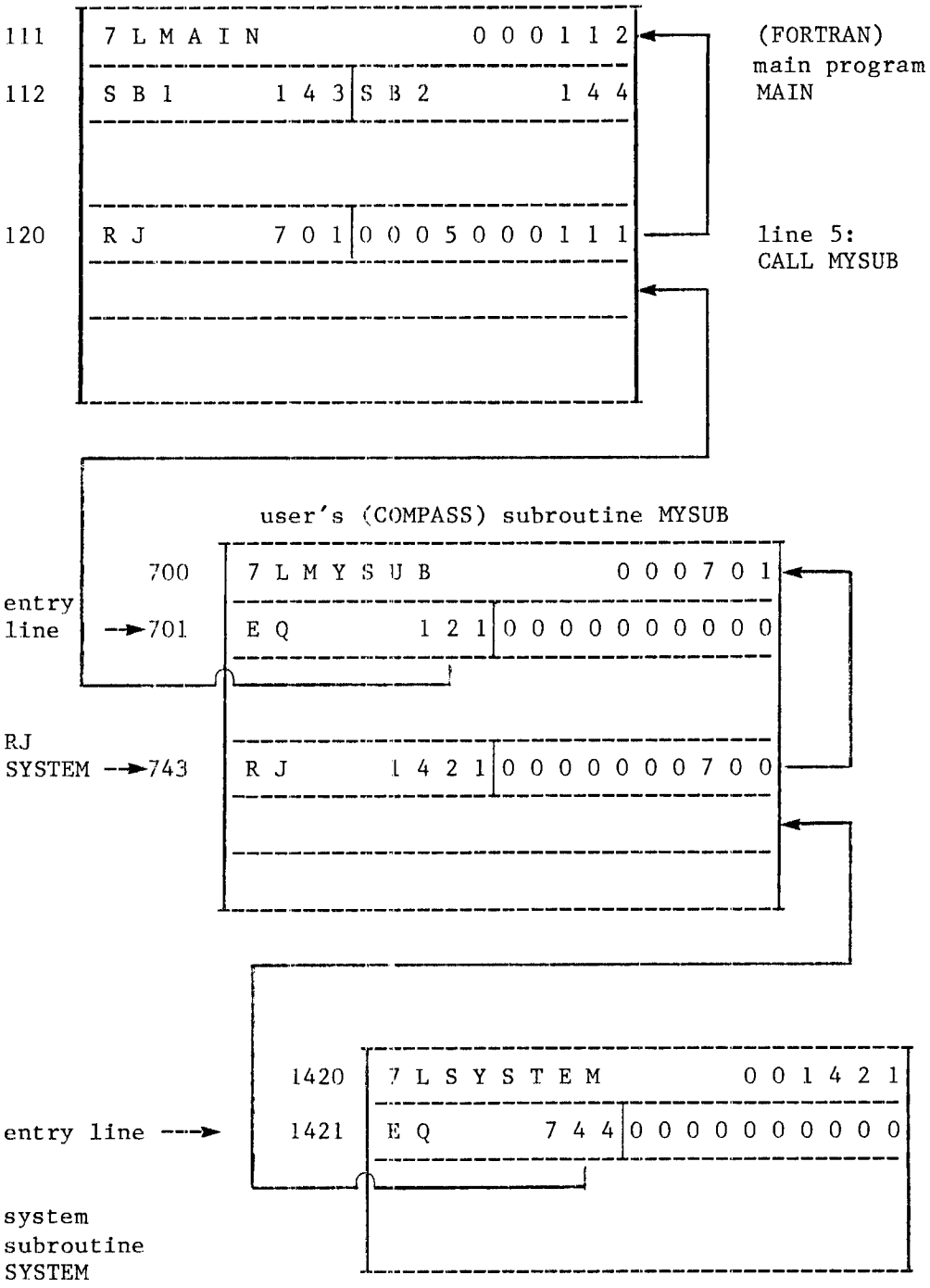

111
7 L M A I N 0 0 0 1 1 2
112
S B 1 1 4 3
S B 2 1 4 4
120
R J 7 0 1
0 0 0 5 0 0 0 1 1 1
(FORTRAN)
main program
MAIN
line 5:
CALL MYSUB
user's (COMPASS) subroutine MYSUB
700
7 L M Y S U B 0 0 0 7 0 1
entry
line
→701
E Q 1 2 1
0 0 0 0 0 0 0 0 0 0 0
RJ
SYSTEM →743
R J 1 4 2 1
0 0 0 0 0 0 0 7 0 0
1420
7 L S Y S T E M 0 0 1 4 2 1
entry line →
1421
E Q 7 4 4
0 0 0 0 0 0 0 0 0 0 0
system
subroutine
SYSTEM

FIGURE 2

4.6 MICROS AND CODE DUPLICATION

So far, we have constructed a set of basic arithmetic macros; we have shown how they can be helpful in debugging, and we have modified them to include one higher-level language feature, mixed-mode arithmetic. In this section, as our final improvement, we shall implement another, more significant, feature of higher-level languages.

Suppose we want to compute the sum of the integer variable HEN and the floating point variable ROOSTER and leave the result, in floating point, in CHICK. In our macro language, we would write:

```
ILOAD     HEN
FADD      ROOSTER
FSTORE    CHICK
```

Compare this with the equivalent series of FORTRAN statements:

```
INTEGER   HEN
REAL  ROOSTER,  CHICK
CHICK  =  HEN  +  ROOSTER
```

In our macro language, we must explicitly state for each operation the mode of the operand or the mode in which we want the result stored. In FORTRAN, on the other hand, the mode is made a property or attribute of the variable; in processing the assignment statement, the compiler uses the modes assigned to the variables involved to determine the instructions to be generated. We shall try to implement a similar arrangement in our macro language.

To do so, we must find a means of assigning an attribute to a symbol. We shall do this by associating with each symbol a second symbol, whose value indicates an attribute of the first symbol. For example, we could form the second symbol by suffixing the first symbol with an M: for a variable VAR, we would use the value of the symbol VARM to indicate the mode of VAR. To hide this apparatus from the user of our macro language, we supply two macros with which the user can indicate the mode of his variables:

```
INTEGER   MACRO     VAR
VAR←M     SET       0
          ENDM
REAL      MACRO     VAR
VAR←M     SET       1
          ENDM
```

Variable names are thus restricted to seven characters; if an eight-character name is used as the argument to one of these macros, a nine-character name will be generated, and hence an error message. Clearly symbols ending in M should also not be used; however, if the user forgets this restriction, and puts in his program

```
VARBM   SET     0
```

(When he already has a variable VARB) he will completely foul up code generation yet get no error message. To avoid this trouble, I shall tell you a secret, which you must promise not to tell the user of our macro language: any special character except + - * / , ^ or blank is allowed in a symbol name (the first character may not be $, = or a digit). Thus, we could use a $ suffix instead of an M; a user is less likely to use a $ in a name accidentally:

```
INTEGER MACRO   VAR
VAR$    SET     0
        ENDM

REAL    MACRO   VAR
VAR$    SET     1
        ENDM
```

($ is a delimiter for macro parameters, so no catentation mark is required). This may all seem quite trivial, especially if you are writing a macro for your own use, but the importance of coding in such a way that bugs are avoided or caught early cannot be overemphasized. "An ounce of prevention is worth a pound of cure" is a severe understatement in programming.

Now that we have devised a means of recording the mode attribute, modification of the arithmetic macros is entirely straightforward. Each of our new macros will invoke the corresponding old integer or floating point macro, depending on the mode of the operand:

```
LOAD    MACRO   ADDR
        IFEQ    ADDR$,0
        ILOAD   ADDR
        ELSE
        FLOAT   ADDR
        ENDIF
        ENDM
```

```
ADD     MACRO   ADDR
        IFEQ    ADDR$,0
        IADD    ADDR
        ELSE
        FADD    ADDR
        ENDIF
        ENDM

SUB     MACRO   ADDR
        IFEQ    ADDR$,0
        ISUB    ADDR
        ELSE
        FSUB    ADDR
        ENDIF
        ENDM

STORE   MACRO   ADDR
        IFEQ    ADDR$,0
        ISTORE  ADDR
        ELSE
        FSTORE  ADDR
        ENDIF
        ENDM
```

If the user forgets to declare a variable INTEGER or REAL before it is referenced in an arithmetic macro, the associated symbol with suffix $ will be undefined, and an error message will result.

As a final veneer for our macro programming efforts, consider the task of modifying our INTEGER and REAL macros to accept lists of variables, like FORTRAN:

```
REAL  FLOAT, SINK, BATHTUB
INTEGER  ENRY, IGGINS
```

Many assemblers, including COMPASS, have some pseudo-instructions specifically for handling such lists. However, in order to illustrate the general capabilities of COMPASS for processing character strings, we shall initially code our fancy INTEGER and REAL macros using only the most basic character manipulating instructions. At the end of this section we shall consider how we might have spared ourselves all this effort by invoking one of the more powerful COMPASS pseudo-instructions. We shall first describe the algorithm for our fancy INTEGER macro in a very informal, hopefully self-explanatory manner. Then we shall take the algorithm, step-by-step, and translate it into COMPASS pseudo-operations. Our new INTEGER macro has one parameter, a

character string which should be a list of symbols; let us call this parameter LIST. A possible algorithm for this macro is

1. Let L= the character string LIST with the characters ",." added at the end

2. Let FIRST = 1 and CURRENT = 1

3. Let CURRENT = CURRENT + 1

4. Let CURRCHAR = character number CURRENT of the string L

5. IF CURRCHAR is a comma,
   a. let NAME = the (CURRENT-FIRST) characters of L, strarting with character number FIRST
   b. invoke the old INTEGER macro with parameter NAME
   c. let first = CURRENT + 1

6. If CURRCHAR is not a period, go to step 3; if it is a period, algorithm is done.

The loop (steps 3 through 6) scans the character string from left to right; CURRENT points to the character position currently being examined. When a comma is encountered (step 5) the characters in the string from position FIRST up to the character preceding the comma are put into NAME; thus NAME will contain the name of the next symbol on the list. After the old INTEGER macro is invoked, FIRST is reset to point to the first character after the comma. The comma is appended to the string in step 1 so that the last list member will be processed; the period is added to mark the end of the string.

In order to be able to implement this in COMPASS, we have to introduce mechanisms for looping and for extracting substrings of character strings. The mechanism for looping or, more precisely, code duplication, is provided by the DUP pseudo-operation. The address field of the DUP may contain an integer, symbol, or expression; this number specifies how many times the following instructions, up to the next ENDD instruction, are to be duplicated. As a simple example, suppose we have an array of dimension 8:

```
ARRAY    BSS      8
```

and we wish to set it to zero. One way of doing this is

```
        MX6     0
        SA6     ARRAY
        SA6     A6+1
        SA6     A6+1
        SA6     A6+1
        SA6     A6+1
        SA6     A6+1
        SA6     A6+1
        SA6     A6+1
```

The identical code be generated by coding

```
        MX6     0
        SA6     ARRAY
        DUP     7
        SA6     A6+1
        ENDD
```

If the dimension of the array were assigned to a symbol

```
SIZE    SET     8
ARRAY   BSS     SIZE
```

the generation of code to zero the array can be controlled by this symbol:

```
        MX6     0
        SA6     ARRAY
        DUP     SIZE-1
        SA6     A6+1
        ENDD
```

The third operation in this family is the STOPDUP. When a STORDUP is encountered in the range of a DUP, duplication stops with the current iteration, regardless of the count in the DUP instruction. The current iteration is completed -- the following instructions up to the ENDD are assembled one last time. The STOPDUP is normally used with an IF instruction to control the number of times code is duplicated. For example, the above instruction sequence can also be coded

```
            MX6         0
            SA6         ARRAY
COUNT       SET         1
            DUP         1000
            SA6         A6+1
COUNT       SET         COUNT+1
            IFGE        COUNT,SIZE
            STOPDUP
            ENDIF
            ENDD
```

When duplication is to be controlled by a STOPDUP,the duplication count (here 1000) will normally be made so large that it will not be reached unless there is a coding error. The DUP and STOPDUP will enable us to implement the "loop" in our algorithm. The second mechanism we required for our algorithm was one to extract substsrings from character strings. This facility is provided by the MICRO operation, which has the form

```
micro-name      MICRO       n1, n2, daaaaaaaad
```

here n1 and n2 may be integers, symbols, or expressions; aaaaaaaa is any string of characters, and d, the delimiter character, is any character not appearing in aaaaaaaa. The value of the micro-name after this operation is the n2 characters of aaaaaaaa starting with character position n1. For example,

```
      NOW       MICRO       1,3,*HOW ARE YOU*
```

assigns the character string "HOW" to the micro NOW. If n2 is omitted, the substring from position n1 through the end of the string is assigned; thus after

```
      NOW       MICRO       1,,*HOW ARE YOU*
```

NOW has the value "HOW ARE YOU ".

After the micro has been defined, the micro name may appear anywhere in any instruction; however, it will be recognized only if the name is immediately preceded and followed by the micro mark ≠. For example, if we define the micro

```
      SETTWO    MICRO       1,,+SET  2+
```

and then use it in

```
      VARF      ≠SETTWO≠
```

COMPASS will first replace the micro by its value:

```
VARF    SET 2
```

and then assemble the instruction as usual. Micro substitution is performed at the same time as catenation (if there are any catenation marks in the line), after substitution for macro parameters but before any other analysis of the instruction. Thus, if we have the micro

```
SWEET   MICRO   1,,$+16$
```

and the macro

```
SETBI   MACRO   P1,P2
        SB1     P1≠P2≠
        ENDM
```

and then invoke the macro

```
        SETBI   VARF,SWEET
```

then micro substitution will be performed

```
        SB1     VARF+16
```

and finally the instruction will be assembled.

Micro names are not symbols. The value of a symbol is a number, while the value of a micro is a character string. Since references to micros are distinguished by the surrounding micro marks, it is possible to have a symbol and micro of the same name. Thus, with the definitions

```
SIX     SET     6
SOX     SET     8
SOX     MICRO   1,,+SIX+
```

the instruction

```
        SB1     SOX
```

assembles to a SB1 8, while

```
        SB1     ≠SOX≠
```

assembles to a SB1 6.

With micro and DUP in hand, we are ready to start work on our algorithm. Before we do, we shall rename our old INTEGER macro "INTEGER .", so that we don't end up with two macros of the same name:

```
INTEGER.  MACRO     VAR
VAR$      SET       0
          ENDM
```

Our algorithm requires two integer "variables;" for these we shall use the symbols FIRST. and CURRENT. (we include a special character in these names so that the user of our macro language will not have a symbol of the same name). Our new INTEGER macro begins with the macro heading

```
INTEGER   MACRO     LIST
```

Step 1 assigns to L the string LIST with "," appended; this clearly calls for a micro:

```
L         MICRO     1,,$LIST,.$
```

In using $ as the string delimiter, we tacitly assume that there will be no $ in LIST itself. Step 2 initializes FIRST. and CURRENT.:

```
FIRST.    SET       1
CURRENT.  SET       1
```

Steps 3 through 6 comprise the scanning loop, which we now know must begin with a DUP:

```
          DUP       72
```

Duplication will be terminated by a conditionally-assembled STOPDUP (step 6); since there can be only 70 characters in a one-card instruction (columns 73 and after are ignored, and there must be blanks between location and opcode and between opcode and address fields), LIST will surely be less than 70 characters, hence L will be less than 72 characters, hence an iteration count of 72 will not be reached. Steps 3 and 4 are straightforward

```
CURRENT.  SET       CURRENT.+1
CURRCHAR  MICRO     CURRENT.,1,$≠L≠$
```

Remember that when a micro is used in the address field of another MICRO operation, both micro marks and a delimiter character are required, since the value of the micro replaces the

micro name and marks in the address field before the instruction is assembled.

Step 5 says "if CURRCHAR is a comma..." This test is basically different from those we have made before: it compares two character strings, rather than the values of two symbols or expressions. For testing character strings there is another IF operation, the IFC, whose form is

```
IFC relation,daaaaadccccccd
```

where "relation" is any relational mnemonic (EQ, NE, GE, LE, GT, LT), "aaaaa" and "ccccc" are the two character strings being compared, and d, the first character after the comma, acts as the delimiter for both strings. As an example, the following macro will generate a store instruction only if the argument to the macro is "A":

```
STOA    MACRO   PAR
        IFC     EQ,*PAR*A*
        SA6     PAR
        ENDIF
        ENDM
```

The IFC test will succeed only if the argument is literally "A"; invoking the macro with another symbol of the same value

```
B       SET     A
        STOA    B
```

will not cause the store instruction to be generated, since it is the argument name, and not its value, which is being tested.

Thus step 5 becomes

```
        IFC     EQ,$≠CURRCHAR≠$,$
NAME    MICRO   FIRST.,CURRENT.-FIRST.,$≠L≠$
        INTEGER. ≠NAME≠
FIRST   SET     CURRENT.+1
        ENDIF
```

Step 6 uses an IFC to stop the scanning loop by assembling a STOPDUP:

```
        IFC     EQ,$≠CURRCHAR≠$.$
        STOPDUP
        ENDIF
```

Having come to the end of the loop, we finally terminate the range of the DUP:

```
            ENDD
```

Collect the instructions we have written over the last few pages and we have

```
INTEGER   MACRO     LIST
L         MICRO     1,,$LIST,.$
FIRST.    SET       1
CURRENT.  SET       1
          DUP       72
CURRENT.  SET       CURRENT.+1
CURRCHAR  MICRO     CURRENT.,1,$≠L≠$
          IFC       EQ,$≠CURRCHAR≠$,$
NAME      MICRO     FIRST.,CURRENT.-FIRST.,$≠L≠$
          INTEGER   ≠NAME≠
FIRST.    SET       CURRENT.+1
          ENDIF
          IFC       EQ,$≠CURRCHAR≠$.$
          STOPDUP
          ENDIF
          ENDD
          ENDM
```

Proud of our fancy new macro, we eagerly try it out:

```
            INTEGER   A,BB,CCC
```

and are, to say the least, a little disappointed when all that comes out is

```
A$        SET       0
```

The trouble is that, the way we have invoked the macro, there appear to be three arguments. COMPASS matches the first one, "A", with the first parameter in the macro definition, LIST; since there are no more parameters in the definition, "BB" and "CCC" are ignored. If we want to include a comma (or blank) in a macro parameter, we must enclose the parameter in parentheses, thus:

```
            INTEGER (A,BB,CCC)
```

In expanding the macro, the parentheses are removed and the text between the parentheses is substituted for the corresponding formal parameter in the macro body. Thus, the first line of the

macro becomes

```
L           MICRO     1,,$A,BB,CCC,.$
```

and this macro call does indeed generate

```
A$          SET       0
BB$         SET       0
CCC$        SET       0
```

Our original goal was to write one such macro for INTEGER declarations and another for REAL declarations. Rather than code this sequence twice, we shall (what else?) write a macro with two parameters: the list, and the macro to be invoked for each item on the list (in the above case, INTEGER.). In fond memory of a pseudo-operation in an earlier assembler which performed essentially the same function, we shall call this macro IRP (iterate prototype).

We will also require the two macros which IRP will invoke: INTEGER. and REAL.. And, finally, the two macros seen by the user of our macro language: INTEGER and REAL.

```
IRP         MACRO     MCNAM,LIST
L           MICRO     1,,$LIST,.4
FIRST.      SET       1
CURRENT.    SET       1
            DUP       72
CURRENT.    SET       CURRENT.+1
CURRCHAR    MICRO     CURRENT.,1,$≠L≠$
            IFC       EQ,$≠CURRCHAR≠$,$
NAME        MICRO     FIRST.,CURRENT.-FIRST.,$≠L≠$
            MCNAM     ≠NAME≠
FIRST.      SET       CURRENT.+1
            ENDIF
            IFC       EQ,$≠CURRCHAR≠$.$
            STOPDUP
            ENDIF
            ENDD
            ENDM

INTEGER.    MACRO     VAR
VAR$        SET       0
            ENDM

REAL.       MACRO     VAR
VAR$        SET       1
            ENDM
```

```
INTEGER   MACRO     VARLIST
          IRP       INTEGER.,(VARLIST)
          ENDM

REAL      MACRO     VARLIST
          IRP       REAL.,(VARLIST)
          ENDM
```

The parentheses around the list are removed when the list is substituted for VARLIST in the macro body; since the list must be enclosed in parentheses on the parameter list of the IRP call, we have to supply a fresh set of parentheses; hence "(VARLIST)".

Although our efforts at developing a macro package over the last few sections have been quite modest, relatively few programmers code macros as fancy as IRP. One factor is that those not very familiar with all the pseudo-operations believe they will spend more time debugging their macro than they could possibly save using it; unless the macro is being developed for general use, this argument is often valid. Another factor is that fancy macros assemble very slowly. In the case of IRP, the ten statements in the range of the DUP must be assembled once for each character in the list; an IRP with a list of six 7-character symbols expands to more than 500 instructions. As machine speeds continue to increase, this factor will decrease in importance, but it cannot be ignored. These problems notwithstanding, this example indicates how a "comprehensive" macro-assembler can be used to assemble a relatively sophisticated "higher-level" language.

Large, complex programs have an inherent tendency to get larger and more complex, particularly if they are heavily used; no matter now many features are included, there is always one more which someone would like to add. This is especially true when program designers are also constant program users, as is generally the case with assemblers (which are almost always written in assembly language). Features which the designer as program user finds useful can then easily find their way into the program.

The sequence of 6000 series assemblers (described on pages 53 and 54) certainly exemplifies this natural growth process. The first assembler manual -- for SIPROS ASCENT -- is a slim little thing describing 16 pseudo-instructions in four pages; the current COMPASS manual (version 3) needs more than 125 pages to describe its 80-plus pseudo-instructions. The largest increment, both in terms of number of pseudo-instructions (several dozen) and size of the reference manual (about one pound) occurred in the great

leap forward from version 1 to version 2 of COMPASS. A number of assembly-time operations which had proven particularly useful but which required complicated macros in version 1 were implemented as pseudo-instructions in version 2; among these was IRP.

In consequence, if you are diligent enough to punch and run the macros described above, you will be rewarded with the error flag "L" as follows:

```
L     IRP         MACRO       MCNAM,LIST
```

and the explanation

```
L     TYPE        ERROR       LOCATION FIELD BAD
```

What this means is that there already is an operation code IRP. If you consult your COMPASS manual, you shall discover that the IRP pseudo-instruction does pretty much the same thing our IRP macro was supposed to do. More precisely, if a macro contains the sequence of instructions

```
                IRP         LIST
                .
                .
                .
                IRP
```

and LIST is a macro parameter whose value is a list of items separated by commas, the instructions between the two IRPs will be assembled once for each item on the list. Furthermore, if the symbol LIST appears in any of the instructions between the two IRPs, it will be replaced by the first item in the list the first time the instructions are assembled, by the second item the second time they are assembled, and so forth.

For example, the following macro will generate code to increment each variable appearing in LIST by INCR:

```
      INK       MACRO       LIST,INCR
                SX2         INCR
                IRP         LIST
                SA1         LIST
                IX6         X1+X2
                SA6         A1
                IRP
                ENDM
```

If invoked by

```
            INK     (BRIG,VISP,ZERMATT),9
```

it will generate

```
            SX2     9
            SA1     BRIG
            IX6     X1+X2
            SA6     A1
            SA1     VISP
            IX6     X1+S2
            SA6     A1
            SA1     ZERMATT
            IX6     X1+X2
            SA6     A1
```

Returning to our arithmetic macros, we see now that they could have been written simply as

```
INTEGER     MACRO   VARLIST
            IRP     VARLIST
VARLIST$    SET     0
            IRP
            ENDM

REAL        MACRO   VARLIST
            IRP     VARLIST
VARLIST$    SET     1
            IRP
            ENDM
```

4.7 THE VALUE OF A SYMBOL

The object of this chapter, as I stated at the outset, has been to present the principles and potentialities of assembly language macro programming. It is not the object of this chapter to replace the COMPASS reference manual; this would be pointless if not impossible, since COMPASS is still a growing language. If I have been successful, you should now have little difficulty in learning the rules and restrictions of COMPASS, the operations we have not discussed and variants on those we have, as you need them from the reference manual.

I must confess, though, that there is one point on which I have intentionally been less than candid until now: the value of a symbol. As we have said several times, the value of a symbol is a number; however, there are numbers and there are numbers. The rest of this section will explore the wisdom of this last remark.

In the earliest systems, the addresses which a program would occupy were fixed at assembly time. The user would tell the assembler that the program would be loaded into memory at execution time starting at location 100, for example. Then, when the assembler encountered a JP Q, where Q was the label on the second word of the program, the assembler would generate a JP 101. This arrangement, known as absolute assembly, was simple but not very satisfactory; if the first routine in a program was made one word longer, all the other routines would have to be reassembled to start one location higher.

To eliminate this problem, the relocatable assembly was devised. In this scheme, the assembler assembles the program as if it were to be loaded starting at location 0; at the same time, it flags those instructions which reference labels in the program (for example, JP Q but not JP 7). A special routine, called the loader, is added to the system to read assembled programs into memory. The user can specify to the loader where in memory his program is to be loaded. Suppose the loader starts loading the program at location 100; then whenever it reads in an instruction which has been flagged by the assembler, it adds 100 to the instructions before storing it. In this way our JP Q, which was translated by the assembler into JP 1, would end up as JP 101, as before.

In a relocatable assembly the user can also reference an external symbol, i.e., a symbol in another routine. An instruction referencing such a symbol is marked by the assembler with the name of the symbol. When this instruction is loaded, the symbol name is recorded on a table; when the loader encounters an entry point of this name while loading another routine, it goes back and fills in the address in the first instruction.

So far we have three kinds of addresses: absolute addresses (i.e., constants, which are not relocated), (program) relocatable addresses, and external addresses. When we come to FORTRAN IV, the situation is further complicated by labeled common. As long as all variables were local (part of the subprogram being compiled), the compiler just needed one counter to keep track of the address to assign to the next statement or variable. If words 0 through 704 (relative to the beginning of the program) had been used, this counter would be at 705; to process

```
                DIMENSION A(10),B(10)
```

the compiler would assign words 705 to 714 to A, words 715 to 724 to B, and leave the counter at 725. With labeled common, the compiler has to maintain a separate counter for each block. When the first declaration for block A is encountered,

```
                COMMON /A/F,G
```

the compiler assigns F to word 0 relative to the start of common block A, G to word 1, and leaves the counter for block A at 2. This may be followed by a declaration of a block B:

```
                COMMON /B/LINE
```

a counter for B is set up as a result. Still later the program may continue the declaration for A:

```
                COMMON /A/H,I
```

The compiler must then return to the counter for block A, allocating H to word 2 relative to the start of common block A, etc.

Common blocks may be declared by COMPASS by means of the USE pseudo-instruction, which has the form

```
                USE  /block-name/
```

This instruction tells the assembler to start using the counter for common block "block-name." For example, the FORTRAN declarations

```
                COMMON  /A/F,G
                COMMON  /B/LINE
                COMMON  /A/H,I
```

would be coded in COMPASS as

```
                USE     /A/
     F          BSS     1
     G          BSS     1
                USE     /B/
     LINE       BSS     1
                USE     /A/
     H          BSS     1
     I          BSS     1
                USE     0
```

The instruction SA1 H would then be translated into SA1 2 and marked common relocatable with common block name A. The loader would set aside four words for block A, and add the starting address of block A to all instructions marked with that block name. The final USE 0 specifies that following code is to go into the usual, program relocatable, block.

Thus we have four kinds of addresses the loader will recognize: absolute, program relocatable, common relocatable, and external. What restrictions does this imply for assembly language coding? In the first place, every address expression must fit into one of these four types. For example, if A and B are program relocatable symbols, the instruction

```
        SB1     A+B
```

is unacceptable because there is no way of indicating to the loader that twice the address of the start of the program is to be added to the instruction. On the other hand,

```
        SB1     A-B
```

is acceptable since the difference of two program relocatable symbols is an absolute expression.

Second, the value of the operands of some pseudo-instructions must be determined at assembly time; these operands must be absolute expressions. One such instruction is DUP:

```
        DUP     A
```

where A is relocatable, is an error, since the assembler cannot wait until the program is loaded to decide how many times to duplicate the following code.

The remaining restrictions on the use of symbols hinge on whether they are defined or not. This question is often confused by the fact that "defined" may mean two different things. By considering the modus operandi of the assembler, we should be able to make all these restrictions clear and logical.

Consider the problem an assembler faces in processing

```
        JP      THERE
THERE   PS
```

The problem is quite simple: when the assembler encounters the JP instruction, it has no idea what the value of THERE is, so it

cannot generate any code. Yet we know the assembler handles such a sequence; what does it do? The solution is equally simple: it reads through the program twice. The first time it decides how much space each instruction occupies, and thus determines the value of every symbol; the second time through it knows the values of all symbols, so it can evaluate address fields and generate code. Each reading through of the source program is called a pass; thus COMPASS, and nearly every other assembler, is a two-pass assembler.

Any expression that affects the amount of code to be generated must be evaluated in pass one. Such an expression may therefore contain only previously defined symbols, i.e., symbols whose value can be determined from the portion of the program already read. Expressions on which this restriction is imposed include the repetition count in DUP instructions, the word count in BSS instructions, and the two expressions which are compared in the various IF instructions. If you look back over the examples in this chapter, you will notice that we have been careful to define symbols (usually by SET instructions at the beginning of the program) before they were tested by IF instructions.

Expressions which do not affect the amount of code generated, principally the address fields of machine instructions, are evaluated during pass two. These expressions may reference any permanently-defined symbol (any symbol defined by its appearance in the location field of a machine instruction or BSS), since the values of such symbols are recorded during pass one. The values of SET-defined symbols, however, cannot be saved from pass one to pass two, since these symbols may take on many values in a program. Such symbols are therefore reset "undefined" at the beginning of pass two, and remain undefined until the first SET instruction for this symbol is encountered. Consequently, SET-defined symbols may not appear in any expressions unless they have been previously defined.

CHAPTER 5

DEBUGGING

5.0 INTRODUCTION

A large portion of the time devoted to program development is spent in correcting errors (bugs) that occur in a program. The process of locating and fixing these bugs is known as debugging. Debugging can be, and usually is, the most tedious part of programming. It is a task that is very rarely escaped, and for the beginning programmer, it is a frequent ritual. Debugging also tends to be an ongoing process, mostly due to poor initial program design, little forethought, and inadequate testing on the part of the programmer.

Bugs fall into two major categories: syntax errors and logic errors. Most compilers and assemblers assist the programmer with syntax errors by pointing out the type of error present and the statement in which it occurs. Some compilers even correct syntax errors. These aids enable the programmer to quickly and easily fix these types of bugs.

On the other hand, logic errors are often hidden deep within the inner workings of a program, and their effects may not be detected immediately after their generation. For example, a division may produce an indefinite, but the program will not abort until this value is used in another arithmetic operation, which may be many instructions later, or even in another subroutine. Logic errors may also be intermittent, that is, they may cause the program to fail only under certain conditions. For these reasons, logic bugs give the programmer a much more formidable challenge than do syntax errors.

Compilers and assemblers do not offer any aid to the programmer for finding logic bugs; bugs of this type require special techniques. These techniques vary with the power of the programming language used by the programmer. Since assembly language is a low-level language with no I/O primitives, its fundamental debugging technique is the examination of register

and memory contents. Through this examination, the debugger can trace erroneous values to the instruction that produced them.

In the past, this examination was done by physically entering memory addresses into a computer by sense switches and reading their contents from a panel of lights located on the computer. This technique required the programmer to have full use of the computer while he was debugging. For a machine the size of the 6600, which costs several dollars a minute to operate, this approach would be expensive. In its place, computer manufacturers have developed a number of debugging aids. These aids range from register and memory dumps to interactive debuggers which allow the user to examine registers and memory, as well as change their contents, while the program is executing.

Two important tools, the assembly listing and the load map, are needed by the debugger before any attempt is made to find bugs. These tools display the memory layout of the program, allowing the programmer to determine which registers and memory locations are of interest. The first part of this chapter will discuss these tools, and the second will discuss the debugging aids available to the COMPASS programmer.

5.1 THE LISTING

Generally, assembly listings generated from COMPASS subprograms consist of at most five sections. These sections are (in order of appearance): the header page, the octal and source listing, assembler statistics, the error directory, and the symbolic reference table. Each of these sections, with the exception of the error directory, appears for each subprogram in the source text. The name of the subprogram appears at the top left hand corner of each page belonging to that subprogram. Pages A1-A4 (Appendix A) show a typical listing.

The header page (A1) is analogous to a table of contents. It designates the major sections of code that are defined in the subprogram, specifying the name, starting address, and length of the subprogram; the names, types, starting addresses, and lengths of the blocks; the names and addresses of the entry points; and a list of all external symbols defined by the EXT pseudo-op. Blocks that appear in the subprogram are usually

common data blocks or program blocks. The name PROGRAM* refers to the primary block of the subprogram, the block into which all instructions and data are put if no USE pseudo-instruction appears. If the subprogram does not contain any common blocks, the block usage description does not appear.

The octal and source listing (A2) is the most informative section of the listing. Each line of the source text is printed alongside the octal code generated for the instruction. Instructions that are packed into one word are displayed in an indented fashion, each successive instruction being indented further. The first instruction packed into a word is preceded by the relative address of the word in which the instructions are to be stored. Notice that no-ops used to fill words are not displayed.

If an instruction contains an address, the octal code may be suffixed by a +, C, or X. These flags indicate how the address is to be treated when the code is being loaded. The "+" indicates that the address is relocatable. The "C" means that the address references an element in a common block. The "X" indicates that the address is external to the subprogram.

If any syntax errors occur in a source statement, a flag appears on that line, in the left hand margin. The flags are single letters. It is possible to have more than one flag appear for a single source statement. Some examples of syntax error flags appear on the listing on page A2.

The assembler statistics appear immediately following the octal and source statement listing. The statistics give information about the size of the subprogram, and the length of time required to assemble it. They are of little interest to the debugger.

The error directory (A3) appears only if syntax errors occur in the source text. Each type of error flag in the octal and source statement listing is displayed with a brief description of the error and the pages in which the error occurs.

The symbolic reference table (A4) is a quick index into the subprogram. It aids the programmer in both debugging and code modification. Each symbol used in the subprogram appears in the table with its value and a series of page/line references. The references specify the pages and lines on which the symbol appears. Some of the page/line references are postfixed by a flag. The flags and their meanings follow:

D	Symbol definition statement (EQU or SET)
E	Symbol defined as an entry point
F	Symbol is used in condition test
L	Symbol used in location field
S	Symbol used in store instruction (SA6 or SA7)
X	Symbol defined as external

If a "U" prefixes a symbol's name, the symbol is undefined, implying one of three conditions: the symbol was not declared external, its definition was omitted, or the symbol is a typographical error.

5.2 THE LOAD MAP

A binary file produced by a compiler or an assembler from a source program is not yet in a form that is executable. The compiler or assembler has coded each common or program block separately, with each one beginning at address zero. Since blocks cannot be in the same memory locations, they must be relocated (see section 4.7). Also, many library routines that are referenced but not included in the source program must be added (for example, math routines such as SQRT and SIN and FORTRAN I/O routines). The addition of library routines, linking of subroutines, relocation of blocks, and all other tasks necessary to get a binary file into executable form are done by the loader.

For each common and program block in the binary file and for added library routines, the loader allocates a block of memory. The first location of a memory block is called the starting address of the common or program block. These starting addresses are used by the loader to adjust the relocatable, external, and common addresses referenced in the program blocks. The type of adjustment is specified by the flag on the octal and source statement listing. The address at which the entire program starts is called the first word address (FWA).

A load map displays the relocation information. It gives the name of each block plus its starting address and length, the name of the file containing the binary code, the date the binary code was created, and other miscellaneous information about the block. Blocks are printed in ascending order of starting addresses. The load map is produced by the control card MAP (PART), which may

appear anywhere prior to the invocation of the loader. (Some systems automatically produce a load map.)

Page A9 contains the load map for the FORTRAN program on page A5 and the COMPASS routine on pages A6-A8. Looking at the map, we see that the common block VECTOR is loaded at the lowest address, 111 (user defined common blocks are always loaded before the first routine which references them). The routine VECLIB is loaded at location 2232 and consists of 15 words of instructions and data. As VECLIB is being loaded, the loader adds the starting address, 2232, to each relocatable address. For example, the assembler generates 0400000012+ for the EQ LOOP1 instruction at the bottom of page A6, but when the routine is reloacated the loader adjusts the code to 0400002244 (2232 + 12). Note that most of the load map consists of "library" routines that have been added to the original binary file (LGO).

5.3 THE DUMP

The most basic debugging aid available to the COMPASS programmer is the dump. A dump is a print-out of the contents of consecutive memory locations. The range of locations printed is stated on the first line of the dump. The contents of four locations are printed on each line, with the address of the first location printed on the left hand side of the page. The message "duplicated lines" appears if successive groups of four locations have the same value as the last group that was printed.

The dump is generated by the control card DMP (start,end); the parameters (octal values) specify the starting and ending memory locations of the dump, respectively. If only one parameter is given, a dump is produced that starts at the first word address (FWA) and terminates at the parameter value. Page A10 illustrates a dump produced by the control statement DMP (100,200).

The dump gives the debugger "postmortem" information, that is, values in memory at the time of program termination (either normal or abnormal). Although the dump does not indicate how these values change during execution, it can give the debugger some clues in finding bugs. For example, the COMPASS routines shown on pages A7 operate on the vector VEC. If we suspected

that one of these routines was malfunctioning, we could use a DMP command to print the final value of this vector. VEC is in the common block VECTOR at the relative address zero. Looking at the load map (A9), we see that VECTOR's starting address is 111. Therefore, we can include a DMP control card with appropriate parameters to see if the correct values are present in the vector. (DMP(111,123) would do the trick if the vector were ten elements long.) By examining the dump, we can determine if the values seem totally incorrect, or if something more subtle has occurred, e.g., only nine partial sums were computed instead of ten. (This error is frequent, and is known as the off-by-one bug.)

A dump is also produced by the operating system when a program terminates abnormally (see example on page A16). It is 100 octal memory locations in length, with the instruction that caused the program to terminate in the middle. Preceding the dump are the contents of the first two words of the user's memory area, RA (reference address), and RA+1. Bits 30-47 of RA are of special interest to the debugger. They specify the memory location of the next instruction word to be executed when the program terminated. The instruction causing termination, therefore, is in the previous word. (On a machine having multiple functional units, like the 6600 (see section 6.1), the instruction causing termination may be located in one of a few previous words.) Above RA and RA+1 is the exchange package. The exchange package gives the values in the registers at program termination, the contents of the memory addresses stored in the A registers, and a series of program parameters. The only program parameter of interest is the field length (FL).

A dump produced by the operating system is very useful to the debugger, and can often lead directly to the instruction causing the program to terminate. This is possible because the register values printed in the exchange package are the values in the registers at the time of the error. Thus, by inspecting these values, important leads can be discovered.

Pages A11-A14 show a COMPASS function WORDS for computing the number of words in a sentence, along with a FORTRAN main program which calls WORDS. Unfortunately, as the dayfile on page A17 shows, this program "bombs out" with a CM OUT OF RANGE message. Let us consider how we would track down the source of this error. The dayfile message means that the program referenced a memory location outside of its field length. Either the program branched to an address outside the field length, or one of the A registers has been set to a value greater than or equal to the field length. Looking at the exchange package (A16), we compare

the field length (FL) with each of the A registers (except of course A0, since assigning a value to A0 does not cause a memory reference). We see that FL is 15200, and the address in A1 is 15236. This tells us that the instruction that terminated the program was a SA1.

Now our job is to find the SA1 instruction. Looking at page A16, we see bits 30-47 of RA contain 4564 (we could also have obtained this address from the dayfile message, CPU ERROR EXIT AT 004564). Since our program was executed on a Cyber 720, which does not have multiple functional units, we know that the instruction causing program termination is in location 4563. Examining the load map, we note that the largest starting address less than 4563 is that of WORDS, 4556. By subtracting the two values, we find that the instruction is in the program WORDS at relative address 5. Looking back at the listing (A13), we see that word 5 contains a SA1 A1+B2; our culprit is found!

5.4 REGDMP

Using the dump and exchange package, we were able to find the instruction that contained a bug, but this did not give us any clues about the cause of the bug. However, we must realize that any postmortem data about a program gives little information about dynamic changes in register and data values. Also, if a program terminates normally, but produces wrong answers, postmortem dumps may be of no help at all.

Thus, a more dynamic way of examining registers is needed -- a method of looking at the values of registers while the program is executing. This is not a trivial task. Any program that is going to allow us to look at the contents of the registers must use the registers itself, but we do not want the contents of the registers to be altered. Impossible you say; well, it isn't. Appendix B contains a routine called REGDMP and its output routine that do the job. REGDMP is a utility that saves the contents of all 24 registers, calls a FORTRAN routine to print their contents, and then restores the previous register values. It is a very cleverly designed routine, and the reader is invited to peruse the code.

To use REGDMP, it must be first declared external to the subprogram. The debugger can then call the routine before or

after any instruction under suspicion (assuming that REGDMP has been installed in your system library). The output lists the location from which the call was made, the number of times the routine was called, and the contents of the 24 registers. We continue the "bug hunt" from the previous section to show how REGDMP is used.

We found that the SA1 A1+B2 instruction terminated the program, but we are still unsure of the underlying cause of the error. We need to examine the values being assigned to A1. Placing a call to REGDMP before the loop gives us the initial address of the array being searched; this is necessary for comparisons to subsequent values of A1. A second call to REGDMP is placed after the SA1 A1+B2. The output of this call gives the needed information about the dynamic changes in A1. (Page A18 shows the code with the REGDMPs added.)

The output of the REGDMP (A19 - 20) displays the changes to A1. The first gives the initial value with subsequent REGDMPs giving values generated in the loop. The program WORDS is designed to search through successive elements of an array, but we see from the values of A1 that it goes through the array in leaps and bounds.

The bug has been found: A1 is being assigned its previous loop value plus the increment value of B2. This produces larger and larger addresses, causing the program to eventually access a location outside its field length. (Note that if the array were smaller, the program would not have aborted, but simply have given incorrect results; in this case, the described technique would still succeed.) We can fix the bug by replacing the SA1 A1+B2 with SA1 A1+1.

5.5 REMARKS

The scenarios of the previous sections describe techniques using the available debugging aids. Considerable practice is required before one can master them. For this reason, debugging is often called an art. The skillful debugger, like the experienced detective, learns to enjoy the hunt and to remember the significance of the many different clues he has seen.

The best form of debugging, though, is good programming. A thorough understanding of a problem and its inputs, plus careful program design decreases the amount of debugging. Well-designed and commented code helps in finding and correcting bugs. Programs with many branches and devious bits of code often make modifications difficult, and can lead to more bugs.

CHAPTER 6

OPTIMIZATION

6.1 MACHINE ARCHITECTURE AND CODE OPTIMIZATION

The goal of this chapter is as modest as that of the last: to describe those features of the 6000 (Cyber 70 Model 74) which the programmer can take advantage of in optimizing his program. We shall not discuss the 6200 or 6400/6500 central processors (Cyber 70 Models 72 and 73) or the Cyber models 171-174, 720, and 730; the optimization possible on these machines is rather limited, and the few pertinent rules are included in both the COMPASS and hardware reference manuals. Nor shall we consider all the details of 6600 instruction timing; this would in fact be quite difficult because 6600's are not all identical in some aspects affecting timing.

The high speed of the 6600 is due in good part to two unusual features of the central processor: multiple functional units and an instruction stack. The 6200, 6400 and 6500 central processors (and Cyber 72, 73, 171-174, 720 and 730), like most computers, have a "unified arithmetic unit," a single "box" of electronics which executes all instructions. As a result, instructions are executed sequentially (i.e., one at a time). In contrast, in the 6600 central processor the hardware to execute the instructions has been divided into several functional units, each of which can execute a few of the central processor instructions. Each functional unit can only handle one instruction at a time, but several units can be operating concurrently. Thus, if two instructions in sequence do not require the same functional unit, and the result of the first instruction is not used as an operand of the second, the two instructions can execute in parallel.

The control unit in the central processor decodes (analyzes) the instructions to determine the functional unit, operand, and result registers required, and dispatches the instructions to the appropriate functional unit. To keep track of all the functional units and the registers they require, it uses a complex maze of registers called the scoreboard. The following outline of the stages in the execution of one instruction gives some idea of the scheduling problems involved:

1. Decode instruction, determine functional unit, result and operand registers;

2. If functional unit is busy (executing another instruction) wait until unit is free;

3. If result register is reserved by another functional unit which is currently busy and which will store its result in this register, wait until the other functional unit stores its result;

4. Pass instruction to functional unit (this is called "instruction issue");

5. Reserve result register for this functional unit, so that a subsequent instruction which uses that register will wait until the current instruction has been completed;

6. Functional unit waits until operands are available (until operand registers are not reserved, as defined in step 3);

7. Instruction is executed by functional unit;

8. Functional unit stores result in result register and clears the reservation of that register.

The control unit can decode and issue an instruction every minor cycle (100ns), so the 6600 has a theoretical limit of 10mips (million instructions per second). Register and functional unit conflicts in typical unoptimized code reduce the speed to something like 3mips. Careful optimization, by reducing register and functional unit conflicts, can sometimes double execution rate, to around 6mips.

The 6600 has ten functional units: branch, Boolean, shift, long add, floating add, divide, (2) multiply, and (2) increment. The multiply and increment units are duplexed; that is, there are two identical functional units of each kind, and an instruction is issued to whichever unit is free. The instructions executed by each functional unit and the execution time for each instruction are tabulated for the convenience of the reader in the last section of this book.

A few general observations can be made about these execution times. One is that branch instructions are abysmally slow; the 6600 is one of the few machines which takes longer to branch than to perform a floating point multiplication. An X-register branch, for example, usually takes 15 minor cycles (1500ns) if the branch is made, and 14 minor cycles if the branch is not taken. During those 14 or 15 cycles, no other instructions are issued. Consequently, an efficient programmer will avoid using branch instructions when equivalent sequences without branches will do as well; a prime example is the absolute value function (see section 3.14). Under certain circumstances jumps will take less time; these conditions will be discussed at the end of this section.

Another significant observation is that, relative to the other instructions, a load from memory takes quite a long time. The set-A instruction executes in three minor cycles; an additional five cycles are required to fetch the word from memory (more if there is a conflict with other memory accesses), so that a total of eight cycles is required to perform a load. As a result, the load instruction should generally not be placed immediately before the instruction which uses the word loaded; whenever possible, the load should be placed a few instructions earlier, so that the memory fetch is overlapped with other calculations. For example, suppose our program must jump to EXIT if either B1 is negative or the contents of SWITCH are non-zero. Until now, we would have coded:

```
NG      B1,EXIT
SA1     SWITCH
NZ      X1,EXIT
```

If the NG branch is not taken, the program must wait eight cycles after the NG is completed until the X1 load is finished and the NZ instruction can start. If we put the load first, however,

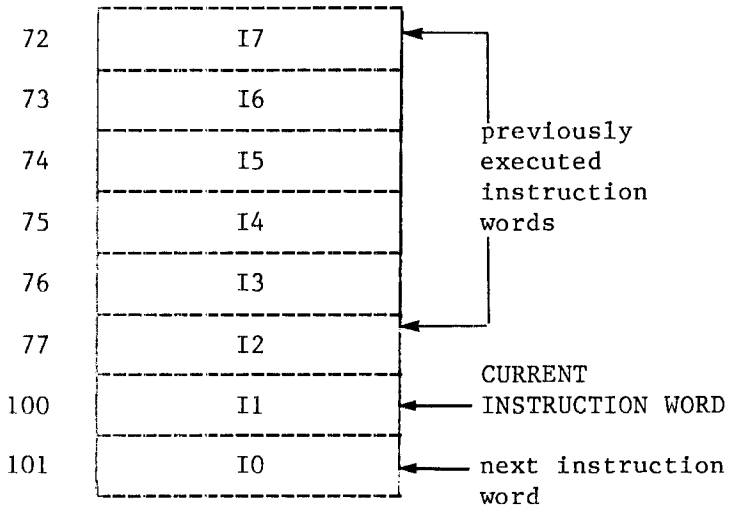
```
SA1     SWITCH
NG      B1,EXIT
NZ      X1,EXIT
```

all but one cycle of the load operation is overlapped with the NG instruction; unless the NG branch is almost always taken, the second sequence is faster.

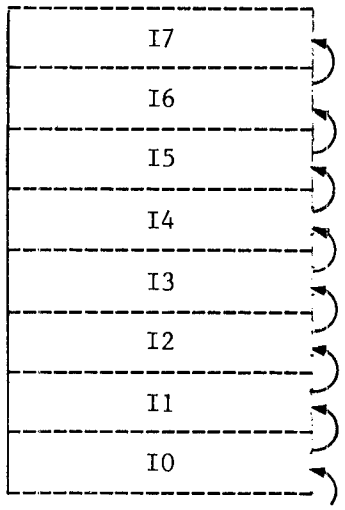
These simple rules go only a short way towards maximizing instruction overlap. A high level of optimization is usually achieved only through a lengthy procedure, which starts with a timing analysis of the code sequence, tabulating instruction

issue, execution start, result available, and functional unit free times for each instruction. This analysis is then scrutinized to determine where the greatest delays occur; instructions are then reordered or registers reassigned to reduce the delay, and the timing analysis is repeated.

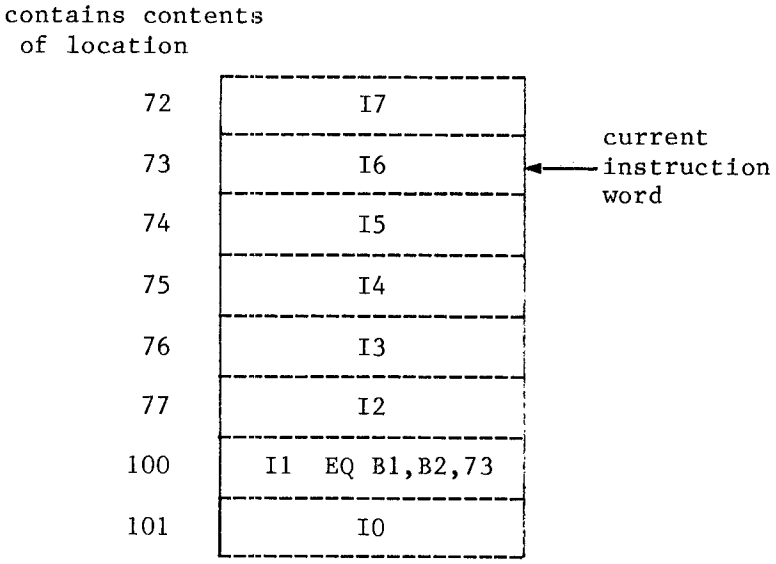
We now turn to the other unusual feature of the 6600 central processor: the instruction stack. The stack is a set of eight 60 bit registers, called I0 through I7. During the sequential execution of instructions (no transfers), I1 holds the current instruction word, I2 through I7 contain previously executed instruction words, and I0 contains the next instruction word:



After all the instructions in the current instruction word have been issued, the contents of each I register are moved into the next-higher-numbered register; the contents of I7 are lost, and the next instruction word from memory is brought into I0:



Normally, when a transfer is made the stack is "flushed" (all its entries are cleared), and the instruction word at the transfer address is brought into I0, and from there into I1. If the target instruction word is already in the stack, however, no memory reference is necessary; the current instruction word pointer is simply reset to point to the I register containing that word. For example, if an instruction in location 100 causes a transfer to 73, the pointer will be set to I6:



This will happen whenever a loop of seven or fewer instruction words (an "in-stack loop") is being executed. As long as the program remains in this loop, all instructions are taken from the stack. Since an in-stack branch need not wait for a new instruction word to be fetched from memory, it takes much less time (600ns less) than an out-of-stack branch. In addition, while in the loop no delays are possible due to instruction fetches. Short program loops may therefore be speeded up significantly if they can be made to fit into the stack.

There are several means for shortening loops which almost fit into the stack. The most obvious method is the removal from the loop of instructions which can just as well be executed before the loop begins. Setting registers outside the loop will also often be helpful. Even if this does not eliminate an instruction in the loop, it may be possible to replace a 30 bit with a 15 bit instruction. A SB1 B1+1 inside a loop, for example, can be replaced by SB7 1 prior to the loop and SB1 B1+B7 in the loop.

Finally, we should note that the unconditional branch instructions always flush the stack, even if the target instruction word is in the stack. The RJ and JP instructions therefore cannot be part of an in-stack loop (this is the reason for using EQ B0,B0,... everywhere instead of JP).

The 7600 uses the same techniques as the 6600 for increasing processor performance--multiple functional units, instruction overlap, and an instruction stack. As a result, most of the techniques described above for optimizing 6600 code are applicable to the 7600 as well. The main difference of the 7600 design is that the functional units are pipelined. A pipelined instruction unit is divided into stages corresponding to the steps required to perform the instructions. The clearest example is floating-point addition, which must be done in four steps: subtracting exponents, shifting one coefficient, adding coefficients, and possibly shifting the result by one bit. In executing one instruction, the operands come in at one end of the "pipe", they move through these four stages, and a result comes out at the other end. The advantage of a pipelined unit is that several pairs of operands can be in the unit, at different stages of processing, at the same time. Once a pair of operands has moved from stage one to stage two, a new pair of operands can be accepted into the first stage. For example, floating addition on the 7600 requires four cycles (110 ns); if there are several successive floating point addition instructions, the first one could start at cycle n, producing a result at cycle n+4; the second could start at cycle n+1, producing a result at cycle n+5; the third could start at cycle n+2, etc. Most 7600 functional

units can accept new operands every cycle; the multiply unit accepts new operands every second cycle, and the divide unit (which is not pipelined) every 18th cycle. The internal design of the 7600 has been carried forward to the Cyber 70 model 76 and the Cyber 170 models 175, 176, 750, and 760.

6.2 OPTIMIZING THE PROGRAMMING EFFORT

The difference between optimization in the small and optimization in the large is the difference between good coding and good programming. We have briefly considered the former in the last section; in this final section I shall venture a few words on the latter.

Efficient program development is largely a matter of exercising good judgment in program design and choice of implementation; as such, it cannot readily be reduced to a set of hard and fast rules. It is usually learned only through bitter experience, when it is learned at all. The most I can hope to do in these last few pages is to make the experience a little less bitter by repeating some warnings I have made before on three points:

1. Design carefully. One of the most common failings of program planners is an optimism concerning programmer ability, which obscures the ease with which even the best programmers can founder in a mass of detail. Even major computer manufacturers with years of experience regularly plan systems an order of magnitude more complex than earlier systems, and then are surprised when they are "ready" years late, if at all, and still contain an order of magnitude more bugs than the earlier systems.

This is not to say that ambitious projects never succeed, but rather that they require careful planning to eliminate needless detail, to develop a simple structure for the entire program, and to prepare complete documentation for the system.

2. Use the appropriate language. If you have digested the last six chapters -- and, more important, have coded enough routines on your own -- you now possess two of the most important skills of any programmer: machine language and macro programming. I hope you have also gained some appreciation for what I feel to be the beauty of this machine and the elegance of some of the tools, such as the macro. Still, one must keep in mind that programming languages are only tools, each with its proper role.

When a suitable high-level language is available, use it; it will make the program easier to write, debug, modify, and read. "Descend" to assembly language when time or space requirements are critical, as they often are in system routines, or when no high-level language is suitable. Use macros whenever appropriate, not just to shorten the source program, but to indicate aspects of the structure of your program.

3. Optimize code only where necessary. Some people find (timing) optimization a lot of fun, but good optimization is very time-consuming (for the programmer) and the resulting code is generally harder to debug, modify, and read. Do it only for code which will be executed very often, such as the SQRT routine or the inner loop of a matrix multiply. And keep in mind that there is another kind of optimization -- minimization of space required -- which is more important for the many system routines which are loaded into memory very often, but executed infrequently (initialization and error routines, for example). Space optimization is generally a matter of register allocation and subroutine design and is, except in the most extreme cases, less tedious than time optimization.

Above all, some perspective and common sense are needed in deciding when and how to optimize. A good system designer is willing to sacrifice some efficiency in individual routines in order to have a cleanly structured system which can be debugged faster and maintained more easily. Don't become one of those people who are so obsessed with saving computer time that they forget that computers are supposed to save people time.

EXERCISES

Given below are twenty-five simple exercises in assembly language programming; these are presented as specifications for subroutines, functions, or macros. The first fourteen can be done after reading Chapter 3, the next seven after Chapter 4. I have intentionally not keyed these exercises to individual sections, since many exercises that can be done after reading part of a chapter may profitably be recoded into more efficient routines after progressing further in the text. The last four problems are related to the discussion of the Compare and Move Unit, section 3.16, but they can all be done on a machine without a CMU.

Solutions to a few exercises which I found of particular interest are included at the end. Readers may also wish to write some of these routines in FORTRAN, and compare the code generated by the compiler with their own assembly language versions. (Both the RUN and FORTRAN Extended compilers will optionally produce a listing of the generated code).

1. Write a function INDEX with three arguments:

```
INDEX (ITEM,LIST,LENGTH)
```

where LIST is a one-dimensional array of LENGTH words, whose value is the index of the first occurrence of ITEM in LIST, starting with LIST (1); if ITEM does not appear in LIST, INDEX returns 0. Stated another way, INDEX returns n if LIST (n) = ITEM and LIST(1) through LIST(N-1) are all ≠ ITEM.

2. Write a subroutine SORT with two arguments:

```
SORT(LIST,LENGTH)
```

where LIST is a one-dimensional array of LENGTH words containing integer values. Sort should rearrange the elements of LIST into ascending sequence, with LIST(1) containing the smallest element.

Remark: Any sorting algorithm will be satisfactory. For those not familiar with any sorting methods, may I suggest an exchange sort, which could be coded in FORTRAN as follows:

```
      LM1 = LENGTH-1
      DO  10  I=1,LM1
      IP1 = I+1
      DO  10  J=IP1,LENGTH
      IF (LIST(I).LE.LIST (J)) GO TO 10
      ITEMP = LIST(I)
      LIST(I) = LIST(J)
      LIST(J) = ITEMP
10    CONTINUE
```

Of course, the assembly language version of this routine should require fewer memory references. Not only is it possible in assembly language to interchange the contents of two variables without using a third, temporary variable, but it is also possible to keep LIST(I) in a register through the loop on J (so that at most one load and one store are required for each iteration of the inner loop).

3. Write a subroutine TRANSPO with two arguments:

```
TRANSPO(MATRIX,N)
```

where MATRIX is an N*N array. TRANSPO should replace MATRIX with its transpose (i.e., interchange every pair of off-diagonal elements MATRIX(I,J) and MATRIX(J,I)).

Remark: This is essentially an exercise in efficient array indexing. Plan your choice of indices carefully before starting to code so that the number of operations in the inner loop will be minimized.

4. Write a function NFACT of one argument

```
NFACT(N)
```

which returns the value of N factorial (if N=0, it returns 1).

Remark: If the integer multiply feature is not installed, the usual FORTRAN loop for factorial,

```
      NFACT = 1
      DO  10  I=2,N
10    NFACT = NFACT*I
```

will most likely generate code to pack NFACT before each multiplication and unpack the product. Your assembly language routine can maintain NFACT in "packed" format (i.e., as an unnormalized floating point number) and only unpack the final result.

5. Write a function EXP of one argument:

EXP(X)

which returns the value of e**x

Remark: Use the series expansion for the exponential function,

exp(x) = 1 + x + x**2/2! + x**3/3! + ...

Keep adding terms until the current term divided by the sum of all previous terms is less than a given number, say 1.E-10.

Remark: If you encounter difficulties for large negative values of x (due to the finite precision of the floating point arithmetic), compute exp(abs(x)) and then use exp(-x) = 1/exp(x).

6. Write a function STDDEV of two arguments:

STDDEV(DATA,L)

where DATA is a one-dimensional array of length L containing floating point values. STDDEV should return the standard deviation of the elements of DATA, defined as

$$\{(1/(L-1))*\sum (DATA(i) - \overline{DATA})**2\}**(1/2)$$

where $\overline{DATA}$ is the average of the elements of DATA, and the sum ranges over i from 1 to L.

Remark: To evaluate the formula given above, the routine must go through DATA twice, first to compute the average and then to compute the sum of the squares of the differences from the average. This formula can be rewritten as

$$\{(1/(L-1))*\{\sum (DATA(i))**2 - (\sum DATA(i))**2/L)\}\}**(1/2)$$

Using this second formula, the standard deviation can be computed after one pass through DATA, in which both the sum and sum of squares are computed.

Remark: As the final step in the routine, to compute the square root, one can call the FORTRAN library SQRT function, just as one would in a FORTRAN routine. Don't forget to include an EXT SQRT in STDDEV.

7. Write a function MULT of two arguments:

MULT(N1,N2)

which returns the 60 bit product of the integers N1 and N2. Two methods are suggested:

(a) Use the "shift and add" method described in Chapter 1. That is, if the low bit of the multiplier is a one, add the multiplicand into the running sum of partial products; then shift the multiplicand left one bit and the multiplier right one bit. Repeat these two steps until the multiplier is zero. This is a very slow method, but is helpful in doing problem 8.

(b) after packing both integers (if the integer multiply feature is not available), compute both the floating and DP products; then combine the low 12 bits of the floating product with the 48 bits of the DP product to form a full 60-bit product.

8. Write a function RBAIEX of two arguments:

RBAIEX(R,I)

where R is floating point and I integer, which returns the value of R**I (RBAIEX stands for real base, integer exponent). The obvious method is to multiply R by itself I times; a more efficient algorithm, however, is analogous to suggestion (a) for the previous problem: Initialize the result to 1. If the low order bit of the exponent is a one, multiply the base into the result. Square the base and halve the exponent (shift right one). Repeat until the exponent is zero.

Remark: Check if the exponent is negative; if so, return 1/R**IABS(I).

9. Write a subroutine similar to CDC (section 3.15) which produces an octal instead of a decimal integer, i.e., according to O20 instead of I20 format.

Remark: This routine is much simpler than CDC, since O20 format always produces 20 digits, no minus sign, and no "R". The algorithm required is in fact very similar to the character unpacking routine, except that here 20 3-bit fields must be unpacked, and each converted to a display code digit. (The divide and find remainder algorithm used in CDC would not work for numbers larger than 2**48 , and hence cannot be used here).

10. Write a subroutine IDN (interpret decimal number) with two arguments:

IDN(NUM,CHAR)

where CHAR is an array of 20 words, each containing one display code character, right justified, zero filled. If CHAR contains a valid decimal integer, including a possible plus or minus sign, IDN should return the value of the integer in NUM. Thus, IDN performs the inverse function from CDC, provided that the absolute value of the number is less than 2**48.

Remark: The value of NUM when CHAR does not contain a valid decimal integer is not specified. The routine should, however, check for such errors as more than one sign character or an invalid character and return some error indication. This error indication should be clearly stated in the comments for the routine.

Remarks: In the usual conversion algorithm, when a new digit is fetched, the value of the number encountered so far is multiplied by ten, and the value of the new digit is added in. If the multiplication is performed by the usual method, there is an inherent limit of 2**48; on the other hand, if the multiplication is effected by a combination of left shifts and additions, numbers up to 2**59 can be accepted.

Remark: The routine should in either case place some limit on the magnitude of numbers which will be accepted, and return an error indication if this limit is exceeded.

EXERCISES

11. Write a subroutine ION (interpret octal number) with two arguments:

```
ION(NUM,CHAR)
```

which is identical to IDN except that the series of digits in CHAR is to be interpreted as an octal integer. This routine is thus the inverse of the one specified in exercise 9.

Remark: Remark 1 from exercise 10 applies here, too. Since numbers of up to twenty digits must be accepted, the multiplication referred to in the second remark must here be performed by a left shift. Finally, since CHAR can contain at most twenty characters, the maximum allowable value, 2**60-1, cannot be exceeded.

12. Combine the routines IDN (exercise 10) and ION (exercise 11) into a subroutine IN with two arguments:

```
IN(NUM,CHAR)
```

as follows: if the series of digits in CHAR is followed by a "B" then interpret the digits as an octal number (as ION would), otherwise interpret the digits as a decimal number (as IDN would).

13. Write a function CHARCT of one argument:

```
CHARCT (NAME)
```

whose value is the number of non-zero characters in NAME (between 0 and 10). For example, CHARCT(1L*) is 1, while CHARCT(1H*) is 10.

14. Write a function DADD of two arguments:

```
DADD(N1,N2)
```

to perform decimal addition. The operands N1 and N2 are each single words containing ten decimal digits in display code (e.g., 10H0000012345); the value of the function should have a similar form.

Remark: Such decimal arithmetic, while not available in FORTRAN, is a standard part of COBOL and PL/1.

Remark: A straightforward algorithm can be written to perform the addition serially, one digit at a time. However, it is possible by careful analysis to devise a routine which adds all ten digits simultaneously, and is consequently much faster.

15. Write a set of four macros, CLOAD, CADD, CSUB, and CSTORE which perform the same operations as LOAD, IADD, ISUB, STORE (Section 4.3), but on complex numbers.

Remark: A complex number is two floating point numbers stored in successive memory words. The first word contains the real part of the number, and the second word the imaginary part.

Remark: The ambitious reader may wish to incorporate these complex operations into the set of mixed-mode arithmetic macros. Conversions should then be performed according to the rules of FORTRAN.

16. Modify the FADD and FSUB macros (section 4.3) to test if the result is infinite, and, if so, call a (FORTRAN) subroutine to print an appropriate error message.

17. Write a function SUBARGS with no argument (invoked in FORTRAN by

```
I = SUBARGS (DUMMY)
```

where the argument is ignored), which returns the number of arguments with which the routine which calls SUBARGS was invoked. For example, if

```
SUBROUTINE ROSSINI (V, E, R, D, I)
INTEGER SUBARGS
K=SUBARGS (DUMMY)
. . .
END
```

is invoked with five arguments

```
CALL ROSSINI (S, E, V, I, L)
```

K will have the value 5, while if it is invoked with three arguments

```
CALL ROSSINI (B, A, R)
```

K will have the value 3.

Remark: Every FORTRAN Extended Subprogram begins with a "SA0 A1" and saves the address of its parameter list in A0 throughout the subprogram (this is why a COMPASS routine called from FORTRAN should not change the value of A0). Therefore, when SUBARGS is called, A0 will point to the argument list of the calling procedure; SUBARGS must determine how long this list is.

18. Modify the INTEGER and REAL macros (p.150) to accept arguments of the form

```
INTEGER ARRAY [100]
```

and generate in this case

```
ARRAY     BSS       100
ARRAY$    SET       0
```

The array "dimension" can be any valid address expression. The macros should continue to accept declarations of "simple variables", without any dimension, and generate a BSS 1.

Remark: Rather than repeat the code for analyzing the argument in INTEGER and REAL, place this code into a single macro which is invoked by both INTEGER and REAL.

Remark: Once this is done, it is a simple matter to modify the fancier INTEGER and REAL macros (p. 162) to accept a list of symbols with dimension declarations, such as

```
REAL (A[20],B[SIZE],C)
```

19. Using the code developed for exercise 18, modify the arithmetic macros to permit reference to subscripted variables:

```
LOAD    A[1]
ADD     B[I]
STORE   C[3]
```

The subscript must be either an integer or a variable name (symbol). If the subscript is an integer, it is added to the array address to yield the effective address; if the subscript is a variable, the contents of the variable are added to the array address to yield the effective address. For example

```
LOAD    A[2]
```

should generate

```
SA5     A-1+2
BX6     X5
```

while

```
LOAD    A[I]
```

should generate

```
SA4     I
SA5     X4+A-1
BX6     X5
```

Note that A-1 is used because, by FORTRAN convention, the address of A(1)= the address of A.

Remark: Symbols and integers are easily differentiated, since our symbols always begin with a letter, integers with a digit.

20. Write a three-way IF macro for the arithmetic macro package, with the form

```
IF3 N,Z,P
```

which branches to Z if the accumulator is zero, to N if it is negative, to P if it is positive. N, Z, and P may be arbitrary address expressions. When two of the address expressions are equal, the sequence of generated branches should be simplified if possible.

21. Modify the LOAD and STORE macros (p. 130) so that if a STORE is immediately followed by a LOAD referencing the same location as the STORE, the LOAD will generate no code. For example,

```
        STORE       A
        LOAD        A
```

should generate simply

```
        SA6         A
```

22. Write a function ICOMPAR of two arguments

```
        ICOMPAR(M,N)
```

such that

```
IF M=N,         ICOMPAR(M,N)=0
IF M>N,         ICOMPAR(M,N)>0
IF M<N,         ICOMPAR(M,N)<0
```

where M and N are any integers.

Remark: If $|M-N| \leq 2^{**}59-1$, the routine IDIF presented in section 3.9 would be a satisfactory solution. If this is not the case, however, subtracting N from M will cause integer overflow, meaning simply that the difference is too large to represent as a 60-bit ones-complement integer. In that event the sign of the difference will not indicate which number is larger. This difficulty may be circumvented by first testing the signs of the arguments or by comparing first the high-order bits and then the low-order bits of the numbers. The latter is analogous to the technique presented in section 3.16 for comparing two 60-bit quantities treated as positive integers. Alternatively, if a CMU is available, the test may be performed by a suitable sequence of compare instructions.

23. Write a function B2Z of one argument

```
        B2Z(CHAR)
```

which returns the value of CHAR, a 10-character string, with each occurrence of a blank character (55 base8) replaced by 00.

This routine can be written in a straightforward manner, using a loop which iterates over the 10 characters. As in the case of CHARCT and DADD, however, (exercises 13 and 14) it is possible to code a faster routine which tests and replaces all 10 characters simultaneously.

24. Develop a set of macros for operating on character strings. In contrast to the arithmetic macros described in Chapter 4, these macros will take their operands from and return their result to memory. A minimal set would include three macros: one for allocating a variable, one for moving a string, one for comparing strings. The allocation macro will have two arguments, the name and length of the string:

```
CVAR        ASTRING,20
```

This would set aside a 2-word (20-character) block called ASTRING, and assign the symbol ASTRING$ the value 20. The latter operation will make it possible for the move and compare macros to determine the length of their arguments. The move macro will take two arguments, the source and target strings:

```
MOVE        FROMSTG,TOSTG
```

The problem arises of what to do if the strings are of different length. The simplest solution is to move the first n characters of FROMSTG to the first n characters of TOSTG, where n is the minimum of the lengths of the two strings. Optionally, one could fill any remaining characters in TOSTG with blanks. The compare macro takes as arguments two strings and a label:

```
IFEQ        ASTRING,BSTRING,INDEED
```

and branches to the label INDEED if the strings are equal. Again a question arises if the lengths of the strings differ. One can say that the strings are equal if the first "n" characters are identical, where "n" is the length of the shorter string, or one may also require that the remaining characters of the longer string all be blanks.

25. Write a function of three arguments

```
SUBSTR(STRING,FIRST,COUNT)
```

where STRING is a character string (array), and FIRST and COUNT are integer variables, FIRST $\geq$ 1, 1 $\leq$ COUNT $\leq$ 10. SUBSTR returns COUNT characters of STRING, beginning with character number FIRST. If COUNT $<$ 10, the substring that is returned in X6 should be left justified and padded on the right with blanks.

Remark: Whereas exercise 24 was intended to indicate the ease with which CMU instructions can handle fixed-length strings beginning at fixed locations, this exercise points up the difficulties when variable-length strings at variable locations are involved. The character count FIRST must be reduced to an offset, in words, from the address of STRING plus a character position. The address, character position, and length must then be shifted into place and incorporated into a move descriptor. Would it be just as easy without the CMU instructions? Try it and compare.

SOLUTIONS TO SELECTED EXERCISES

1. Straightforward solution:

```
         IDENT   INDEX
**************************************************
*                                                *
*     FUNCTION INDEX(ITEM, LIST, LENGTH)         *
*     SEARCHES THE ARRAY LIST, OF LENGTH WORDS,  *
*     FOR ITEM.  RETURNS ARRAY INDEX OF FIRST    *
*     OCCURRENCE OF ITEM IF FOUND, ELSE 0.       *
*                                                *
**************************************************
         ENTRY     INDEX
INDEX    BSS       1
         SA4       A1+1        X4=STARTING ADDRESS OF LIST
         SA3       A1+2        X3=ADDRESS OF LENGTH
         SA1       X1          X1=ITEM SOUGHT
         SA3       X3
         SB3       X3          B3=LENGTH OF LIST
         SB4       0           B4=COUNT=0
LUP      SA2       X4+B4       FETCH NEXT LIST ELEMENT
         IX5       X1-X2
         ZR        X5,HIT      JUMP IF IT MATCHES ITEM SOUGHT
         SB4       B4+1        INCREMENT COUNT
         LT        B4,B3,LUP   LOOP IF NOT AT END OF LIST
         SX6       B0
         EQ        INDEX       RETURN 0 - ITEM NOT FOUND
HIT      SX6       B4+1
         EQ        INDEX       RETURNS ARRAY INDEX
         END
```

Notes:

(a) This routine will successfully match an item of -0 with a list element of +0 (or vice versa); -0-(+0)=-0 and +0-(-0)=+0, and the ZR test will branch on either +0 or -0. To accept only LIST elements which match ITEM bit-for-bit, we must take the logical instead of the integer difference

```
         BX5       X1-X2
```

and branch out of the loop only on X5 = +0. The latter may be accomplished by

```
         NG        X5,NEG
         ZR        X5,HIT
   NEG   SB4       B4+1
```

or by

```
        CX5         X5
        ZR          X5,HIT
```

The count instruction is slightly faster than the branch on the 6600 but much slower on the 6400 and 6500.

(b) We must add one to B4 at HIT because the LIST element at address LIST+i has array index i+1.

A more devious solution:

```
        IDENT   INDEX
        ENTRY   INDEX
INDEX   BSS     1
        SA2     A1+1        X2=STARTING ADDRESS OF LIST
        SA3     A1+2        X3=ADDRESS OF LENGTH
        SA1     X1          X1=ITEM SOUGHT
        SB2     X2          B2=STARTING ADDRESS OF LIST
        SA3     X3          X3=LENGTH OF LIST
        SA4     X3+B2       FETCH WORD AFTER END OF LIST
        BX6     X1
        SA6     A4          PUT SOUGHT ITEM THERE
        SA2     B2          FETCH FIRST ELEMENT OF LIST
        SB7     1
 LUP    IX5     X1-X2
        SA2     A2+B7       FETCH NEXT ARRAY ELEMENT
        NZ      X5,LUP      LOOP IF PREVIOUS ELEMENT≠ITEM
        BX7     X4
        SX6     A2-B2       X6=INDEX OF MATCHING ENTRY
        SA7     A4          RESTORE WORD AFTER END OF LIST
        IX0     X3-X6       X0=ARRAY LENGTH-INDEX
        PL      X0,INDEX    IF INDEX.LE.LENGTH, RETURN INDEX
        SX6     B0          ELSE RETURN ZERO
        EQ      INDEX
```

Notes:

(a) This routine puts the item sought after the end of the list so that it will always be found by the search loop, even if the item does not appear in the list. Consequently, the loop need not check each time if it has reached the end of the list. However, when a match is found, the routine must determine whether an element of the list or the item put after the end of the list was found.

(b) The warning in note (a) for the first solution also applies here.

(c) The SA2 is put after the IX5 in the loop so that the time required to fetch the next element will overlap the branch execution (on the 6600).

(d) The constant 1 is preloaded into B7 so that the search loop can fit into one word.

8. Simple algorithm:

```
         IDENT RBAIEX
**************************************************
*                                                *
*    RBAIEX(R,I) WHERE R IS FLOATING,            *
*    I IS INTEGER. RETURNS R**I                  *
*                                                *
**************************************************

         ENTRY  RBAIEX
RBAIEX   BSS    1
         SA2    A1+1        X2=ADDRESS OF EXPONENT
         SA1    X1          X1=BASE
         SA2    X2          X2=EXPONENT
         SA4    =1.
         BX6    X4          X6=1.
         ZR     X2,RBAIEX   IF EXPONENT=0,RETURN 1
         BX3    X2
         AX3    60
         BX3    X2-X3       X3=ABSOLUTE VALUE OF EXPONENT
         SX0    1
LUP      FX6    X6*X1       X6=X6*BASE
         IX3    X3-X0       EXPONENT=EXPONENT-1
         NZ     X3,LUP      IF EXPONENT ≠ 0, LOOP
         PL     X2,RBAIEX   IF EXPONENT POSITIVE, RETURN
         FX6    X4/X6       IF EXPONENT NEGATIVE, COMPUTE
         EQ     RBAIEX      RECIPROCAL AND RETURN
         END
```

8. Fast algorithm:

```
         IDENT     RBAIEX
         ENTRY     RBAIEX
RBAIEX   BSS       1
         SA2       A1+1        X2=ADDRESS OF EXPONENT
         SA1       X1          X1=BASE
         SA2       X2          X2=EXPONENT
         SA4       =1.
         BX6       X4          X6=1.
         ZR        X2,RBAIEX   IF EXPONENT=0,RETURN 1
         BX3       X2
         AX3       60
         BX3       X2-X3       X3=ABSOLUTE VALUE OF EXPONENT
         MX5       59
LUP      BX7       -X5*X3      X7=LOW ORDER BIT OF EXPONENT
         ZR        X7,BITOFF
         FX6       X6*X1       IF BIT=1,MULTIPLY BASE INTO X6
BITOFF   AX3       1           HALVE EXPONENT
         FX1       X1*X1       SQUARE BASE
         NZ        X3,LUP      IF 1 BITS REMAIN IN EXPO, LOOP
         PL        X2,RBAIEX   IF EXPONENT POSITIVE, RETURN
         FX6       X4/X6       IF EXPONENT NEGATIVE,COMPUTE
         EQ        RBAIEX      RECIPROCAL AND RETURN
         END
```

Notes:

(a) The fast algorithm is based on the following analysis. If we write the exponent as a binary number with bits b(n). . .b(0):

```
I = b(n)*2**n + b(n-1)*2**(n-1) + ... + b(1)*2**1 + b(0)*2**0
```

then

```
R**I=R**(b(n)*2**n + b(n-1)*2**(n-1) +...+ b(1)*2**1 + b(0)*2**0)

=R**(b(n)*2**n)*R**(b(n-1)**2(n-1))*...*R**(b(1)*2**1)*R**(b(0)*1)
```

since each b(k) = 0 or 1, this means that the factor R**(2**k) is to be included in the result only if b(k)=1. The loop in the routine generates successively R**(2**k), k = 0, 1, 2, ... and multiplies it into X6 if the corresponding bit in I, b(k)=1.

(b) The second version requires about log(base 2) I iterations of the loop, and thus is much faster than the first version, which requires I iterations.

13. Straightforward solution:

```
        IDENT CHARCT
**********************************************
*                                            *
*          CHARCT(WORD) RETURNS NUMBER OF    *
*          NON-ZERO CHARACTERS IN WORD       *
*                                            *
**********************************************
        ENTRY     CHARCT
CHARCT  BSS       1
        SA2       X1          X1=WORD
        SB7       10
        SX6       0           X6=CHARACTER COUNT=0
        MX5       6           FORM ONE CHARACTER MASK
LUP     BX0       X5*X1       MASK OUT HIGH-ORDER CHARACTER
        ZR        X0,NOCHAR   IF NON-ZERO,
        SX6       X6+1            INCREMENT CHARACTER COUNT
NOCHAR  LX1       6           SHIFT WORD LEFT ONE CHARACTER
        SB7       B7-1
        NZ        B7,LUP      IF 10 CHAR NOT EXAMINED, LOOP
        EQ        CHARCT
        END
```

Faster method:

```
        IDENT     CHARCT
        ENTRY     CHARCT
CHARCT  BSS       1
        SA1       X1          X1=WORD
        SA2       =4040404040404040404040B
        BX6       X1
        DUP       5           OR ALL SIX BITS OF A CHARACTER
        LX1       1           INTO THE HIGH-ORDER
        BX6       X6+X1       BIT OF THE CHARACTER
        ENDD
        BX6       X6*X2       MASK OUT HIGH-ORDER BITS OF CHARS
        CX6       X6          RETURN NUMBER OF 1 BITS
        EQ        CHARCT
        END
```

Note: Fast character manipulation routines, such as the above, are often obtained by processing a number of characters simultaneously. The sequence of five shift and logical sum operations (within the DUP) leaves in the high-order bit of each character in X6 the logical sum of all six bits of the character

in the argument. After the logical product only the high-order bit of each character is left; it will be non-zero if any bit in the original character was non-zero. Thus, the number of one bits in X6 before the count ones instruction is executed equal the number of non-zero characters in WORD.

14.

```
        IDENT DADD
**************************************************
*                                                *
*    DADD(ADDEND1,ADDEND2)  RETURNS THE SUM,     *
*    OF THE TWO DISPLAY-CODE DECIMAL ARGUMENTS   *
*                                                *
**************************************************
        ENTRY  DADD
DADD    BSS    1
        SA2    A1+1
        SA1    X1      X1,X2=TWO ARGUMENTS TO BE SUMMED
        SA2    X2
        IX3    X1+X2   FORM INTEGER SUM
        SA4    =6060606060606060606OB
        SA5    =3333333333333333333B
        BX6    X3*X4   FOR EACH CHAR IN X3>=66B
        BX7    X6      (NO CARRY OUT OF CHAR POSN)
        LX7    57      SET CORR CHAR IN X6=66B,
        BX6    X6+X7   OTHER CHARS IN X6=0
        IX6    X3-X6   SUBTRACT 66B FROM ALL CHARS >=66B
        IX6    X6+X5   ADD 33B TO ALL CHARS
        EQ     DADD    RETURN
        END
```

Note: In discussing binary-coded-decimal addition (p. 14), we mentioned that the addition algorithm must check for digits greater than ten in the sum, and propagate a carry to the next digit. Storing the decimal numbers in display code neatly solves the problems of propagating carries; if the sum of two digits is greater than 9, the sum of their display codes will be greater than 77B, and a carry will be propagated into the next character by the IX3 X1+X2. For example,

```
                          display code
  17                        34  42
+ 25                       +35  40
  --                       ------
  42                        72  02  integer sum
                            37  35  final result
```

If no carry occurred out of a given character position, the character will be between 66B (sum of two display code zeros) and 77B; we must subtract 33B to get the proper display code character for the result. If a carry out did occur, the character will be between 00 and 11B; 33B must be added to obtain the proper display code character. This routine "corrects" all characters simultaneously through some devious bit manipulation. First, 66B is subtracted from all characters $\geq$ 66B; second, 33B is added to all characters.

17.

```
        IDENT SUBARGS
**************************************************
*                                                *
*     SUBARGS TAKES NO ARGUMENTS: IT RETURNS     *
*     THE NUMBER OF ARGUMENTS PASSED TO THE      *
*     ROUTINE WHICH INVOKED SUBARGS              *
*                                                *
**************************************************
        ENTRY   SUBARGS
SUBARGS BSS     1
        SA2     A0              X2=FIRST WORD OF ARGUMENT LIST
                                TO ROUTINE THAT CALLED SUBARGS
        SX6     0               X6=ARGUMENT COUNT =0
        ZR      X2,SUBARGS      IF NO ARGUMENTS, RETURN
NEXTARG SA2     A2+1            FETCH NEXT WORD OF ARGUMENT LIST
        SX6     X6+1            INCREMENT ARGUMENT COUNT
        NZ      X2,NEXTARG      IF NOT AT END OF ARGUMENTS, LOOP
        EQ      SUBARGS         RETURN
        END
```

20.

```
IF3     MACRO     N,Z,P
        ZR        X6,Z
        IFC       NE,$N$P$
        PL        X6,P
        ENDIF
        EQ        N
        ENDM
```

Note: The IFC NE,NP$ will succeed unless N and P are literally identical character for character. The reader might be inclined to use instead

```
        IFNE      N,P
```

which would succeed unless the values of N and P are equal. However, the latter IF would be an error unless both N and P were previously defined. Since this would not always be the case, the test on the values of N and P cannot be used.

STORAGE ALLOCATION.

ADDRESS	LENGTH
0	15
15	

BINARY CONTROL CARDS.

```
IDENT  VECLIB
END
```

BLOCKS	TYPE	ADDRESS	LENGTH
PROGRAM*	LOCAL	0	15
VECTOR	COMMON	0	25

ENTRY POINTS.

INIT	0+	PSUM	6+

?

```
                                        IDENT  VECLIB

                                *       VECLIB IS A LIBRARY OF ROUTINES WHICH PERFORM OPERATIONS
                                *       ON A VECTOR IN THE COMMON BLOCK VECTOR

                                        ENTRY  INIT
                                        ENTRY  PSUM

                                        USE    /VECTOR/
 0                      24      VEC     BSS    20          THE VECTOR, MAXIMUM SIZE OF 20 ELEMENTS
24                       1      SIZE    BSS    1           ACTUAL SIZE OF THE VECTOR
                                        USE    0


                                *       INIT INITIALIZES THE ELEMENTS OF THE VECTOR TO THE VALUE OF
                                *       THE PARAMETER PASSED TO IT.  CALLING SEQUENCE AS FOLLOWS:
                                *             CALL INIT(INTEGER)

 0                       1      INIT    BSS    1
 1  53110                               SA1    X1          X1=VALUE USED FOR INITALIZATION
          12611                         BX6    X1          X6=VALUE USED FOR INITALIZATION
                5120000024 C            SA2    SIZE        GET ACTUAL SIZE OF THE VECTOR
 2  63120                               SB1    X2          B1=SIZE OF THE VECTOR
          5160000000 C                  SA6    VEC         SET THE FIRST ELEMENT
 3  6120000001                          SB2    1           B2=COUNTER
 4  0621000000 +            LOOP        GE     B2,B1,INIT  CHECK IF WE ARE DONE
                5066000001              SA6    A6+1        IF NOT, SET NEXT ELEMENT
 5  6122000001                          SB2    B2+1        INCREMENT COUNTER
                0400000004 +            EQ     LOOP        GO FOR MORE


                                *       PSUM COMPUTES THE PARTIAL SUM OF A VECTOR, VEC(I) IS EQUAL
                                *       TO THE SUM OF THE ELEMENTS FROM 1 TO I.  CALLING SEQUENCE:
                                *             CALL PSUM

 6                       1      PSUM    BSS    1
 7  5110000000 C                        SA1    VEC         X1=FIRST ELEMENT
          10611                         BX6    X1          X6=FIRST ELEMENT
10  5120000000                          SA2    SIXE        GET ACTUAL SIZE OF THE VECTOR
          63120                         SB1    X2          B1=SIZE OF THE VECTOR
11  6120000001                          SB2    1           B2=COUNTER
12  0621000006 +            LOOP        GE     B2,B1,PSUM  ARE WE ALL DONE
                5011000001              SA1    A1+1        GET NEXT ELEMENT
13  36616                               IX6    X1+X6       COMPUTE NEW PARTIAL SUM
                                                           (OLD PARTIAL SUM PLUS NEW ELEMENT)
          54610                         SA6    A1          STORE THE PARTIAL SUM
                6122000001              SB2    B2+1        INCREMENT COUNTER
14  0400000004 +                        EQ     LOOP        GO FOR IT
15                                      END

                43300B CM  STORAGE USED              49 STATEMENTS        6 SYMBOLS
                           MODEL 73 ASSEMBLY      0.333 SECONDS         16 REFERENCES

                                2 ERRORS IN VECLIB
```

ERROR DIRECTORY.

D TYPE ERROR DOUBLY DEFINED SYMBOL. THE FIRST DEFINITION HOLDS
OCCURRED ON PAGES 2

U TYPE ERROR UNDEFINED SYMBOL. VALUE ASSUMED 0.
OCCURRED ON PAGES 2

?

SYMBOLIC REFERENCE TABLE.

INIT	0	PROGRAM*	2/06	E	2/19	L	2/26		
LOOP	4	PROGRAM*	2/26	L	2/29		2/42	L	2/48
PSUM	6	PROGRAM*	2/07	E	2/36	L	2/42		
SIXE			2/39						
SIZE	24	VECTOR	2/11	L	2/22				
VEC	0	VECTOR	2/10	L	2/24	S	2/37		

```
      PROGRAM MAIN       73/74   OPT=1            ** A5 **        FTN 4.8+518        80/12/03. 21.42.51       PAGE     1

     1                PROGRAM MAIN(OUTPUT)
                      COMMON /VECTOR/ VEC(20),SIZE
                      INTEGER VEC,SIZE
                      SIZE = 10
     5                CALL INIT(1)
                      CALL PSUM
                      STOP
                      END


       SYMBOLIC REFERENCE MAP (R=1)

ENTRY POINTS
 2062  MAIN

VARIABLES     SN  TYPE           RELOCATION
   24  SIZE      INTEGER            VECTOR          0  VEC       INTEGER    ARRAY    VECTOR

FILE NAMES        MODE
    0  OUTPUT

EXTERNALS          TYPE   ARGS
       INIT                 1                          PSUM                0

COMMON BLOCKS   LENGTH
       VECTOR       21

STATISTICS
  PROGRAM LENGTH                71B      57
  BUFFER LENGTH               2003B    1027
  CM LABELED COMMON LENGTH      25B      21
         52000B CM USED
?
```

STORAGE ALLOCATION.

ADDRESS	LENGTH
0	15
15	

BINARY CONTROL CARDS.

```
IDENT  VECLIB
END
```

BLOCKS	TYPE	ADDRESS	LENGTH
PROGRAM*	LOCAL	0	15
VECTOR	COMMON	0	25

ENTRY POINTS.

INIT	0+	PSUM	6+

?

```
                                        IDENT  VECLIB

                                *       VECLIB IS A LIBRARY OF ROUTINES WHICH PERFORM OPERATIONS
                                *       ON A VECTOR IN THE COMMON BLOCK VECTOR

                                        ENTRY  INIT
                                        ENTRY  PSUM

                                        USE    /VECTOR/
 0                    24    VEC       BSS    20          THE VECTOR, MAXIMUM SIZE OF 20 ELEMENTS
24                     1    SIZE      BSS    1           ACTUAL SIZE OF THE VECTOR
                                        USE    0


                                *       INIT INITIALIZES THE ELEMENTS OF THE VECTOR TO THE VALUE OF
                                *       THE PARAMETER PASSED TO IT.  CALLING SEQUENCE AS FOLLOWS:
                                *             CALL INIT(INTEGER)

 0                     1    INIT      BSS    1
 1  53110                             SA1    X1          X1=VALUE USED FOR INITALIZATION
         10611                        BX6    X1          X6=VALUE USED FOR INITALIZATION
               5120000024 C           SA2    SIZE        GET ACTUAL SIZE OF THE VECTOR
 2  63120                             SB1    X2          B1=SIZE OF THE VECTOR
         5160000000 C                 SA6    VEC         SET THE FIRST ELEMENT
 3  6120000001                        SB2    1           B2=COUNTER
 4  0621000000 +          LOOP        GE     B2,B1,INIT  CHECK IF WE ARE DONE
               5066000001             SA6    A6+1        IF NOT, SET NEXT ELEMENT
 5  6122000001                        SB2    B2+1        INCREMENT COUNTER
               0400000004 +           EQ     LOOP        GO FOR MORE


                                *       PSUM COMPUTES THE PARTIAL SUM OF A VECTOR, VEC(I) IS EQUAL
                                *       TO THE SUM OF THE ELEMENTS FROM 1 TO I.  CALLING SEQUENCE:
                                *             CALL PSUM

 6                     1    PSUM      BSS    1
 7  5110000000 C                      SA1    VEC         X1=FIRST ELEMENT
               10611                  BX6    X1          X6=FIRST ELEMENT
10  5120000024 C                      SA2    SIZE        GET ACTUAL SIZE OF THE VECTOR
               63120                  SB1    X2          B1=SIZE OF THE VECTOR
11  6120000001                        SB2    1           B2=COUNTER
12  0621000006 +          LOOP1       GE     B2,B1,PSUM  ARE WE ALL DONE
               5011000001             SA1    A1+1        GET NEXT ELEMENT
13  36616                             IX6    X1+X6       COMPUTE NEW PARTIAL SUM
                                                         (OLD PARTIAL SUM PLUS NEW ELEMENT)
         54610                        SA6    A1          STORE THE PARTIAL SUM
               6122000001             SB2    B2+1        INCREMENT COUNTER
14  0400000012 +                      EQ     LOOP1       GO FOR IT
15                                    END

          43300B CM  STORAGE USED              49 STATEMENTS        6 SYMBOLS
                     MODEL 73 ASSEMBLY      0.320 SECONDS         16 REFERENCES
```

?

SYMBOLIC REFERENCE TABLE.

INIT	0	PROGRAM*	2/06	E	2/19	L	2/26
LOOP	4	PROGRAM*	2/26	L	2/29		
LOOP1	12	PROGRAM*	2/42	L	2/48		
PSUM	6	PROGRAM*	2/07	E	2/36	L	2/42
SIZE	24	VECTOR	2/11	L	2/22		2/39
VEC	0	VECTOR	2/10	L	2/24	S	2/37

FWA OF THE LOAD 111
LWA+1 OF THE LOAD 5437

TRANSFER ADDRESS -- MAIN 2220

PROGRAM ENTRY POINTS -- MAIN 2220

PROGRAM AND BLOCK ASSIGNMENTS.

BLOCK	ADDRESS	LENGTH	FILE	DATE	PROCSSR	VER	LEVEL	HARDWARE	COMMENTS
/VECTOR/	111	25							
MAIN	136	2074	LGO	80/12/03	FTN	4.8	518	666X I	PROGRAM OPT=1
VECLIB	2232	15	LGO	80/12/03	COMPASS	3.6	518		
/STP.END/	2247	1							
/FCL.C./	2250	26							
/Q8.IO./	2276	145							
Q2NTRY=	2443	1	SL-FORTRAN	80/10/14	COMPASS	3.6	518		FCL INITIALIZATION ROUTINE.
/FCL=ENT/	2444	41							
FCL=FDL	2505	40	SL-FORTRAN	80/10/14	COMPASS	3.6	518		FCL CAPSULE LOADING
FEIFST=	2545	3	SL-FORTRAN	80/10/14	COMPASS	3.6	518		CONVERTED DATA STORAGE
FORSYS=	2550	516	SL-FORTRAN	80/10/14	COMPASS	3.6	518		FORTRAN OBJECT LIBRARY UTILITIES.
FORUTL=	3266	25	SL-FORTRAN	80/10/14	COMPASS	3.6	518		FCL MISC. UTILITIES.
GETFIT=	3313	74	SL-FORTRAN	80/10/14	COMPASS	3.6	518		LOCATE A FIT GIVEN A FILE DESCRIPTOR
SYSAID=	3407	1	SL-FORTRAN	80/10/14	COMPASS	3.6	518		LINK BETWEEN SYS=AID AND INITIALIZATION C@
CPU.CPM	3410	5	SL-SYSLIB	80/10/14	COMPASS	3.6	518		79/05/10. 80/03/20. CONTROL POINT MANAGER@
CPU.SYS	3415	40	SL-SYSLIB	80/10/13	COMPASS	3.6	518		PROCESS SYSTEM REQUEST.
CMF.ALF	3455	162	SL-SYSLIB	80/10/13	COMPASS	3.6	518		CMM V1.1 - ALLOCATE FIXED.
CMF.CSF	3637	6	SL-SYSLIB	80/10/13	COMPASS	3.6	518		CMM V1.1 - CHANGE SPECS FIXED.
CMM.FFA	3645	14	SL-SYSLIB	80/10/13	COMPASS	3.6	518		CMM V1.1 - FIXED FREE ALGORITHM.
CMF.FRF	3661	36	SL-SYSLIB	80/10/13	COMPASS	3.6	518		CMM V1.1 - FREE FIXED.
CMF.GSS	3717	22	SL-SYSLIB	80/10/13	COMPASS	3.6	518		CMM V1.1 - GET SUMMARY STATISTICS.
CMM.KIL	3741	12	SL-SYSLIB	80/10/13	COMPASS	3.6	518		CMM V1.1 - DEACTIVATE CMM.
CMM.MEM	3753	7	SL-SYSLIB	80/10/13	COMPASS	3.6	518		
CMM.R	3762	206	SL-SYSLIB	80/10/13	COMPASS	3.6	518		CMM V1.1 - RESIDENT SUBROUTINES.
CMF.SLF	4170	22	SL-SYSLIB	80/10/13	COMPASS	3.6	518		CMM V1.1 - SHRINK AT LWA FIXED.
CTL$RM	4212	435	SL-SYSLIB	80/10/14	COMPASS	3.6	518		CRM CONTROLLING ROUTINE.
ERR$RM	4647	25	SL-SYSLIB	80/10/14	COMPASS	3.6	518		CRM ERROR PROCESSOR ENTRY.
LIST$RM	4674	67	SL-SYSLIB	80/10/14	COMPASS	3.6	518		CRM - ALLOCATE SPACE FOR LIST OF FILES
RM$SYS=	4763	5	SL-SYSLIB	80/10/14	COMPASS	3.6	518		CRM - POST RA+1 REQUEST
/FDL.COM/	4770	14							
FDL.RES	5004	211	SL-SYSLIB	80/10/01	COMPASS	3.6	518		FAST DYNAMIC LOADER RESIDENT.
FDL.MMI	5215	222	SL-SYSLIB	80/10/01	COMPASS	3.6	518		FDL MEMORY MANAGER INTERFACE.

.390 CP SECONDS 21600B CM STORAGE USED 2 TABLE MOVES

?

```
- CM DUMP FROM     100 TO     200.                    ** A10 **

   100      54000 00000 01000 00001      00532 60000 00000 05437      00000 00000 00000 00000      00000 00000 00000 00000
   104      00000 00000 00000 05437      00000 00000 00000 00000      00000 00000 00000 00000      00000 00000 00000 00000
   110      15011 11600 00000 02220      00000 00000 00000 00001      00000 00000 00000 00002      00000 00000 00000 00003
   114      00000 00000 00000 00004      00000 00000 00000 00005      00000 00000 00000 00006      00000 00000 00000 00007
   120      00000 00000 00000 00010      00000 00000 00000 00011      00000 00000 00000 00012      60000 00000 04004 00123
   124      60000 00000 04004 00124      60000 00000 04004 00125      60000 00000 04004 00126      60000 00000 04004 00127
   130      60000 00000 04004 00130      60000 00000 04004 00131      60000 00000 04004 00132      60000 00000 04004 00133
   134      60000 00000 04004 00134      00000 00000 00000 00012      17252 42025 24000 00001      00000 00000 00000 00207
   140      00000 00000 00000 00000      00000 00000 00000 00000      00000 00000 00000 00000      00000 00000 00000 00000
         DUPLICATED LINES.
   150      00000 00000 00600 00000      00000 00022 00000 00000      00000 00000 00000 00000      14000 00000 00000 00000
   154      00000 00000 00000 00000      00000 00006 00000 00000      00000 00000 00000 00000      00000 00000 00000 02003
   160      00000 00000 00000 00000      00000 00000 00000 00000      00000 00000 00000 00000      00000 00000 00000 00000
         DUPLICATED LINES.
?
```

```
¬
              PROGRAM MAIN       73/74   OPT=1            ** All **        FTN 4.8+518         80/12/17. 22.17.06       PAGE      1

        1                PROGRAM MAIN(INPUT,OUTPUT)
                         INTEGER SENT(160),WORDS
                         READ 5,(SENT(I),I=1,80)
                         READ 5,(SENT(I),I=81,160)
        5                NUMBER = WORDS(SENT,160)
                         PRINT 6,NUMBER
                         STOP
                       5 FORMAT(80A1)
                       6 FORMAT(* THE NUMBER OF WORDS IS *,I5)
       10                END


         SYMBOLIC REFERENCE MAP (R=1)

   ENTRY POINTS
    4137  MAIN

   VARIABLES     SN  TYPE           RELOCATION
    4203  I         INTEGER                          4204  NUMBER    INTEGER
    4205  SENT      INTEGER    ARRAY

   FILE NAMES        MODE
       0  INPUT     FMT           2054  OUTPUT    FMT

   EXTERNALS          TYPE   ARGS
         WORDS      INTEGER    2

   STATEMENT LABELS
    4172  5       FMT                      4174  6       FMT

   STATISTICS
     PROGRAM LENGTH               437B     287
     BUFFER LENGTH               4006B    2054
           52000B CM USED
?
```

STORAGE ALLOCATION.

ADDRESS	LENGTH
0	15
15	

BINARY CONTROL CARDS.

```
IDENT  WORDS
END
```

ENTRY POINTS.

WORDS 0+

?

```
                                IDENT  WORDS

                          *     WORDS IS A FUNCTION WHICH COUNTS THE NUMBER OF WORDS IN
                          *     A SENTENCE OR PHRASE.  A WORD IS A SEQUENCE OF CHARACTERS
                          *     SEPARATED BY ONE OR MORE BLANKS.  THE SENTENCE IS AN ARRAY
                          *     OF CHARACTERS, ONE CHARACTER PER WORD, LEFT JUSTIFIED,
                          *     BLANK FILLED.
                          *          CALLING SEQUENCE:
                          *               NUMBER = WORDS(SENTENCE,LENGTH)

                                ENTRY  WORDS
 0                    1   WORDS BSS    1
 1  5021000001                 SA2    A1+1        X2=ADDRESS OF LENGTH
           53110               SA1    X1          A1=STARTING ADDRESS OF SENTENCE
                53220          SA2    X2          X2=LENGTH OF SENTENCE
 2  63320                      SB3    X2          B3=LENGTH OF SENTENCE
      66100                    SB1    B0          B1=FLAG, 0=BLANKS  1=NONBLANKS
           6120000001          SB2    1           B2=CHARACTER COUNTER
 3  76600                      SX6    B0          X6=WORD COUNTER
      5150000014 +             SA5    BLANK       X5=LEFT JUSTIFIED BLANK
                13115          BX1    X1-X5       IS THE FIRST CHARACTER A BLANK
 4  0301000005 +               ZR     X1,LOOP     IF SO, JUMP
           6110000001          SB1    1           IF NOT, FLAG=NONBLANK
 5  54112                 LOOP SA1    A1+B2       GET NEXT CHARACTER
      13115                    BX1    X1-X5       IS THE CHARACTER A BLANK
           0311000010 +        NZ     X1,OVER     IF NOT, JUMP
 6  0410000011 +               EQ     B1,NEXT     IS IT THE FIRST BLANK BETWEEN WORDS
           7266000001          SX6    X6+1        IF SO, INCREMENT WORD COUNT
 7  66100                      SB1    B0          SET FLAG TO INDICATE BLANK ENCOUNTERRED
      0400000011 +             EQ     NEXT
10  6110000001            OVER SB1    1           SET FLAG TO INDICATE NONBLANK
11  6122000001            NEXT SB2    B2+1        INCREMENT CHARACTER COUNTER
           0723000005 +        LT     B2,B3,LOOP  DID WE EXAMINE ALL CHARACTERS?  IF NOT,LOOP
12  0410000000 +               EQ     B1,WORDS    WAS THE LAST CHARACTER A BLANK?  YES,RETURN
           7266000001          SX6    X6+1        SENTENCE ENDED WITH A WORD, INCREMENT COUNT
13  0400000000 +               EQ     WORDS       RETURN
14  5555555555555555555  BLANK DATA   1H          DEFINE A LEFT JUSTIFED BLANK
15                             END

          43300B CM  STORAGE USED              38 STATEMENTS        5 SYMBOLS
                     MODEL 73 ASSEMBLY      0.245 SECONDS         14 REFERENCES
```

SYMBOLIC REFERENCE TABLE.

BLANK	14	PROGRAM*	2/20	2/37 L			
LOOP	5	PROGRAM*	2/22	2/24 L	2/33		
NEXT	11	PROGRAM*	2/27	2/30	2/32 L		
OVER	10	PROGRAM*	2/26	2/31 L			
WORDS	0	PROGRAM*	2/11 E	2/12 L	2/34	2/36	

FWA OF THE LOAD 111
LWA+1 OF THE LOAD 13107

TRANSFER ADDRESS -- MAIN 4250

PROGRAM ENTRY POINTS -- MAIN 4250

PROGRAM AND BLOCK ASSIGNMENTS.

BLOCK	ADDRESS	LENGTH	FILE	DATE	PROCSSR	VER	LEVEL	HARDWARE	COMMENTS
MAIN	111	4445	LGO	80/12/17	FTN	4.8	518	666X I	PROGRAM OPT=1
WORDS	4556	15	LGO	80/12/17	COMPASS	3.6	518		
/STP.END/	4573	1							
/FCL.C./	4574	26							
/Q8.IO./	4622	145							
Q2NTRY=	4767	1	SL-FORTRAN	80/10/14	COMPASS	3.6	518		FCL INITIALIZATION ROUTINE.
/FCL=ENT/	4770	41							
COMIO=	5031	33	SL-FORTRAN	80/10/14	COMPASS	3.6	518		COMMON CODED I/O ROUTINES AND CONSTANTS.
FCL=FDL	5064	40	SL-FORTRAN	80/10/14	COMPASS	3.6	518		FCL CAPSULE LOADING
FECMSK=	5124	41	SL-FORTRAN	80/10/14	COMPASS	3.6	518		INITIALIZE CONSTANTS.
FEIFST=	5165	3	SL-FORTRAN	80/10/14	COMPASS	3.6	518		CONVERTED DATA STORAGE
FLTIN=	5170	156	SL-FORTRAN	80/10/14	COMPASS	3.6	518		COMMON FLOATING INPUT CONVERTER.
FLTOUT=	5346	311	SL-FORTRAN	80/10/14	COMPASS	3.6	518		COMMON FLOATING OUTPUT CODE
FMTAP=	5657	357	SL-FORTRAN	80/10/14	COMPASS	3.6	518		CRACK APLIST AND FORMAT FOR KODER/KRAKER.
FORSYS=	6236	516	SL-FORTRAN	80/10/14	COMPASS	3.6	518		FORTRAN OBJECT LIBRARY UTILITIES.
FORUTL=	6754	25	SL-FORTRAN	80/10/14	COMPASS	3.6	518		FCL MISC. UTILITIES.
GETFIT=	7001	74	SL-FORTRAN	80/10/14	COMPASS	3.6	518		LOCATE A FIT GIVEN A FILE DESCRIPTOR
INCOM=	7075	145	SL-FORTRAN	80/10/14	COMPASS	3.6	518		COMMON INPUT FORMATTING CODE
INPC=	7242	207	SL-FORTRAN	80/10/14	COMPASS	3.6	518		FORMATTED READ FORTRAN RECORD.
KODER=	7451	465	SL-FORTRAN	80/10/14	COMPASS	3.6	518		OUTPUT FORMAT INTERPRETER.
KRAKER=	10136	375	SL-FORTRAN	80/10/14	COMPASS	3.6	518		PROCESS FORMATTED FORTRAN INPUT.
OUTC=	10533	150	SL-FORTRAN	80/10/14	COMPASS	3.6	518		FORMATTED WRITE FORTRAN RECORD.
OUTCOM=	10703	154	SL-FORTRAN	80/10/14	COMPASS	3.6	518		COMMON OUTPUT CODE
SYSAID=	11057	1	SL-FORTRAN	80/10/14	COMPASS	3.6	518		LINK BETWEEN SYS=AID AND INITIALIZATION CO
CPU.CPM	11060	5	SL-SYSLIB	80/10/14	COMPASS	3.6	518		79/05/10. 80/03/20. CONTROL POINT MANAGERH
CPU.SYS	11065	40	SL-SYSLIB	80/10/13	COMPASS	3.6	518		PROCESS SYSTEM REQUEST.
CMF.ALF	11125	162	SL-SYSLIB	80/10/13	COMPASS	3.6	518		CMM V1.1 - ALLOCATE FIXED.
CMF.CSF	11307	6	SL-SYSLIB	80/10/13	COMPASS	3.6	518		CMM V1.1 - CHANGE SPECS FIXED.
CMM.FFA	11315	14	SL-SYSLIB	80/10/13	COMPASS	3.6	518		CMM V1.1 - FIXED FREE ALGORITHM.
CMF.FRF	11331	36	SL-SYSLIB	80/10/13	COMPASS	3.6	518		CMM V1.1 - FREE FIXED.
CMF.GSS	11367	22	SL-SYSLIB	80/10/13	COMPASS	3.6	518		CMM V1.1 - GET SUMMARY STATISTICS.
CMM.KIL	11411	12	SL-SYSLIB	80/10/13	COMPASS	3.6	518		CMM V1.1 - DEACTIVATE CMM.
CMM.MEM	11423	7	SL-SYSLIB	80/10/13	COMPASS	3.6	518		
CMM.R	11432	206	SL-SYSLIB	80/10/13	COMPASS	3.6	518		CMM V1.1 - RESIDENT SUBROUTINES.
CMF.SLF	11640	22	SL-SYSLIB	80/10/13	COMPASS	3.6	518		CMM V1.1 - SHRINK AT LWA FIXED.
CTL$RM	11662	435	SL-SYSLIB	80/10/14	COMPASS	3.6	518		CRM CONTROLLING ROUTINE.
ERR$RM	12317	25	SL-SYSLIB	80/10/14	COMPASS	3.6	518		CRM ERROR PROCESSOR ENTRY.
LIST$RM	12344	67	SL-SYSLIB	80/10/14	COMPASS	3.6	518		CRM - ALLOCATE SPACE FOR LIST OF FILES
RM$SYS=	12433	5	SL-SYSLIB	80/10/14	COMPASS	3.6	518		CRM - POST RA+1 REQUEST
/FDL.COM/	12440	14							
FDL.RES	12454	211	SL-SYSLIB	80/10/01	COMPASS	3.6	518		FAST DYNAMIC LOADER RESIDENT.
FDL.MMI	12665	222	SL-SYSLIB	80/10/01	COMPASS	3.6	518		FDL MEMORY MANAGER INTERFACE.

EXCHANGE PACKAGE. ** A16 **

P	0	A0	0	B0	0	(A0)	0001 0045 6400 0000 0000
RA	414200	A1	15236	B1	1	(A1)	0000 0000 0000 0000 0000
FL	15200	A2	4312	B2	137	(A2)	0000 0000 0000 0000 0240
EM	7007	A3	7243	B3	240	(A3)	0000 0000 0074 1500 0000
RAE	0	A4	4304	B4	12	(A4)	5143 3301 3452 5555 5555
FLE	0	A5	4572	B5	52	(A5)	5555 5555 5555 5555 5555
MA	3400	A6	4640	B6	1	(A6)	5555 5555 5555 5555 5555
		A7	4662	B7	0	(A7)	2012 0000 0000 0000 4632

X0	7777 7777 7777 7777 7700
X1	0000 0000 0000 0000 0000
X2	0000 0000 0000 0000 0240
X3	0000 0000 0074 1500 0000
X4	0000 0000 0000 0000 4632
X5	5555 5555 5555 5555 5555
X6	0000 0000 0000 0000 0005
X7	2012 0000 0000 0000 4632

(RA)	0001 0045 6400 0000 0000
(RA+1)	0000 0000 0000 0000 0000

CM DUMP FROM 4524 TO 4624.

4524	05555 55555 55555 55555	55555 55555 55555 55555	55555 55555 55555 55555	55555 55555 55555 55555
4530	01555 55555 55555 55555	11555 55555 55555 55555	04555 55555 55555 55555	55555 55555 55555 55555
4534	55555 55555 55555 55555	17555 55555 55555 55555	06555 55555 55555 55555	55555 55555 55555 55555
4540	55555 55555 55555 55555	55555 55555 55555 55555	24555 55555 55555 55555	10555 55555 55555 55555
4544	05555 55555 55555 55555	11555 55555 55555 55555	22555 55555 55555 55555	55555 55555 55555 55555
4550	20555 55555 55555 55555	01555 55555 55555 55555	22555 55555 55555 55555	24555 55555 55555 55555
4554	31555 55555 55555 55555	55555 55555 55555 55555	04000 04257 00000 00000	50210 00001 53110 53220
4560	63320 66100 61200 00001	76600 51500 04572 13115	03010 04563 61100 00001	54112 13115 03110 04566
4564	04100 04567 72660 00001	66100 04000 04567 46000	61100 00001 61000 46000	61220 00001 07230 04563
4570	04100 04556 72660 00001	04000 04556 61000 46000	55555 55555 55555 55555	04000 06400 61000 46000
4574	55555 55555 55555 55555	40404 04040 40404 04040	11162 02524 00000 00000	17252 42025 24000 00000
4600	17171 27432 14774 13155	20001 20726 42717 30565	00000 00000 00000 00000	00000 00000 00000 04244
4604	00000 00000 00000 00310	00000 00000 00000 00000	00000 00000 00000 00000	00000 00000 00000 00000
4610	00000 00000 00000 00000	00000 00000 00000 00000	00000 00000 00000 00024	00000 00000 00000 00000
4614	00000 00000 00000 07640	20000 00000 00000 10455	00000 00000 00000 00000	00000 00000 00000 04620
4620	00000 00000 00000 00000	00000 00000 00000 00000	55555 55555 55555 55555	55150 51655 55555 55555
4624	55555 55555 55552 41755	55555 55555 03171 50555	55555 55555 55552 41755	55552 41005 55555 50111

?

```
 ANUABSO. 80/10/17.NYU ACADEMIC COMPUTING FACILITY.        NOS 1.4-518-NYU-21

13.19.07.KPM.
13.19.07.UCCR, AB50,      0.056KCDS.
13.19.07.USER,MCAULFK,.
13.19.08.CHARGE,122A30Y,BOOK.
13.19.08.FTN.
13.19.11.      .637 CP SECONDS COMPILATION TIME
13.19.11.MAP,PART.
13.19.11.LGO,PL=200.
13.19.14. CPU ERROR EXIT AT  004564.
13.19.14. CM OUT OF RANGE.
13.19.14. ILLEGAL INSTRUCTION.
13.19.14.UEAD,      0.002KUNS.
13.19.14.UEPF,      0.013KUNS.
13.19.14.UFMS,      2.005KUNS.
13.19.14.UECP,      1.394SECS.
13.19.14.AESR,      2.142UNTS.
13.25.16.UCLP, AB51,      0.145KUNS.
```

```
                                IDENT  WORDS
                              *
                              *       WORDS IS A FUNCTION WHICH COUNTS THE NUMBER OF WORDS IN
                              *       A SENTENCE OR PHRASE.  A WORD IS A SEQUENCE OF CHARACTERS
                              *       SEPARATED BY ONE OR MORE BLANKS.  THE SENTENCE IS AN ARRAY
                              *       OF CHARACTERS, ONE CHARACTER PER WORD, LEFT JUSTIFIED,
                              *       BLANK FILLED.
                              *            CALLING SEQUENCE:
                              *                 NUMBER = WORDS(SENTENCE,LENGTH)

                                ENTRY  WORDS
                                EXT    REGDMP
 0                        1    WORDS  BSS    1
 1  5021000001                       SA2    A1+1          X2=ADDRESS OF LENGTH
              53110                  SA1    X1            A1=STARTING ADDRESS OF SENTENCE
                   53220             SA2    X2            X2=LENGTH OF SENTENCE
 2  63320                            SB3    X2            B3=LENGTH OF SENTENCE
       66100                       SB1    B0            B1=FLAG, 0=BLANKS  1=NONBLANKS
             6120000001            SB2    1             B2=CHARACTER COUNTER
 3  76600                            SX6    B0            X6=WORD COUNTER
       5150000016 +                 SA5    BLANK         X5=LEFT JUSTIFIED BLANK
                  13115             BX1    X1-X5         IS THE FIRST CHARACTER A BLANK
 4  0301000006 +                     ZR     X1,LOOP       IF SO, JUMP
             6110000001             SB1    1             IF NOT, FLAG=NONBLANK
 5  0100000000 X                     RJ     REGDMP
 6  54112                    LOOP    SA1    A1+B2         GET NEXT CHARACTER
       0100000000 X                 RJ     REGDMP
 7  13115                            BX1    X1-X5         IS THE CHARACTER A BLANK
       0311000012 +                 NZ     X1,OVER       IF NOT, JUMP
10  0410000013 +                     EQ     B1,NEXT       IS IT THE FIRST BLANK BETWEEN WORDS
             7266000001             SX6    X6+1          IF SO, INCREMENT WORD COUNT
11  66100                            SB1    B0            SET FLAG TO INDICATE BLANK ENCOUNTERRED
       0400000013 +                 EQ     NEXT
12  6110000001               OVER    SB1    1             SET FLAG TO INDICATE NONBLANK
13  6122000001               NEXT    SB2    B2+1          INCREMENT CHARACTER COUNTER
            0723000006 +             LT     B2,B3,LOOP    DID WE EXAMINE ALL CHARACTERS?  IF NOT,LOOP
14  0410000000 +                     EQ     B1,WORDS      WAS THE LAST CHARACTER A BLANK?  YES,RETURN
             7266000001             SX6    X6+1          SENTENCE ENDED WITH A WORD, INCREMENT COUNT
15  0400000000 +                     EQ     WORDS         RETURN
16  5555555555555555555      BLANK   DATA   1H           DEFINE A LEFT JUSTIFED BLANK
17                                   END

          43300B CM  STORAGE USED              41 STATEMENTS        6 SYMBOLS
                     MODEL 73 ASSEMBLY      0.267 SECONDS         17 REFERENCES
```

?

BLOCK	ADDRESS	LENGTH	FILE	DATE	PROCSSR VER LEVEL	HARDWARE	COMMENTS
FDL.RES	13021	211	SL-SYSLIB	80/10/01	COMPASS 3.6 518		FAST DYNAMIC LOADER RESIDENT.
FDL.MMI	13232	222	SL-SYSLIB	80/10/01	COMPASS 3.6 518		FDL MEMORY MANAGER INTERFACE.

.679 CP SECONDS 30400B CM STORAGE USED 6 TABLE MOVES

THIS IS REGISTER DUMP NUMBER 1 CALLED FROM LOCATION 004564

X0	77777777777777777700	A0	000000	B0	000000
X1	55555555555555555555	A1	004317	B1	000000
X2	00000000000000000240	A2	004312	B2	000001
X3	00000000007750000000	A3	007576	B3	000240
X4	00000000000000005165	A4	004304	B4	000012
X5	55555555555555555555	A5	004574	B5	000052
X6	00000000000000000000	A6	005173	B6	000001
X7	20120000000000005165	A7	005215	B7	000000

THIS IS REGISTER DUMP NUMBER 2 CALLED FROM LOCATION 004564

X0	77777777777777777700	A0	000000	B0	000000
X1	55555555555555555555	A1	004321	B1	000000
X2	00000000000000000240	A2	004312	B2	000002
X3	00000000007750000000	A3	007576	B3	000240
X4	00000000000000005165	A4	004304	B4	000012
X5	55555555555555555555	A5	004574	B5	000052
X6	00000000000000000000	A6	005173	B6	000001
X7	20120000000000005165	A7	005215	B7	000000

THIS IS REGISTER DUMP NUMBER 3 CALLED FROM LOCATION 004564

X0	77777777777777777700	A0	000000	B0	000000
X1	55555555555555555555	A1	004324	B1	000000
X2	00000000000000000240	A2	004312	B2	000003
X3	00000000007750000000	A3	007576	B3	000240
X4	00000000000000005165	A4	004304	B4	000012
X5	55555555555555555555	A5	004574	B5	000052
X6	00000000000000000000	A6	005173	B6	000001
X7	20120000000000005165	A7	005215	B7	000000

THIS IS REGISTER DUMP NUMBER 4 CALLED FROM LOCATION 004564 ** A20 **

X0	7777777777777777700	A0	000000	B0	000000
X1	24555555555555555555	A1	004330	B1	000000
X2	00000000000000000240	A2	004312	B2	000004
X3	00000000007750000000	A3	007576	B3	000240
X4	00000000000000005165	A4	004304	B4	000012
X5	55555555555555555555	A5	004574	B5	000052
X6	00000000000000000000	A6	005173	B6	000001
X7	20120000000000005165	A7	005215	B7	000000

THIS IS REGISTER DUMP NUMBER 5 CALLED FROM LOCATION 004564

X0	7777777777777777700	A0	000000	B0	000000
X1	55555555555555555555	A1	004335	B1	000001
X2	00000000000000000240	A2	004312	B2	000005
X3	00000000007750000000	A3	007576	B3	000240
X4	00000000000000005165	A4	004304	B4	000012
X5	55555555555555555555	A5	004574	B5	000052
X6	00000000000000000000	A6	005173	B6	000001
X7	20120000000000005165	A7	005215	B7	000000

THIS IS REGISTER DUMP NUMBER 6 CALLED FROM LOCATION 004564

X0	7777777777777777700	A0	000000	B0	000000
X1	55555555555555555555	A1	004343	B1	000000
X2	00000000000000000240	A2	004312	B2	000006
X3	00000000007750000000	A3	007576	B3	000240
X4	00000000000000005165	A4	004304	B4	000012
X5	55555555555555555555	A5	004574	B5	000052
X6	00000000000000000001	A6	005173	B6	000001
X7	20120000000000005165	A7	005215	B7	000000

THIS IS REGISTER DUMP NUMBER 7 CALLED FROM LOCATION 004564

X0	7777777777777777700	A0	000000	B0	000000
X1	11555555555555555555	A1	004352	B1	000000
X2	00000000000000000240	A2	004312	B2	000007
X3	00000000007750000000	A3	007576	B3	000240
X4	00000000000000005165	A4	004304	B4	000012
X5	55555555555555555555	A5	004574	B5	000052
X6	00000000000000000001	A6	005173	B6	000001
X7	20120000000000005165	A7	005215	B7	000000

THIS IS REGISTER DUMP NUMBER 8 CALLED FROM LOCATION 004564

X0	7777777777777777700	A0	000000	B0	000000
X1	06555555555555555555	A1	004362	B1	000001
X2	00000000000000000240	A2	004312	B2	000010

STORAGE ALLOCATION.

ADDRESS	LENGTH
0	206
206	

BINARY CONTROL CARDS.

```
IDENT Q8REGZ
END
```

ENTRY POINTS.

Q8REGZ	0+	Q8RSTZ	141+
Q8SAVZ	44+	REGDMP	0+

EXTERNAL SYMBOLS.

Q8PRGZ

?

```
                                IDENT Q8REGZ
                                ENTRY Q8REGZ,Q8SAVZ,Q8RSTZ
                                EXT Q8PRGZ
                           *
                           *    Q8REGZ,Q8SAVZ,Q8RSTZ AND Q8PRGZ, A SET OF PROGRAMS TO SAVE AND RES
                           *    C. P. REGISTERS WITHOUT DESTROYING THE CONTENTS OF ANY REGISTER
                           *    MODIFIED TO WORK UNDER SCOPE 3.0 BY STEVE JASIK - CDC
                           *    MODIFIED TO WORK UNDER SCOPE 3.1 BY EDE JACOBY AND DAVID ROSENBERG
                           *
                                ENTRY  REGDMP
 0  0200400000 +             REGDMP  JP      *+400000B
                     0 +     Q8REGZ  EQU     REGDMP
 1  0100000044 +                     RJ      Q8SAVZ       .SAVE THE OPERATING (X,A,B) REGISTERS
 2  5110000000 +                     SA1     Q8REGZ       .PICK UP THE JUMP BACK TO THE CALLING ROUTI
              20136                  LX1     30           .RETURN ADDRESS TO LOWER 18 BITS
 3  7271777776                       SX7       X1-1             .X7=ADDRESS CALLED FROM
              5170000012 +           SA7       TEMP2        .SAVE CALLING ADDRESS IN CORE
                           *    SET UP ARGUMENTS TO REGPNT
 4  5110000007 +                     SA1     ARGADD       .SET CALLING SEQUENCE FOR Q8PRGZ
              0100000000 X           RJ      Q8PRGZ       .PRINT THE CONTENTS OF THE REGISTERS
 5  0100000141 +                     RJ      Q8RSTZ       .RESTORE THE CONTENTS OF THE OPERATING REGS
 6  0400000000 +                     EQ      Q8REGZ       .RETURN TO CALLING ROUTINE
 7  0000000000000000000013 +  ARGADD  VFD     60/SAVX0
10  0000000000000000003012 +          VFD     60/TEMP2
11                        1           BSSZ    1
12                        1  TEMP2    BSS     1            .TEMPORARY FOR STORAGE OF CALLING ADDRESS
                           *
                           *    END OF Q8REGZ
                           *
                           *    CONTENTS OF C. P. REGISTERS ARE STORED HERE
                           *
13                        1   SAVX0    BSS      1          .THE CONTENTS OF X0 WILL BE STORED HERE.
14                        1   SAVX1    BSS      1          .THE CONTENTS OF X1 WILL BE STORED HERE.
15                        1   SAVX2    BSS      1          .THE CONTENTS OF X2 WILL BE STORED HERE.
16                        1   SAVX3    BSS      1          .THE CONTENTS OF X3 WILL BE STORED HERE.
17                        1   SAVX4    BSS      1          .THE CONTENTS OF X4 WILL BE STORED HERE.
20                        1   SAVX5    BSS      1          .THE CONTENTS OF X5 WILL BE STORED HERE.
21                        1   SAVX6    BSS      1          .THE CONTENTS OF X6 WILL BE STORED HERE.
22                        1   SAVX7    BSS      1          .THE CONTENTS OF X7 WILL BE STORED HERE.
23                        1   SAVA0    BSS      1          .A0 RIGHT ADJUSTED WITH SIGN EXTENSION.
24                        1   SAVA1    BSS      1          .A1 RIGHT ADJUSTED WITH SIGN EXTENSION.
25                        1   SAVA2    BSS      1          .A2 RIGHT ADJUSTED WITH SIGN EXTENSION.
26                        1   SAVA3    BSS      1          .A3 RIGHT ADJUSTED WITH SIGN EXTENSION.
27                        1   SAVA4    BSS      1          .A4 RIGHT ADJUSTED WITH SIGN EXTENSION.
30                        1   SAVA5    BSS      1          .A5 RIGHT ADJUSTED WITH SIGN EXTENSION.
31                        1   SAVA6    BSS      1          .A6 RIGHT ADJUSTED WITH SIGN EXTENSION.
32                        1   SAVA7    BSS      1          .A7 RIGHT ADJUSTED WITH SIGN EXTENSION.
33                        1   SAVB0    BSS      1          .THIS WORD WILL BE ZEROED TO INDICATE B0.
34                        1   SAVB1    BSS      1          .B1 RIGHT ADJUSTED WITH SIGN EXTENSION.
35                        1   SAVB2    BSS      1          .B2 RIGHT ADJUSTED WITH SIGN EXTENSION.
36                        1   SAVB3    BSS      1          .B3 RIGHT ADJUSTED WITH SIGN EXTENSION.
37                        1   SAVB4    BSS      1          .B4 RIGHT ADJUSTED WITH SIGN EXTENSION.
40                        1   SAVB5    BSS      1          .B5 RIGHT ADJUSTED WITH SIGN EXTENSION.
41                        1   SAVB6    BSS      1          .B6 RIGHT ADJUSTED WITH SIGN EXTENSION.
42                        1   SAVB7    BSS      1          .B7 RIGHT ADJUSTED WITH SIGN EXTENSION.
                           *
                           *    ROUTINE TO SAVE THE C. P. REGISTERS
```

```
                                *
 43  21432301263255000044 +     +          VFD    42/6HQ8SAVZ,18/Q8SAVZ
 44  00000000000000000000        Q8SAVZ    DATA Ø
 45  0670000047 +                          PL       B7,BIT16  .IS BIT 17 OF B7 ON, IF NOT BRANCH TO BIT16
                 0400000046 +              EQ       BØ,BØ,RJ17     .YES. GO INDICATE THAT BIT 17 WAS ON.
 46  0100000046 +               RJ17       RJ       RJ17      .INDICATE THAT BIT 17 OF B7 WAS ON.
 47  66777                      BIT16      SB7      B7+B7     .BRING BIT 16 OF B7 INTO THE SIGN POSITION.
           0670000051 +                    PL       B7,BIT15  .WAS BIT 16 (NOW BIT 17) OF B7 ON.
 5Ø  0100000050 +               RJ16       RJ       RJ16      .YES. INDICATE THAT BIT 16 OF B7 WAS ON.
 51  66777                      BIT15      SB7      B7+B7     .BRING BIT 15 OF B7 INTO THE SIGN POSITION.
           0670000053 +                    PL       B7,BIT14  .WAS BIT 15 (NOW BIT 17) OF B7 ON.
 52  0100000052 +               RJ15       RJ       RJ15      .YES. INDICATE THAT BIT 15 OF B7 WAS ON.
 53  66777                      BIT14      SB7      B7+B7     .BRING BIT 14 OF B7 INTO THE SIGN POSITION.
           0670000055 +                    PL       B7,BIT13  .WAS BIT 14 (NOW BIT 17) OF B7 ON.
 54  0100000054 +               RJ14       RJ       RJ14      .YES. INDICATE THAT BIT 14 OF B7 WAS ON.
 55  66777                      BIT13      SB7      B7+B7     .BRING BIT 13 OF B7 INTO THE SIGN POSITION.
           0670000057 +                    PL       B7,BIT12  .WAS BIT 13 (NOW BIT 17) OF B7 ON.
 56  0100000056 +               RJ13       RJ       RJ13      .YES. INDICATE THAT BIT 13 OF B7 WAS ON.
 57  66777                      BIT12      SB7      B7+B7     .BRING BIT 12 OF B7 INTO THE SIGN POSITION.
           0670000061 +                    PL       B7,BIT11  .WAS BIT 12 (NOW BIT 17) OF B7 ON.
 6Ø  0100000060 +               RJ12       RJ       RJ12      .YES. INDICATE THAT BIT 12 OF B7 WAS ON.
 61  66777                      BIT11      SB7      B7+B7     .BRING BIT 11 OF B7 INTO THE SIGN POSITION.
           0670000063 +                    PL       B7,BIT10  .WAS BIT 11 (NOW BIT 17) OF B7 ON.
 62  0100000062 +               RJ11       RJ       RJ11      .YES. INDICATE THAT BIT 11 OF B7 WAS ON.
 63  66777                      BIT1Ø      SB7      B7+B7     .BRING BIT 10 OF B7 INTO THE SIGN POSITION.
           0670000065 +                    PL       B7,BIT9   .WAS BIT 10 (NOW BIT 17) OF B7 ON.
 64  0100000064 +               RJ1Ø       RJ       RJ1Ø      .YES. INDICATE THAT BIT 10 OF B7 WAS ON.
 65  66777                      BIT9       SB7      B7+B7     .BRING BIT 9 OF B7 INTO THE SIGN POSITION.
           0670000067 +                    PL       B7,BIT8   .WAS BIT 9 (NOW BIT 17) OF B7 ON.
 66  0100000066 +               RJ9        RJ       RJ9       .YES. INDICATE THAT BIT 9 OF B7 WAS ON.
 67  66777                      BIT8       SB7      B7+B7     .BRING BIT 8 OF B7 INTO THE SIGN POSITION.
           0670000071 +                    PL       B7,BIT7   .WAS BIT 8 (NOW BIT 17) OF B7 ON.
 7Ø  0100000070 +               RJ8        RJ       RJ8       .YES. INDICATE THAT BIT 8 OF B7 WAS ON.
 71  66777                      BIT7       SB7      B7+B7     .BRING BIT 7 OF B7 INTO THE SIGN POSITION.
           0670000073 +                    PL       B7,BIT6   .WAS BIT 7 (NOW BIT 17) OF B7 ON.
 72  0100000072 +               RJ7        RJ       RJ7       .YES. INDICATE THAT BIT 7 OF B7 WAS ON.
 73  66777                      BIT6       SB7      B7+B7     .BRING BIT 6 OF B7 INTO THE SIGN POSITION.
           0670000075 +                    PL       B7,BIT5   .WAS BIT 6 (NOW BIT 17) OF B7 ON.
 74  0100000074 +               RJ6        RJ       RJ6       .YES. INDICATE THAT BIT 6 OF B7 WAS ON.
 75  66777                      BIT5       SB7      B7+B7     .BRING BIT 5 OF B7 INTO THE SIGN POSITION.
           0670000077 +                    PL       B7,BIT4   .WAS BIT 5 (NOW BIT 17) OF B7 ON.
 76  0100000076 +               RJ5        RJ       RJ5       .YES. INDICATE THAT BIT 5 OF B7 WAS ON.
 77  66777                      BIT4       SB7      B7+B7     .BRING BIT 4 OF B7 INTO THE SIGN POSITION.
           0670000101 +                    PL       B7,BIT3   .WAS BIT 4 (NOW BIT 17) OF B7 ON.
1ØØ  0100000100 +               RJ4        RJ       RJ4       .YES. INDICATE THAT BIT 4 OF B7 WAS ON.
1Ø1  66777                      BIT3       SB7      B7+B7     .BRING BIT 3 OF B7 INTO THE SIGN POSITION.
           0670000103 +                    PL       B7,BIT2   .WAS BIT 3 (NOW BIT 17) OF B7 ON.
1Ø2  0100000102 +               RJ3        RJ       RJ3       .YES. INDICATE THAT BIT 3 OF B7 WAS ON.
1Ø3  66777                      BIT2       SB7      B7+B7     .BRING BIT 2 OF B7 INTO THE SIGN POSITION.
           0670000105 +                    PL       B7,BIT1   .WAS BIT 2 (NOW BIT 17) OF B7 ON.
1Ø4  0100000104 +               RJ2        RJ       RJ2       .YES. INDICATE THAT BIT 2 OF B7 WAS ON.
1Ø5  66777                      BIT1       SB7      B7+B7     .BRING BIT 1 OF B7 INTO THE SIGN POSITION.
           0670000107 +                    PL       B7,BIT0   .WAS BIT 1 (NOW BIT 17) OF B7 ON.
1Ø6  0100000106 +               RJ1        RJ       RJ1       .YES. INDICATE THAT BIT 1 OF B7 WAS ON.
1Ø7  66777                      BIT0       SB7      B7+B7     .BRING BIT Ø OF B7 INTO THE SIGN POSITION.
           0670000111 +                    PL       B7,ENDRJ  .WAS BIT Ø (NOW BIT 17) OF B7 ON.
11Ø  0100000110 +               RJØ        RJ       RJØ       .YES. INDICATE THAT BIT Ø OF B7 WAS ON.
```

```
111  65770                  ENDRJ   SB7     A7-B0     .TEMPORARILY HOLD C(A7) IN B7.
          5170000022 +              SA7     SAVX7     .SAVE X7 IN MEMORY.
                   75760            SX7     A6-B0     .MOVE C(A6) INTO X7.
112  5170000031 +                   SA7     SAVA6     .SAVE A6 IN MEMORY.
               5160000021 +         SA6     SAVX6     .SAVE X6 IN MEMORY.
113  77670                          SX6     B7-B0     .MOVE C(A7) INTO X6.
          6170000001                SB7     1         .B7 WILL REMAIN EQUAL TO 1 THROUGH THE END.
                   54677            SA6     A7+B7     .SAVE A7 IN MEMORY.
114  43700                          MX7     0         .PRETEND THIS IS (SX7 B0) FOR SAVING B0.
          77610                     SX6     B1-B0     .MOVE C(B1) INTO X6.
               54767                SA7     A6+B7     .SAVE B0 (ALWAYS ZERO) IN MEMORY.
                    77720           SX7     B2-B0     .MOVE C(B2) INTO X7.
115  54677                          SA6     A7+B7     .SAVE B1 IN MEMORY.
          77630                     SX6     B3-B0     .MOVE C(B3) INTO X6.
               54767                SA7     A6+B7     .SAVE B2 IN MEMORY.
                    77740           SX7     B4-B0     .MOVE C(B4) INTO X7.
116  54677                          SA6     A7+B7     .SAVE B3 IN MEMORY.
          77650                     SX6     B5-B0     .MOVE C(B5) INTO X6.
               54767                SA7     A6+B7     .SAVE B4 IN MEMORY.
                    77760           SX7     B6-B0     .MOVE C(B6) INTO X7.
117  54677                          SA6     A7+B7     .SAVE B5 IN MEMORY.
          10600                     BX6     X0        .MOVE C(X0) INTO X6.
               54767                SA7     A6+B7     .SAVE B6 IN MEMORY.
                    10711           BX7     X1        .MOVE C(X1) INTO X7.
120  5160000013 +                   SA6     SAVX0     .SAVE X0 IN MEMORY.
               10622                BX6     X2        .MOVE C(X2) INTO X6.
                    54767           SA7     A6+B7     .SAVE X1 IN MEMORY.
121  10733                          BX7     X3        .MOVE C(X3) INTO X7.
          54677                     SA6     A7+B7     .SAVE X2 IN MEMORY.
               10644                BX6     X4        .MOVE C(X4) INTO X6.
                    54767           SA7     A6+B7     .SAVE X3 IN MEMORY.
122  10755                          BX7     X5        .MOVE C(X5) INTO X7.
          54677                     SA6     A7+B7     .SAVE X4 IN MEMORY.
               75600                SX6     A0-B0     .MOVE C(A0) INTO X6.
                    54767           SA7     A6+B7     .SAVE X5 IN MEMORY.
123  75710                          SX7     A1-B0     .MOVE C(A1) INTO X7.
          5160000023 +              SA6     SAVA0     .SAVE A0 IN MEMORY.
                   75620            SX6     A2-B0     .MOVE C(A2) INTO X6.
124  54767                          SA7     A6+B7     .SAVE A1 IN MEMORY.
          75730                     SX7     A3-B0     .MOVE C(A3) INTO X7.
               54677                SA6     A7+B7     .SAVE A2 IN MEMORY.
                    75640           SX6     A4-B0     .MOVE C(A4) INTO X6.
125  54767                          SA7     A6+B7     .SAVE A3 IN MEMORY.
          75750                     SX7     A5-B0     .MOVE C(A5) INTO X7.
               54677                SA6     A7+B7     .SAVE A4 IN MEMORY.
                    54767           SA7     A6+B7     .SAVE A5 IN MEMORY.
126  43101                          MX1     1         .SET X1 = 4000000000000000000000.
          5120000046 +              SA2     RJ17      .FETCH THE (RJ *) USED TO INDICATE BIT 17.
127  5130000137 +                   SA3     RJINS     .FETCH A (RJ RJ17) INSTRUCTION.
               66277                SB2     B7+B7     .SET B2 = 2.
                    20140           LX1     32        .SET X1 = 0000000002000000000000.
130  10633                          BX6     X3        .SET X6 = (RJ RJ17 $ NO $ NO).
          21267                     AX2     55        .POSITION BIT 17 IN BIT 1 OF X2.
               6130000020           SB3     16        .B3 WILL COUNT ITERATIONS THROUGH THE LOOP.
131  10722                          BX7     X2        .B7 WILL BE RECONSTRUCTED IN X7.
          54222                     SA2     A2+B2     .FETCH THE (RJ *) USED TO INDICATE BIT 16.
132  55622                  LOOP    SA6     A2-B2     .STORE A WORD OF THE FORM (RJ * $ NO $ NO)
```

```
          21270                         AX2     56          .SHIFT THE SAVED BIT TO POSITION 0 OF X2.
               36661                    IX6     X6+X1       .ADD 2 TO THE ADDRESS OF THE (RJ *) INST.
                    67337               SB3     B3-B7       .DECREASE THE ITERATION COUNTER BY 1.
133  12772                              BX7     X7+X2       .OR THIS BIT INTO THE RECONSTRUCTION OF B7.
          54222                         SA2     A2+B2       .FETCH THE NEXT (RJ *) INSTRUCTION.
               20701                    LX7     1           .MAKE ROOM FOR THE NEXT BIT OF B7.
134  0630000132 +                       PL      B3,LOOP     .IF WE HAVE NOT YET RESTORED BIT 0 GO LOOP.
               20751                    LX7     41          .PUT BIT 17 OF B7 IN POSITION 59 OF X7.
                    55622               SA6     A2-B2       .RESTORE THE (RJ *) INSTRUCTION FOR BIT 0.
135  21752                              AX7     42          .RIGHT ADJUST B7 IN X7 WITH SIGN EXTENSION.
          5170000042 +                  SA7     SAVB7       .SAVE B7 IN MEMORY.
136  0400000044 +                       EQ      Q8SAVZ
                                   *
137  0100000046 +                  RJINS   RJ      RJ17        .THIS IS STORED AT RJ17, NOT EXECUTED HERE.
                                   *
                                   *       ROUTINE TO RESTORE THE C. P. REGISTERS
                                   *
140  21432223243255000141 +        +       VFD     42/6HQ8RSTZ,18/Q8RSTZ
141  00000000000000000000          Q8RSTZ  DATA    0
142  5110000034 +                          SA1     SAVB1
               6110000001                 SB1     1
143  76710                              SX7     B1          .X7 = 1
          43022                         MX0     18
               76710                    SX7     B1
                    20022               LX0     18
144  5120000176 +                       SA2     RSRB1
               11301                    BX3     X0*X1
                    15220               BX2     -X0*X2
145  20725                              LX7     21
          6120000006                    SB2     6
                    12623               BX6     X2+X3
146  54620                              SA6     A2
147  54111                         SETBS   SA1     A1+B1
          67221                         SB2     B2-B1
               11301                    BX3     X0*X1
                    36227               IX2     X2+X7
150  12623                              BX6     X2+X3
          54661                         SA6     A6+B1
               0520000147 +             NZ      B2,SETBS
151  5130000173 +                       SA3     W11
               5150000020 +             SA5     SAVX5
152  43404                              MX4     4
          20463                         LX4     51
               15634                    BX6     -X4*X3
153  0325000154 +                       PL      X5,W10
               12646                    BX6     X4+X6
154  54630                         W10     SA6     A3
          43274                         MX2     60
               6272777777               SB7     X2+777777B
155  5150000023 +                       SA5     SAVA0
               53057                    SA0     X5+B7       .RESTORE A0
                    54451               SA4     A5+B1
156  53147                              SA1     X4+B7       .RESTORE A1
          54541                         SA5     A4+B1
               53257                    SA2     X5+B7       .RESTORE A2
                    54451               SA4     A5+B1
157  53347                              SA3     X4+B7       .RESTORE A3
```

```
          54541                          SA5     A4+B1
                53457                    SA4     X5+B7     .RESTORE A4
160  5150000031 +                        SA5     SAVA6
                53557                    SA5     X5+B7
                     10655               BX6     X5
161  55650                               SA6     A5-B0     .RESTORE A6
          5150000032 +                   SA5     SAVA7
                     53557               SA5     X5+B7
162  10755                               BX7     X5
          55750                          SA7     A5-B0     .RESTORE A7
                5150000013 +             SA5     SAVX0
163  10055                               BX0     X5        .RESTORE X0
          54551                          SA5     A5+B1
               10155                     BX1     X5        .RESTORE X1
                     54551               SA5     A5+B1
164  10255                               BX2     X5        .RESTORE X2
          54551                          SA5     A5+B1
               10355                     BX3     X5        .RESTORE X3
                     54551               SA5     A5+B1
165  10455                               BX4     X5        .RESTORE X4
          5150000021 +                   SA5     SAVX6
                     10655               BX6     X5        .RESTORE X6
166  54551                               SA5     A5+B1
          10755                          BX7     X5        .RESTORE X7
                5150000020 +             SA5     SAVX5
167  26515                               UX5     B1,X5
          20513                          LX5     11
               26525                     UX5     B2,X5
                     20513               LX5     11
170  26535                               UX5     B3,X5
          20513                          LX5     11
               26545                     UX5     B4,X5
                     20513               LX5     11
171  26555                               UX5     B5,X5
          20513                          LX5     11
               26565                     UX5     B6,X5
172  5150000030 +                        SA5     SAVA5
               53557                     SA5     X5+B7     .RESTORE A5
173  43500                    W11        MX5     0         .START RESTORATION OF X5
          27565                          PX5     B6,X5
               20561                     LX5     49
                     27555               PX5     B5,X5
174  20561                               LX5     49
          27545                          PX5     B4,X5
               20561                     LX5     49
                     27535               PX5     B3,X5
175  20561                               LX5     49
          27525                          PX5     B2,X5
               20561                     LX5     49
                     27515               PX5     B1,X5     .RESTORATION  OF X5 COMPLETED
176  46000                    RSRB1      NO
          46000                          NO
               6117000000                SB1     B7+0      .RESTORE B1
177  46000                    RSRB2      NO
          46000                          NO
               6127000000                SB2     B7+0      .RESTORE B2
200  46000                    RSRB3      NO
```

?

```
          46000                         NO
                6137000000              SB3     B7+0        .RESTORE B3
201  46000                    RSRB4     NO
          46000                         NO
                6147000000              SB4     B7+0        .RESTORE B4
202  46000 ¬¬ (               RSRB5     NO
          46000                         NO
                6157000000              SB5     B7+0        .RESTORE B5
203  46000                    RSRB6     NO
          46000                         NO
                6167000000              SB6     B7+0        .RESTORE B6
204  46000                    RSRB7     NO
          46000                         NO
                6177000000              SB7     B7+0        .RESTORE B7
205  0400000141 +                       EQ      Q8RSTZ
                              *      END OF REGISTER DUMPING PACKAGE
206                                    END

                46400B CM  STORAGE USED              302 STATEMENTS       79 SYMBOLS
                           MODEL 73 ASSEMBLY       2.067 SECONDS        157 REFERENCES
```

SYMBOLIC REFERENCE TABLE.

ARGADD	7	PROGRAM*	2/19		2/23	L			
BIT0	107	PROGRAM*	3/53		3/55	L			
BIT1	105	PROGRAM*	3/50		3/52	L			
BIT10	63	PROGRAM*	3/23		3/25	L			
BIT11	61	PROGRAM*	3/20		3/22	L			
BIT12	57	PROGRAM*	3/17		3/19	L			
BIT13	55	PROGRAM*	3/14		3/16	L			
BIT14	53	PROGRAM*	3/11		3/13	L			
BIT15	51	PROGRAM*	3/08		3/10	L			
BIT16	47	PROGRAM*	3/04		3/07	L			
BIT2	103	PROGRAM*	3/47		3/49	L			
BIT3	101	PROGRAM*	3/44		3/46	L			
BIT4	77	PROGRAM*	3/41		3/43	L			
BIT5	75	PROGRAM*	3/38		3/40	L			
BIT6	73	PROGRAM*	3/35		3/37	L			
BIT7	71	PROGRAM*	3/32		3/34	L			
BIT8	67	PROGRAM*	3/29		3/31	L			
BIT9	65	PROGRAM*	3/26		3/28	L			
ENDRJ	111	PROGRAM*	3/56		4/01	L			
LOOP	132	PROGRAM*	4/57	L	5/07				
Q8PRGZ	0	EXTERNAL*	2/03	X	2/20				
Q8REGZ	0	PROGRAM*	2/02	E	2/12	D	2/14	2/22	
Q8RSTZ	141	PROGRAM*	2/02	E	2/21		5/18	5/19 L	7/15
Q8SAVZ	44	PROGRAM*	2/02	E	2/13		3/02	3/03 L	5/12
REGDMP	0	PROGRAM*	2/10	E	2/11	L	2/12		
RJINS	137	PROGRAM*	4/49		5/14	L			
RJ0	110	PROGRAM*	3/57	L	3/57				
RJ1	106	PROGRAM*	3/54	L	3/54				
RJ10	64	PROGRAM*	3/27	L	3/27				
RJ11	62	PROGRAM*	3/24	L	3/24				
RJ12	60	PROGRAM*	3/21	L	3/21				
RJ13	56	PROGRAM*	3/18	L	3/18				
RJ14	54	PROGRAM*	3/15	L	3/15				
RJ15	52	PROGRAM*	3/12	L	3/12				
RJ16	50	PROGRAM*	3/09	L	3/09				
RJ17	46	PROGRAM*	3/05		3/06	L	3/06	4/48	5/14
RJ9	104	PROGRAM*	3/51	L	3/51				
RJ3	102	PROGRAM*	3/48	L	3/48				
RJ4	100	PROGRAM*	3/45	L	3/45				
RJ5	76	PROGRAM*	3/42	L	3/42				
RJ6	74	PROGRAM*	3/39	L	3/39				
RJ7	72	PROGRAM*	3/36	L	3/36				
RJ8	70	PROGRAM*	3/33	L	3/33				
RJ9	66	PROGRAM*	3/30	L	3/30				
RSRB1	176	PROGRAM*	5/26		6/51	L			
RSRB2	177	PROGRAM*	6/54	L					
RSRB3	200	PROGRAM*	6/57	L					
RSRB4	201	PROGRAM*	7/03	L					
RSRB5	202	PROGRAM*	7/06	L					
RSRB6	203	PROGRAM*	7/09	L					
RSRB7	204	PROGRAM*	7/12	L					
SAVA0	23	PROGRAM*	2/40	L	4/37	S	5/50		
SAVA1	24	PROGRAM*	2/41	L					
SAVA2	25	PROGRAM*	2/42	L					
SAVA3	26	PROGRAM*	2/43	L					
SAVA4	27	PROGRAM*	2/44	L					
SAVA5	30	PROGRAM*	2/45	L	6/37				

SYMBOLIC REFERENCE TABLE.

SAVA6	31	PROGRAM*	2/46	L	4/04	S	6/03	
SAVA7	32	PROGRAM*	2/47	L	6/07			
SAVB0	33	PROGRAM*	2/48	L				
SAVB1	34	PROGRAM*	2/49	L	5/20			
SAVB2	35	PROGRAM*	2/50	L				
SAVB3	36	PROGRAM*	2/51	L				
SAVB4	37	PROGRAM*	2/52	L				
SAVB5	40	PROGRAM*	2/53	L				
SAVB6	41	PROGRAM*	2/54	L				
SAVB7	42	PROGRAM*	2/55	L	5/11	S		
SAVX0	13	PROGRAM*	2/23		2/32	L	4/25 S	6/11
SAVX1	14	PROGRAM*	2/33	L				
SAVX2	15	PROGRAM*	2/34	L				
SAVX3	16	PROGRAM*	2/35	L				
SAVX4	17	PROGRAM*	2/36	L				
SAVX5	20	PROGRAM*	2/37	L	5/41		6/25	
SAVX6	21	PROGRAM*	2/38	L	4/05	S	6/21	
SAVX7	22	PROGRAM*	2/39	L	4/02	S		
SETBS	147	PROGRAM*	5/33	L	5/39			
TEMP2	12	PROGRAM*	2/17	S	2/24		2/26 L	
W10	154	PROGRAM*	5/45		5/47	L		
W11	173	PROGRAM*	5/40		6/39	L		

```
      1               SUBROUTINE Q8PRGZ(RGSTRS,LOCTN)
                      DIMENSION INT(8),RGSTRS(24)
                      DATA NUMBER,INT/1,0,1,2,3,4,5,6,7/
                      PRINT 1, NUMBER,LOCTN
      5               NUMBER =NUMBER+1
                      PRINT 2,(INT(I),RGSTRS(I),INT(I),RGSTRS(I+8),INT(I),RGSTRS(I+16),
                     1   I=1,8)
                      PRINT 3
                      RETURN
     10              1 FORMAT(////10X,* THIS IS REGISTER DUMP NUMBER *,I3,
                      1  * CALLED FROM LOCATION *,O6/1H )
                     2 FORMAT(11X,1HX,I1,3X,O20,7X,1HA,I1,3X,O6,7X,1HB,I1,3X,O6)
                     3 FORMAT(1H )
                       END


        SYMBOLIC REFERENCE MAP (R=1)

  ENTRY POINTS
      3  Q8PRGZ

  VARIABLES     SN  TYPE            RELOCATION
    102  I          INTEGER                          103  INT       INTEGER    ARRAY
      0  LOCTN      INTEGER               F.P.        35  NUMBER    INTEGER
      0  RGSTRS     REAL       ARRAY      F.P.

  FILE NAMES        MODE
       OUTPUT       FMT

  STATEMENT LABELS
     61  1       FMT                        72  2       FMT                       100  3       FMT

  LOOPS  LABEL    INDEX     FROM-TO    LENGTH    PROPERTIES
     14           I           6 6        15B              EXT REFS

  STATISTICS
    PROGRAM LENGTH              117B       79
          52000B CM USED
?
```

APPENDIX C: MORE ABOUT PASSING PARAMETERS

PROCEDURES WITH A VARIABLE NUMBER OF ARGUMENTS

One of the virtues of the FORTRAN Extended calling sequence is the ease of writing functions with a variable number of arguments. As a simple example of this, we shall present a function that returns the maximum of its integer arguments (this is the same as the MAX1 built-in function). This function may take any number of arguments.

```
          IDENT     MAX
          ENTRY     MAX
MAX       BSS       1
          SA2       X1
          BX6       X2          X6 = FIRST ARGUMENT
MAXLOOP   SA1       A1+1        X1 = ADDRESS OF NEXT ARGUMENT
          ZR        X1,MAX      IF NO MORE ARGUMENTS, RETURN
          SA2       X1          X2 = NEXT ARGUMENT
          IX3       X6-X2       NEW ARGUMENT > X6 (CURRENT MAXIMUM)
          PL        X3,MAXLOOP
          BX6       X2          YES, REPLACE X6 BY NEW ARGUMENT
          EQ        MAXLOOP
          END
```

A related problem, counting the number of arguments to a function or subroutine, is presented in exercise 17.

RUN COMPILER CALLING SEQUENCE

As we noted in our discussion of parameter transmission (section 3.6), CDC's original FORTRAN compiler, named RUN, used a quite different calling sequence. Instead of passing the addresses of parameters as a list in memory, it placed the addresses of the first (up to) 6 parameters in registers B1 to B6 (if only one parameter, in B1, and so forth). For example, the IDIF function (section 3.9) would have been coded

```
          IDENT     IDIF
          ENTRY     IDIF
IDIF      BSS       1
          SA1       B1          X1=ARGUMENT 1
          SA2       B2          X2=ARGUMENT 2
          IX6       X1-X2       X6=ARGUMENT 1-ARGUMENT 2
          EQ        IDIF        RETURN
          END
```

What if a routine has more than six parameters? The addresses of the remaining parameters are stored in memory immediately before the traceback word of the routine being called (note: for RUN, the traceback word must be immediately before the entry line and contain, in the low 18 bits, the argument count of the subprogram). As an example we shall consider ISUM8, a function of 8 integer arguments which returns as its value the sum of these eight numbers. The code for ISUM8 is:

```
          IDENT     ISUM8
ARG7      BSS       1
ARG8      BSS       1
TRACEBAK  VFD       42/7LISUM8   ,18/8
          ENTRY     ISUM8
ISUM8     BSS       1
          SA1       B1         X1=ARGUMENT 1
          SA2       B2         X2=ARGUMENT 2
          SA3       B3         X3=ARGUMENT 3
          SA4       B4         X4=ARGUMENT 4
          IX6       X1+X2      X6=SUM=ARG1 + ARG2
          IX6       X6+X3      SUM = SUM + ARG3
          IX6       X6+X4      SUM = SUM + ARG4
          SA1       B5         X1= ARGUMENT 5
          SA2       B6         X2= ARGUMENT 6
          SA3       ARG7       X3= ADDRESS OF ARGUMENT 7
          SA3       X3         X3= ARGUMENT 7
          SA4       ARG8       X4= ADDRESS OF ARGUMENT 8
          SA4       X4         X4= ARGUMENT 8
          IX6       X6+X1      SUM=SUM + ARG5
          IX6       X6+X2      SUM=SUM + ARG6
          IX6       X6+X3      SUM=SUM + ARG7
          IX6       X6+X4      SUM=SUM + ARG8
          EQ        ISUM8      RETURN
          END
```

The FORTRAN routine will put the addresses of the seventh and eigth arguments in ARG7 and ARG8 before calling ISUM8. A well-formed traceback word, including the count of the number of arguments, is essential for routines with more than six arguments. The calling routine needs to know the number of arguments the called routine expects, in order to store the addresses of arguments after the sixth in the right place.

APPENDIX D

CENTRAL PROCESSOR INSTRUCTION TIMINGS

APPENDIX D: CENTRAL PROCESSOR INSTRUCTION TIMINGS

			CENTRAL PROCESSOR			
Opcode	Mnemonic	Unit(2)	6200 72	6400 6500 72	6600 74	171
00	PS	Branch				
01	RJ K	"	24	21	13	43
02	JP Bi+K	"	16	13	14	36
030	ZR Xj,K	"				
031	NZ Xj,K	"				
032	PL Xj,K	"				
033	NG Xj,K	"				
034	IR Xj,K	" branch	16	13		36
035	OR Xj,K	" taken				
036	DF Xj,K	"				
037	ID Xj,K	" branch				
04	EQ Bi,Bj,K	" not	16	5		19
	(ZR Bi,K)	taken				
05	NE Bi,Bj,K	"				
	(NZ Bi,K)					
06	GE Bi,Bj,K	"				
	(PL Bi,K)					
07	LT Bi,Bj,K	"				
	(NG Bi,K)					
10	BXi Xj	Boolean	8	5	3	17
11	BXi Xj*Xk	"	8	5	3	19
12	BXi Xj+Xk	"	8	5	3	19
13	BXi Xj-Xk	"	8	5	3	19
14	BXi -Xk	"	8	5	3	17
15	BXi -Xk*Xj	"	8	5	3	19
16	BXi -Xk+Xj	"	8	5	3	19
17	BXi -Xk-Xj	"	8	5	3	19
20	LXi jk	Shift	9	6	3	19
21	AXi jk	"	9	6	3	19
22	LXi Bj,Xk	"	9	6	3	19
23	AXi Bj,Xk	"	9	6	3	19
24	NXi Bj,Xk	"	10	7	4	20
25	ZXi Bj,Xk	"	10	7	4	20
26	UXi Bj,Xk	"	10	7	3	19
27	PXi Bj,Xk	"	10	7	3	19

INSTRUCTION TIMINGS

172	173 174	175	720	730	750	760	Notes	Page
								38
36	29	28	36	25	28	20		40
29	22	26	29	22	26	18		38
								41
								41
								41
								42
29	22	26	29	20	26	18		89
								89
							(3)	89
								90
12	5	3	12	5	2	2		43
								43
								43
								44
								44
								44
								44
								45
10	4	2	10	3	2	2		63
12	6	2	12	5	2	2		63
12	6	2	12	5	2	2		63
12	6	2	12	5	2	2		64
10	4	2	10	3	2	2		64
12	6	2	12	5	2	2		64
12	6	2	12	5	2	2		64
12	6	2	12	5	2	2		64
12	6	2	12	5	2	2		96
12	6	2	12	5	2	2		96
12	6	2	12	5	2	2		99
12	6	2	12	5	2	2		99
13	7	3	13	6	3	3		72
13	7	3	13	6	3	3		72
12	6	2	12	5	2	2		103
12	6	2	12	5	2	2		102

APPENDIX D: CENTRAL PROCESSOR INSTRUCTION TIMINGS

Opcode	Mnemonic	Unit(2)	6200 72	6400 6500 72	6600 74	171
30	FXi Xj+Xk	Add	14	11	4	24
31	FXi Xj-Xk	"	14	11	4	24
32	DXi Xj+Xk	"	14	11	4	24
33	DXi Xj-Xk	"	14	11	4	24
34	RXi Xj+Xk	"	14	11	4	24
35	RXi Xj-Xk	"	14	11	4	24
36	IXi Xj+Xk	Long Add	9	6	3	19
37	IXi Xj-Xk	"	9	6	3	19
40	FXi Xj*Xk	Multiply	60	57	10	71
41	RXi Xj*Xk	"	60	57	10	71
42	DXi Xj*Xk	"	60	57	10	71
43	MXi jk	Shift	9	6	3	19
44	FXi Xj/Xk	Divide	60	57	29	71
45	RXi Xj/Xk	Divide	60	57	29	71
460	NO		6	3	1	17
464	IM					
465	DM					
466	CC					
467	CU					
47	CXi Xk	Divide	71	68	8	80
50	SAi Aj+K	Increment				
51	SAi Bj+K	" i=0:	9	6		19
52	SAi Xj+K	" i=1-5:	15	12		35
53	SAi Xj+Bk	" i=6,7:	13	10		24
54	SAi Aj+Bk	"				
55	SAi Aj-Bk	"				
56	SAi Bj+Bk	"				
57	SAi Bj-Bk	"				
60	SBi Aj+K	"	8	5	3	18
61	SBi Bj+K	"	8	5	3	18
62	SBi Xj+K	"	8	5	3	18
63	SBi Xj+Bk	"	8	5	3	18
64	SBi Aj+Bk	"	8	5	3	18
65	SBi Aj-Bk	"	8	5	3	18
66	SBi Bj+Bk	"	8	5	3	18
67	SBi Bj-Bk	"	8	5	3	18

APPENDIX D: CENTRAL PROCESSOR INSTRUCTION TIMINGS

172	173 174	175	720	730	750	760	Notes	Page
17	11	4	16	9	4	4		71
17	11	4	16	9	4	4		71
17	11	4	16	9	4	4		71
17	11	4	16	9	4	4		71
17	11	4	16	9	4	4		71
17	11	4	16	9	4	4		72
12	6	2	12	5	2	2		66
12	6	2	12	5	2	2		66
64	58	5	63	57	5	5		77
64	58	5	63	57	5	5		77
64	58	5	63	57	5	5		77
12	6	2	12	5	2	2		98
64	58	20	63	57	20	20		84
64	58	20	63	57	20	20		85
10	3	1	10	3	1			48
							(4)	115
							(4)	113
							(4)	120
							(4)	116
73	67	2	73	67	2	2		
12	5	2	12	5	2	2		59
28	21	23	25	18	23	15	(5)	59
17	10	2	15	8	2	2		59
								59
								59
								59
								59
								59
11	5	2	11	4	2	2		61
11	5	2	11	4	2	2		61
11	5	2	11	4	2	2		61
11	5	2	11	4	2	2		61
11	5	2	11	4	2	2		61
11	5	2	11	4	2	2		61
11	5	2	11	4	2	2		61
11	5	2	11	4	2	2		61

APPENDIX D: CENTRAL PROCESSOR INSTRUCTION TIMINGS

Opcode	Mnemonic	Unit(2)	6200 72	6400 6500 72	6600 74	171
70	SXi Aj+K	Increment	9	6	3	19
71	SXi Bj+K	"	9	6	3	19
72	SXi Xj+K	"	9	6	3	19
73	SXi Xj+Bk	"	9	6	3	19
74	SXi Aj+Bk	"	9	6	3	19
75	SXi Aj-Bk	"	9	6	3	19
76	SXi Bj+Bk	"	9	6	3	19
77	SXi Bj-Bk	"	9	6	3	19

1. All instruction times are given in minor cycles (also called clock periods), which are

100 ns for 6000 and Cyber 70 series
50 ns for models 171-174, 720, and 730
25 ns for models 175, 750, and 760

2. Functional unit designations are for the 6600 and Cyber 70 model 74. In addition to units specified above,

for opcode 02: address calculation done in increment unit
for opcodes 03x: tests done in long add unit
for opcodes 04-07: tests done in increment unit

Functional unit relationships for models 175, 750, and 760 are the same except:

opcodes 24 and 25: normalize unit
opcodes 26 and 27: Boolean unit
opcode 47: population count unit

3. Branch timings for 6600 and Cyber 70 model 74:

	opcode 03x	opcode 04-07
branch in stack, branch taken	9	8
branch in stack, branch not taken	11	10
branch out of stack, branch taken	15	14
branch out of stack, branch not taken	14	13

For models 175, 750, and 760, if the branch is taken and the instruction branched to is in the instruction stack, 3 cycles are required.

172	173 174	175	720	730	750	760	Notes	Page
12	6	2	12	5	2	2		55
12	6	2	12	5	2	2		55
12	6	2	12	5	2	2		55
12	6	2	12	5	2	2		57
12	6	2	12	5	2	2		57
12	6	2	12	5	2	2		57
12	6	2	12	5	2	2		57
12	6	2	12	5	2	2		57

4. Instructions available only on models 72, 73, 171 (optional), 172, 173, 174, 720, and 730. The timing formulas for these instructions are quite complex; refer to the hardware manual for your specific machine.

5. On the 6600 and model 74, data is available in the X register 8 cycles after the instruction begins.

INDEX