

Interpreting Machines

THE COMPUTER SCIENCE LIBRARY

Operating and Programming Systems Series

PETER J. DENNING, *Editor*

<i>Halstead</i>	A Laboratory Manual for Compiler and Operating System Implementation
<i>Spirn</i>	Program Behavior: Models and Measurements
<i>Halstead</i>	Elements of Software Science
<i>Franta</i>	A Process View of Simulation
<i>Organick and Hinds</i>	Interpreting Machines: Architecture and Programming of the B1700/B1800 Series

OPERATING AND PROGRAMMING SYSTEMS SERIES

Interpreting Machines:
Architecture and
Programming of the
B1700/B1800 Series

Elliott I. Organick

The University of Utah

James A. Hinds

Burroughs Corporation



NORTH-HOLLAND · NEW YORK
NEW YORK · OXFORD · SHANNON

Elsevier North-Holland, Inc.
52 Vanderbilt Avenue, New York, New York 10017

Distributors outside the United States and Canada:
THOMOND BOOKS
(A Division of Elsevier/North-Holland Scientific Publishers, Ltd).
P. O. Box 85
Limerick, Ireland

©1978 by Elsevier North-Holland, Inc.

Library of Congress Cataloging in Publication Data

Organick, Elliott Irving, 1925–
Interpreting Machines: Architecture and Programming of the B1700/B1800 Series.

(Operating and programming systems series) (The Computer science library)

Includes index.

1. Burroughs B1726 (Computer) I. Hinds, James A., joint author. II. Title.

QA76.8.B85073 001.6'4'04 77-12127

ISBN 0-444-00241-3

ISBN 0-444-00242-1 pbk.

Manufactured in the United States of America

Contents

Preface	ix
Chapter 1 <i>Universal Host Computers</i>	1
1.1 Structure of Storage in the B1700 Family of Computers	4
1.2 The B1726 Model of Storage	7
Chapter 2 <i>The B1700 as an Interpreting Machine</i>	10
2.1 Instruction Decoding	12
Chapter 3 <i>Organization of the B1726 Microprocessor</i>	16
3.1 Data Fetch and Addressing	16
3.2 Data Examination and Manipulation	28
3.2.1 The arithmetic capability or “Function Box”	31
3.2.2 Arithmetic tidbits	32
3.3 Instruction Decoding	33
3.4 Control	37
Chapter 4 <i>The B1700 Computation Environment</i>	42
4.1 The Burroughs Concept of “Codefile”	45
4.2 Constructing a Computation Environment	46
4.3 Implementing a Complete MIL Program	47
4.4 Declarations in MIL	50
4.5 Literals	57
4.6 Input/Output in the McMIL Language	59
4.6.1 Declaring files	64
4.7 The LOADER (Details)	68
Chapter 5 <i>The Structure of an Interpreter</i>	74
5.1 Detection and Response to Faults and Interrupts	74
5.2 The Host Environment	80
5.3 The Shell Concept	81
5.4 Moving Top Down on the Interpreter Structure	87
5.4.1 Storage representation for the target machine	88
5.4.2 Discussion of MIL code for ADD routine	98

Chapter 6	<i>MIL Coding for Data Manipulation</i>	104
6.1	Arithmetic of the 24-Bit Function Box	104
6.2	VALIDATE.DECIMAL: Case Study for a Utility Routine	106
6.3	BINARY.TO.FA	121
6.4	ADDRESS.TO.BINARY	124
6.5	EFFECTIVE.ADDRESS	127
6.6	The ADD Routine	137
6.7	UNPACK.AND.WRITE	147
6.8	Chapter Summary	149
Chapter 7	<i>The Split-Level Control Store</i>	150
7.1	Control over the Use of H-store	152
7.2	Microinstruction Fetch from H-Store or G-Store	155
7.3	Embedding Tabulated Data in MIL Programs	157
7.4	Transfer of Microcode from G-Store to H-Store	161
7.5	Transferring Control to Another Interpreter	164
7.6	Summary	168
Appendix A—Abridged MIL Reference Guide		170
	Directory	170
1	Notation	172
2	Executable MIL Statements	173
3	Nonexecutable MIL Statements	205
4	Special MIL Expressions	210
Appendix B—Abridged Reference Guide to the B1726		212
1	B1726 Register Summary	212
2	Testable Bits for IF Statements	217
3	Microinstructions: Syntax and Semantics	219
Appendix C—A User's Guide to McMIL and SMACK		258
1	McMIL Statement Syntax	259
2	McMIL Statements for the Operation of the SMACK Processor	259
3	McMIL Statements for Documentation	260
4	McMIL Statements Used to Format the Listing	260
5	McMIL Statements for Storage Allocation and Addressing	261
6	McMIL Statements for Debugging	262
7	McMIL Statements for MCP Interface	263
8	McMIL Statements for MCP Communication	264

Contents	vii
Appendix D—Loader Primer	269
1 The JCL Command Sequences	269
2 Syntax of the Loader Card Deck	270
3 Example	274
Appendix E—McMIL Listing for an Abridged SAMOS Interpreter	275
Appendix F—The SAMOS Computer	304
1 The SAMOS Execution Cycle	306
2 Instruction Repertoire	307
3 Error Conditions	307
4 The SAMOS Loader	307
Index	311

Preface

The Burroughs B1700 family of computers exhibits a new style of architecture. These computers are known as interpretive definable-field machines. Their normal mode of execution is the interpretation of *other* computers, virtual or real. A system designed to interpret other computer systems should have a flexible storage-accessing mechanism so that bit strings of arbitrary length may be fetched and processed under control of the programmer. The definable-field feature of the B1700 family supports efficient interpretation of instructions and promotes effective use of storage. Overviews of these features were presented by W. T. Wilner in a series of papers in 1972 [“Design of the B1700”, pp. 489–497, and “B1700 Memory Utilization”, pp. 579–586, in *AFIPS Conference Proceedings*, Vol. 41, Part 1, and “Microprogramming Environment of the Burroughs B1700” in *IEEE Computer Society COMPCON72*, pp. 103–106.]

Innovative systems such as the B1700 and its successors are attractive laboratory facilities for education and research in computer science, especially for software engineering studies, including the design and evaluation of new or special-purpose computer and data-base systems, and for studies in software portability.

This book describes the architecture of the Burroughs B1700 family, with primary attention given to the B1726 computer system, its internal structure, and how it may be programmed for the emulation of other computer systems. The book may have only limited appeal to computer-system specialists who are looking for reasons to select one computer organization over another. We do not address the comparative strengths and weaknesses of the B1700. We do not address such interesting questions as why interpretation is important and when it is to be preferred over the more conventional compiler-based general-purpose systems popular today. We do not dwell on the history of interpretation nor on its potential for the future. (We only hint at the promise for multilevel interpreters.) Finally, we do not suggest other applications of the B1700 architecture, say for database computing. Rather, our objective is to help the person who is already motivated to learn the “insides” of the B1700 and who wants the knowhow to implement an interpreter at the microcode level.

The book grew out of a set of notes written for upper-level undergrad-

uate computer-science students who have some prior knowledge of conventional computer-system organization and low-level language programming. Students at the University of Utah have used these notes in a software laboratory course in which the major objective is to produce a microcoded emulator for a fairly simple computer, e.g., a PDP-11. For more advanced students who expect to use the B1700 for research, the same notes have been useful for self-study as a supplement to or replacement for available reference manual literature.

The programming language introduced and used in this text, McMIL, is an enhanced version of MIL (Micro Instruction Language, an assembler for which is supplied by Burroughs). The McMIL superset of MIL contains statement types which can be used by the programmer to simplify the generation of MIL instruction sequences that correctly interface a MIL interpreter program with the system environment (e.g., for achieving interrupt handling, i/o management, file system services, and process switching).

The text consists of seven chapters and several appendices. The first three chapters focus on the architecture of the B1700 family as interpreting machines, on the internal structure of the B1700 processor, and on its (symbolic) micro-level machine language. The next three chapters show ways to write micro-level programs. A major case study vehicle that is used is a simulator for the hypothetical computer SAMOS outlined in Appendix F. It is in Chapters 4, 5, and 6 that the assembly language MIL and its McMIL enhancements are thoroughly illustrated. Methodologies of higher-level language programming including stepwise decomposition, clean structure, and good documentation are applied in translating from problem statements expressed in relatively abstract terms to concrete McMIL programs. Appendices A, B, and C are intended as reference manuals for MIL, for the actual computer system's register and instruction semantics, and for the McMIL extensions, respectively. (Appendix D provides additional reference materials used for setting up test runs of an interpreter, and Appendix E offers listings of the toy SAMOS interpreter and a sample test run. The toy interpreter may be used in a set of exercises as a study vehicle and point of departure for some interesting modifications and enhancements.) Chapter 7 examines the fine points in the control structure of the B1726 as a microprogram processor.

These seven chapters intentionally focus on the existing hardware of the B1700 family for use in design and implementation of interpreters and are to a great extent independent of the supporting software supplied by Burroughs. It is expected that another book would be useful for focusing on the structure and functions of the Burroughs software,

including the operating system (MCP) and the critically important central module (known as GISMO) which serves as an i/o-device driver, process switcher, i/o buffer server, and interface with the MCP. Such a book would provide the reader with a serious look at the (system-controlled) environment which supports the execution of programs one has learned to write and test.

The authors acknowledge with deep appreciation the support of our colleagues, students, and secretarial friends at Utah who have helped us assemble this text. We are also most fortunate for the support received from the Burroughs Corporation. Many persons within Burroughs helped make the project at Utah and this book, one of the byproducts, a reality and, we hope, a success. We are grateful to all of these individuals. In particular, the project could not have become a reality without the help and confidence of R. R. Johnson, R. D. Merrell, and R. S. Barton, members of the Burroughs engineering organization who were early advocates of the B1700 as a system worthy of serious attention and use in computer-science and engineering studies. This book is published with the permission of the Burroughs Corporation.

E. I. Organick
Salt Lake City, Utah

J. A. Hinds
Goleta, California

Chapter 1

Universal host computers

An important characteristic of conventional (von Neumann) computer systems is the *control mechanism*, or *processor*, which is designed to decode and execute a sequence of instructions fetched from storage (Figure 1.1). The processor generally has at least two groups of registers: one for control, and one for “processing information”. The first set of registers is mainly used for controlling the sequence of instructions in the program and for decoding each instruction so that it can be properly executed. The second set of registers, nearly but not totally unrelated to the first set, is used in carrying out the execution of decoded instructions. Generally speaking, execution involves fetching (or storing) data from (or to) storage, or examination and manipulation of data fetched from storage or produced by the execution of preceding instructions.

The picture of the computing machine given in Figure 1.1 is clearly incomplete, since it lacks a connection to the storage in the outside world. The input/output (i/o) controls and devices provide channels for information to flow from or to the computing machine and the “outside” storage which may consist of various media (tapes, disks, displays, printed paper, etc.) For the present discussion we shall ignore i/o transfers to outside storage.

The tasks of actually decoding and executing each instruction of the computing machine are *primitive*. The programmer normally cannot influence the manner in which these tasks are carried out. In all early computers these primitives were achieved by hardware circuitry. In many recent computer designs they are implemented as sequences of microsteps or microprograms which are themselves interpreted by hardware circuitry. By one means or another these microprograms are often made inaccessible to the programmer, so that interpretation of the instructions that a user programmer might compose remains primitive; i.e., he has no influence over the interpretation mechanism.

Although the programmer of a computer of this class may not vary the primitive behavior of such a computer, he may as an expedient compose a *simulator* (or *emulator*) program whose function is to interpret programs for other machines. The logic of the programmed interpreter is

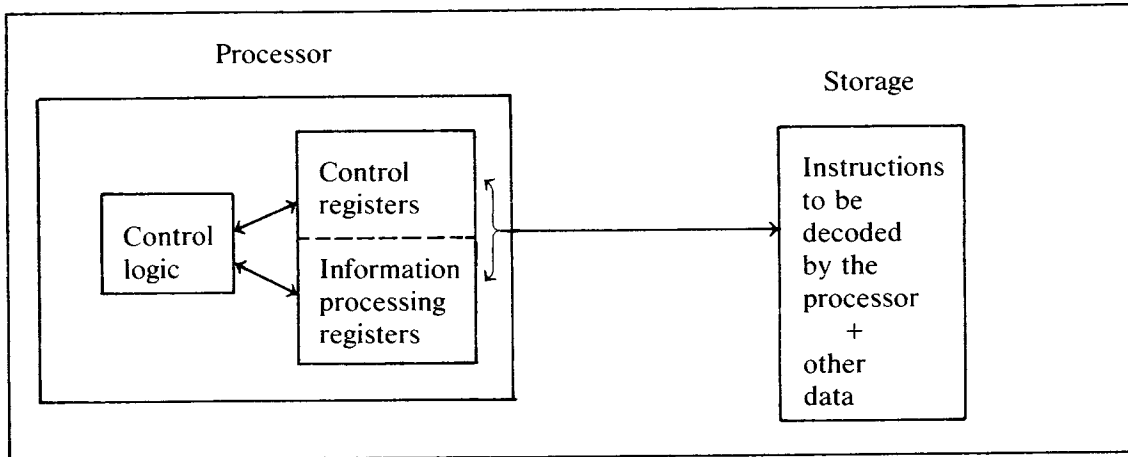


Figure 1.1. A view of a typical computer architecture.

completely under the control of the user. Not only can he vary the steps of the decoding mechanism, but he also can select whatever execution logic he chooses.

The user has a wide spectrum of redesign opportunities available. It may be that he wishes to simulate a machine that offers only a slightly different set of responses from that of the basic machine, e.g., augment its instruction set with a few more instructions, or alter the interpretation of the existing instructions. On the other end of the spectrum, he may have in mind the simulation of a machine having an entirely different set of instructions, with formats quite different from that of the "host" machine and having quite different semantics. For example, he may have in mind to emulate on a PDP-9 a PDP-15, a SAMOS machine,¹ or a FORTRAN machine. The first one (PDP-15) is just an extension of the PDP-9 itself (i.e., has only a few new instructions.) The second (SAMOS), though quite different in its semantics (having decimal arithmetic rather than binary) is roughly similar in the syntax and semantic power of its instructions to that of the PDP-9. Thus the formats of both SAMOS and PDP-9 instructions are fixed in length and have a small number of fixed subfields, both use index registers, etc. On the other hand, the instructions of FORTRAN have variable formats, a variable number of subfields, and a much greater range of semantic complexity than those of the PDP-9.

Figure 1.2 is a first view of a two-level host/guest system, consisting of a host, or H-machine, which functions as an interpreter of another computer system—G, for guest. Recursion in computer organization is

¹ A hypothetical computer used for instructional purposes in certain introductory computer science courses. (See Appendix F.)

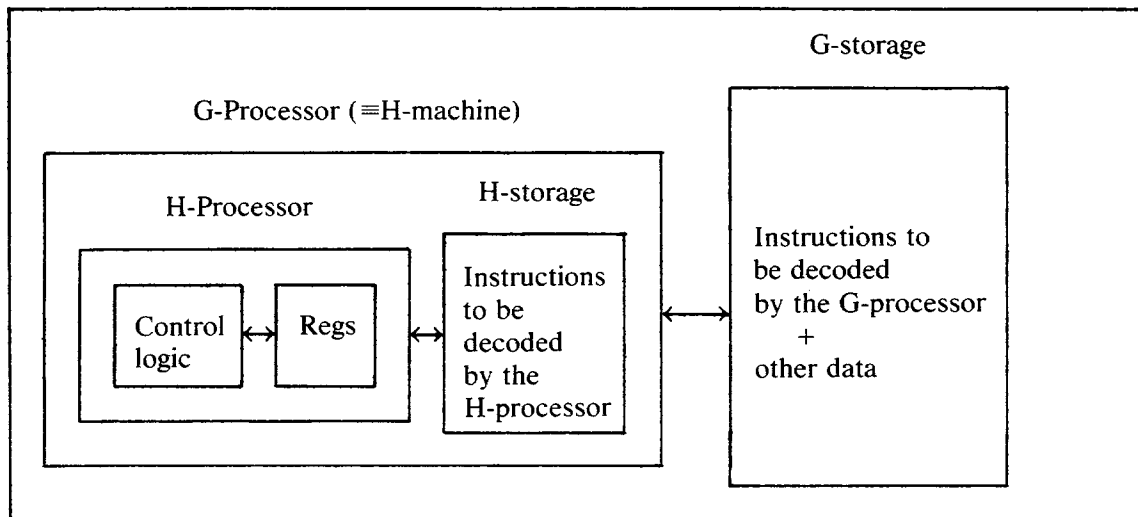


Figure 1.2. Structure of a two-level host/guest system.

clearly implied in this view.² Here we examine it from the inside out. The H or host processor consists of control logic and some storage (the registers). The H-machine consists of the H-processor and storage for its instructions (H-storage). But the H-machine in turn functions as a processor for another machine G, so the H-machine is in effect a G-processor. Adding “outside” storage for the H-machine forms a new machine (the G-machine). The outside storage for the G-machine is not actually shown in Figure 1.2, but its existence is implied (as was the outside storage for the machine depicted in Figure 1.1). In principle this recursion can be extended, since the G-machine might be designed to behave as a processor for some other machine G-G (guest of guest) and be coupled to storage containing programs for the G-G machine, etc.

There have always been practical trade-offs in building interpretive systems of this type. If the instruction set of the host machine and its registers is sufficiently different from that of its guest, the H-language subroutines which interpret G-language instructions may become long (and occupy a lot of H-machine storage). Also the time required to interpret a G-language instruction sequence on the H-machine may far exceed the time required to execute a “comparable” H-language instruction sequence executed on the same H-machine. Ratios of 10 to 100 for G-time/H-time are not uncommon. Even so, interpreters built to run on conventional computer systems are valued widely.

Since any machine may in principle be coded to behave as a host for any guest machine, it is also feasible that the same host may behave at

² The concept that a processor may be viewed as having a recursive structure was first brought to the authors’ attention by Robert S. Barton.

different times like the processor for any of a number of different guests. The backing store for an H-machine may contain interpreters for different guests. These interpreter programs may be swapped in and out of H-storage by some scheduling discipline, so that during discrete time slices the H-machine in fact acts like first one G-processor and then another. The duration of the time slices may be days, minutes, or seconds (or less), depending on the "swapping" technology that is used. Whatever the size of the time slice during which one of the interpreters is active, it should now be easy to accept the fact that any host may behave as a *universal* host, i.e., a host for a variety of guests.

Even so, few actual computer systems have been designed for applications in which they behave typically as hosts, much less as universal hosts for other machines. The B1700 class of computers, however, is one system which was indeed intended to behave mainly as a universal host. As we study it we shall hope to see in what ways its special features support such behavior.

The B1700 family of computer models, produced by the Burroughs Corporation, has been recently augmented with upgraded versions called B1800. In this book we will use the term "B1700" to refer to all members of this augmented class of computer systems except when we explicitly mention one member. For these systems the machine language of the host processor (H-language) is defined by the same base set of 16-bit microinstructions. Moreover, these systems have essentially the same internal logical structure, differing only in the mechanisms for accessing microinstructions. The B1700 has also been called an "interpretive definable field machine" because the programs and data executed by its interpreters are accessed from a storage that is viewed as an ordered set of *fields* (bit strings), each of *definable* length.

1.1 STRUCTURE OF STORAGE IN THE B1700 FAMILY OF COMPUTERS

To satisfy requirements of a universal host machine, the H-machine processor must have access to microprograms of many interpreters, one for each guest machine. One way to translate this requirement into an implementation is to imagine that the H-processor actually has access to several H-stores, each holding an interpreter for a different guest machine. Naturally, the processor must then be capable of switching from one H-store to another so that the system can multiprogram among several active interpreters. Storage technology and storage management techniques that have been developed over the past 15 years suggest several cost-effective ways by which such a system can be implemented.

Three related approaches have been taken in the B1700 family, one for each of three models within this family. These models are the B1710, B1720 and B1800.

The first approach (simplest, least expensive in hardware and slowest) is found in the B1710 model. Here (Figure 1.3) main storage is allocated into separate sections, some representing H-store and some representing G-store. The section representing H-store holds the microprograms that comprise the interpreter for a G-machine. The figure shows only one G-store and one H-store section represented, but in principle and in practice the main store is large enough to hold several of each.

Each H-store holds the microprograms that constitute the interpreter for a G-machine. The B1700 processor can be initialized to begin fetching and executing microinstructions from any H-store section of main storage using a G-store section as its workspace. At any given moment the B1700 processor knows about (has access to) only one H-store and one G-store representation in main storage. Switching interpreters implies resetting registers of the B1700 processor so it has access to a different H-store/G-store pair.

The B1800 model uses a similar principle for the representation of H- and G-stores in main storage, but is able to fetch microinstructions more rapidly through the use of a "cache memory" (Figure 1.4). The cache holds copies of blocks of microinstructions transferred from the main store as needed. The access to a microinstruction, when it is found in the cache (the usual case), is roughly an order of magnitude faster than the access to a microinstruction that must first be brought to the cache

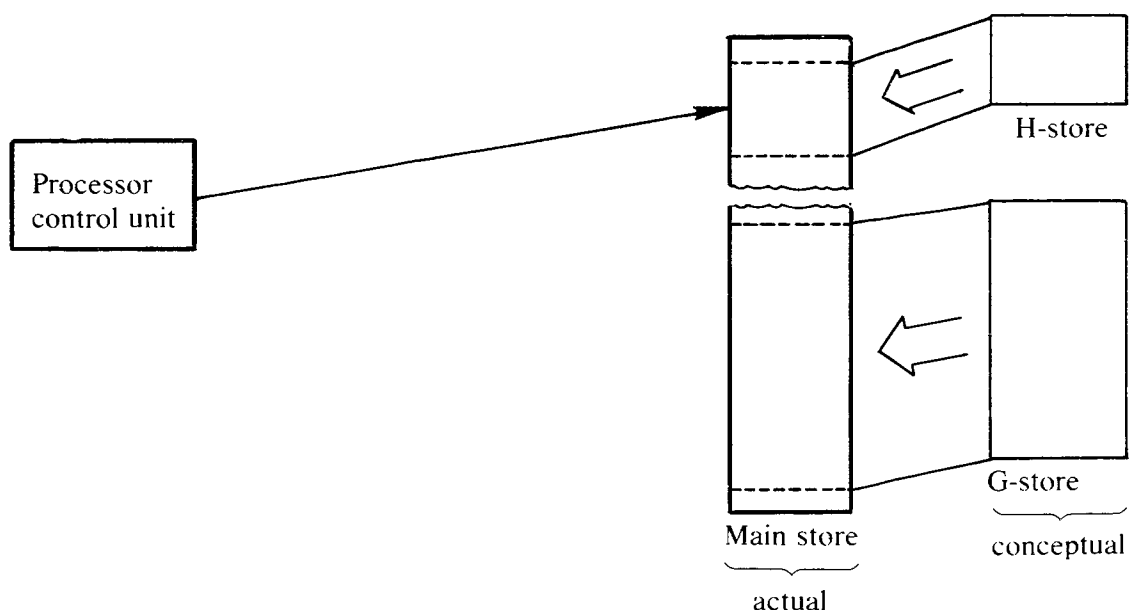


Figure 1.3. B1710 Processor access to H-store code.

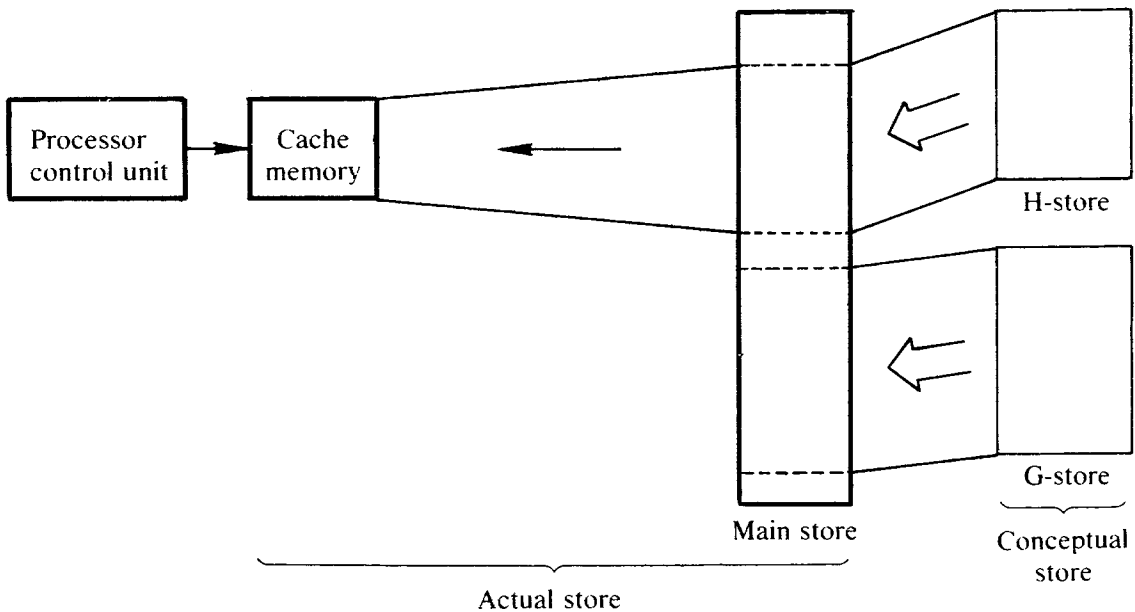


Figure 1.4. B1800 Processor access to H-store code.

from main store. The size of the cache is large enough to contain an entire interpreter, or at least that portion of it that is most frequently executed.

The B1720 model uses a less elegant but quite effective method for speeding up the fetching of microinstructions. A second storage unit, here called *fast control store*, is added to the system (Figure 1.5). This unit is large enough to hold the most frequently used portions of one or

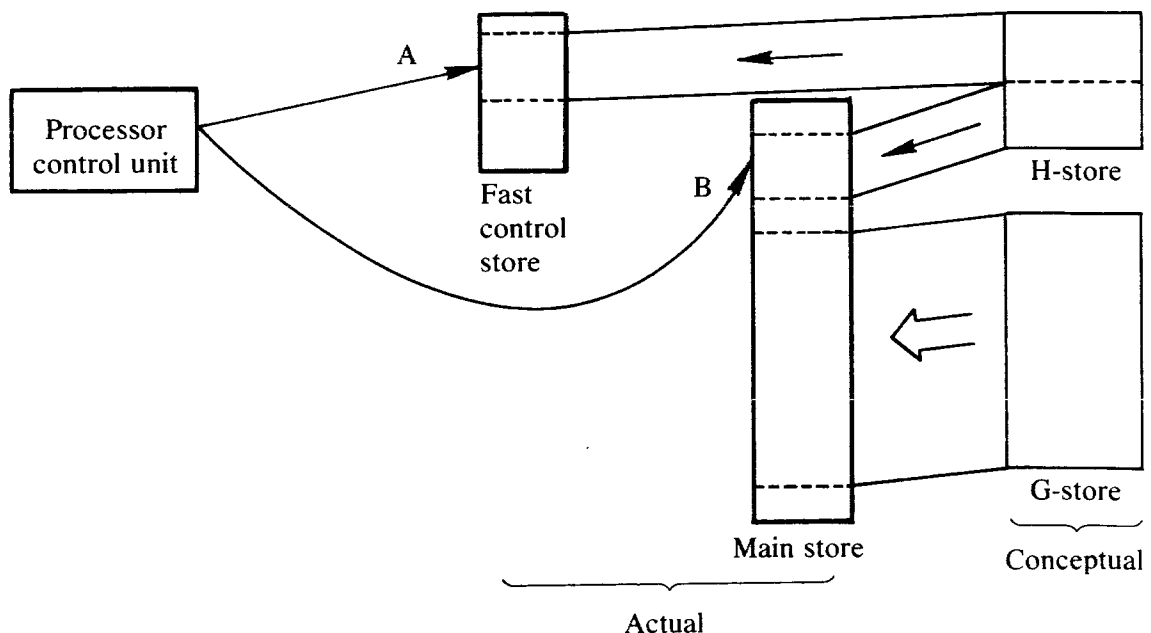


Figure 1.5. B1720 Processor access to H-store code.

more interpreters, space permitting. H-store is represented in part in the fast control store and in part in main store, depending on the size of the available fast control store. Extra base registers are provided in the B1720 processor for use in determining the access path needed to fetch the next microinstruction, a path leading either to the fast control store (path A) or to the main store (path B). Other things being equal, the B1720 and the B1800 degrade gracefully to B1710-like performance as the size of fast control store or the size of the cache, respectively, is reduced to zero. Chapter 7 of this book deals with these details.

Other differences exist between the B1710, B1720 and B1800 models than those just mentioned, but they are unimportant for the purposes of this book. Nevertheless, to avoid fuzziness, we shall always be as specific as possible about which model we are discussing. Because the authors' experience at the University of Utah has been primarily with the B1720 model, in particular the variant known as the B1726, this book will describe the B1726; but in so doing it also describes the related models to a very large extent. When we have occasion to discuss one of the other models, we will be careful to identify it.

1.2 THE B1726 MODEL OF STORAGE

We can now gain additional initial perspective by focusing on how storage in the B1726 achieves the effect of a universal host machine. A typical mainstore, which Burroughs refers to as S-memory (S for string), normally has a size of at least 64K bytes (2^{19} bits). The fast control store, which Burroughs refers to as M-memory (M for microinstruction) usually has a size in the range 4K to 8K bytes, enough to hold at least 2048 H-language, 16-bit microinstructions.

Let us first assume that the B1726 is busy executing programs for only one G-machine. [Later we will consider the more general case of two or more different G-machines as simultaneous guests on the host B1726.] And further, let us assume that the one G-machine interpreter needed consists of about 4096 microinstructions, or twice that of the available H-store. Then we expect that at some point in time the main-store S-memory will hold half of the G-machine interpreter. If there are more G-machine language programs active (i.e., being executed in multiprogramming mode), then storage will be needed for procedures of each program and for the data sets of each program. [If two or more programs shared certain procedures, duplicate copies of those (reentrant) procedures will not be needed. So the remainder of S-memory will be occupied by various procedures and data structures of the active programs of the guest machine.]

Any time the host machine needs to execute a microinstruction from H-store that is not in the M-memory, one of three approaches can be taken:

1. A block of microinstructions, including the ones currently needed, can be swapped in from S-memory, replacing a selected block of microcode now present.
2. So long as the microcode has the attributes of a pure procedure (read only), a simple overlaying strategy will also work, making swapping unnecessary. This also assumes that a backup copy of the entire interpreter is kept in S-memory.
3. Since the B1726 processor is so designed that individual microinstructions can also be fetched into the instruction register directly from S-memory (not just directly from M-memory), only the frequently needed microinstructions need be fetched from M-memory.

When blocks of microinstructions are needed in control store, approach 2 is used. (Approach 1 is never needed or used, since microcode is treated as pure procedure.) The B1726 executive system known as “MCP” (*Master Control Program*) also uses approach 3, since H-store microinstructions may be fetched directly from either M-memory or from S-memory.

To summarize, our conceptual G-store maps onto the physical storage called S-memory, and our conceptual H-store maps, to a first approximation, onto the physical storage called M-memory; but in actuality, since M-memory is a relatively scarce resource, H-store maps onto S-memory as well. It will be convenient and simpler to adopt the more ideal view, that of a one-to-one correspondence, which is H-store onto M-memory and G-store onto S-memory. We will take this simpler view in the next five chapters without loss of rigor. In the last chapter (Chapter 7), however, we will need to examine the details of the actual mapping between conceptual and actual host stores in the B1726 system.

To appreciate the motivation for the “two-level control store” of the B1726, it is important to observe the following.

1. Because the M-memory is regarded as a relatively scarce resource, the different interpreters being multiprogrammed can if necessary reside on and be executed entirely from S-memory. The operating system has responsibility for keeping track of which physical storage resources currently hold the interpreters, and is able to redistribute all or part of each interpreter among the two levels of storage as deemed appropriate.

2. If the operating system allocates the most frequently used portions of one or more interpreters to M-memory, there need be little need for frequent reallocation. This is because in general much more is known about the control structure and frequency of use of an interpreter and its parts than is known about the higher-level language programs that will be interpreted; hence it is possible to dedicate a portion of the M-memory resource to particular interpreters for relatively long periods of time with better effect than would be possible in a conventional system whose fastest storage is used for currently executing user or system code that is derived from compilers.

Chapter 2

The B1700 as an interpreting machine

The computer known as the B1700, or more precisely (in our laboratory) the B1726, is a system designed to make easy and as efficient as possible the interpreting of a wide variety of instruction sets. The machine language of the B1700 host is very low-level, resembling the microcode of other systems whose programmable machine language is at a higher level. A very low-level machine language is advantageous for programs that interpret instructions which are at the semantic level of conventional machine language or even higher-level languages. Although we shall refer to the machine language of the B1700 as microcode, we should take care to avoid the heretofore common connotation of microcode as something fixed (e.g., read only) and inaccessible to the computer user. In our case “microcode” is merely the manufacturer’s name for the machine language of the B1700.

The B1726 is a general-purpose computer. Like all such machines it may be programmed (in this case in microcode) to interpret another machine. There is, however, one major practical difference. Most general-purpose computers have instruction sets designed to go with a storage organization that is word- or byte-oriented. Bit strings fetched from storage are always taken in fixed chunks (words or bytes) aligned on chunk boundaries. Moreover, the length of the machine’s instruction is always made strictly compatible with the chunk sizes fetched from storage. Usually the length of an instruction is one chunk or a multiple thereof. We take it for granted, for example, that for the 18-bit word-organized storage of the PDP-9 there is a companion instruction set, every member of which is an 18-bit chunk. Instructions of the PDP-9 are fetched or stored in units of 18 bits on aligned 18-bit boundaries (i.e., $\equiv 0$ modulo 18.) Now the PDP-9 may not be a perfect computer, but without knowing more about its internal organization, we may assume that what it does best is done on chunks of 18 bits. Thus, if we were to use the PDP-9 to interpret instructions for a 17, 19, or 20 bit computer, we would expect to see a waste of storage as well as a distinct loss of efficiency in both the decoding and execution functions of the interpreter. What we have just said about the PDP-9 applies equally well to all such conventional word- or byte-organized systems.

Is the B1726 any different in this respect? Very definitely, yes, but the difference is somewhat subtle. On the one hand, its own machine language involves a set of microinstructions of fixed length (16 bits). However, within this repertoire are instructions which control the width and position of bit-string fetches (and stores) between storage and the processor. Thus, to fetch a string of 17 bits from bit addresses 19367 through 19383 takes no more and no fewer B1726 machine instructions than, say, fetching a string of 21 bits from bit addresses 8001 through 8021.

These controls for fetches and stores actually only regulate the flow of bit strings of from 1 to 24 bits in length. For transmission of chunks greater than 24 bits, the B1726 provides simple but powerful iteration controls in its machine-language repertoire. So although it takes more instructions (and more time) to fetch a 25-bit chunk than a 24-bit chunk, all chunks in the range 25 through 48 are in the same get/put class (instructions and time for a fetch or store), as are chunks in the range 49 through 72, 73 through 96, etc.

Apart from the crucial capability for defining fields of bits (chunks) and transmitting them from or to storage, the B1726 organization resembles in essence the familiar von Neumann architecture of a modern (e.g., 4th generation) sequential stored program digital computer. Because of its special orientation (and objective) as an interpreting machine, however, the structure of the processor, at first glance, appears to be more complicated than a conventional processor. Even so, it is easy to gain a simplifying view of this structure if one realizes that the processor performs only four types of activities; one can gain an integrated understanding by studying these activities one by one. There are interconnecting data and control paths between the registers used to implement each activity so a complete understanding of the processor can come only after *all* these interconnections and interrelationships are recognized. The four activities are

1. Data fetch and addressing
2. Data examination and manipulation
3. Decoding of higher-level language instructions
4. Control

We will look briefly at each of these.

1. There is a group of registers associated with the control and transmission of chunks to and from data storage. These include, for example, registers to hold the starting address of a bit field in storage, field length, etc., as well as registers to serve as receivers (from storage) or sources (to storage). Other registers, in a block

- known as the *scratchpad*, are useful for holding bit addresses and lengths for frequently referenced fields in storage.
2. There is a group of registers associated with the arithmetic and logical functions of the computer, so that the B1726 can be micro-coded to perform the conventional types of arithmetic and logic needed in everyday (simpler) computer and systems applications.
 3. A few registers are available for use as *local storage*. Some of these are endowed with special properties, very useful in the decoding phase of the interpreting process (e.g. shift and rotate, and in addition, extraction and testing of subfields).
 4. The machine has what amounts to an instruction or program line counter and an instruction register for controlling the sequence of microinstructions and for holding the microinstruction that is being decoded and executed by the *hardware*. In addition, there is a small stack whose main use is for holding return addresses for microprocedure calls (a control stack.) Address modification and other dynamic altering of B1726 microinstructions is made easy by utilizing a feature that ORs the operand of the preceding instruction with the next instruction. In many conventional machines this feature is achieved using index or base registers.

There are a number of explicit and implicit “connections” between the registers involved in the four classes of functions of the processor. Learning all these connections will take some time; the best way is by first studying some examples (case studies) of short microcode sequences. By tracing these one can incrementally accumulate an understanding of the whole process and be able to start writing B1700 microcode and/or “critiquing” microcode written by someone else.

The explicit connections referred to in the preceding paragraph are those spelled out in each microinstruction. For example, each MOVE microinstruction explicitly names the *source* and *sink* registers involved in the move operation. In essence, it is possible to move bit strings from any register to any other register. Of course, there are certain exceptions—for example, the microinstruction that causes a fetch from storage names the sink register explicitly, but does not name the registers holding the bit address or length of the source field in storage. These registers are always implied, as is (for example) the accumulator in the conventional SUB instruction of SAMOS.

2.1 INSTRUCTION DECODING

In the introduction we hinted that the B1726 instruction set and machine organization were designed especially to facilitate interpreting

of guest-language instructions. That's a broad statement. There are potentially an infinite number of guest languages with an infinite variety of instruction formats. It is surely not the case that the B1726 decodes with equal facility the instructions of every possible guest or G-machine. First note, however, that for most actual machines (potential G-machines) the instructions formats are *regular*. That is, they have such characteristics as fixed length and fixed fields, or a very small number of lengths and a small number of different fields and field lengths, according to the subclass within the repertoire. Based on this set of characteristics, SAMOS, for example, has a regular instruction format. So does the PDP-9 or even the IBM System 370. Regularity is popular in actual machines for minimizing the complexity of the interpreting hardware and/or micrologic so as to gain maximum speed or economy.

But it is certainly possible to imagine other G-machine languages where the instruction formats are, may, or should be highly irregular or *exotic*. If the G-machine has a phrase-structured language such as ALGOL (or any of the so-called higher-level languages), chances are the instruction format will be regarded as exotic in comparison with those of most everyday conventional computers.

In a well-designed interpreting machine the work of decoding should be roughly proportional to the complexity of the instruction format—and this appears to be true for the B1726 design. Whether regular or exotic, decoding is easiest on the B1726 if the operation code field is at or near the left end of the instruction. But fortunately, this is the case in nearly every machine design we have seen. Why is this so? Well, op-code fields are typically positioned at the left end of an instruction, with operand fields following, to conform with the customary functional notation of mathematics, e.g., $f(a,b)$ for a two-operand operator f . To fetch and decode such an instruction, two steps are necessary.

1. First we position the storage pointer to the storage address of the first bit in the instruction, and then read into a B1726 register as many bits as are needed to examine the entire op-code field. For a G-machine with 256 or fewer distinct binary op-codes (that covers most G-languages), the op-code field might then be no more than 8 bits in length.¹
2. In the B1726 there is a 24-bit special decoding register, known as the transform register T, which has been endowed with special logic. In

¹ Of course there are various ways of designing operation codes, which for the sake of efficiency, might lead to op-codes of various lengths, some less than 8 bits and some more, in this case. [But, for pure binary computers it is certainly unlikely that an op-code field greater than 24 bits would ever be required.]

particular, there are B1726 machine-language instructions which test bits of T and bit subfields of T, extract them, and move them to other registers. The B1726 possesses other features which enable a rapid jump (much like a FORTRAN computed GO TO) to the appropriate subroutine, where further analysis of the instruction or its execution can commence. For such jumps, the op-code extracted from the special decoding register T serves as index value for the jump (i.e., jump to *here* plus *index*.)

If the G-language is regular, and if the instructions are 24 bits or less in length, then the entire instruction can be analyzed directly from one loading of the T-register described above. If they are of regular format, but greater than 24 bits, then two or more successive loadings of T from storage may be required. After each loading of T, subfields can be extracted from T, analyzed, and held as necessary in other B1726 registers that serve as local or temporary storage.

For each new loading of T the storage pointer must be reset to the bit address of the next field in storage. There is a special *field address* register in the B1726 called FA which is used for the purpose of holding the storage address of the next 24-bit (or smaller) chunk. To go with the FA-register, there is an adder dedicated for the purpose of incrementing or decrementing FA. One can specify activation of this adder, for adding or subtracting a small constant (0 through 24), as part of the microinstruction that uses FA. For example,

READ 8 BITS TO T INC FA

specifies that T is to be loaded with an 8-bit field from G-store beginning at the address given by FA. Following the fetch from G-store, FA is to be incremented by 8. (The incrementation of FA overlaps the fetch of the next micro instruction from H-store.)

Another register called the *field length* register, FL, is provided in the B1726, and is also outfitted with a dedicated adder. The content of the FL-register is often used as an iteration counter for loop control during the transfer of long fields (>24 bits) to or from storage. If an instruction subfield is of variable length, the value in the FL-register can be preset to the (current) field length and then decremented and tested (against zero).

The specification for activating the FL's adder, like that for the FA's, is made part of the transfer microinstruction (READ or WRITE), e.g.,

WRITE 12 BITS TO T INC FA AND DEC FL, so there is no extra cost in B1726 machine time to carry out the address and counter arithmetic during each transit of the transfer loop. In this way, a series of variable-size subfields can be read from G-storage, the size being dependent on the analysis of preceding fields within the same G-language instruction.

Chapter 3

Organization of the B1726 microprocessor

In the preceding section we identified four kinds of activities performed by the B1726 microprocessor.¹ We now look at these one by one in more detail.

3.1 DATA FETCH AND ADDRESSING

Before an interpreter can decode a higher-level instruction, it must be fetched from the store that holds it. We have called that store the guest store, or G-store (although the Burroughs literature calls it “S-memory”). We assume that before interpretation begins, the G-language program has been loaded into one portion of G-store and a workspace has been allocated for data storage for the same program.

The left side of Figure 3.1 shows the G-store on a long “stick” to represent a bit string. A subfield to be used as the workspace is marked off by values in a pair of bounds registers called BR (*base register*) and LR (*limit register*). (The workspace is used for variables, constants, temporary storage, saved copies of registers during temporary interruption of the interpretation process, etc.). The bounds registers are used mainly to protect against accidentally writing into sections of G-store lying outside the workspace. The sections outside the workspace normally hold G-language instructions, i.e., code, system-manipulated input/output buffers, and workspaces for other computations that are being multiprogrammed with this one. We assume that the interpreter is provided with the size and location of the workspace and that the base and limit registers are set prior to interpreting the first G-language instruction.

The hardware logic of the microprocessor checks each G-store data

¹ We use the term microprocessor to mean *a processor of microinstructions* (i.e., as a shorthand for *microinstruction processor*). We do not intend to imply that the B1726 computer system is a tiny computer consisting of a few large-scale integrated (LSI) chips. A principal reference describing the processor is: Burroughs, “B1700 Systems Reference Manual”, Burroughs Corporation, Detroit, 1972, Form 1057165.

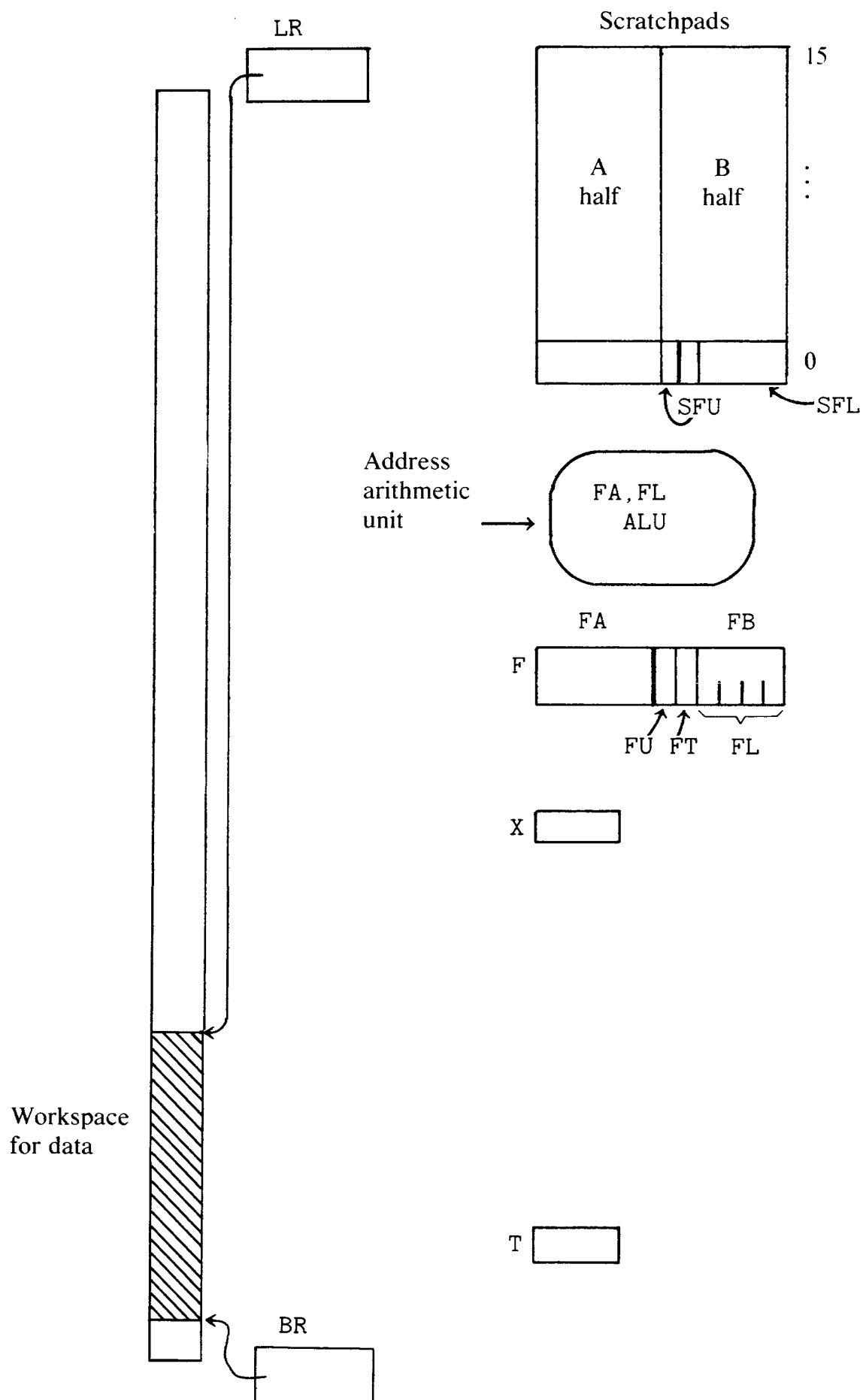


Figure 3.1. Data fetch and addressing.

address before using it against values of the LR and BR register. If the LR and BR registers are preset properly, and if they are not improperly reset during execution of the G-language program, the programmer can be assured that the process of interpretation will not damage system tables. But caution must be exercised, since there are no constraints in the machine language against assigning values to LR and BR. Having said all this about the protection role of LR and BR, we will for the most part now ignore these two registers, taking for granted that the microcoder who develops an interpreter will use these key registers properly.

We are now ready to see how inputs *from* G-store (reads) and outputs *to* G-store (writes) are executed. The read (or write) action is a hardware procedure that has three parameters.

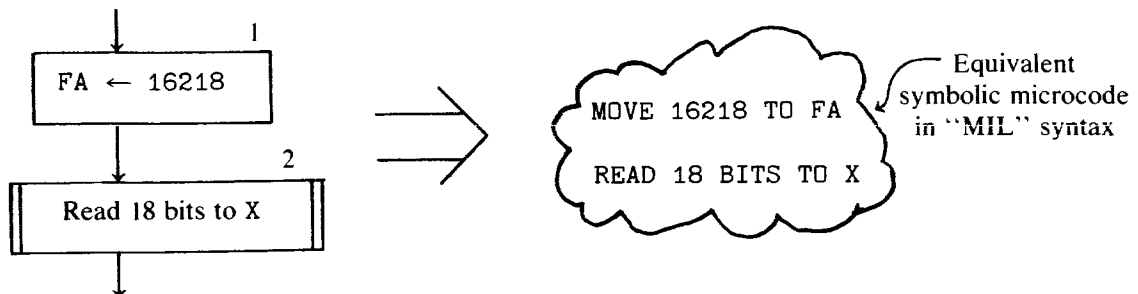
Bit address of the field in G-store

Length of that field

Register in the microprocessor that is to serve as *sink* (for a read) or *source* (for a write)

An argument value for the first parameter is (must be) always provided by presetting the FA register with the bit address of the beginning of the G-store field. [In Figure 3.1 we see that the F-register is a double-length register, the left half being the 24-bit FA register. G-stores of up to 2^{24} bits are possible in the B1726, so an absolute address is 24 bits long.] The second argument may be specified explicitly in the read (write) microinstruction, or by a default rule. The third argument is always specified explicitly. The registers X and T shown in Figure 3.1 are two of four registers that can be named as the sink (source) registers in a read (write) microinstruction.

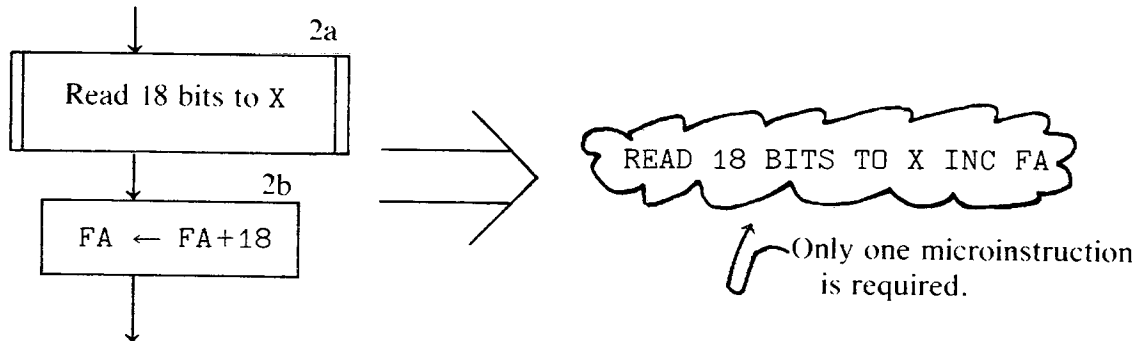
Example Suppose we are trying to read an 18-bit field located at bit address 16218 into the X-register of the microprocessor. The steps we want to execute are



What could be simpler? This sequence would cause the transfer of 18 bits at address FA to register X. The X-register is one of four 24-bit

registers (others are Y, T, and L) which may serve as a sink (or source) for reads (or writes) from (or to) G-store.

One READ (or WRITE) instruction is sufficient to transfer from 1 to 24 bits. [Fields of fewer than 24 bits are regarded as right-justified in the sink (or source). Zeros pad the left end of the sink register when the input field is less than 24 bits long.] How are fields of length greater than 24 bits to be transferred? Clearly, a microinstruction loop is needed such that upon each transit of the loop, up to 24 bits are shipped. Of course, FA must be properly incremented for use in succeeding reads or writes. A special arithmetic unit is provided in the B1726, shown in Figure 3.1. Using this facility we can make the READ (or WRITE) instructions specify incrementation (or, if we like, decrementation) of FA immediately following the transfer of bits from/to G-store. In particular,



Combining the incrementing of FA with the READ (or WRITE) in this way will cut down the number of instructions needed for loops involving repeated transfers. For example, the loop in Figure 3.2 shows how one might control the transfer of a sequence of ten 18-bit fields (or one 180-bit field taken as ten 18-bit chunks) into the microprocessor, where the starting address in G-store is 1600.

We already know how easy it is to map boxes 1 and 3 into B1726 symbolic microcode (or MIL,² for *micro implementation language*). It is not however, straightforward to map box 2 above into microcode, simply because on the B1726 there is no circuitry to perform a logical comparison on the value in FA. As mentioned at the end of Chapter 2, the B1726 designers have solved this problem in another, relatively convenient, way. They provided another register, FL, in the lower 16 bits of the right half of F to serve several purposes, including that of a loop counter. The address arithmetic unit will increment (or decrement) FL as well as FA, if such action is specified in a READ or WRITE microinstruction. Moreover, contents of FL *can* be compared with zero. A skip or GO TO based on this comparison can then be taken based on

² Appendix A of this book is an abridged reference manual for the MIL language.

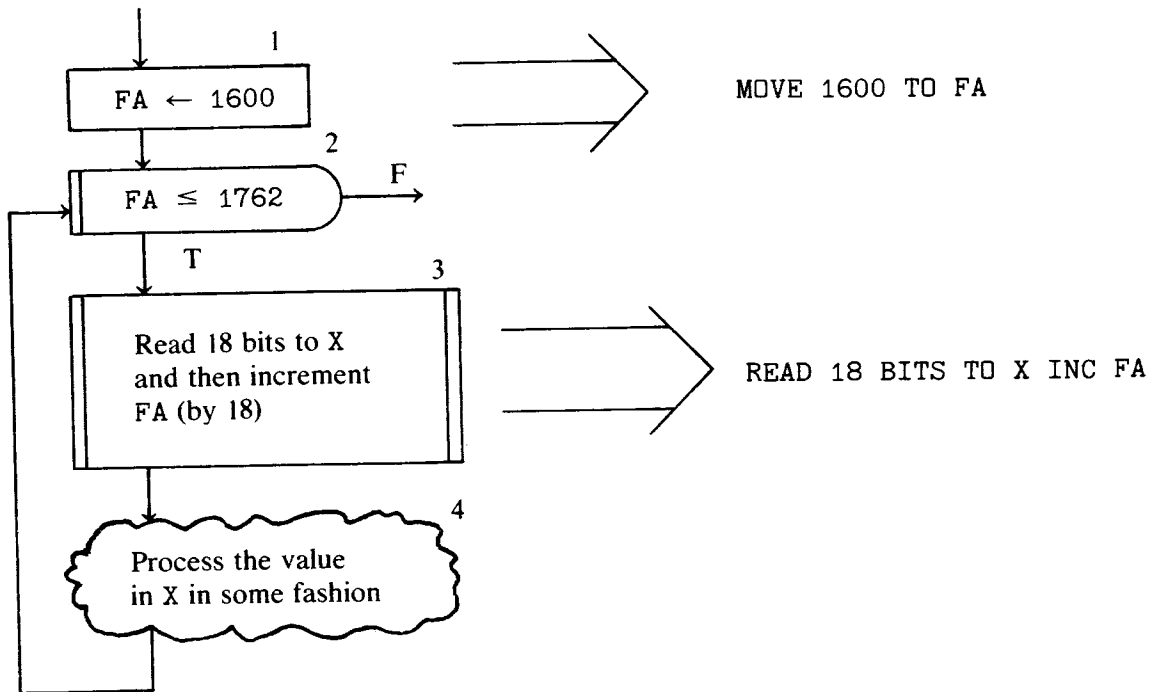


Figure 3.2. Controlling a loop for reading a sequence of ten 18-bit fields.

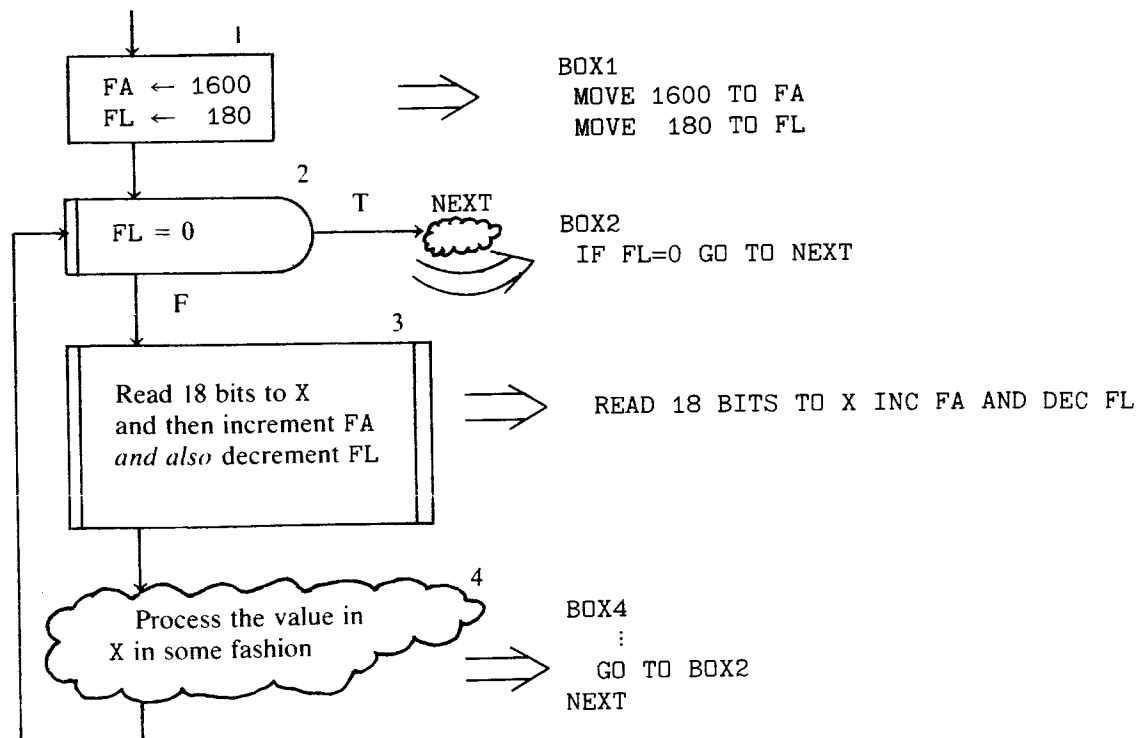


Figure 3.3. Input loop and equivalent MIL code.

this test. Figure 3.3 shows a modified flowchart for the above input loop showing the use of the FL register as a loop control counter, counting by 18.

The approach taken in Figure 3.3 is all well and good when the width of the field moved from (or to) G-store is a multiple of the chunk size transferred during one READ or WRITE. What about other cases? For example, suppose we wish to transfer a field of 2001 bits, up to 24 bits at a time—i.e., as a sequence of 83 chunks of 24 bits, followed by one 9-bit chunk—starting at bit address 22759. It will be most convenient if all READs can be performed by one instruction which is part of the loop, including the one that transfers the residue of 9 bits. Figure 3.4 shows a flowchart representation of this type of read loop. Here again the FL register serves as a loop-control counter, but it also has one other important role.

These two illustrations (Figures 3.3 and 3.4) show how the FL register gets its name, i.e., the *field-length* register. This register may be initially assigned the actual length of the long G-store field to be transferred.

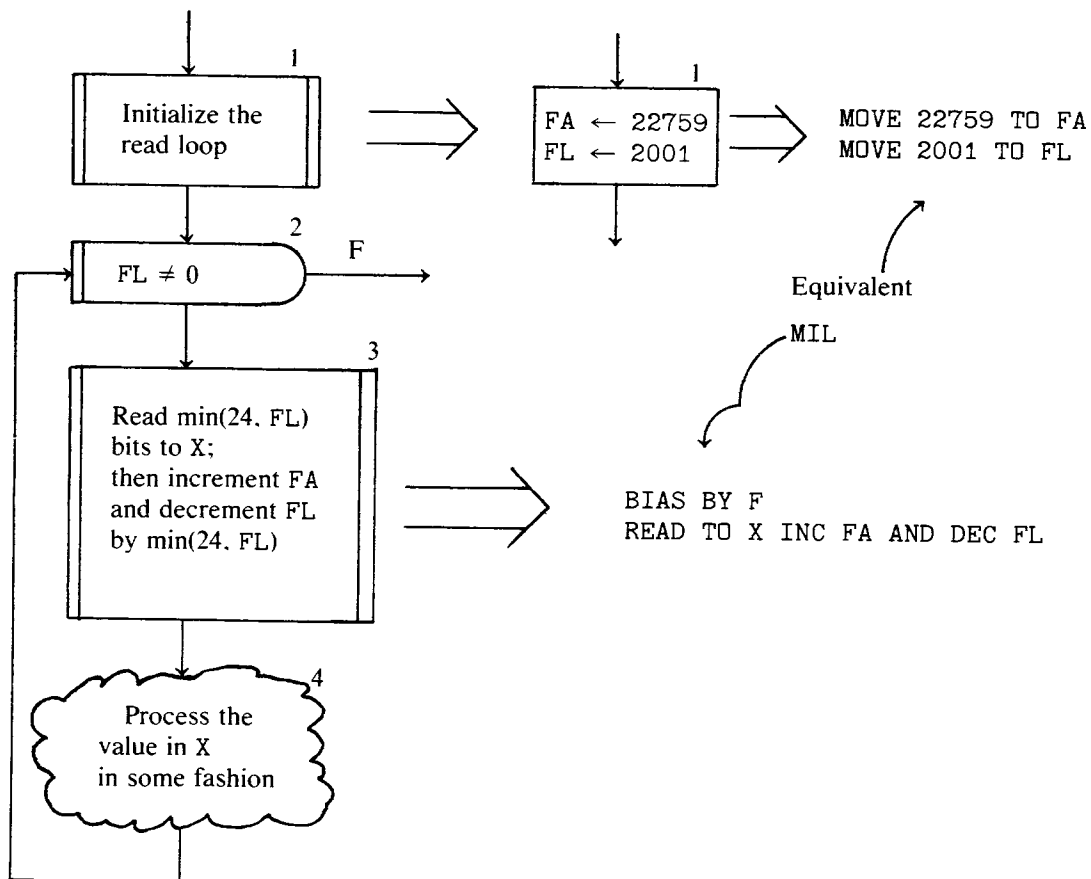


Figure 3.4. Read loop to transfer a field of length FL, CPL bits at a time, into the X-register.

Fields up to 2^{16} bits long may be accommodated in the B1726. [That's why FL is 16 bits wide.] We want to set the size of the chunk transferred by the READ instruction to the minimum of 24 and the current value of FL, which is to be decremented by 24 following each READ. When there is a residue chunk of 1 to 23 bits remaining to be transferred, it will be transferred on the last transit to the loop, the size of the residue being $\min(24, FL)$.

To see how this *residue control* is achieved in the equivalent B1726 operations, we must note two more facts about the semantics of the B1726 READ (and WRITE) microinstructions.

1. If the chunk size is not explicitly specified in the READ (or WRITE) instruction (or if it is explicitly specified as zero), a default value is chosen as the current value in the special length control or CPL register (not shown in Figure 3.1 but shown in Fig. 3.8 below).
2. The READ (or WRITE) instruction may be preconditioned by execution of a so-called BIAS instruction, which sets the default chunk size by assigning to CPL the minimum of 24 and the value specified in that BIAS instruction. For example, executing the microinstruction

BIAS BY F

prior to executing a READ with an *unspecified* chunk size amounts to

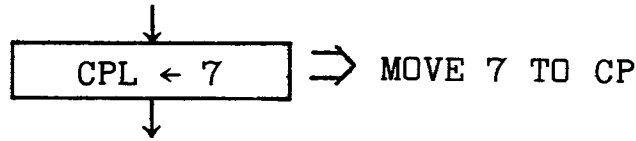
$$CPL \leftarrow \min(24, FL),$$

which is precisely what is wanted for handling the residue in our field-transfer algorithm of Figure 3.4. In that instance FL had the value 9 and CPL the value 24 when the BIAS instruction was about to be executed for the last time. Executing the BIAS instruction at that point gives CPL the new value 9, so the chunk size used in the succeeding READ is 9.

Here are two final notes regarding field transfers:

1. In the example of Figure 3.4 we imagined that we wanted to minimize the number of READs by transferring 24-bit subfields (the largest chunk that can be transferred at one time). We are, of course, free to specify chunks of less than 24 bits. For example, had we chosen to transfer subfields of 7 bits each, only two coding

changes are required. In box 1, we need to insert the instruction



In addition, the BIAS instruction of box 3 must be written as

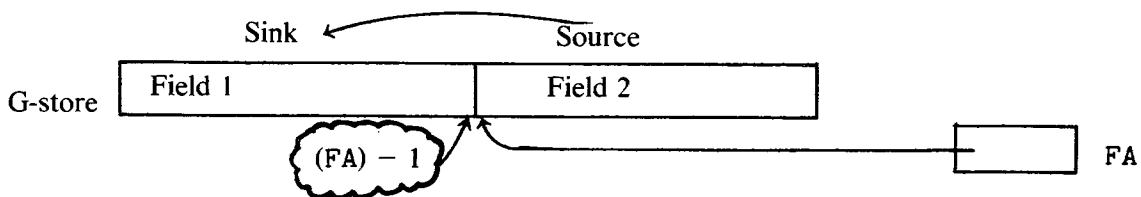
BIAS BY F AND CP

The semantics of this variant of the BIAS instruction is

$CPL \leftarrow \min(24, FL, CP)$

Executing this instruction before each READ (or WRITE) guarantees that CPL will be set to the minimum of 7 (the value first assigned to CPL) and any lower value that may be eventually assigned to FL. Each READ will now transfer a chunk of 7 bits, except possibly the last transfer, which may be less than 7 bits. Incrementing of FA and decrementing of FL will now be done by 7 instead of 24.

2. READ and WRITE microinstructions may proceed not only *forward*, (i.e., from low to high G-store bit addresses, which is the usual way) but also in *reverse* (i.e., from high to low G-store bit addresses). The READ REVERSE or WRITE REVERSE option is provided by the B1726 designers so the programmer can gain increased efficiency in certain types of transfer operations. For example, with FA set at a bit address as shown below,



one can READ TO X (forward) from field 2 of G-store, and later WRITE REVERSE FROM X into field 1 without having to use a different value of FA. Suppose, for example, that fields 1 and 2 are each 16 bits long, with the address of the leftmost bit of field 2 currently in FA. If field 2 contains the character string “BC”, then upon executing the sequence

READ 16 BITS TO X
 WRITE 16 BITS REVERSE FROM X

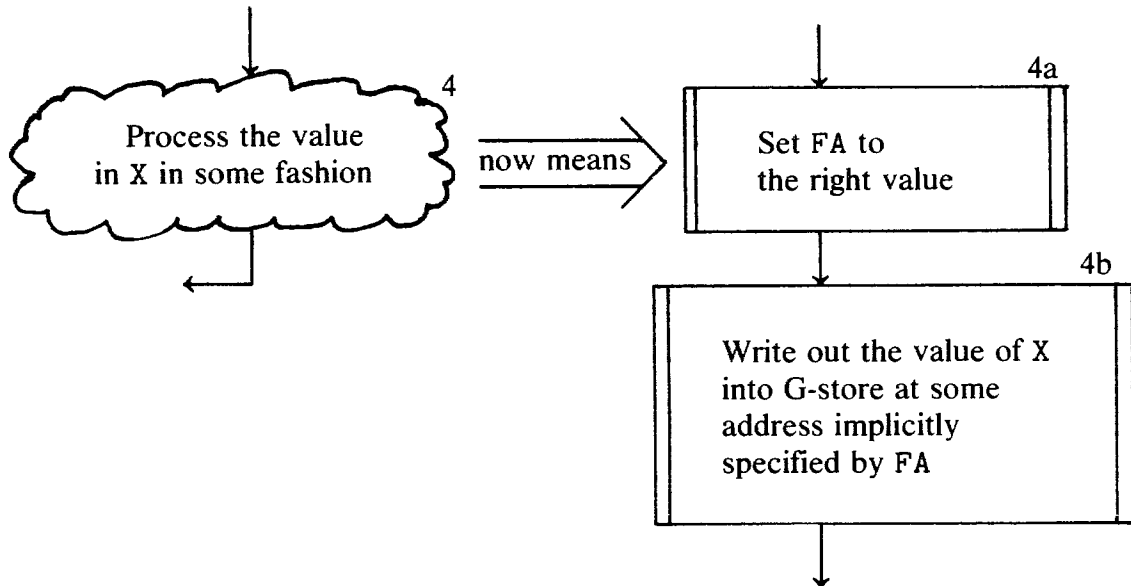
the value stored in field 1 is also “BC”. A REVERSE transfer does

not reverse the order of the information in the transferred copy. It only changes the significance of FA from the field beginning at address FA to the field ending at $FA - 1$.

We have now explained the functions of most of the registers shown in Figure 3.1 except the “scratchpads”. These are a set of 16 double-length registers which may be used as local storage for any purpose, but are especially convenient for the temporary storage of G-storage field *descriptors*, i.e. (address, length) pairs.

One obvious use of the scratchpads occurs to us if we consider the simple problem of copying a bit string from one part of G-store to another. For example, we shall consider the problem of copying a field of 2001 bits from a starting address of 22759 to a new field, starting at (say) 26721.

We saw in Figure 3.4 how to flowchart the task of moving this bit string from G-store into the microprocessor via register X. Now we want to take the chunks originally brought into X one by one, and write them out to G-store. Thus



But to achieve this objective, we need temporary storage to save and restore alternately the current values of FA for the source and sink fields on each transit of the loop. We can use scratchpad registers for this purpose, as suggested in Figure 3.5. S1A is used for temporary storage of the sink address, and SOA as temporary storage for the source address. The FL register is used as a loop control counter.

Another way to achieve the alternation of source and sink addresses, which saves two MIL instructions, takes advantage of the XCH microin-

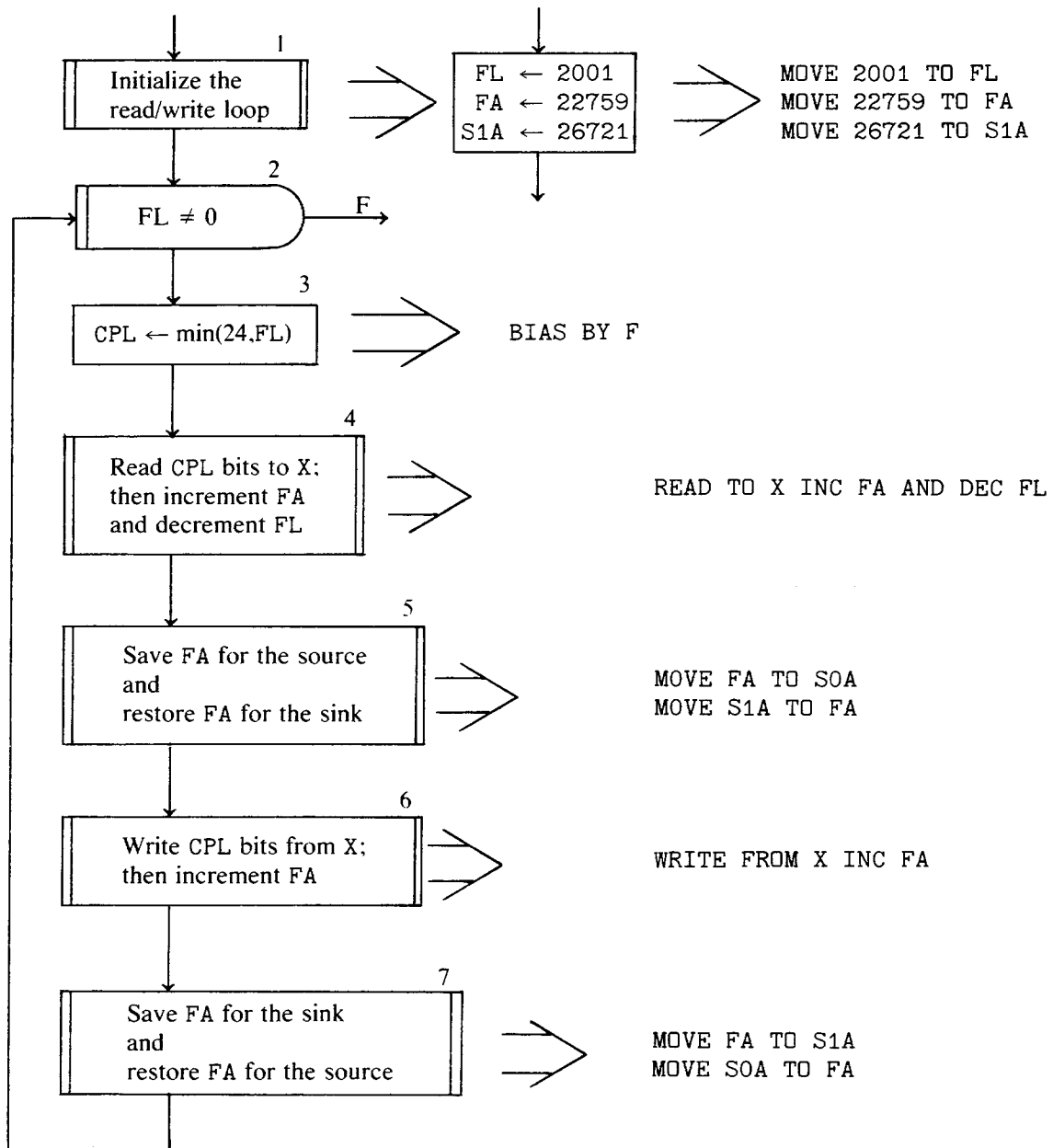


Figure 3.5. G-store-to-G-store copy loop for a 2001-bit field, using separate scratchpad registers for saving source and sink addresses.

struction to interchange the contents of any scratchpad register (all 48 bits) with F. [The general form of the XCH is

$$\text{XCH } \langle \text{scratch1} \rangle \text{ F } \langle \text{scratch2} \rangle,$$

which “simultaneously” moves a copy of F into scratch2 and a copy of scratch1 into F. In our special use of XCH in Figure 3.6, scratch1 and scratch2 are the same registers. We assume that the XCH operation uses

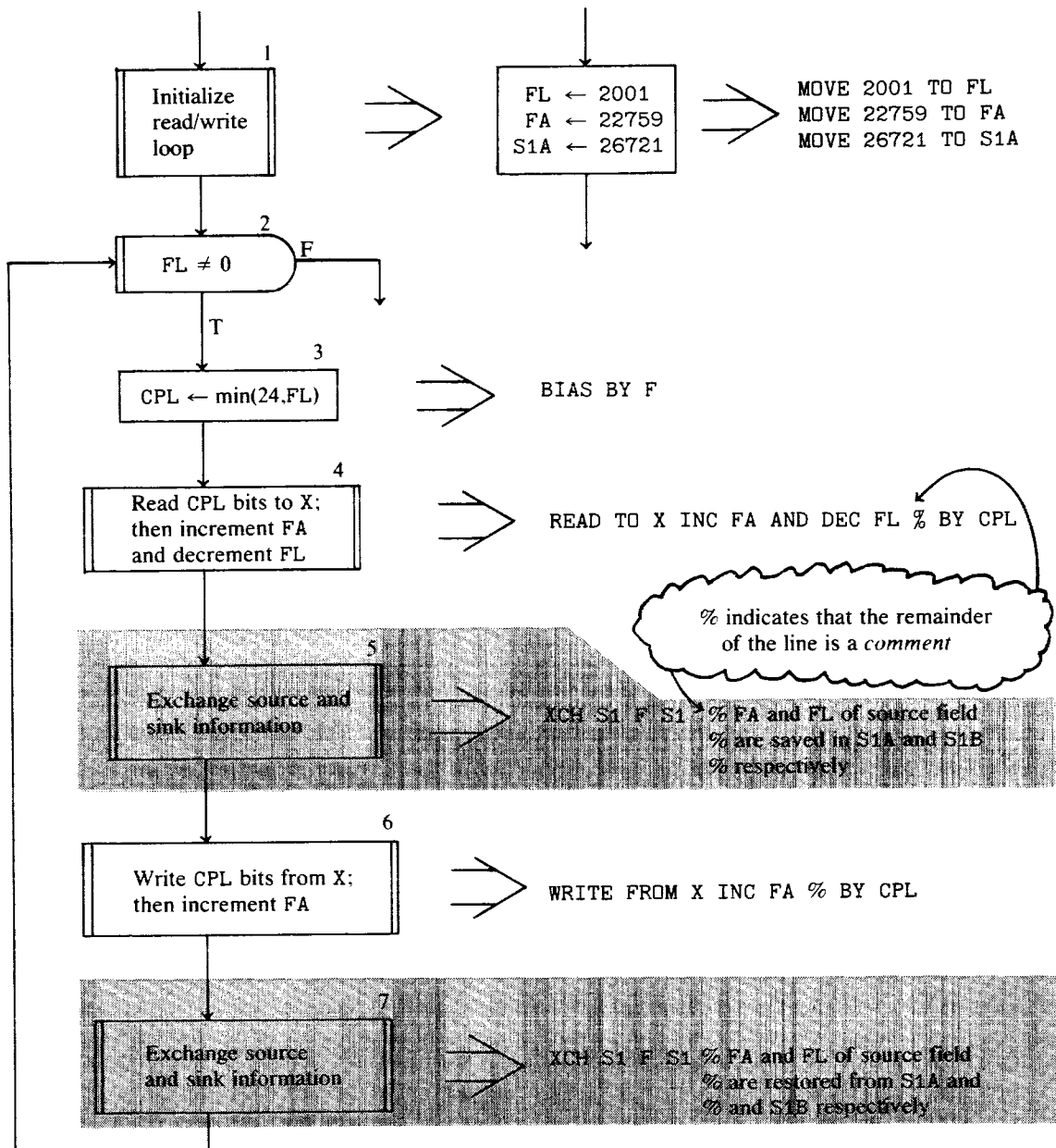
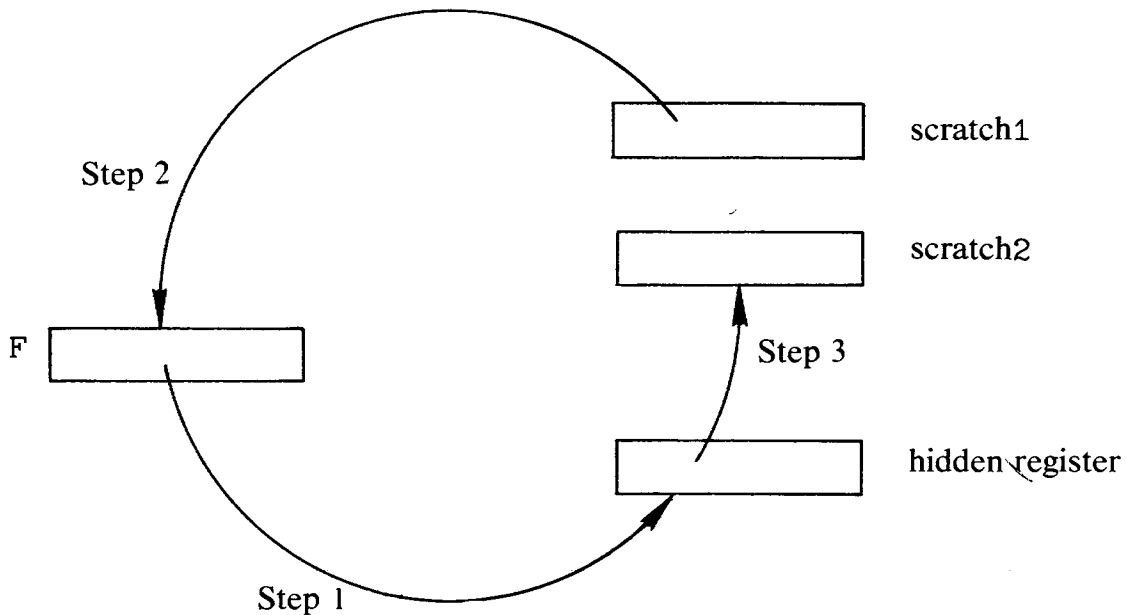


Figure 3.6. Same copy program as in Figure 3.5 except for coding of boxes 5 and 7, which uses the XCH instruction and scratchpads S1A and S1B. (This program uses S1B but not S0A.)

a hidden register to simulate the simultaneity inherent in the exchange



where step 1 must precede steps 2 and 3.]

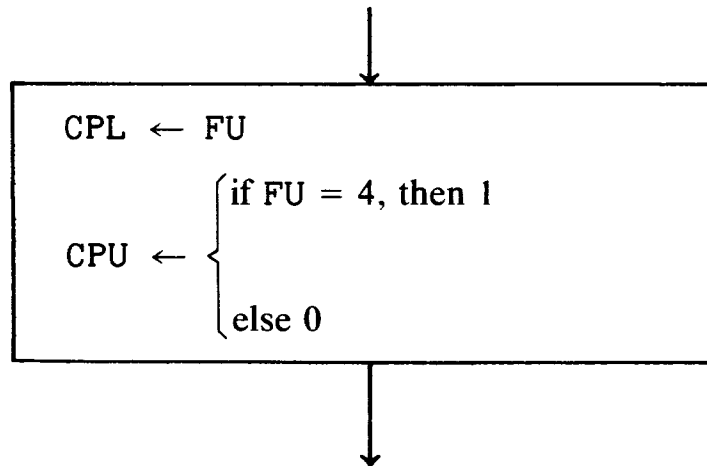
As the operations on the data flowing from G-store and back again become more elaborate, it will be increasingly convenient to hold at one time a variety of descriptors in the scratchpad registers. That is why 16 double registers hardly seems too many. [There are, however, some good arguments for not making the scratchpad too big. We shall discuss this issue in connection with the job of saving and restoring the state of a computation following an interrupt.]

At this point we have discussed all the registers shown in Figure 3.1 except the 16-bit register SFL (in the lower-order portion of SOB, the right half of SO) and several 4-bit registers, namely FU, FT, and SFU. The SFL register may be used as a limit value against which FL may be compared. That is, the hardware senses the relative magnitudes of FL and SFL (<, =, >) and sets a bit in one of several control registers to be discussed later. Testing that bit can be used as a basis for branching, i.e., skipping the next microinstruction.

The SFL field may also be hardware-sensed in a variant of the BIAS instruction. For example, BIAS BY F AND S means $CPL \leftarrow \min(24, FL, SFL)$. It might be used, for instance, when SO and F contain descriptors for two fields and the size of the next read or write chunk is to be based on the smaller of the length fields FL and SFL of the respective descriptors, thus avoiding destruction of G-store information when the source field is longer than the destination field.

The four-bit registers FU, FT, and SFU have no important hardware function for data transmission to and from G-store. [Actually, the

contents of FU may be sensed in a rarely used variant of the BIAS instruction, BIAS BY UNIT, in which UNIT is the key word that refers to FU. The contents of FU characterize the type of data (i.e., bit string, 4-bit decimal, 8-bit decimal). The meaning of BIAS BY UNIT is



This has the effect of setting the chunk size to that of FU. The significance of the CPU and the value assigned to it is secondary. We will discuss the significance of the special CPU register later when we discuss the so-called “24-bit function box”.] Data stored in FU and FT may be regarded as “addenda” to the address and length of a descriptor. In particular, when it is necessary to carry a *type* description for a field, such information may be held in FU and/or FT.

3.2 DATA EXAMINATION AND MANIPULATION

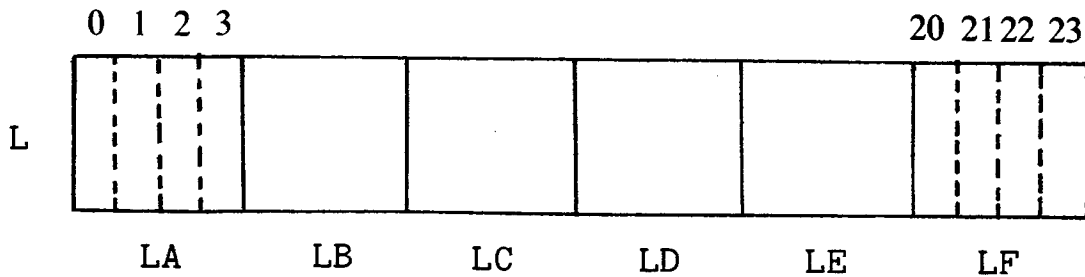
Once data have arrived from G-store into the H-processor, there are a wide variety of facilities for examining and processing them. As mentioned earlier, there are actually four registers, X, Y, L, and T, that may serve as receivers. Each is 24 bits long, and each has a set of distinct functional properties such that, depending on what one wants to do with the data arriving from G-store, one particular receiver register (X, Y, L, or T) may be more suitable than another. Of course, data can be moved (copied) from the receiver register to another one using the MOVE microinstruction. For example, if data has been read to T, it can then be moved (copied) to X:

```

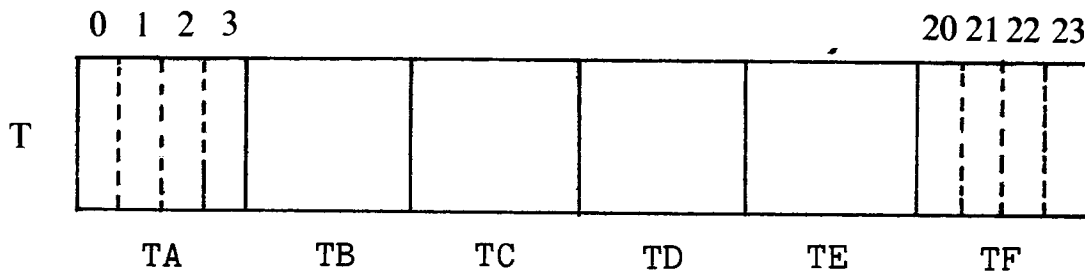
READ 15 BITS TO T
MOVE T TO X
  
```

The L and T registers are further subdivided into individually address-

able 4-bit subregisters, known as LA, LB, LC, LD, LE, and LF:

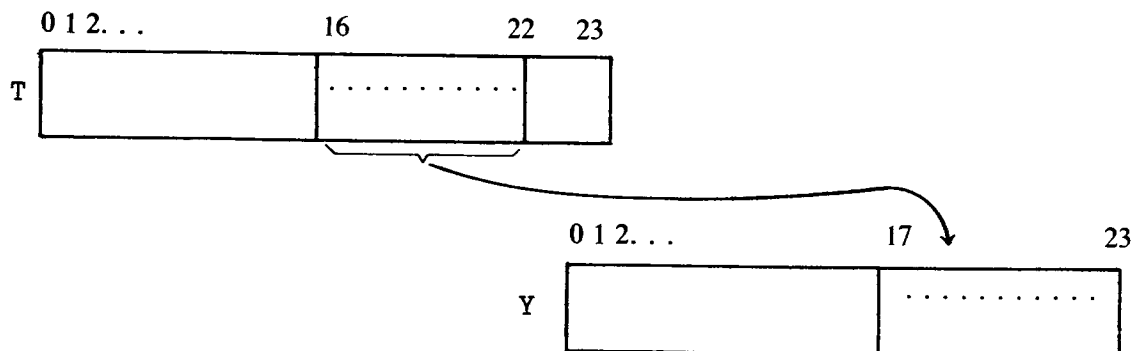


and TA, TB, TC, TD, TE, and TF:



That is, each of these subregisters may be mentioned by name in a microinstruction. For example, MOVE LC TO TD would cause the four bits in LC to be copied to the TD field of T. The individual bits of a four-bit register are addressable from left to right with subscripts 0, 1, 2, or 3 respectively. Thus LC(3) is the same as L(11).

The T-register has rather special (and powerful) transformational properties. Its contents may also be tested as a basis for conditional branching. Perhaps even more interesting is the fact that *any* subfield of T may be copied and moved to any receiver register (X, Y, L, or T), using the so-called EXTRACT microinstruction. For example, a copy of the seven-bit field in positions T₁₆ through T₂₂ can be assigned to the lower-order seven bits of Y (and Y padded with leading zeros), e.g.,



The (MIL) microinstruction to achieve this type of copy is

EXTRACT 7 BITS FROM T(16) TO Y.

Extraction in the direct sense just described cannot be performed on the other receiver registers. However, two of the registers, X and Y, may be shifted or rotated left or right to isolate a desired subfield.

Any bit within L or T, or pair of bits that are within any one four-bit subregister of L or T, may be tested (compared for equality against particular values) for use in selection steps. Some example selection steps, mapped into MIL code, are given in Figure 3.7. The illustrations

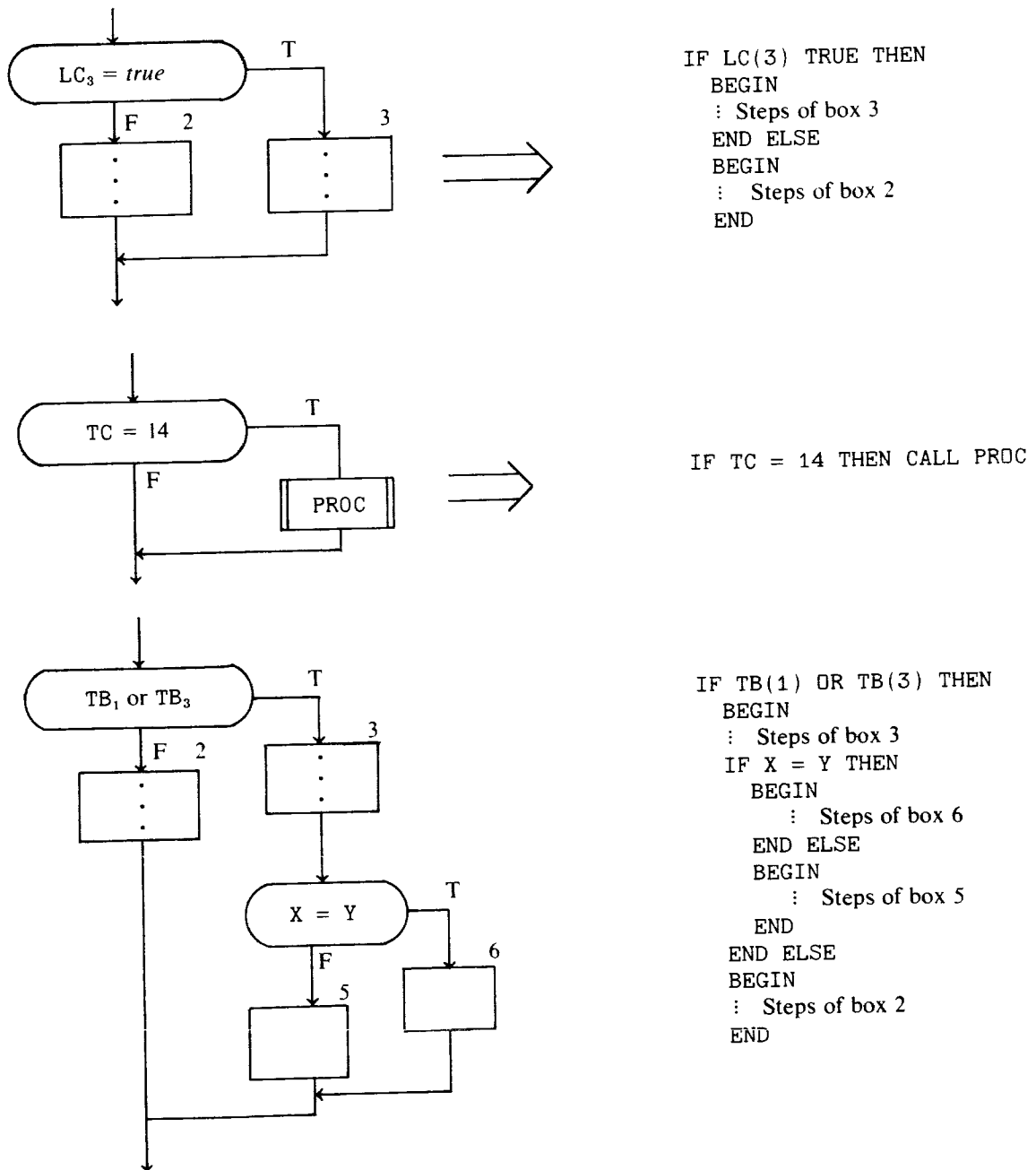
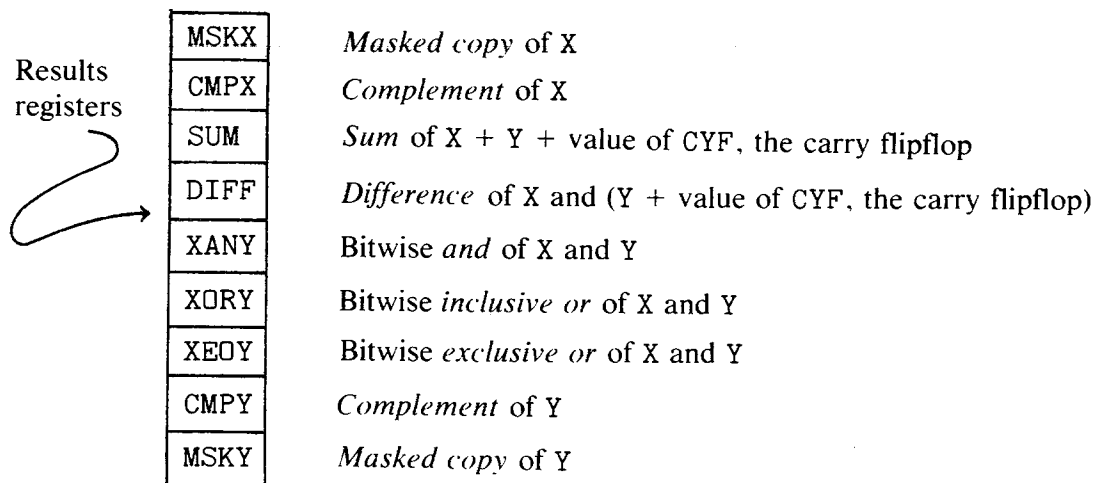


Figure 3.7. Examples of selection steps based on contents of bits or pairs of bits within a subregister of L or T, or based on a comparison of X and Y.

given in this figure are intended only to provide an idea how easy it is to express tests based on the contents of the receiver registers or on one or two bits within one subregister. The syntax of the MIL IF statement is detailed in Appendix A.

3.2.1 The arithmetic capability or “function box”

The two registers X and Y are inputs to a 24-bit *function box* whose combinational logic provides a variety of arithmetic and logical results as output. These results are available in a block of nine 24-bit “result” registers, one machine cycle (167 nanoseconds) after a new value is assigned to either X or Y. The registers, whose meaning is given below, are shown in Figure 3.8.



The length of each result is controlled by the value assigned to our old friend, the CPL register. A value of $m \leq 24$ in CPL allows m -bit results to appear in each result register. For example, if $CPL = 7$, then only the low-order 7 bits of the results appear in the result registers. [The high-order portion of the result register is padded with zeros.] Thus if X contains the binary number 100010_2 and Y contains the binary number 1000101_2 , and if $CPL = 4$, then MSKX (masked copy of X) contains the binary number 10_2 and MSKY contains 101_2 . At the same time, SUM contains 111_2 , XORY contains 111_2 , etc. All bits to the left of the fourth bit are then zero in each result register. So by controlling the value in CPL, the microprogrammer can generate results of any length up to 24 bits.

Carries for sums and borrows for differences are indicated in separate, one-bit result registers, CYL and CYD respectively. These carry-out registers may be tested as a basis of an (IF) selection step. Carry values may also be copied into the carry-in flipflop register CYF for input to the

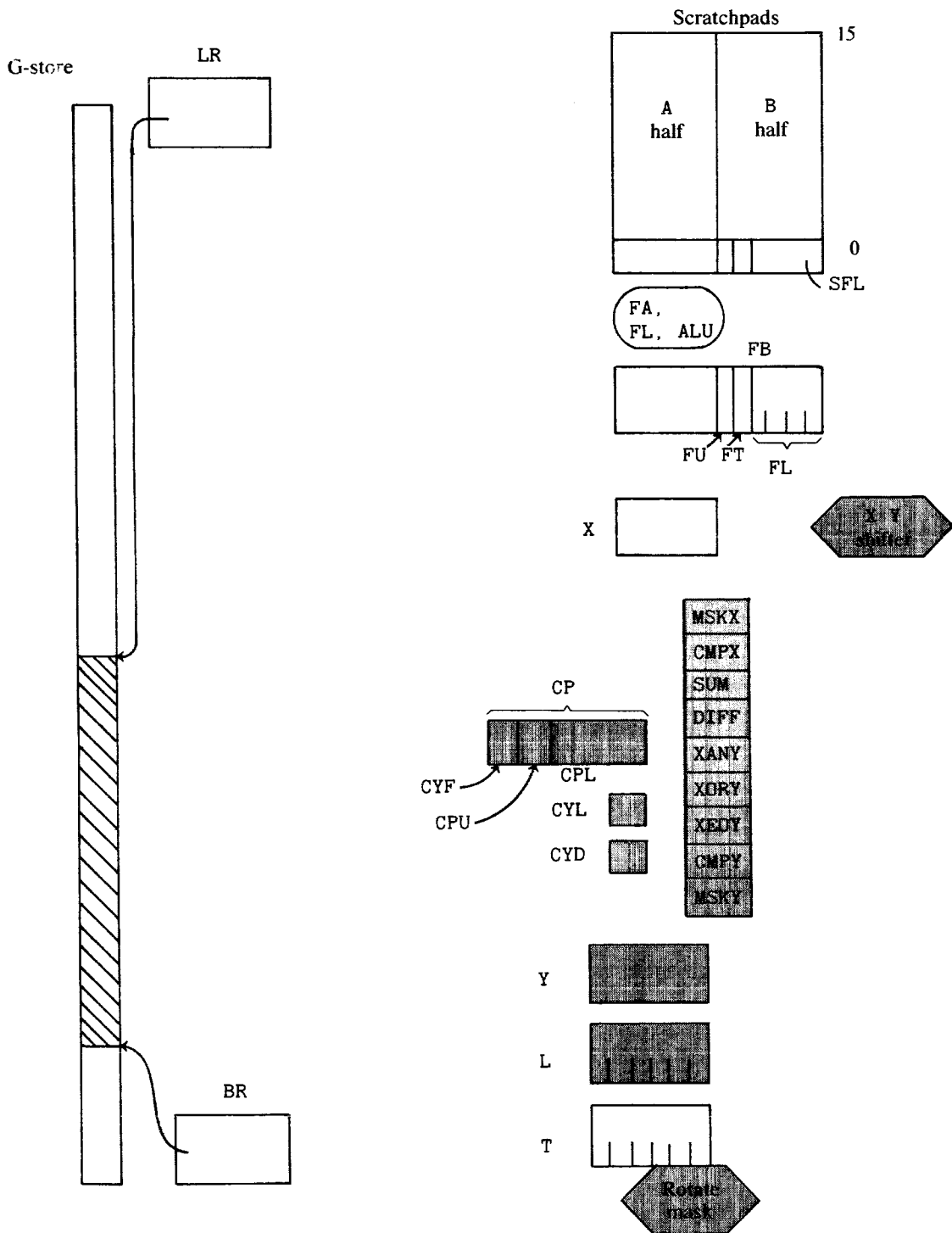


Figure 3.8. Additional registers for data examination and manipulation.

function box on a next sum or difference of X and Y, as might be required, say, in simulating a multiple-precision add or subtract.

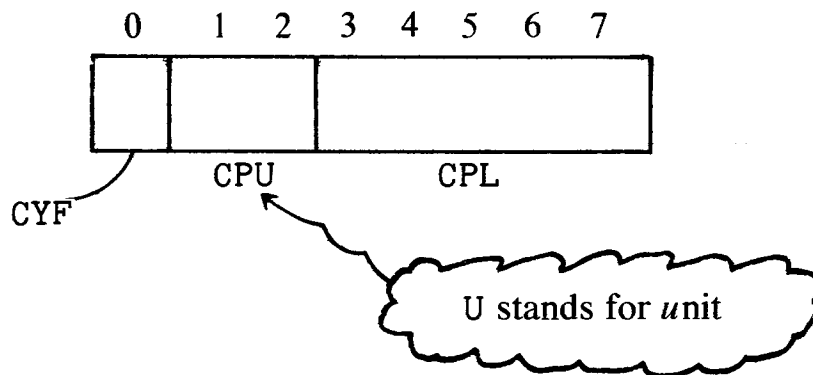
3.2.2 Arithmetic tidbits

To do a multiply or a divide, one is forced to use a microcoded subroutine built on repeated adds or repeated subtracts. There is, of

course, no floating-point arithmetic primitive in the B1726, so this too must be microprogrammed.

Adds and subtracts can be performed in binary or in 4-bit decimal. In 4-bit decimal arithmetic, each 4-bit subfield of the operands X and Y and of the results in the SUM and DIFF registers is binary-coded 0 through 9.

What determines whether X- and Y-registers will be regarded by the hardware as binary or as 4-bit decimal? There is a special 2-bit register called CPU, whose value controls this arithmetic interpretation of X and Y. The CPU is a companion or “cellmate” of the CPL register, both being housed, along with the carry-in flipflop CYF, in the 8-bit CP (control parallel) register.



The microprogrammer can, of course, also set or test the value in CPU and hence can cause the controls to switch back and forth from decimal to binary arithmetic for the results he wants.

Taking stock, the CP register plays an important role in control of arithmetic and (to a lesser extent) of logical operations resulting from the inputs X, Y, and CYF. The subregister CPL controls the length of each result, and the subregister CPU controls the unit (binary or 4-bit decimal) on which arithmetic will be performed. For an overview of the data examination and manipulation capabilities based on the receiver registers X, Y, L, and T, we have now said enough.

3.3 INSTRUCTION DECODING

At the risk of oversimplification, we can say that a machine instruction consists of an op-code followed by several operands (none, one, or more). If this is true for the machine that is to be interpreted by the B1726 host, then instruction decoding can be thought of as consisting of these steps:

1. Determine, by examination of the op-code field, which microcoded subroutine must be called to carry out the intent of that instruction. (We will call it the “operator subroutine”.)

2. Evaluate each operand on the basis of the information in each operand field.
3. Call the operator subroutine determined in step 1.

If step 2 has been executed before step 3, then the operator subroutine is in effect supplied arguments which are the results produced in step 2. However, if steps 2 and 3 are interchanged, then the operator subroutine must be “smart” enough to know how each operand is to be found and evaluated so as to execute the intended semantics of the operator.

In simple (i.e., regular) machines, not only is there a fixed (or at least small) number of operands for each instruction, but each operand has a simple interpretation. For example, in SAMOS each operand may be regarded as atomic (no substructure) and is a number representing a location in the SAMOS store. For such simple machines it is probably of small consequence (except for limited efficiency tradeoffs) whether or not step 2 precedes step 3.

When operands differ in description according to the statement types in which they appear, as in the case of higher-level (more exotic) machine like FORTRAN, there is greater justification for the interpreter designer to delegate to each operator subroutine the job of deciding how its associated operands are to be fetched or stored. Thus, each operand in a FORTRAN-like machine would probably have several components, such as its *type*, in what *table* (work space) the cell for this operand may be found, the *offset* within this table, and the *length* of the operand. For the remainder of this discussion we wish to keep things simple, so we'll assume we are dealing with a regular machine in which it is quite feasible to execute step 2 before step 3.

With this case in mind, and without loss of too much generality, we can further narrow the discussion to the case of a one-address machine like SAMOS. For such a machine there are typically, at most, three fields following the op-code field, e.g., operand, index register indicator, and possibly an indirect address indicator. The order is immaterial. Since each field is in a fixed position within the instruction, it is a relatively easy matter to write microinstructions which fetch the “next” instruction from G-store and *extract* each of its fields. The T-register is ideally suited for this. If the instruction is greater than 24 bits in width, several fetches will be needed, and more of the “local registers” such as L or even X and Y might be used. As each operand is isolated it can be evaluated and its value saved in an agreed-upon scratchpad register.

What do we mean by evaluation of an operand? Take the case where we are simulating a machine M whose storage consists of r cells, each s bits wide. We assume that the storage for M ($r \times s$ bits) will have been allocated in G-store beginning at some absolute address, represented (say) by the symbol B . Then evaluating an operand whose value is (say)

v amounts to a mapping from v to an absolute address in G-store. In this case it is

$$\text{map}(v) = B + s \times v$$

where $0 \leq v \leq r$. Of course, the word width s is a constant for machine M . Moreover, B is fixed for a particular interpretation of a program for M . B is based on the contents of the B1726 base register BR.³ [It may be useful to let the value of r be a parameter of the interpreter so that the storage size of M can be specified anew on each simulation.]

The sequence of microinstructions which performs this mapping must therefore know where to find B , probably s , and r . Very likely B will be found in an agreed-upon scratchpad register.

Must address arithmetic of the type required in computing $\text{map}(v)$ be performed entirely by the function box? If so, there could be some congestion-induced overhead in the use of registers X and Y . If one or both of these registers hold data values which are to be written out to G-store at an address which is about to be computed in the function box, then data values in X and Y will have to be moved and may have to be given a "round-trip ride" to and from some available scratchpad while the function box is put to use computing the target address—e.g., as shown in Figure 3.9.

To avoid a lot of this overhead the designers of the B1700 attached a full 24-bit adder to the FA register, making it possible to do the most frequently needed address arithmetic (computing offsets by addition and/or subtraction) outside the function box. One can add to or subtract from FA the value of any of the 16 left half scratchpads' registers (S_iA , $i = 0, 1, \dots, 15$). Thus in the instruction sequences

```
MOVE BR   TO FA           % MOVE BASE FROM BR TO FA
ADD  S7A  TO FA
MOVE FA   TO S9A
```

or

```
MOVE S10A TO FA           % MOVE BASE FROM S10A TO FA
SUBTRACT S11A FROM FA
MOVE FA   TO S15A
```

the FA register plays the role of a conventional accumulator (with addend or subtrahend addressed from the left half of the scratchpad and with augend or minuend from any register). We see that if address computation only requires computing an offset from some base, then the

³ We assume that BR is fixed in value during the interpretation process, but in a multiprogramming environment it may be that a program's data space may be relocated before its execution is completed, so in fact BR is not strictly a constant.

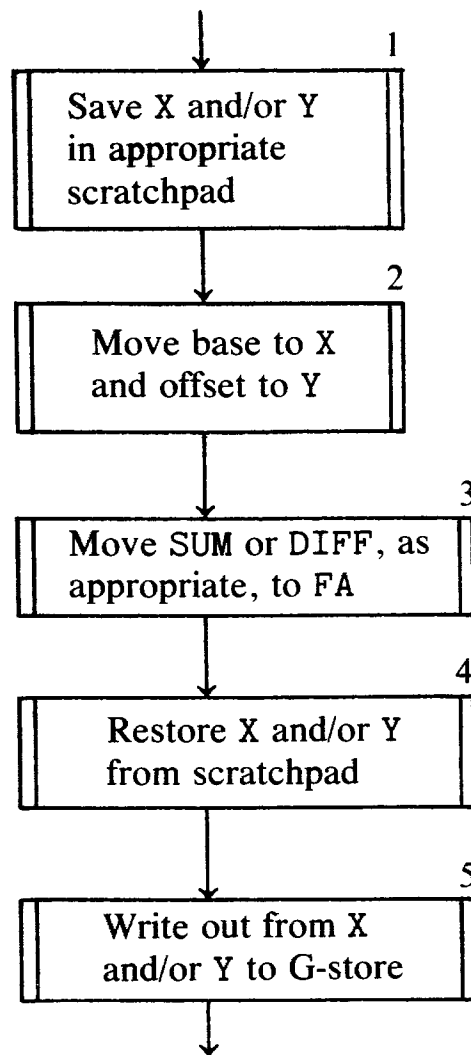
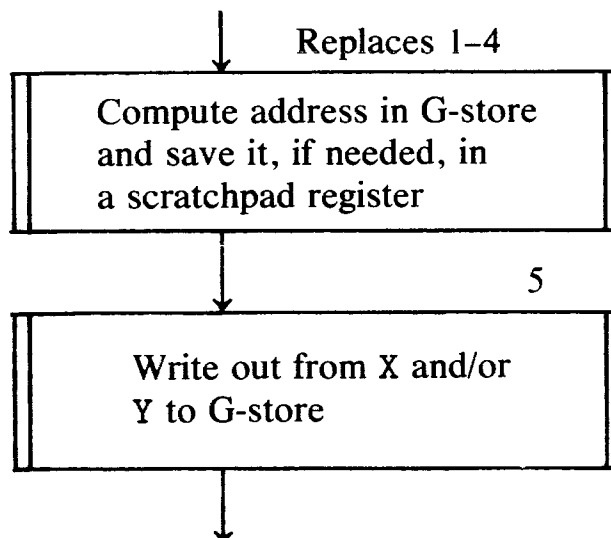


Figure 3.9.

wasteful sequence of 5 steps in Figure 3.9 reduces to



We now suppose that, having computed $\text{map}(v)$, this value is stored in another agreed-upon register. The values of other subfields that define the operand, such as the index register indicator, can also be stored in agreed slots. The operator subroutine may then be called and it will know, by convention, where to find the values of its arguments, for instance in scratchpad registers S3, S4, and S5. The mechanics of calling the appropriate operator subroutine and returning from it will be discussed in the next section.

3.4 CONTROL

We are now ready to examine more closely the actual control structure of the B1726 microprocessor. Code executed by the B1726 consists of sequences of 16-bit microinstructions. [The machine language of the B1726 is itself a somewhat regular one.] Microcode is regarded as *invariant*, i.e., it shouldn't be altered as a result of being used (interpreted).

To the extent possible, microcode is kept in a separate program store (H-store), whose mode of access is primarily read-only, and where microinstructions are relatively well protected against being accidentally clobbered. Of course the program store has to be loaded periodically with new batches of microcode, so the program store (usually referred to in the literature as *control store*) must also be (and is) writeable. [In fact, there are several ways to write into control store. Use of the special OVERLAY instruction causes code to be shipped from G-store to H-store, and it is also possible to load microcode into H-store from a console cassette tape or from the console switches.] The control store on the University of Utah B1726 has, for example, a capacity of 2048 microinstructions (4096 bytes).

The ordinary cycle for interpreting a microinstruction begins with a fetch from H-store from the location pointed to by the A-register (Figure 3.10), which is the program counter for the hardware. The fetched microinstruction goes to the M-register, whence it is decoded by the hardware for execution.

How does the OR box attached to M help us? The actual operation of the hardware in the execution of a B1700 instruction is

1. The microinstruction at location (A) is ORed into M.
2. A is incremented by 16.
3. Instruction in the M-register is decoded, and gates are set appropriately for execution.
4. M is cleared (preparatory for next instruction).
5. The instruction is executed.

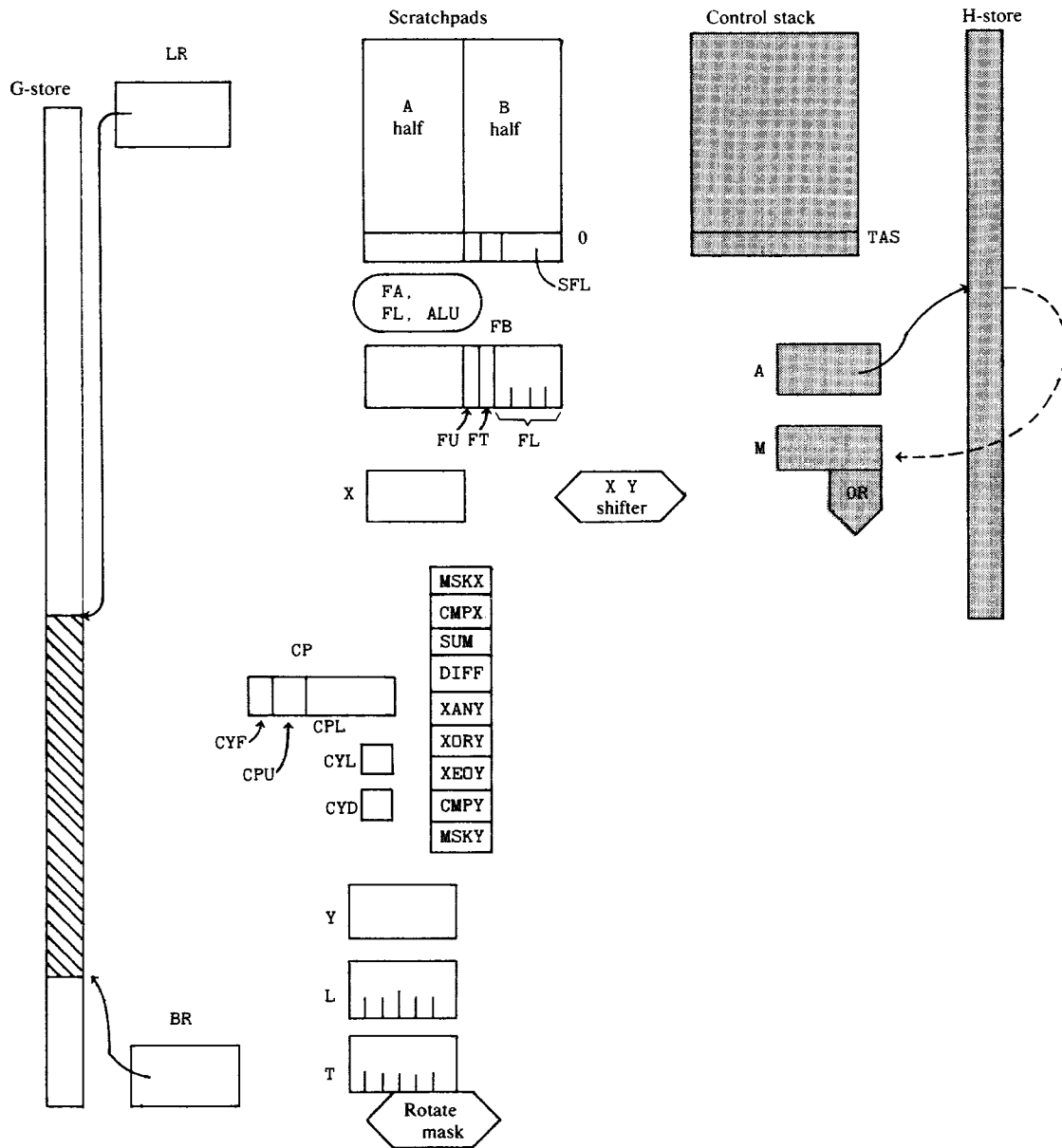


Figure 3.10. Additional registers for control, i.e., interpretation of microinstructions.

Of course, before the *very first instruction* of a program is executed, the M register must be cleared. This always happens during the initial startup procedure. Suppose that some instruction caused data to be moved to the M register—for example, MOVE 3 TO M. During step 5 the value specified (3 in this case) would be moved to M and would then “participate” in step 1 of the next instruction cycle. Hence the value moved to M can control (modify) the effect of the next instruction. Note several things about this modification.

1. Logical ORing of all 16 bits allows modification of any field of a B1700 operation.

2. The effect lasts only for one instruction execution.
3. The actual H-store is not changed in any way.

Example Suppose scratchpad registers S12A, S13A, S14A, and S15A contain a list of 4 possible offsets to a base address of a list in G-store. Further, assume we know that TA has a value in the range 0 through 3 inclusive, corresponding to one of the offsets found in S12, S13, S14, or S15. We would like to use the scratchpad ADD instruction to add the offset from the desired scratchpad into FA. The following sequence of microinstructions accomplishes this objective.

```

MOVE TA TO M
ADD S12A TO FA
:
```

The scratchpad ADD instruction names S12A as the source, but the address field designating S12A will be ORed with a copy of TA previously moved to M, producing an “effective address” that designates the desired scratchpad register: S12A for TA=0, S13A for TA=1, S14A for TA=2, or S15A for TA=3.

[*Warning:* We must be careful in the use of the ORing feature, or we may get unexpected results. For example, what is the effect of MOVE TA TO M in the following sequence when TA is again assumed to have a value in the range 0 through 3?

```

MOVE TA TO M
ADD S11A TO FA
:
```

[*Answer:* No effect. Why?]

Another important application of the ORing property of M is in achieving a multiway branch, using a “jump table”. For example, suppose we wish to decode a 3-bit op-code to achieve an 8-way branch to one of 8 operator subroutines. Assume that 3-bit op-code is located in bit positions 5 through 7 of the T-register. Then the following instruction sequence would do the trick.

```

EXTRACT 3 BITS FROM T (5) TO L
MOVE L TO M
JUMP FORWARD
GO TO ROUTINE1
GO TO ROUTINE2
:
GO TO ROUTINE7
GO TO ROUTINE8
```

Explanation First we take advantage of the EXTRACT instruction to

pull out of T a copy of the 3-bit field and move it to another receiver register L. The value moved to L will be padded with high-order zeros. This value is then moved to M.

The JUMP is an unconditional branch whose operand is a signed displacement, e.g., JUMP to “here”+5 or JUMP to “here”-10, where “here” is the current contents of the A-register, now pointing at the *next* instruction. The special JUMP FORWARD option means JUMP to “here”+0. But we have ORed into this instruction an unsigned integer in the range 0 through 7, so we will have an *effective* JUMP to one of the eight succeeding instructions, each being a GO TO to a different operator subroutine.

Note that in no case where we take advantage of the ORing feature of the M-register, do we in any way alter the instruction residing in H-store. This ORing feature permits, in a limited way, the instruction modification capabilities made possible in more conventional machines using *index registers*.

Besides indexed JUMPs, of course, we can have arbitrary jumps limited only to displacements (+ or -) of no greater than 4096. In the MIL symbolic language the usual form of such an instruction is

GO TO label

which is converted by the MIL assembler into a jump instruction with an appropriate displacement.

Conditional branching is achieved by having an IF statement mapped into an appropriate SKIP or TEST microinstruction.⁴ It should not be necessary to become familiar with the detailed syntax of either the SKIP or TEST microinstructions, as the proper ones are generated by the MIL assembler from the higher-level IF statement. Direct use of SKIP and TEST instructions should be avoided. Section 2 of Appendix B lists the registers (and bits within them) that may be tested in an IF statement.

Labels are declared implicitly by their occurrence in the label position of a MIL statement, i.e., the first item on a card (beginning somewhere in columns 1 through 5.) Labels may be global in scope, in which case they must of course be unique, or they may be local in scope, in which case two or more occurrences of the same local label are permitted. A local label is declared with its first character immediately preceded by a period (.) character. Hence local labels are often spoken of as *point* labels. To transfer control to a point or local label using a GO TO

⁴ The SKIP conditionally skips the next microinstruction according to the truth or falsity of a bit field in some specified 4-bit register, possibly interpreted under control of a specified *mask*. The TEST conditionally jumps up to ± 16 microinstructions, depending on the condition of a specified bit in a designated 4-bit register.

statement, one must indicate the jump direction (+ for forward or – for backward) in the GO TO statement, e.g., GO TO +LABEL or GO TO –LABEL. The jump is then made forward or backward to the first occurrence of the possibly non-unique LABEL identifier.

There are other microinstructions whose execution exercises control over the flow path of the computation. Of course, there is a HALT instruction and a no-op (NOP) instruction, with the usual meanings. However, perhaps the two most important control instructions beyond the jump and (conditional) skip instructions are the CALL and EXIT instructions used for microsubroutine procedure calls and returns. These instructions operate with the help of a special *stack*, onto which return addresses are pushed on each CALL and from which return addresses are popped on each EXIT.

Executing a CALL instruction pushes the contents of the A-register, (i.e., the program counter) onto the stack. At the time of this push, the program counter already has as its value the address of the next instruction, which is the wanted return address. Thus, executing a matching EXIT microinstruction pops the top entry of the stack and places it in the A-register. After an EXIT is executed, the next instruction executed will be the one whose address is now in A, which is the return address of the subroutine.

The stack is a special set of registers (32 in all) which can only be dealt with as a pure *pushdown device*, that is, only its top entry can be addressed. This entry has the name TAS (*top of A stack*). Because each entry in the control stack is a 24-bit register, one can, with care, push any 24-bit datum onto the stack.

Example MOVE X TO TAS pushes a copy of the value of X onto the stack, and MOVE TAS TO FA pops the top element off the stack and assigns it to FA.

Since there will ordinarily be some excess capacity in the stack, there may be occasions when the stack can be used for storing data that are local to the current procedure activation, provided of course they are “used up”, i.e., all popped off prior to executing the return (EXIT) step. In other words, care must therefore be taken to maintain a balance of executed pushes and pops during each procedure activation.

The foregoing discussion has been a brief sketch of the control aspects and features of the B1726 microprocessor. Later chapters will include a series of case studies (code vignettes), some of which will show how these control features may be exploited. Section 1 of Appendix B gives a summary of the B1726 registers and their properties and uses.

Chapter 4

The B1700 computation environment

The designers of the B1700 software system supposed that the MCP (operating system) would serve a queue of jobs, each remarkably similar in computation structure. We need to become familiar with this computation structure, since any interpreter we build must fit into that structure or else the valuable services of the MCP will not be easily accessible.

Every computation served by the MCP is assumed to consist of two parts, a MIL-coded *program part*, assumed to be an interpreter, and a *data part*, which usually consists of several components. One component, always present, serves as an interface with the MCP. That interface, known as the *run-structure nucleus*, has a standard format, and must be generated and loaded into G-store before the computation can start.

Whether the program part is actually an interpreter is really immaterial to the MCP as long as appropriate formatting conventions for the computation structure are adhered to, as in the construction and use of the run-structure nucleus. The program part is the MIL object code produced by the MIL assembler. It is easiest, and quite practical, to assume that the entire program part resides in the program store of the B1700, i.e., H-store. However, in reality the MIL object program may be segmented, some segments being loaded into H-store and the rest into G-store. The MCP will take care of loading the interpreter in this fashion.

When microcode is split between H- and G-store, certain hardware control registers, not yet described, are pre-loaded by the MCP to condition the microinstruction-fetching mechanism so that each instruction is fetched from the appropriate store in which it resides. No extra program steps are needed to fetch a G-store-resident instruction.¹ Only the hardware speed of the fetch is different (1 microsecond for G-store versus 167 nanoseconds for H-store). Via MIL statements, a programmer may advise the MCP which segments should be loaded, space permitting, into the preferred H-store.

¹The hardware organization details by which this "neat trick" is accomplished are discussed in Chapter 7.

During the computation, messages are transmitted to/from the program and the MCP via “mailboxes” in the run-structure nucleus. Also placed in this interface, usually preset at load time, are key parameters of the computation, for instance, i/o device and file identification and attributes of files, the size and makeup of the remainder of the data part, and the name (file identification) of the associated interpreter. This information is used by the MCP in determining, for instance, what i/o service to offer the computation when coded messages requesting service are sent to the MCP, and where to find the interpreter whenever the MCP is ready to give control back to the computation. [We are implying here that, in the interplay between a computation and the MCP, each computation plays the role of a *coroutine* with respect to the MCP.] There is a lot to know about the run-structure nucleus if one expects to construct an interpreter that fully exploits the MCP’s service functions. Fortunately, we will be able to get started knowing a bare minimum about this nucleus. (We will see why this is so later.)

The data part of a computation has other important components (at least one) besides the system-oriented run-structure nucleus. There needs to be a read/writeable workspace area which the program (interpreter) can use for local storage. Of primary interest to us as beginners is the section of the read/write workspace called the STATIC region.

The workspace required for one program may be so large that overlaying strategies will be needed to conserve G-store. For this reason an overlayable or DYNAMIC region of the read/write workspace, contiguous with the STATIC region, may also be specified (Figure 4.1). It is the responsibility of the programmer (i.e., the author of the interpreter) to overlay (or swap) from the disk portions of the workspace into the DYNAMIC region, if such space is needed.

For many interpreters the G-language code being interpreted is invariant (i.e., the process of the interpretation causes no alteration of the G-language code). Such code may therefore be regarded as *read-only*. Only the read/writeable section of the workspace need be placed within the region bracketed by the BR and LR registers.² For this reason invariant, code to be interpreted (higher-level language code) is usually loaded into G-store on an as-needed basis as *code segments*, outside the BR-LR region. Code segments belong to an address space that is logically and physically separate from the read/writeable workspace.

We now begin to perceive an important distinction between an interpreter for a higher-level programming language like FORTRAN or COBOL, whose code is invariant under interpretation, and an inter-

² These registers are sensed by the hardware so as to prevent attempts to write outside the bracketed region.

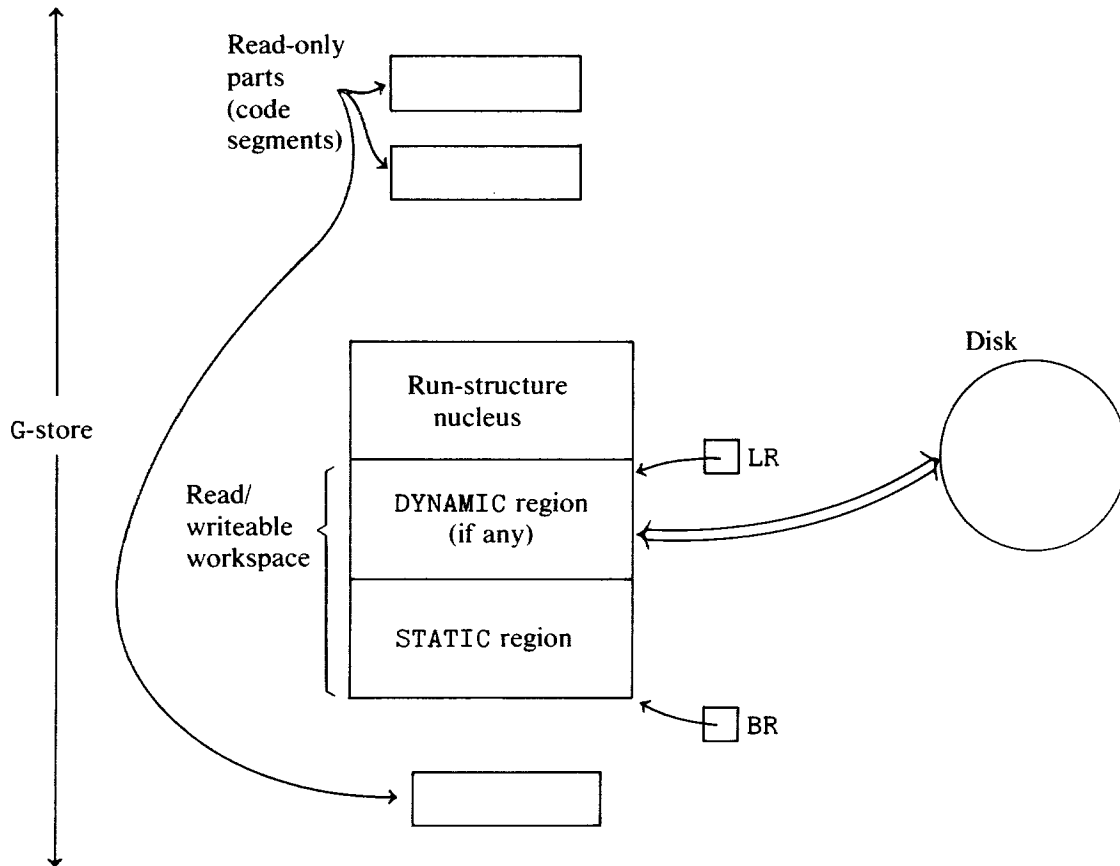


Figure 4.1. Accessing environment for a B1700 computation. Note that the run-structure nucleus is always allocated immediately “above” the BR-LR workspace.

preter (or simulator) for an actual machine, whose storage must be regarded as read/writeable. For example, an interpreter for the SAMOS machine must simulate its storage. SAMOS code to be interpreted must be loaded (written) into the simulated storage. Moreover, the very process of executing a SAMOS read (RWD) instruction results in *writing* into the simulated storage. We see that as long as our main interest is the construction of interpreters for actual machines whose storage is read/writeable as in SAMOS, the workspace for the interpreter will need no read-only code segments.

If the storage space of an interpreted machine is small enough, as it may be for a first-attempted simulation, it should be possible to ignore the DYNAMIC region. We shall assume that the STATIC region can be allocated large enough for adequately representing the storage and the various registers of the emulated machine with room to spare for the local storage needed by a MIL-coded interpreter. [For example, to simulate a 100-word SAMOS machine (88 bits per 11-character word), no more than 10,000 bits of STATIC storage are needed to represent the

store and the various registers. The implementation of a SAMOS simulator is considered as a case study in Chapters 5 and 6. Appendixes E and F provide further details.]

4.1 THE BURROUGHS CONCEPT OF "CODEFILE"

In Burroughs B1700 terminology, the entire data part of a computation, (all its components as discussed above) is specified by a *codefile*. Each codefile is typically generated by a Burroughs-provided *compiler*. For example, consider what happens when a COBOL program is processed by a Burroughs-supplied COBOL compiler, which is compatible with the MCP and produces code for the Burroughs-supplied COBOL interpreter. The COBOL compiler, taking the user's COBOL program as input, not only generates code segments to be interpreted by the COBOL interpreter, but also generates and formats the run-structure nucleus and various constants that are to be preset in the STATIC and DYNAMIC regions. The generated nucleus also contains a system pointer to the particular (COBOL) interpreter which will "execute" the codefile. [The interpreter may be regarded as the MCP's coroutine and the codefile as the accessible environment for the coroutine.]

Once the codefile has been prepared, it is kept "on ice" as an ordinary disk-resident file and read by the MCP whenever the user issues a command to execute it. When that happens, the MCP loads the data part of the codefile into G-store and then *invokes* it by sending control to the start point of the indicated interpreter.

To summarize, note the following points.

1. In the context of the B1700 MCP, the data part of a computation structure is generated as part of the process of *compilation* and saved as a codefile named by the user.
2. The program part is a file of MIL object code whose name is specified to the MCP at the time the codefile is created (by a `COMPILE` command).
3. Once the codefile is completed as a file in the system storage, it can be executed (`EXECUTE` command).
4. When the MCP responds to an `EXECUTE` command naming that codefile, the MCP will, in effect, load the codefile and its associated interpreter³ and, when ready to do so, pass control to the start point of the interpreter. Loading the codefile implies

³ If a copy of the interpreter is already loaded, perhaps executing some other codefile at this time, this step is skipped. The interpreter is reentrant, so only one copy ever needs to be resident in addressable storage.

allocating storage and loading a copy of the run-structure nucleus, the read/write STATIC region, and the initial DYNAMIC region if any.

4.2 CONSTRUCTING A COMPUTATION ENVIRONMENT

We have described the nature of a B1700 computation environment and we have explained how such environments come into existence under control of the MCP during “normal” use of the system, as conceived by the system’s designers. If we are to implement arbitrary MIL programs on the B1700, also under MCP control, we will need some convenient system for constructing (compiling) computation environments tailored to the needs of our individual MIL programs (interpreters). No such general-purpose environment constructor is currently available as a Burroughs software product, since the machine is not marketed for use by customers who are MIL coders. (Ordinary customers are expected to code in one of the several higher-level languages for which MIL-coded interpreters are already provided.)

One general-purpose constructor program of the type needed was developed by Hinds.⁴ This program is on file in the University of Utah system and is regarded by the MCP as a compiler (since its purpose is to construct codefiles). Hinds named his compiler the LOADER, but it is really a *codefile maker*. To use the LOADER, one has to learn its language. We will see later that this language is relatively simple to learn. [A LOADER program is a sequence of statements which is compiled into a codefile description appropriate for the MIL program one would like to execute. The syntax of such a program will be described briefly in the next section; for those needing it, a reference document is given as Appendix D.]

Taking stock to this point, we see that to implement and then execute a MIL-coded computation on the B1700 under the MCP, we need to successfully complete a 3-step process:

1. Request the MCP to assemble a MIL object file using the MIL assembler, given a symbolic MIL program. The name given to this assembled object file will later be regarded by the MCP, in step 3, as the name or identifier of an interpreter.
2. Request the MCP to use the LOADER to compile a LOADER program into a codefile. The request includes a name to be associated with the codefile to be generated.

⁴J. A. Hinds, “SMACK, a System for Operating System Interface for the Burroughs B1700”, M.S. Dissertation, Department of Computer Science, State University of New York at Buffalo, 1975.

3. If, in step 1, the given name of the MIL object is `MY.MIL`, and if in step 2 the name specified for the codefile is `MY.CODEFYL`, then this step requests the MCP to execute `MY.CODEFYL` using the `MY.MIL` interpreter.

4.3 IMPLEMENTING A COMPLETE MIL PROGRAM

We could continue providing additional details on the use of the MIL assembler and the `LOADER` in more or less top-down fashion, until you had enough information to begin constructing a MIL program—perhaps even a complete simulator program. But at this point it seems more fruitful to make a new beginning and proceed from the bottom up by building up a complete (though trivial) program and identifying by a discovery process the concepts, constructs, and tools we still need to learn to complete our understanding.

In the spirit of learning in small steps, we want first to imagine that our MIL program represents a very simple everyday algorithm. For the moment, let's assume we have succeeded in getting our loaded program to begin executing. You may rightly ask, "What algorithms are we now able to code in MIL?" The MIL we have seen so far allows us to perform only processing steps that are *internal* to the machine. We saw no way to actuate and use a card reader, start up a line printer, open or close a file, rewind a tape, etc. How far can we really get without such i/o instructions? What about declarations? How do we *name* the variables of our program, i.e., associate names with storage cells, so we can easily map from a flowchart language variable to specific fields in storage or to registers of the processor? Are we stuck with all those funny names for the registers of the H-processor, or can we rename them, using "decent" mnemonics that are especially applicable to the problem at hand?

All or at least most of these missing pieces become evident when we tackle even the simplest of problems. For example, suppose the problem is to write and test a MIL program that inputs a line of 80 characters from a data card, echoes that line on the line printer, and outputs the inverted line on the printer, repeating this process until the card deck (input file) is exhausted, and then prints a sign-off message such as `THE END`.

Figure 4.2 is a flowchart and legend of what we want. Study it for a moment. Our present knowledge of MIL coding will allow us to map the heart of the algorithm (boxes 5 through 8), but not boxes 1 through 4 or 9 through 12, which involve declarations and i/o operations. So let's start by coding what we know how to code, and then learn how to do the rest via a series of digressions into some new topics.

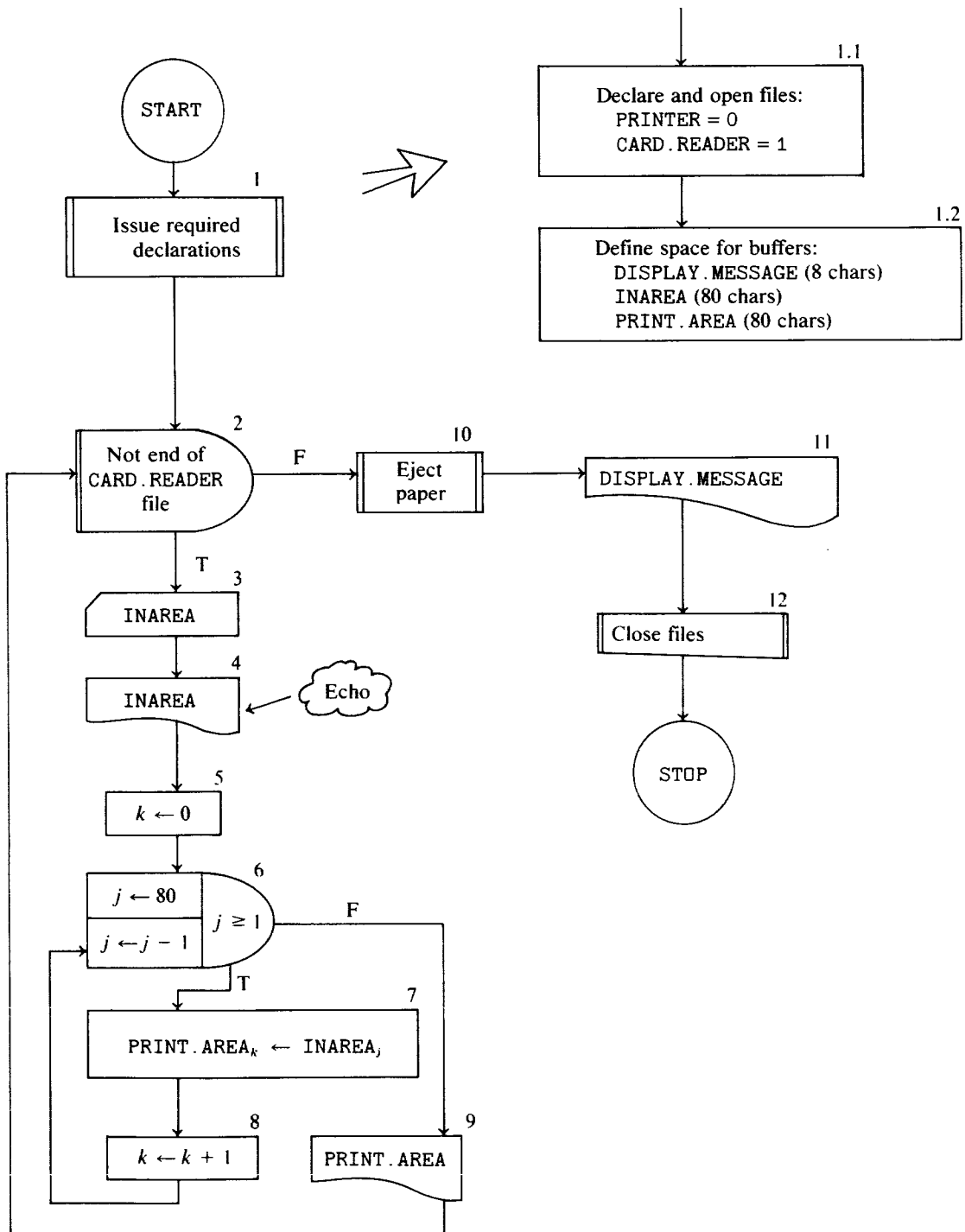
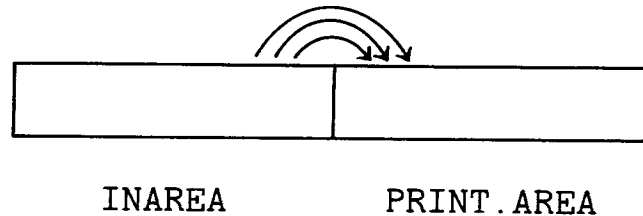


Figure 4.2. Flowchart algorithm for displaying inverted card images.

We assume that suitable declarations (box 1.2) will make the symbols for the buffers INAREA and PRINT.AREA stand for bit offsets from the base of the data storage region. Let us further assume that we can declare these 80-character buffers as *contiguous* fields.



Inverting a card image amounts to moving $INAREA_{79}$ to $PRINT.AREA_0$, then $INAREA_{78}$ to $PRINT.AREA_1$, etc.

We seem to need two starting addresses for the loop of boxes 6, 7, and 8, namely that of $INAREA_{79}$ and $PRINT.AREA_0$. Here, however, we can make use of the little trick: If we read $INAREA_{79}$ into a receiver register using the REVERSE option, then $PRINT.AREA_0$ is the proper starting address for the READ REVERSE of $INAREA_{79}$. It is of course also the proper starting address for $PRINT.AREA_0$. This means that to code box 7 for the first transit of the inversion loop, we will need to compute only one absolute G-storage address, that of $PRINT.AREA_0$. This address is the sum of BR (which holds the absolute address of the base of the program's workspace) and the declared offset value for PRINT.AREA. The sum can be formed in register FA if we remember to use the adder associated with FA. This technique assumes that one operand is in FA and the other is in a scratchpad register. The procedure is

```

MOVE BR TO SOA           % SOA SELECTED TO HOLD VALUE
                        % OF BR
MOVE PRINT.AREA TO FA    % "PRINT.AREA" IS A SYMBOL
                        % WHICH, BY DECLARATION, HAS
                        % BEEN EQUATED TO SOME
                        % INTEGER
ADD SOA TO FA
MOVE FA TO S2A          % SAVE COPY OF COMPUTED
                        % ADDRESS IN S2A

```

We can represent the loop counter j (range 80 to 1) as FL if we set FL to 80×8 , step it down by 8, and then test for $FL = 0$.

BOX6

```

    MOVE 80*8 TO FL           % THE SET PART OF BOX6
.LP  IF FL = 0 GO TO BOX9    % ESCAPE FROM INNER LOOP
% BOX7. USE Y AS THE CHARACTER RECEIVER
    READ 8 BITS REVERSE TO Y DEC FA AND DEC FL
                                % DECREMENTS THE COUNTER
                                % AND ALSO ACCOMPLISHES
                                % THE PURPOSE OF BOX8
    XCH S2 F S2              % GET POINTER TO SINK
                                % AND SAVE POINTER TO SOURCE
    WRITE 8 BITS FROM Y INC FA
    XCH S2 F S2              % RESTORE POINTER TO SOURCE
    GO TO -LP                % "-LP" MEANS THE PRECEDING
                                % LABEL, ".LP"

```

The READ REVERSE instruction decrements FL and FA, thus accomplishing the counter and indexing arithmetic for INAREA. The WRITE instruction increments FA, thus accomplishing the indexing on PRINT.AREA, as it is not necessary to keep a second counter, corresponding to k . Hence box 5 needs no counterpart in the MIL code.

A moment's reflection explains why we saved in S2A a second copy of the computed absolute address for PRINT.AREA. We need one copy to be successively decremented and another copy to be successively incremented. The XCH instructions are used to interchange these values. FL is saved along with FA on the first XCH and is restored on the second XCH.

Having found a way to code the inner loop of Figure 4.2, we are now ready to consider how declarations are coded. Then we will be ready to consider the i/o steps.

4.4 DECLARATIONS IN MIL

Three types of declaration statements are available in MIL: MACROs, DEFINEs, and DECLAREs.

MACROs allow us to define templates for instruction sequences that are to be generated upon request to the assembler.⁵ This is a powerful language feature about which we will have more to say later.

⁵ A macro definition, by analogy with a procedure definition, can be invoked via a macro call anywhere in the body of a MIL program. The MIL assembler responds to a macro call by substituting for it a copy of the template, after making string substitutions as indicated by the arguments supplied by the macro call.

A DEFINE statement permits the MIL coder to use an arbitrary symbol as a surrogate (or substitute) for a standard symbol, literal, or previously DEFINED symbol. For example,

```
DEFINE LEN.OF.INAREA = 640#
```

would allow us to recode the first line of BOX6 in the more meaningful way

```
BOX6
    MOVE LEN.OF.INAREA TO FL % SET PART OF BOX6
```

Mnemonic-valued symbols are nearly always preferable to numeric constants for two reasons.

1. The documentation has greater clarity.
2. If the substituted symbol is used several times in the program, then a later decision to change the constant—e.g., from 640 to 768 (in going from 80- to 96-column cards)—requires only a change in the DEFINE statement, in this case to DEFINE LEN.OF.INAREA = 768#. Reassembly of the program will then substitute, for every occurrence of LEN.OF.INAREA, the new value 768.

DEFINES can also be used to advantage for “christening” scratchpad registers with new names that characterize their storage function in the given program. For example, we chose SOA as the register to hold a copy of the BR value. Why not rename SOA as BR.VALUE?

```
DEFINE BR.VALUE = SOA#
```

While we are at it, we might rename S2A as IMAGE.ADDRESS and S2 as IMAGE.DESCRIPTOR. With these definitions, our code for the inner loop is now as shown in Figure 4.3.

The scope of a DEFINE, unless otherwise constrained, is the entire MIL program. We can narrow or localize the scope very easily. Scopes may be nested as in ALGOL declarations. The syntactic device for blocking the DEFINES is a pair of BEGIN, END statements where the BEGIN statement is followed by the phrase LOCAL.DEFINES, as suggested in Figure 4.4.

A study of the purely fictitious example in Figure 4.4 reveals the following: The symbol LEN.OF.INBUFFER means S5B everywhere in the program. The symbol ADDR.INBUFFER means S5A everywhere in level 0 and level 1 of the program. In block C (level 2) the same symbol means S6A. In Block A and in Block C the symbol ADDR.CARD.COUNTER is associated with S5A. Within block B the

```

% COMPUTE CARD IMAGE.ADDRESS AND SAVE COPY
  MOVE BR TO BR.VALUE
  MOVE PRINT.AREA TO FA
  ADD BR.VALUE TO FA
  MOVE FA TO IMAGE.ADDRESS
BOX6
  MOVE LEN.OF.INAREA TO FL % THE SET PART OF BOX6
.LP IF FL = 0 GO TO BOX9 % ESCAPE FROM INNER LOOP
% BOX7. USE Y AS THE CHARACTER RECEIVER
  READ 8 BITS REVERSE TO Y DEC FA AND DEC FL
  % DECREMENTS COUNTER
  XCH IMAGE.DESSCRIPTOR F IMAGE.DESSCRIPTOR
  WRITE 8 BITS FROM Y INC FA % INC PART IS BOX8
  XCH IMAGE.DESSCRIPTOR F IMAGE.DESSCRIPTOR
  GO TO -LP

```

Figure 4.3. Code for the inner loop.

symbol ADDR.INDEX.REG is also associated with S5A. (In block C, therefore, S5A has three different aliases.)

A DECLARE statement⁶ (almost identical with the UPL⁷ DECLARE) permits the MIL coder to define data structures *and* associate with each a G-store, base-relative address. The MIL assembler maintains a "location counter" which is started at zero at the beginning of each assembly. That counter is incremented as MIL processes each DECLARE, the amount of the increment being the size of the declared data structure. Use of the DECLARE allows the programmer to associate a program variable with a specific field in G-store. This can best be illustrated for our card-image inversion problem. Box 1.2 of Figure 4.2 can be coded using a DECLARE as follows.

```

DECLARE
  DISPLAY.MESSAGE CHARACTER(8),
  INAREA           CHARACTER(80),
  PRINT.AREA       CHARACTER(80);

```

The effect of processing this declaration will be to associate DIS-

⁶ The DECLARE statement was not described in the original MIL reference manual printed by Burroughs. An alternate method for naming and sizing data spaces was available to students using James A. Hinds's SMACK system. The alternate approach used the =BSS macro call in McMIL.

⁷ "B1700 Systems User Programming Language (UPL) Reference Manual", Burroughs Corporation, Detroit, December 1973, Form No. 1067170.


```

Line No.
1      % LEVEL 0
2      DEFINE LEN.OF.INBUFFER      = S5B #
3      DEFINE ADDR.INBUFFER        = S5A #
      :
10     [ BEGIN % BLOCK A (LEVEL1)
11     [ LOCAL.DEFINES
12     [   DEFINE ADDR.CARD.COUNTER = S5A #
      :
20     [   END   % BLOCK A
      :
30     [ BEGIN % BLOCK B (LEVEL1)
31     [ LOCAL.DEFINES
32     [   DEFINE ADDR.INDEX.REG    = S5A #
      :
40     [   [ BEGIN % BLOCK C (LEVEL2)
41     [   [ LOCAL.DEFINES
42     [   [   DEFINE ADDR.CARD.COUNTER = S5A #
43     [   [   DEFINE ADDR.INBUFFER    = S6A #
      :
50     [   [   END   % BLOCK C
      :
60     [   END   % BLOCK B
      :
70     % END OF PROGRAM

```

Figure 4.4. Use of the LOCAL.DEFINES feature for localizing the scope of DEFINE statements.

PLAY.MESSAGE with bit address 0, INAREA with bit address 64, and PRINT.AREA with bit address 704 (=64 + 640). More elaborate DECLAREs involving structured data types, as in UPL, are also permitted—for example,

```

DECLARE 01 STRUC,
        02 C,
        03 D BIT(20),
        03 E BIT(30),
        02 G CHARACTER(3);

```

It is not necessary to specify type and length for STRUC and C, which are *group items*. Clearly STRUC is a 20 + 30 + 3×8 or 74-bit record, and

C is a 20 + 30 or 50-bit subrecord. Using this definition, the absolute G-store address of STRUC, or for any subfield of STRUC, can be computed as we have done in the past. For example, to move the first character of G to the T register,

```

MOVE BR TO SOA    % MOVE LEFTMOST CHARACTER
MOVE G  TO FA     % OF G TO T
ADD SOA TO FA
READ 8 BITS TO T

```

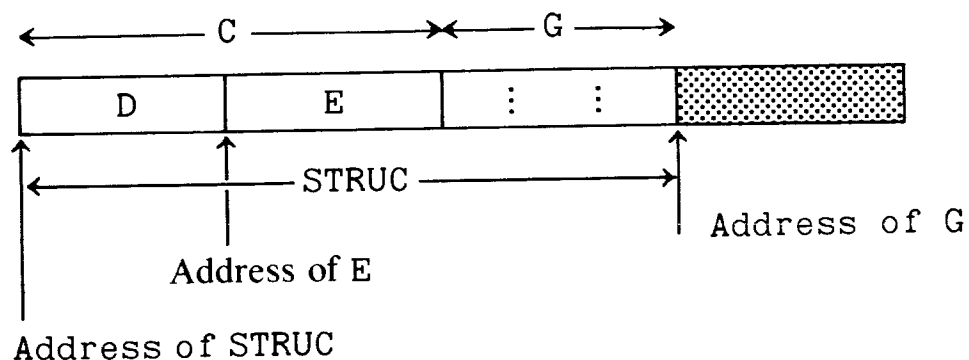
The declared attribute of a variable, a structure, or any part of a structure may include the word REVERSE so that the MIL assembler will associate with that identifier an address that is appropriate for READ REVERSE and WRITE REVERSE microinstructions. For example, if the characters of the component G of STRUC are to be brought into T in REVERSE mode (i.e., one at a time, last character first), then we might declare STRUC as

```

DECLARE 01 STRUC
        02 C,
        03 D BIT(30),
        03 E BIT(20),
        02 G CHARACTER(3) REVERSE;

```

Now the address associated with G is the ending address of G, plus 1, i.e.,



So, to bring into the T-register the last character of G, we might use code such as

```

MOVE BR TO SOA
MOVE G  TO FA
ADD SOA TO FA
READ 8 BITS REVERSE TO T

```

Space for arrays may also be declared. Thus,

```
DECLARE FF(5) BIT(10);
```

defines a space for a 5-element array named FF, each element 10 bits long. Any item or group item within a structure may also be an array, but arrays may not be nested.

```
DECLARE 01 Q(5) BIT(48),
        02 B CHARACTER(3),
        02 C FIXED;
```

defines an array structure Q having five 48-bit elements, each being a pair of components B and C, as shown below.

Q ₀		Q ₁		Q ₂		Q ₃		Q ₄	
B ₀	C ₀	B ₁	C ₁	B ₂	C ₂	B ₃	C ₃	B ₄	C ₄

The addresses associated with identifiers Q, B, and C are those corresponding to the *first* elements, Q₀, B₀, and C₀, respectively. Of course, we have no subscript expressions in MIL, so to reference a particular element of an array, other than the first element, requires execution of an appropriate microinstruction sequence provided by the programmer.

MIL provides several useful special features as companion pieces to the DECLARE statement. These can be studied in Appendix A. One feature is mentioned here. The MIL assembler treats the expression

```
DATA.LENGTH(<declared identifier>)
```

as a function call and returns the bit length of the <declared identifier>, (or, if an array, the length of one array element for that identifier) as though that length had appeared in an explicit DEFINE declaration. Thus,

```
MOVE DATA.LENGTH(Q) TO S3A
```

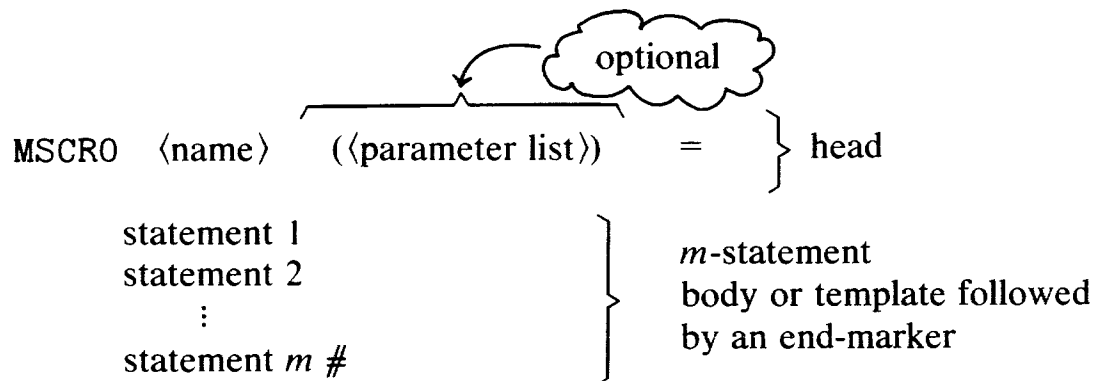
would assign 48 to S3A in the context of the preceding array declaration for Q. Subsequently, one could advance from one element of B or of C to another by incrementing, with the offset DATA.LENGTH(Q) held in S3A

```
MOVE S4A TO FA      % ADDR OF C(I)
ADD S3A TO FA       % COMPUTE ADDR OF C(I+1)
READ 24 BITS TO L   % L GETS C(I+1)
MOVE FA TO S4A      % ADDR OF NEW C(I)
```

A more comprehensive example could be useful here to summarize

what we have learned so far about MIL declarations. Instead, let us expand the concept of MIL macros.

A MIL *macro definition* consists of a head and a body. The head contains the name of the macro and an optional formal parameter list. The body, terminated by the # mark, is the instruction template, consisting of a sequence of one or more MIL statements



Subsequent to encountering such a definition, the MIL assembler will respond to any recurrence of <name> by first substituting for it the corresponding *body*. Each recurrence of <name> is regarded as a *macro call*. If the macro definition includes a formal parameter list, then each corresponding macro call must include a matching argument list. In substituting the statements of the body for <name>, each instance of a formal parameter will be replaced (by string substitution) with its

```
MACRO GET.NEXT.ELEMENT(CURRENT.ELEM,LENGTH,RECEIVER) =
  MOVE CURRENT.ELEM TO FA
  READ LENGTH BITS TO RECEIVER INC FA
  MOVE FA TO CURRENT.ELEM #
  (a)
```

```
DECLARE Q(50) FIXED;
DEFINE CURRENT.Q = S12A #
:
MOVE Q(0) TO CURRENT.Q % SET UP CURRENT.Q
MOVE BR TO FA % WITH ABS. ADDRESS
ADD CURRENT.Q TO FA % OF
MOVE FA TO CURRENT.Q % FIRST ELEM OF Q
:
GET.NEXT.ELEMENT (CURRENT.Q, DATA.LENGTH(Q), X)
IF X = 0 GO TO ZERO.CASE
(b)
```

Figure 4.5.

matching argument. [*Warning*: In the current MIL implementation, a parameter may *not* represent a statement label.]

Example Imagine that a frequently needed step of an algorithm is to scan the next item of an array, given the address of the last item scanned, and the length of an item (assumed to be ≤ 24 bits.) The next item is to be brought to one of the four possible receiver registers for examination. We shall assume that all available addresses are relative to the base register. A possible MIL macro definition is given in Figure 4.5(a).

We might want to use this macro to test if the next element of an array is zero. After having DECLARED the array Q, defined the register CURRENT.Q, and initialized the latter with the absolute address of Q_0 , we can later issue a macro call [underscored statement in Figure 4.5(b)] to place the value of Q_i in the receiver register X and then adjust the pointer, CURRENT.Q, to point to Q_{i+1} . This macro call will *expand* to (i.e., be replaced by)

```
MOVE CURRENT.Q TO FA
READ DATA.LENGTH(Q) BITS TO X INC FA
MOVE FA TO CURRENT.Q
```

We see that the body of GET.NEXT.ELEMENT will be inserted into the MIL code in place of the call, but with the argument strings substituted for the corresponding parameter. When the MIL assembler reads the new lines of code, it will assemble microinstructions for which DATA.LENGTH(Q) will be evaluated as 24 and CURRENT.Q evaluated as S12A, by virtue of declarations previously encountered.

4.5 LITERALS

We have been illustrating the use of literals in a number of MIL statements and declarations. It is time we explained the rules by which one writes a MIL literal and the rules that the MIL assembler uses to interpret them.

Literals come in three “flavors”; they are character strings, digit strings, and decimal integers.

1. *Character strings* are enclosed by quotation marks, e.g., "CAT", "99", "□□□", etc. Such a string may have up to 3 characters. Thus, the statement MOVE "CAT" TO X would be mapped to an instruction to assign to X a string of 24 bits representing the EBCDIC encoding of "CAT".
2. *Digit strings* may be specified in base 2, 4, 8, or 16. Each digit string is enclosed by “at” signs (@). The digit string is preceded

by a base indicator enclosed in parentheses. The indicator (bits per digit) is 1 for base two, 2 for base four, 3 for base eight, and 4 for base sixteen [The indicator is optional for a base-16 digit string.] For example,

Base	Digit string	Meaning
2	@(1)101@	Binary number 101_2
4	@(2)3211@	Base-4 number 3211_4
8	@(3)62715@	Octal number 62715_8
16	@(4)AF0C9@	Hexadecimal number $AF0C9_{16}$
16	@AF0C9@	Hexadecimal number $AF0C9_{16}$

Thus all of the following MIL instructions are equivalent

```
MOVE @(4)3218@ TO L
MOVE @3218@ TO L
MOVE @(3)30450@ TO L
MOVE @(1)0011000100101000@ TO L
```

The effect of each of these instructions will be to assign the specified digit string, converted to a bit string, to L, right-justified, with left zero fill.

3. *Decimal integers.* Unsigned or positive decimal integers are converted to unsigned binary integers. Negative decimal integers are converted to 2s-complement form.

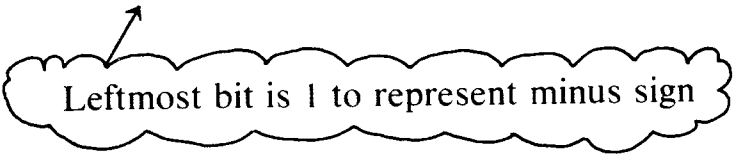
Examples

1. MOVE 24 TO CP is equivalent to MOVE @(1)00111000@ TO CP.
2. DEFINE LENGTH = 65# is equivalent to
DEFINE LENGTH = @(1)1000001@ #.
3. MOVE -3 TO X is equivalent to

```
MOVE @FFFFFFD@ TO X  % 2'S COMPLEMENT OF 3
                    % EXPRESSED IN HEX
```

Note that if we want a different representation for -3 moved to X, such as a signed magnitude representation, we must specify the negative integer as a digit string, e.g.,

```
MOVE @800003@ TO X
```



Leftmost bit is 1 to represent minus sign

4.6 INPUT/OUTPUT IN THE McMIL LANGUAGE

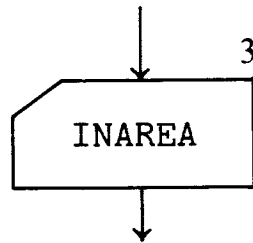
The language MIL has been extended by James A. Hinds in a system called SMACK (*system macro*). SMACK includes a package of about 20 powerful macro definitions. A number of these macros define instruction sequences for communicating input/output requests to the MCP. As mentioned earlier, the MCP has responsibility for executing i/o functions and file-system primitives. Getting the MCP to carry out these steps amounts to sending it an appropriate message called a “communicate”. Because sending these messages involves use of the run-structure nucleus, and because we would prefer insofar as possible to regard such activity as off limits (at least while we maintain amateur status as MIL coders), the SMACK macro definitions will answer our need admirably. We shall be able to code our interactions with the MCP as *macro calls* and thus avoid the risk of generating code that the MCP cannot understand or digest. [Using this approach, we in effect delegate to SMACK the role of (MCP) interface specialist.] We have only to become familiar with the available SMACK macros and how to use them. The language MIL is really extended when we permit calls to the predefined macros of SMACK. Macro calls are known as “E-statements” (E is for extension). Together with regular MIL statements, they form the superset known as McMIL. An E-statement is distinguished from ordinary MIL statement by beginning with an equal sign (=) in column 1.

A McMIL program (i.e., a MIL program that has been enriched with E-statements) is processed in two stages. In the first phase, each E-statement (macro call) is expanded into a sequence of MIL statements. Upon completion of the first or preprocessing phase, the program is ready to be assembled by the MIL assembler. The second phase completes the process of producing the microcode and places the object code on file for use as an interpreter.⁸

We introduce a few of the important McMIL statements here. A complete reference manual (user’s guide) for McMIL and SMACK is included in Appendix C and should be consulted when more information is needed.

⁸ R. A. Belgard, while with Burroughs, and later at the University of Utah, developed a set of macro definitions (coded in MIL) known as BIOPSI. These definitions are comparable to those of SMACK. The BIOPSI macros may, however, be invoked using ordinary MIL macro call statements, provided that the definitions are included in the same source program. The inclusion of these definitions is achieved easily, using control commands of the form & LIBRARY <file name>, where <file name> designates the file holding the BIOPSI definitions. Advanced MIL programmers at the University of Utah now prefer the use of BIOPSI over SMACK because symbolic assembly is faster using BIOPSI since only one processing phase is needed.

Consider the problem of coding box 3 of Figure 4.2,



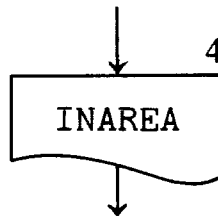
Here we wish to fill the buffer INAREA from the CARD.READER file. The McMIL equivalent of box 3 is:

= BUFFER READ USING INAREA FILE CARD.READER

The underscored parts of this statement are the arguments of this particular SMACK macro. [Note that the format of a SMACK macro call is quite different from the functional form used in MIL, which is:

$\langle \text{macrocall} \rangle ::= \langle \text{macroname} \rangle | \langle \text{macroname} \rangle (\langle \text{argument list} \rangle).$

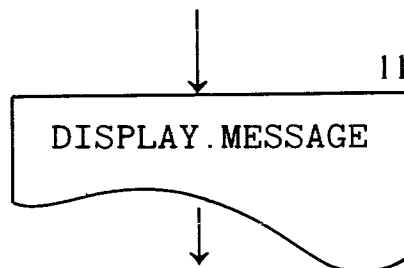
The same macro is used for coding box 4,



but on printer output we specify line spacing. One writes

= BUFFER WRITE USING INAREA FILE PRINTER OPT SINGLE

Likewise box 11,



may be coded as

=BUFFER WRITE USING DISPLAY.MESSAGE FILE PRINTER OPT SINGLE

The OPT SINGLE means “space the printer carriage *one line after*

printing”. The possible spacing options are

OPT SINGLE	spaces one line after printing
OPT DOUBLE	spaces two lines after printing
OPT EJECT	skips to next page after printing
OPT ADVANCE	spaces no lines after printing (allows <i>overprinting</i>)

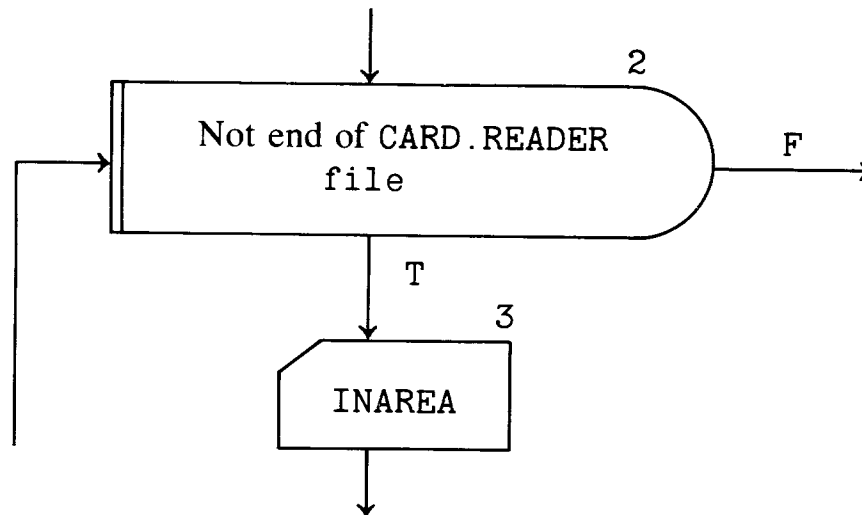
If no spacing option is specified in a WRITE statement the default is *no advance*.

We may also wish to send the same message, such as “THE END”, to the console printer. If so, the key word needed is DISPLAY rather than WRITE, e.g., =BUFFER DISPLAY USING DISPLAY.MESSAGE. No file need be specified, since the console printer cannot be attached to a user’s computation, as a file can by *opening* it. No spacing option is required either.

The BUFFER READ macro call may also specify a branch on *end of file*. For our problem we can state

```
BOX2AND3
=BUFFER READ USING INAREA FILE CARD.READER
      ON EOF GO TO BOX10
```

because this McMIL statement is really the equivalent of flowchart boxes 2 and 3 combined,



Box 10 may be coded by appending the EJECT option to a (dummy) print step which prints no characters:

```
BOX10
=OUTPUT 0 BYTES CORE PRINT.AREA FILE
      PRINTER OPT EJECT
```

```

Line
1  ?EXECUTE MCMIL
2  ?CONVERT
3  ?DATA CARDS
4      DEFINE PRINTER          = 0 #
5      DEFINE CARD.READER      = 1 #
6      DEFINE BASE.OF.INTERPRETER = S1A # % SEE TEXT FOR EXPLANATION
7      DECLARE
8      DISPLAY.MESSAGE CHARACTER(8),
9      INAREA                  CHARACTER(80),
10     PRINT.AREA              CHARACTER(80);
11  =INITIALIZE
12  % EXECUTABLE PORTION OF MIL PROGRAM BEGINS HERE
13  =SECTION CARD.INVRT
14  BOX2AND3
15  =BUFFER READ USING INAREA FILE CARD.READER ON EOF GO TO BOX10
16  % BOX4
17  =BUFFER WRITE USING INAREA FILE PRINTER OPT SINGLE
18  %
19      BEGIN % INNER LOOP,
20      LOCAL.DEFINES
21          DEFINE LEN.OF.INAREA      = 640 # % IN BITS
22          DEFINE BR.VALUE           = SOA #
23          DEFINE IMAGE.ADDRESS      = S2A #
24          DEFINE IMAGE.DESRIPTOR    = S2 #
25  % COMPUTE CARD IMAGE.ADDRESS AND SAVE A COPY

```

```

26         MOVE BR TO BR.VALUE
27         MOVE PRINT.AREA TO FA
28         ADD BR.VALUE TO FA
29         MOVE FA TO IMAGE.ADDRESS
30 BOX6
31         MOVE LEN.OF.INAREA TO FL % SET PART OF BOX6
32 .LP     IF FL=0 GO TO BOX9           % ESCAPE FROM INNER LOOP
33 % BOX7.USE Y AS THE CHARACTER RECEIVER
34         READ 8 BITS REVERSE TO Y DEC FA AND DEC FL % DEC PART OF BOX6
35         XCH IMAGE.DESRIPTOR F IMAGE.DESRIPTOR
36         WRITE 8 BITS FROM Y INC FA % INC PART OF BOX8
37         XCH IMAGE.DESRIPTOR F IMAGE.DESRIPTOR
38         GO TO -LP
39     END % INNER LOOP
40 BOX9
41 =BUFFER WRITE USING PRINT.AREA FILE PRINTER  OPT DOUBLE
42     GO TO BOX2AND3
43 BOX10
44 =OUTPUT 0 BYTES CORE PRINT.AREA FILE PRINTER OPT EJECT
45 % BOX11
46 =BUFFER WRITE USING DISPLAY.MESSAGE FILE PRINTER  OPT DOUBLE
47 % BOX12
48 =STOP
49 =TERMINATE CARD.INVRT
50 =END

```

Figure 4.6. Source deck for card inversion program coded in MIL and McMIL.

4.6.1 Declaring files

Notice that `CARD.READER` and `PRINTER` are declared in flowchart box 1 as the file names we intend to associate with the card reader and printer *devices*. We need to declare these devices as files and open them. But actually, the information specified in `FILE` declarations belongs in the codefile, rather than in the MIL program, so these declarations will be placed in the program that the `LOADER` will process.

Both the codefile and the MIL program must know about the input and output devices and files used in the program by some correlated numbering scheme. As indicated in box 1.1 of Figure 4.2, the output device is to be known as file 0 and the input device as file 1 of the program. (Of course, within the MIL program we use the mnemonics `PRINTER` and `CARD.READER` in place of their numeric (internal) names 0 and 1.

Each file is automatically opened on the first attempt to read from it (or write to it). Files are also automatically closed upon termination of the computation. The implicit open and close operations are achieved via the `SMACK` subroutines called by MIL code generated from the `=BUFFER READ` and `=BUFFER WRITE` `McMIL` statements. `McMIL` statements for explicit open and close look like this:

```
=OPEN INVENTORY.FILE WITH INPUT
=OPEN CHECKS WITH OUTPUT
```

and

```
=CLOSE INVENTORY.FILE
=CLOSE CHECKS
```

Before we look at the `LOADER` details and what is needed for this problem, we had better tie up all the loose strands developed so far. Figure 4.6 shows a `McMIL` program listing which, when assembled, will comprise the MIL object code equivalent to Figure 4.2. New code, not yet motivated, appears underscored (in Figure 4.6) and is explained in the next paragraphs.

Explanation of the underscored statements in Figure 4.6 `MCP control cards`. Line 1 invokes the `McMIL` preprocessor. Line 2 asks the `MCP` to translate our deck from `BCD (026)` to `EBCDIC (029)` codes, before it is processed by the `SMACK` subsystem; the data to be converted to `EBCDIC` are in the file named `CARDS`, as indicated on line 3. Line 50, `?END`, marks the end of the data deck.

Line 6 (`DEFINE BASE.OF.INTERPRETER = S1A`) is a `SMACK` requirement. `SMACK` needs to have one scratchpad register, which it

knows by the name `BASE.OF.INTERPRETER`, set aside for its own housekeeping chores. By reserving it using the `DEFINE` statement, we satisfy this `SMACK` requirement. Any 24-bit scratchpad register may be selected except `S0A`. We chose `S1A`.

The next `SMACK` requirement appears on line 11, ahead of the first executable statement of the program (`= INITIALIZE`). This macro call causes `SMACK` to insert at the very beginning of the MIL program a section of code which includes a set of subroutines needed for MCP communication.

These instructions compute and place at a strategic place within the run structure nucleus the resume point of our MIL program (regarded, you will recall, as a coroutine to the MCP). Other instructions in this section save and restore the scratchpad registers just before and just after control is shifted to and from the MCP, respectively.

The first statement following `=INITIALIZE` should be an `=SECTION` card such as the one on line 13 (`=SECTION CARD.INVRT`), which names the section that follows `CARD.INVRT`. The principal purpose of an `=SECTION` statement is to indicate that you want `SMACK` to generate comment cards for each succeeding McMIL statement so the source code to the MIL assembler will be readable (well documented) when you get the assembly listing from the MIL assembler (see Figure 4.7). `SMACK` will also reproduce the name you give on the `=SECTION` card on each line of the assembly listing that follows (until another `=SECTION` card is encountered, if any, after which the name for that section would be reproduced). Each `=SECTION` card generates an `END` for the preceding section and a `BEGIN` for the current section. (The final `END`—i.e., the one just before `FINI` in Figure 4.7—is generated by the `=TERMINATE` statement discussed below.) Study Figure 4.7 to see the effect of the `=SECTION` on line 13 of Figure 4.6.

Line 48 (`=STOP`) generates code such that execution of our microprogram will be terminated, storage released, all files not otherwise closed explicitly closed, and control returned to the MCP.

Line 49 contains the terminate macro call (`=TERMINATE CARD.INVRT`). This macro call is used to terminate the `SMACK` processing of our program (phase 1) and shift control to the MIL assembler to process our “expanded” MIL program. By placing an identifier of our choice on this `=TERMINATE` card (in this case `CARD.INVRT`), we name the file of microcode that the MIL assembler will produce and place it on disk storage. We can retrieve our MIL object code or apply it later as an interpreter by referring to it as `CARD.INVRT`.

We have just cited the essential `SMACK` requirements that must be

BURROUGHS B1700 MIL COMPILER, MARK V.0(01/24/76 18:05)

CARD.INVRT

MONDAY, MAY 02, 1977, 06:28 PM.

99

BLOCK NAME	CODE	MEMORY ADDRESS	SOURCE IMAGE	SEQUENCE	SEGMENT NAME	OBJ DECK ADDRESS
			DEFINE PRINTER = 0 #	[000001] C		
			DEFINE CARD.READER = 1 #	[000002] C		
			DEFINE BASE.OF.INTERPRETER = S1A # % SEE TEXT FOR EXPLANATION	[000003] C		
			DECLARE	[000004] C		
		[000000]	DISPLAY.MESSAGE CHARACTER (80),	[000005] C		
		[000280]	INAREA CHARACTER (80),	[000006] C		
		[000500]	PRINT.AREA CHARACTER (80);	[000007] C		
			BEGIN CARD.INVRT	[000186] C		
			% SECTION CARD.INVRT	[000187] C		
			BOX2AND3	[000188] C		
			% BUFFER READ USING INAREA FILE CARD.READER ON EOF GO TO BOX10	[000189] C		
			% BOX4	[000199] C		
			% BUFFER WRITE USING INAREA FILE PRINTER OPT SINGLE	[000200] C		
			%	[000208] C		
			BEGIN % INNER LOOP.	[000209] C		
			LOCAL.DEFINES	[000210] C		
			DEFINE LEN.OF.INAREA = 640 # % IN BITS	[000211] C		
			DEFINE BR.VALUE = S0A #	[000212] C		
			DEFINE IMAGE.ADDRESS = S2A #	[000213] C		
			DEFINE IMAGE.DESRIPTOR = S2 #	[000214] C		
			% COMPUTE CARD IMAGE.ADDRESS AND SAVE A COPY	[000215] C		
			MOVE BR TO BR.VALUE	[000216] C		[008B0]
			MOVE PRINT.AREA TO FA	[000217] C		[008C0]
CARD.INVRT	2680	AT [008B0]				
CARD.INVRT	9800	AT [008C0]				

CARD. INVRT 05 00 AT [008D0]					
CARD. INVRT 08 00 AT [008E0]					
CARD. INVRT 28 82 AT [008F0]					
		ADD BR. VALUE TO FA			
		MOVE FA TO IMAGE. ADDRESS	[000218]	G	[008D0]
	BOX6		[000219]	C	[008E0]
		MOVE LEN. OF. INAREA TO FL % SET PART OF BOX6	[000220]	C	[008F0]
			[000221]	C	
	.LP	IF FL = 0 GO TO BOX9 % ESCAPE FROM INNER LOOP	[000222]	C	[00C00]
	% BOX7	USE Y AS THE CHARACTER RECEIVER	[000223]	C	[00C10]
		READ 8 BITS REVERSE TO Y DEC FA AND DEC FL % DEC PART OF BOX6	[000224]	C	[00C20]
		XCH IMAGE. DESCRIPTOR F IMAGE. DESCRIPTOR	[000225]	C	
		WRITE 8 BITS FROM Y INC FA % INC PART OF BOX8	[000226]	C	[00C30]
		XCH IMAGE. DESCRIPTOR F IMAGE. DESCRIPTOR	[000227]	C	[00C40]
		GO TO -LP	[000228]	C	[00C50]
	END %	INNER LOOP	[000229]	C	[00C60]
	BOX9		[000230]	C	[00C70]
	% BOX10	BUFFER WRITE USING PRINT. AREA FILE PRINTER OPT DOUBLE	[000231]	C	
		GO TO BOX2AND3	[000239]	C	
	BOX10		[000240]	C	[00010]
	% BOX11	OUTPUT 0 BYTES CORE PRINT. AREA FILE PRINTER OPT EJECT	[000241]	C	
	% BOX12	BUFFER WRITE USING DISPLAY. MESSAGE FILE PRINTER OPT DOUBLE	[000250]	C	
	% BOX13	STOP	[000251]	C	
	END		[000259]	C	
	FINI		[000260]	C	
			[000266]	C	
			[000267]	C	

NUMBER OF ERRORS DETECTED = 000
 NUMBER OF WARNING MESSAGES = 000
 MICRO INSTRUCTION COUNT = 00236

CAUTION: % SUBSET WAS NOT SPECIFIED; THEREFORE, THIS
 PROGRAM SHOULD NOT BE USED ON A 81712/81714.

Figure 4.7.

present in every MIL program processed by SMACK

1. A BASE.OF.INTERPRETER definition
2. An =INITIATE
3. An =SECTION
4. An =STOP
5. An =TERMINATE

A number of other SMACK macros will be found useful even for MIL beginners.

In calling SMACK macros the user must understand one important constraint that is imposed in the version described in this text: All SMACK macro calls involving input and output generate code that assumes there is nothing of interest to the user in the hardware stack. For example, if the MIL programmer leaves anything in the stack before issuing an =BUFFER READ . . . , that information will have been destroyed when control reaches the user's next MIL statement. This means that one may only issue such E-statements in the top level of a MIL program. E-statement i/o cannot therefore be executed from within a user-coded MIL subroutine called in the usual way from the top level, because the return pointer to the caller will be lost.

Usually the programmer can get around this limitation in one of several ways. He may for example, set a global switch which is tested upon return to the top level, to determine if the i/o step should be executed. Alternatively, routines that must issue i/o calls can be treated as coroutines to the top-level program (i.e., routines reached by GO TOs rather than by CALLs and returned from by GO TOs rather than by EXITs).

At this point, the reader is advised to study (once, quickly) the McMil and SMACK User's Guide (Appendix C) for an overview of the available macros and the services they perform and also for a review of what has been said so far about this important support system.

4.7 THE LOADER (DETAILS)

In our overview discussion of the LOADER we said that a LOADER program is a sequence of statements which is compiled into a codefile description. For each MIL program we wish to make operational, we must provide an appropriate LOADER program. Let us do this, by way of example, for CARD. INVRT, the MIL program of Figure 4.5. [Full details on the LOADER can be found in Appendix D.]

The information we are expected to supply falls into three categories (and should be supplied in that order).

1. program parameter specifications
2. scratchpad settings
3. FILE and DATA declarations

Under *program parameter* specifications we may supply a number of attributes of the associated MIL program, including the overall dimensions and makeup of the workspace. For example, `INTERP = CARD.INVRT` gives the *name* of the associated MIL program, and `STATIC = 5500` is an example of a workspace parameter. In fact, for the simple MIL programs we will be writing these two specifications, *name of MIL program* and *size of STATIC*, are really all we need to make.

If we want specific initial *scratchpad settings* other than all zeros, we could next specify their values. We are not likely to want to specify nonzero initial values for our codefiles, so the scratchpad-settings component of our `LOADER` program may be left empty.

We will always want to give some file descriptions, even in the case where our MIL program uses the card-reader and line-printer devices as the only files. Each `FILE` statement associates an internal name with a physical input/output device and supplies, implicitly or explicitly, a list of attributes for that device. File numbers (internal names) are assigned from 0 in the order of appearance of the `FILE` statements in the `LOADER` program. For example, if the first `FILE` statement is

FILE	NAME	=	PRINTER	PRINTER
			⏟	⏟
			Local	Hardware
			name	type

then the file named `PRINTER` will be understood by the MCP as file 0 of this codefile. The file 0 is of hardware type `PRINTER` (as opposed to `TAPE`, etc.). Default attributes of the declared hardware type (e.g., 80-byte records) will be generated by the `LOADER` for eventual placement in the file information block⁹ for file 0. (Note that our MIL program defines `PRINTER` as 0 on line 4 of Figure 4.6, so the names used for file 0 in the MIL program and in the codefile are actually correlated via the number 0 and not by use of the identical local names on both programs. Different local names could have been used with the same net effect.)

⁹ The run-structure nucleus has a few noncontiguous appendages. Among these is a *file dictionary* with pointers to a set of file information blocks, one for each declared file.

If the next FILE statement is

```
FILE NAME = CARD.READR READER;
           Local      Hardware
           name      type
```

then the LOADER will be given sufficient information to permit the generation of a file information block for file 1, the CARD.READR file, by specifying READER as the hardware type. [Most of the options and file attributes (e.g., number of buffers, locks, record size, blocking factors, etc.) recognized by the operating system and which can be specified in FILE declarations in higher-level languages such as UPL, can be expressed in FILE statements of the LOADER language, but we won't need to use these refinements for our beginning work.]

The final item in the LOADER program is the DATA statement for specifying initial values for the STATIC section of the workspace. For example

```
DATA "THE END";
```

specifies that the first 8 character positions of the workspace (beginning at base-relative 0) are to be initialized with the string "THE END". For our MIL program the variable whose address begins at base-relative 0 is DISPLAY.MESSAGE. No value is input for this variable, yet in box 10 we print out its contents (line 46). The above DATA statement guarantees that when equivalent of line 46 is executed, the message THE END will be printed.

The last statement of a LOADER program is FINI. Figure 4.8 shows

```
?COMPILE CARD/INVERTER WITH LOADER LIBRARY
?CONVERT
?DATA CARDS

INTERP=CARD.INVRT  STATIC=5500; } Program parameter specs
;                               } Scratchpad settings (empty)

FILE NAME=PRINTER PRINTER;     }
FILE NAME=CARD.READR READER;   } FILE and DATA declarations
DATA "THE END";
FINI;
?END
```

Figure 4.8. The LOADER program named for use with the CARD.INVERT interpreter.

```

?EXECUTE CARD/INVERTER
?DATA CARD.READR
NOW IS THE TIME FOR ALL...
ABLE WAS I ERE I SAW ELBA
MADAM IM ADAM
      ...EVE...
?END

```

Figure 4.9. The job deck for testing CARD/INVERTER.

the LOADER program ready for the card reader, complete with MCP control cards. We ask to *compile* the program, arbitrarily named CARD/INVERTER, using LOADER as our compiler, and we ask to place the compiled codefile in the library.

Once we have both the MIL object program and the codefile in the LIBRARY, we can execute the codefile using a job deck like the one shown in Figure 4.9. The output will appear as in Figure 4.10.

Since our MIL program fails to trim off trailing blanks from each data card that is processed, each output line appears *right-justified*, in contrast with the echoed data card images, which appear *left-justified* at least for those data cards with nonblanks in column 1).

Exercise. Modify CARD.INVRT so that blanks are stripped off the right end of each data card before the inversion is made into PRINT.AREA, thus producing inverted card images that are *left justified* as shown in Figure 4.11. Data cards that are entirely blank should be echoed, but their inverses should not be printed.

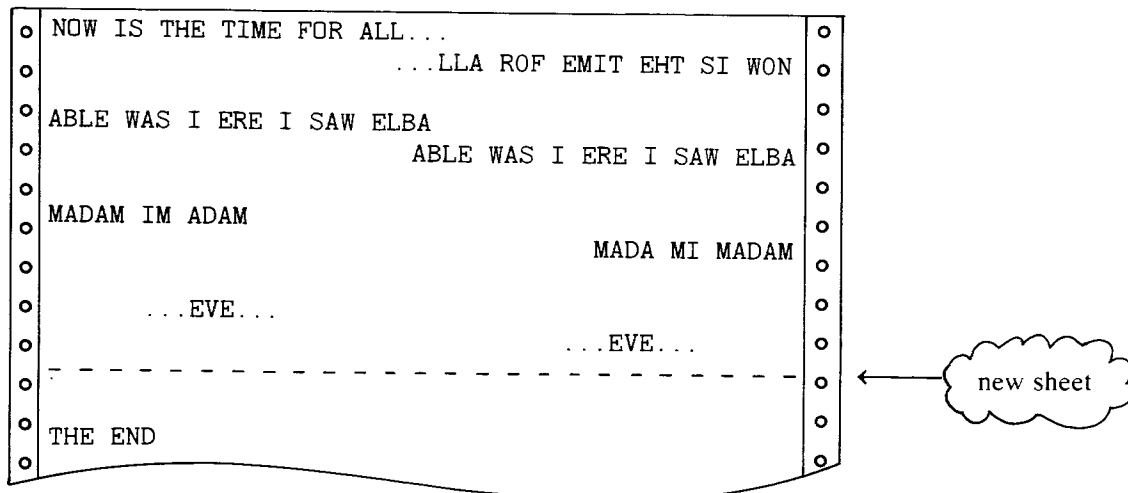


Figure 4.10. Output of program executing CARD/INVERTER.

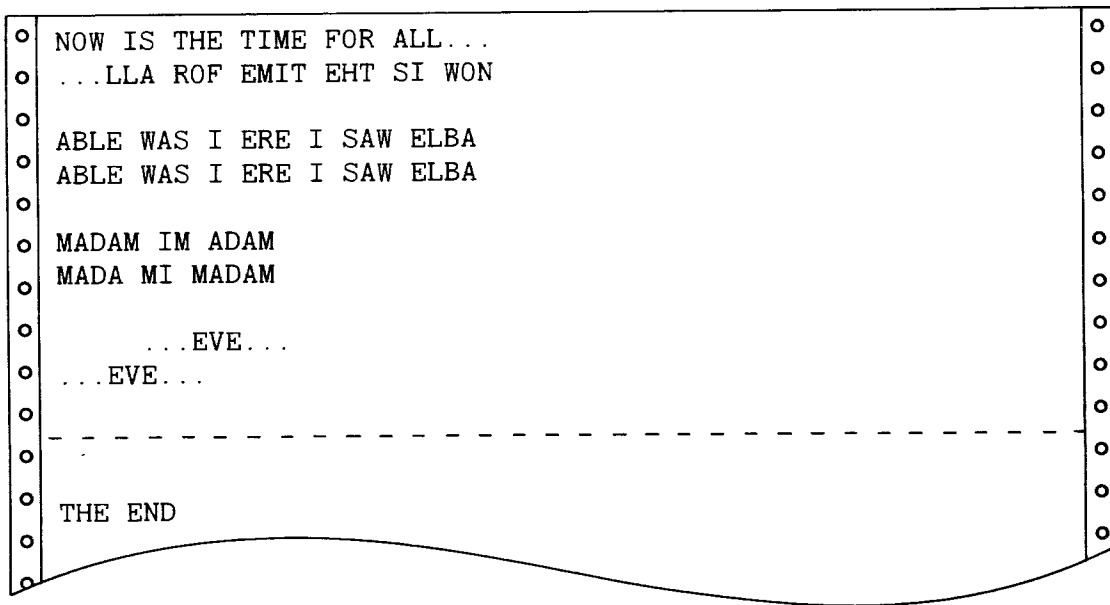


Figure 4.11. Desired output achieved by trimming off trailing blanks before inverting each card image.

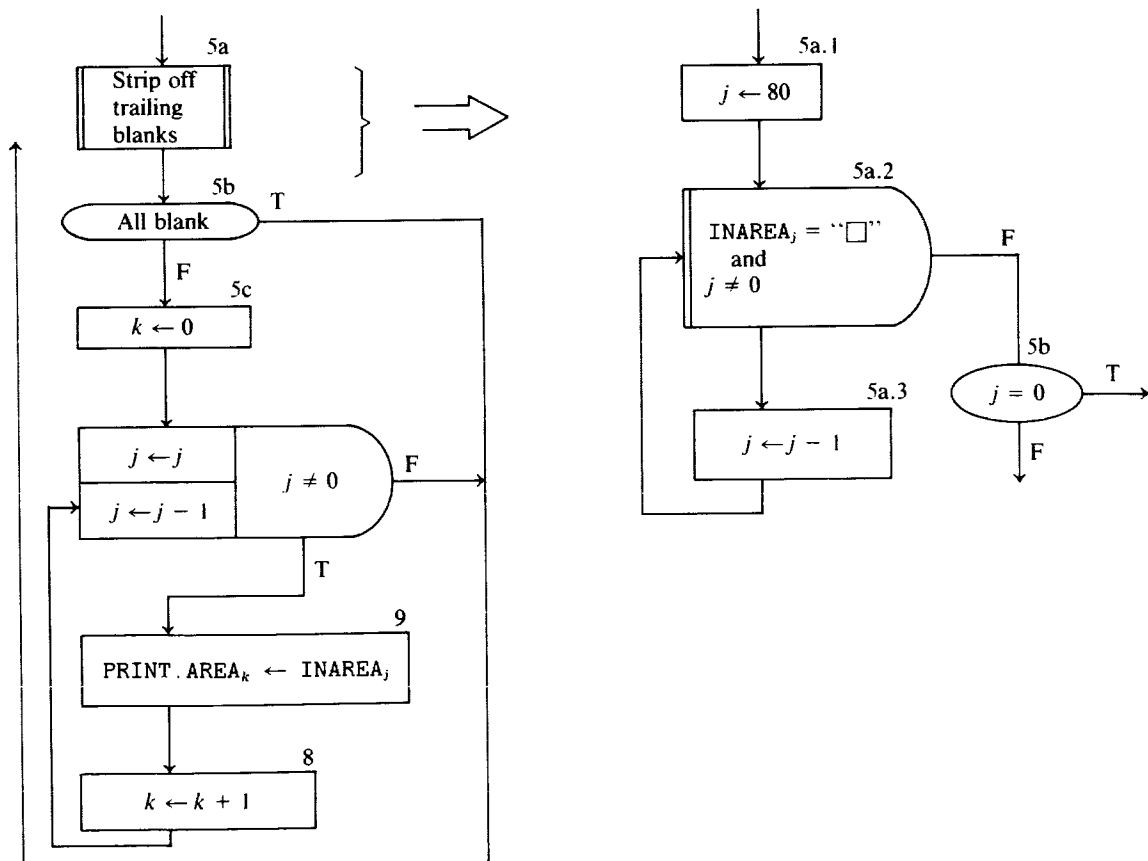


Figure 4.12. Modified logic of inner loop for CARD.INVERT to left justify the output.

The overall structure of `CARD.INVRT` can remain the same. The loop structure of boxes 5 through 8 can be replaced as shown in Figure 4.12 to suggest the logic now required. To solve this problem requires more practice in MIL coding. No new McMIL statements are needed, however. Will any change be needed to the `CARD/INVERTER`, i.e., to the `LOADER` program?

Chapter 5

The structure of an interpreter

An interpreter algorithm, as we discuss it in this chapter, is one that imitates the fetch-execute cycle of a particular von Neumann-style computer. Figure 5.1 suggests the characteristic structure of such an interpreter. Its main feature is a loop, each transit of which corresponds to the fetch and execute of one instruction in the program of the target machine. To be sure, not every interpreter need be structured precisely this way, but this skeleton is sufficiently representative to be instructive. As we discuss this structure we will expand it both top down, by providing more of its details, and also bottom up, by suggesting environment structure in which the interpreter is nested.

5.1 DETECTION AND RESPONSE TO FAULTS AND INTERRUPTS

An interpreter algorithm should be capable of detecting when things go wrong with the program being interpreted (faults). The interpreter can signal the nature of the fault encountered in two ways.

1. By printing explicit messages and shifting control to code in its environment, i.e., to its host;
2. By reflecting to its caller (here we regard the interpreter as a subroutine) the nature of the fault and returning control to its caller, leaving to the caller the responsibility for reacting properly.

The second approach is attractive, and we shall mainly pursue it in subsequent discussions. This approach allows us to keep the size of the interpreter procedure small and at the same time purchase adequate flexibility through modularity.

Normally the designer of an interpreter cannot be expected to recognize in advance all possible faults. Usually he can think of the most obvious ones such as those listed in Table 5.1.

Switches set within the loop's body (details of boxes 3 through 6 of

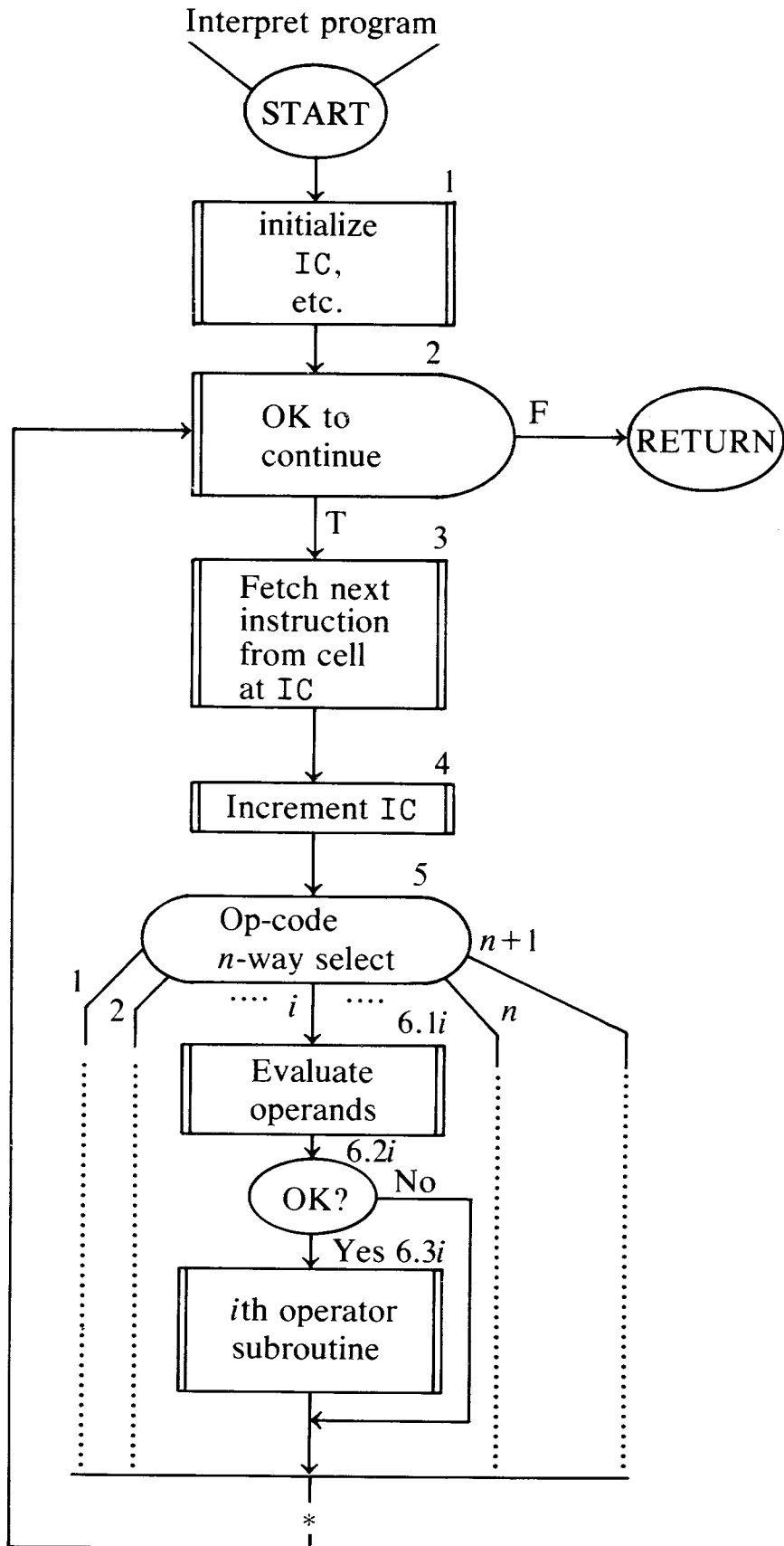


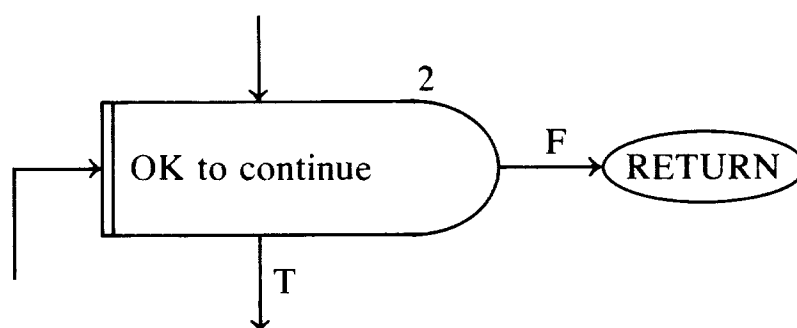
Figure 5.1. Skeletal structure of an interpreter. IC refers to the instruction counter of the target machine. The point marked * is discussed later in the text.

TABLE 5.1 Partial List of Faults to Be Sensed by an Interpreter

FAULT TYPE OR OTHER SPECIAL CONDITION	MECHANISM FOR SENSING	RESPONSE ^a
Runaway computation	Number of interpreted instructions (a work counter value) exceeds some given or declared limit	Abort the program
Invalid opcode	Failure of a table lookup or other search	Abort the program
Invalid operand address	Comparison against storage address limits	Abort the program
Invalid IC value	Comparison against storage address limits	Abort the program
End of file sensed on attempt to execute an input step	System-sensed using the ON EOF option in = BUFFER READ	Possibly terminate the run

^a The easiest response to each fault is to cause the interpretation process to be terminated (abort the program) and possibly give a dump (display storage registers.) More sophisticated responses, such as restarts using a new data set or some given IC value, may be possible. [In this book we shall not emphasize responses.]

Figure 5.1) will be tested in the “interior” of box 2,



to cause return to the interpreter’s host or caller. We shall examine some of these details momentarily.

In addition to faults encountered that are intrinsic to the program being interpreted, there may be other reasons for discontinuing interpretation, at least temporarily. Our interpreter must be responsive to the needs of its host environment. System interrupt signals may arrive while the interpreter is executing its loop body. Some of these signals imply that the host system should respond “immediately”; others are less urgent. The interpreter must periodically pass control back to some system routine which specializes in analyzing and responding to these interrupt conditions. [On the B1700, that specialist routine is called

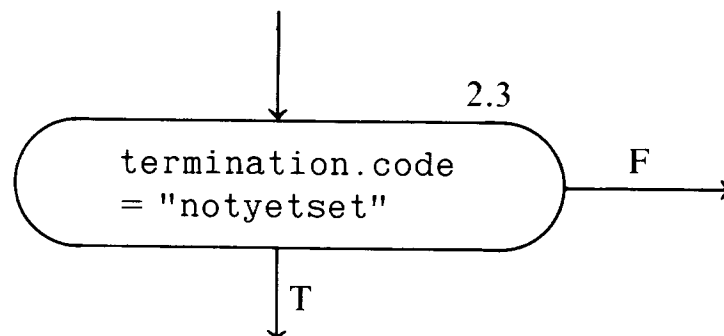
GISMO; it interfaces with various routines of the MCP when a comprehensive response to an interrupt is called for.]

A *courteous* interpreter must not execute too long “at one stretch” without checking to see if such system interrupts have arrived, or else it may be too late for the host system to give proper response. In our perhaps naive first design, we shall assume that such a check need be made only once in each transit of the interpretation loop, as suggested in the details of box 2, given in Figure 5.2.

We should bear in mind that an actual computer usually executes a hardware check for interrupts at the completion of each fetch/execute (instruction) cycle, so by letting our interpreter make a (software) check for interrupts at the corresponding point in its cyclic process, we cause it to mimic an actual machine. If, for certain exit paths from box 5 of the interpreter (Figure 5.1), the total time for executing one transit of the interpretation loop will actually be “discourteous” to the system that hosts this interpreter, then it is the responsibility of the interpreter designer to insert additional checks for interrupts along such “long paths”.

Fortunately it is very easy for a MIL coder to insert a check for interrupts. The McMIL macro call =CHECK INTERRUPTS generates MIL code that calls a SMACK subroutine that checks the state of all physical devices and determines if any system service is required at this time. If so, the scratchpads are saved, and control is passed (in the coroutine sense) to the MCP (via GISMO). Upon resumption of control, scratchpads are restored; but note that register values for X, Y, T, L, CA, CB, FA, FB, and TAS will have been *lost*. The statement immediately following =CHECK INTERRUPTS is then executed. We see, therefore, that the logic of boxes 2.1 and 2.2 of Figure 5.2 is accomplished for us by this single McMIL statement.

The other test in box 2, namely



suggests that we may use one multivalued switch, here named `termination.code` which can be preset in box 1 of the interpreter to a value equivalent to “notyetset”. Interpretation continues as long as the

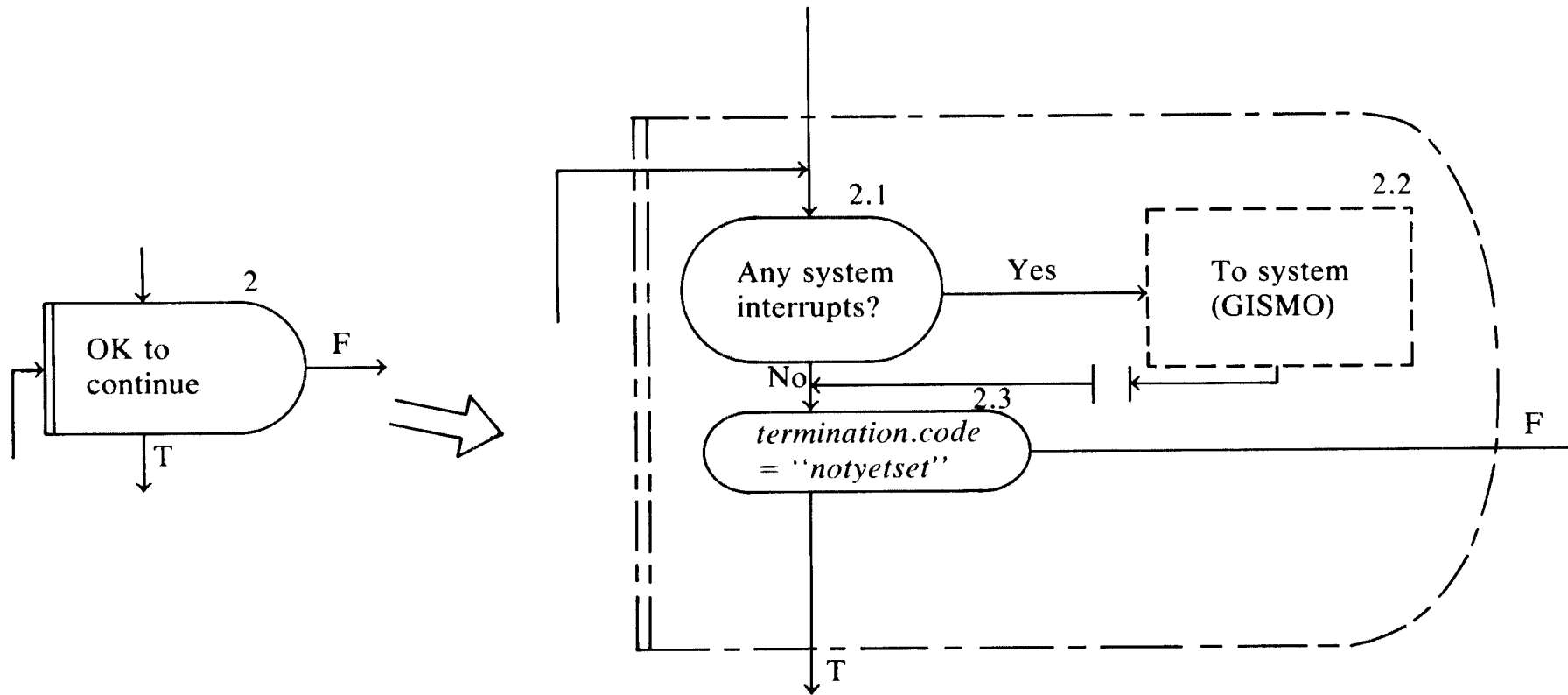
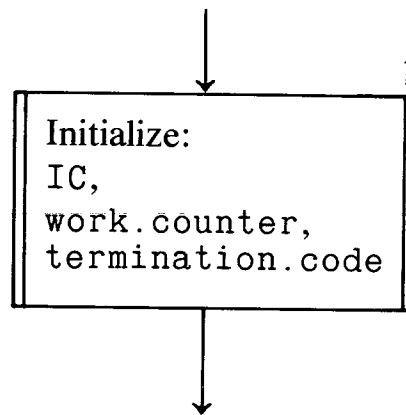


Figure 5.2. Details of box 2. The symbol $\leftarrow | \leftarrow$ means resumption after control is passed back from the system procedures after some delay while servicing interrupts. It is assumed that the variable `termination.code` is initialized in box 1 of Figure 5.1 to the value "notyetset", and may be altered in the body of the loop controlled by box 2.

switch remains in this condition. It is assumed that recognition of any fault such as those in Table 5.1 results in setting `termination.code` to a specific value properly understood by the interpreter's caller. Likewise, any *normal termination* of the interpreted program, such as results from executing a halt instruction or from attempting to read past an end-of-file condition, is also assumed to set `termination.code` to a value meaningful to the interpreter's caller.

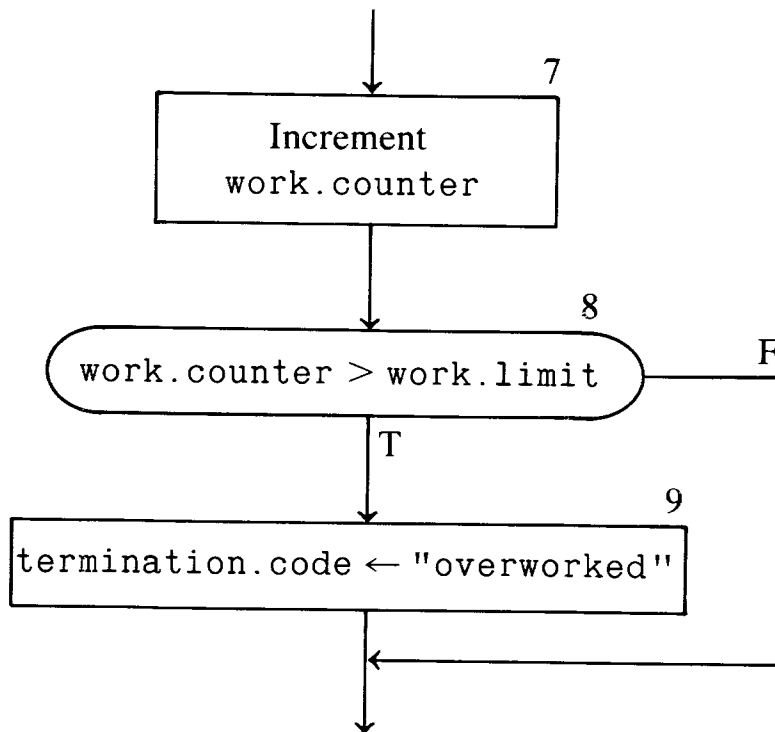
From the above discussion, we see that

1. Box 1 of the interpreter should now include the detail



The variable `work.counter` is to be used as a counter to keep track of the number of instructions that have been interpreted.

2. The point marked by * in Figure 5.1 [prior to return (looping back to box 2)] should now be represented by the additional steps



3. Various places within the interior of boxes 3, 4, 5, and 6 may (should) include tests for fault or termination conditions that, when encountered, result in the setting of `termination.code` to an appropriate value.

5.2 THE HOST ENVIRONMENT

Before looking further into details of the interpreter's loop body, one should consider the choices available for possible environments in which to embed the interpreter structure. Basically, there are two choices.

1. Embed the interpreter directly in the MCP (the MCP treats the interpreter as a coroutine and spawns it directly).
2. Embed the interpreter within an outer *shell*, which in turn is embedded directly in the MCP (the MCP treats the shell as a coroutine and the shell communicates with the interpreter in some appropriate manner—e.g., as a coroutine or as a subordinate procedure).

We shall consider the ramifications of each choice.

If we opt for choice 1, we will be using the interpreter much as the B1700 designers intended. Namely, each time the MCP passes control to an interpreter, there already exists in the compiled run structure some code that is ready to be interpreted. For interpreters like UPL, FORTRAN, COBOL, etc., that code is a compiled *user program*, originating from UPL, COBOL, or FORTRAN source text in the respective language. A new codefile is needed for each such user program that is to be interpreted. However, for interpreters which are machine simulators, the code that is initially resident in the run structure at the start of interpretation may be either a user program or a system program. Let us consider each case in turn using the PDP-9 simulator for illustration.

If we want the simulator to interpret only one PDP-9 program (e.g., a PDP-9 *user program*), then it is sufficient to compile into the run structure a copy of that user code, and this code will be interpreted by the simulator. When interpretation is complete, control will return to the MCP. To interpret another PDP-9 user program would then require a new codefile be created.

More than likely, however, we will want our simulator to be capable of interpreting a series of PDP-9 user programs. For this purpose code initially compiled into the run structure should be one or more PDP-9 *system programs*. The emulator begins execution of these programs, which in turn cause the loading of still other PDP-9 programs, e.g., user programs. In fact, all we would need to preload into the run structure is

a small PDP-9-coded self-loader. Of course, the run structure must be large enough to represent the storage space for the full PDP-9 simulated storage, so that PDP-9 programs much larger than the self-loader will also fit in.

Most of the early general-purpose digital computers, especially those built before modern read-only memories were available, were designed for use with self-loader programs in mind. The self-loader was itself loaded by a simple hardware circuit. When activated, this circuit would read one data record containing the self-loader code into a preset (fixed) base address of storage, set the instruction counter to this base address, and commence execution.

Here is a further ramification of choice 1 to embed the interpreter directly in the MCP. The MCP will not know anything about *our* interpreter (e.g., the PDP-9 simulator). Therefore, to handle PDP-9 program faults, it will be the interpreter's responsibility to supply specific calls on the MCP (*i/o* requests coded as McMIL E-statements) which spell out precisely what and how error messages, dumps, etc. are to be displayed. Such *i/o* requests will have to be completed before the interpreter can pass control back to the MCP for the purpose of terminating the execution.

If we opt for choice 2, an interpreter within an outer shell, we have somewhat greater flexibility (at least for simulators) at somewhat added cost. The shell, also coded in MIL, can provide the structure of an environment tailored for the machine we want to simulate. With the advent of integrated-circuit technology, many operating-system features, (e.g., self-loaders and editors) are being built into the hardware and/or read-only firmware. For example, in the case of the SAMOS machine the loader function (described in Appendix F) is regarded as built into the hardware. Any number of such built-in features can be simulated by coding them into the shell. In the next section we elaborate further on implementation using a shell. To make the discussion more concrete we shall assume, with little loss of generality, that the shell being designed is for the SAMOS computer.

5.3 THE SHELL CONCEPT

Figure 5.3 suggests the structure of a simple shell which has some very useful properties. This shell calls on the interpreter (box 7 of Figure 5.3) only when it has successfully copied a complete (SAMOS) program from the input card file into the simulated (SAMOS) storage located within the workspace of the run structure. Whenever the interpreter returns control to the shell, the latter is able to respond intelligently to the `termination.code` reflected back to it by the interpreter, and

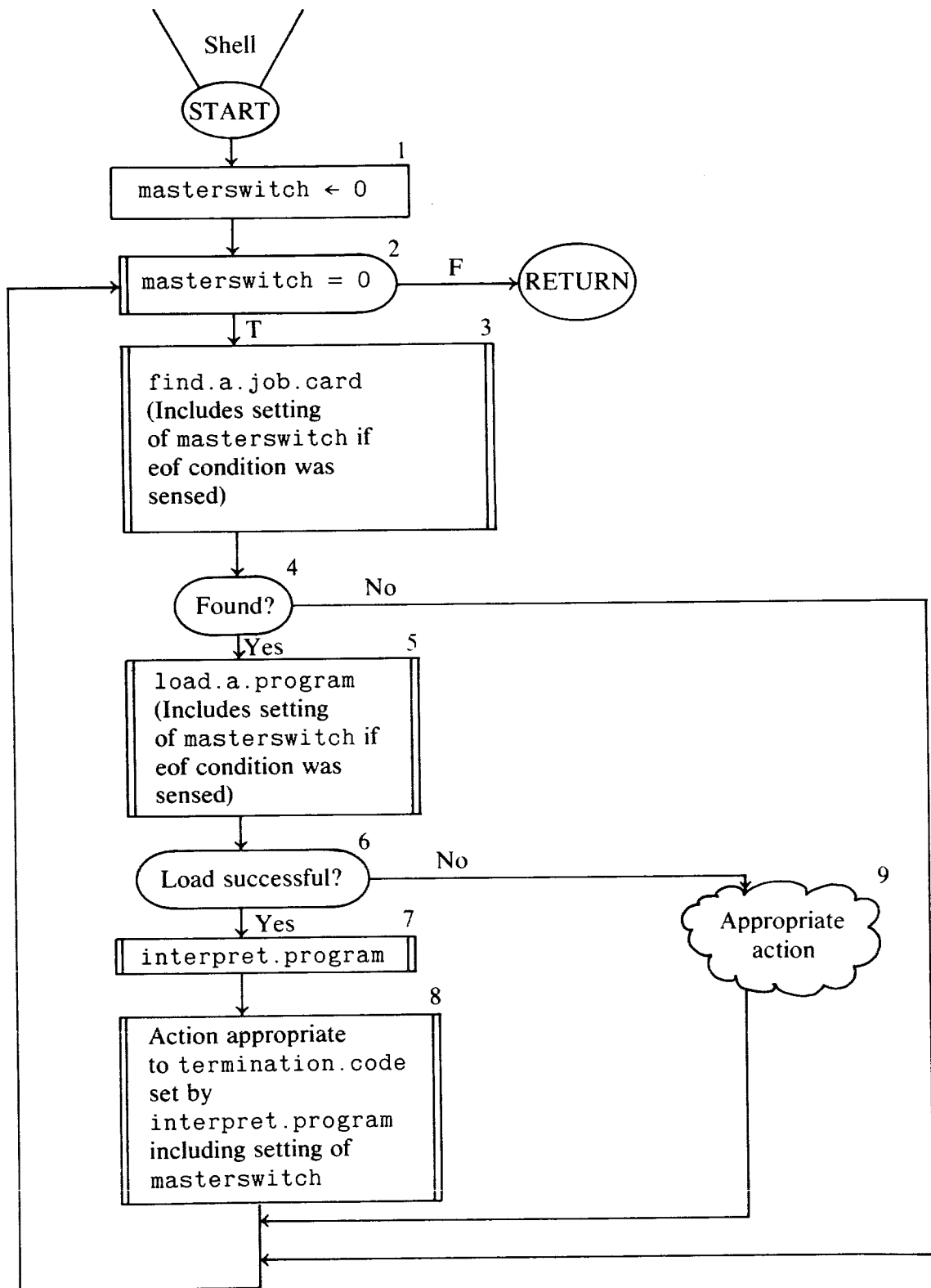


Figure 5.3. First view of a possible shell for an interpreter. This shell behaves like a simple batch operating system that processes jobs from a single input file and halts when this file is exhausted.

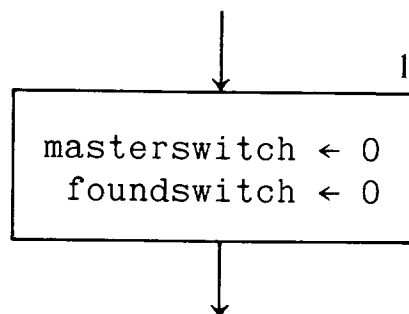
following this response, process another (and another) SAMOS “job” from the input file in a similar way. (No new codefile need be prepared for the new job.)

The particular shell of Figure 5.3 behaves like a simple batch operating system for the SAMOS computer. The shell is programmed under the assumption that the input file is a sequence of jobs, each headed by a control card (identified by * in column 1, for example).

When the end-of-file indicator on the input file is sensed, the master-switch that controls the outermost loop is set to force a return to the shell’s environment (in this case to the MCP). Any of the subprocedures called by the shell (box 3, 5, or 7) can sense the end-of-file (eof) condition. In case `interpret.program` senses the eof condition, the code in box 8 can set the masterswitch.

We are persuaded here to add further detail for the shell. For example, Figures 5.4 and 5.5 define possible structures for the `find.a.job.card` and `load.a.program` procedures.

The `find.a.job.card` procedure is basically a search loop, looking for a card image corresponding to a control card (* card). If, during the search, an end-of-file condition is reached, the masterswitch is set (box 4). When a * card is found, the foundswitch is set to 1 to force exit from the loop at box 1. This switch may be reset upon return to the shell proper, which must decide (box 4 of Figure 5.3) which condition caused control to return from `find.a.job.card`. The details of box 4 are shown in Figure 5.6. The very first time `find.a.job.card` is called, we must guarantee that the foundswitch is in the reset position. This requirement is satisfied simply by initializing the foundswitch in box 1 of the shell, i.e., by revising it to



The logic of `load.a.program` is clear from inspecting the top-level description on the left of Figure 5.5. The details for box 2 and box 3 of this description will depend on the particular machine being simulated. Even so, the details given in Figure 5.5 are as independent of the particular target machine as we know how to make them. We have in mind that after clearing the registers and storage of the target machine

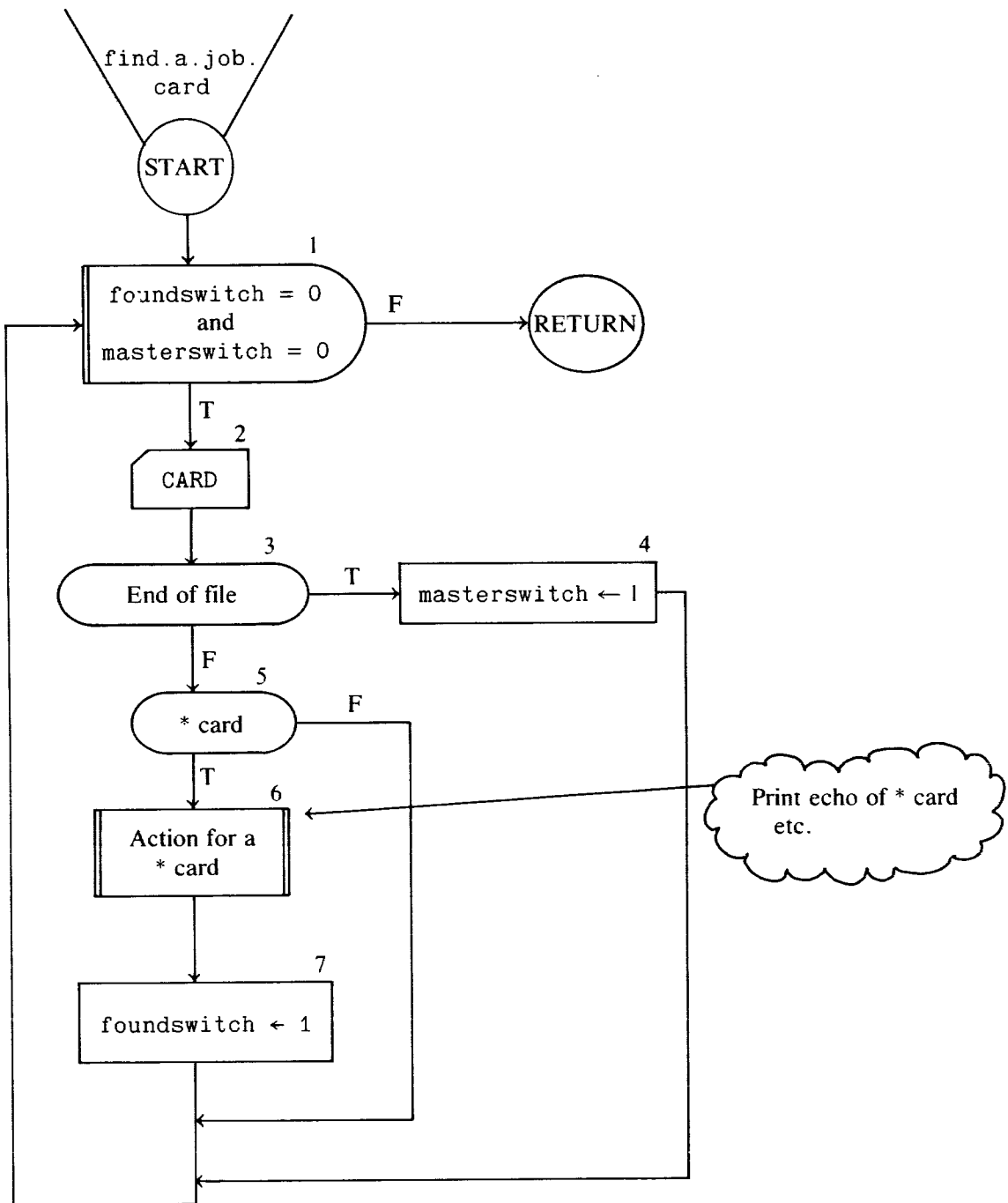


Figure 5.4. A possible structure for `find.a.job.card`.

(or after initializing these cells to some value representing “undefined value”), the procedure would read a sequence of cards (actually *program cards*), each containing one (or more) target-machine instructions, which are to be moved into consecutive cells of the simulated storage. The addresses of these cells are governed in this case by the `location.counter`, set initially to zero. [Actually, any other starting

value for the `location.counter` would do if the starting value were specified as an input parameter of the procedure.]

The end of the program-card sequence is sensed by recognizing some kind of sentinel card, e.g., a blank card, as in the case of SAMOS. When all program cards have been processed, loading is completed, and this fact is reflected back to the shell by recording an "OK" value for the indicator variable, `code`. If, prior to sensing the sentinel card, a control

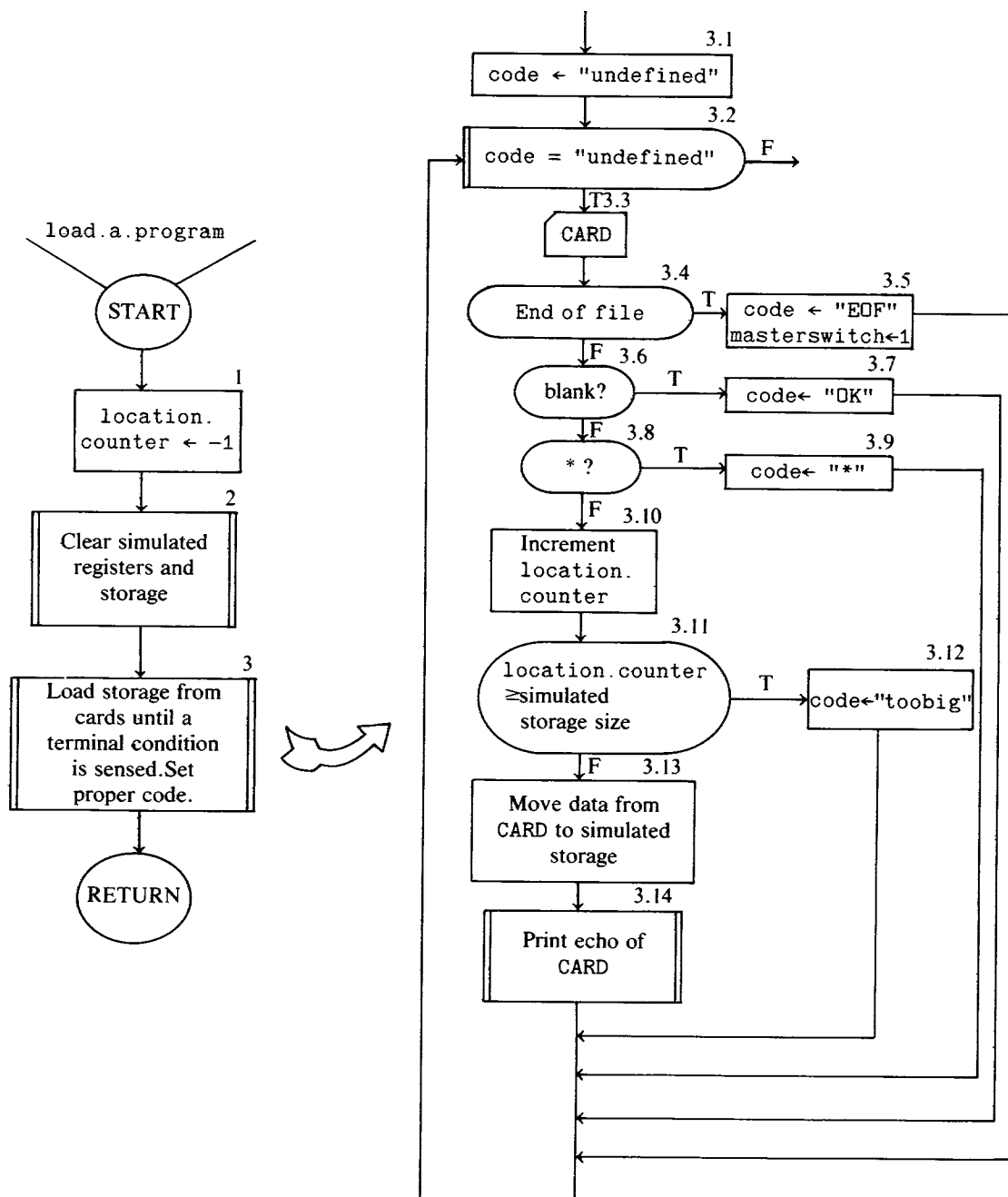


Figure 5.5. A possible structure for `load.a.program`.

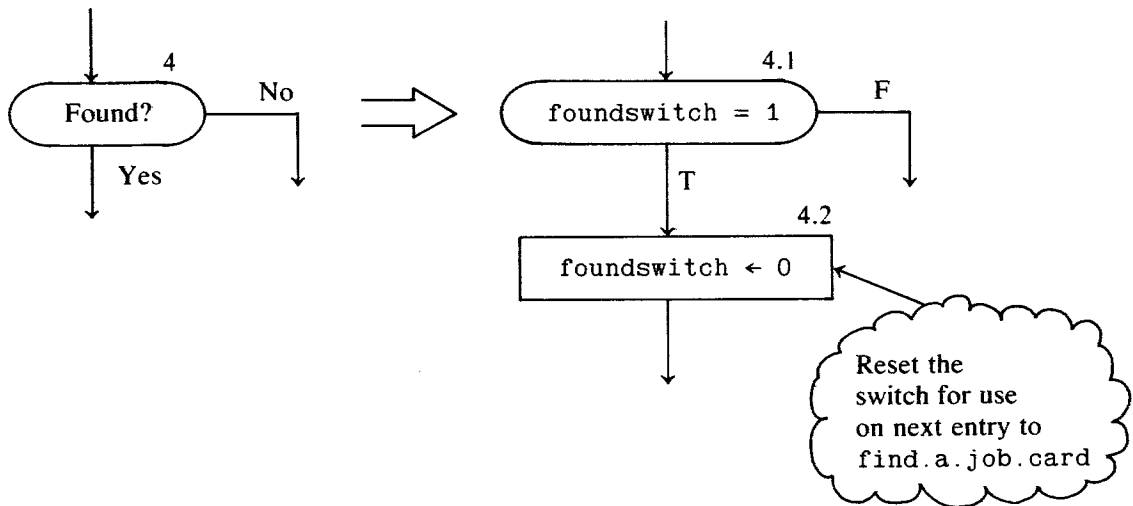


Figure 5.6.

card such as the * card is sensed, the program deck is deemed incomplete; the loading process is regarded as a failure, and this fact is reflected back to the shell by assigning the "*" value to code. Likewise, if the end-of-file condition is sensed before the expected sentinel is reached, that condition must be reflected back to the shell also.

Another condition which should be sensed to denote failure of the loading process is an attempt to load a program which cannot fit into storage of the simulated (target) machine. This condition is easily detected (box 3.11 of Figure 5.5), and the value "TOOBIG" is reflected back. We have now motivated all the details of boxes 6 and 9 of the shell that are suggested in Figure 5.7.

Now that we have considered one possible control structure for the shell of an interpreter, you can probably improve on it or embellish it to achieve one of a series of other objectives to make the interpreter's *human interface* more effective for your purposes. But before you begin making improvements, it is a good idea to see how well you can code this shell in MIL. You'll have to make some additional design choices, depending on the particular computer you decide to simulate.

When coding the shell in MIL you should recall the admonition in Section 4.5 that MIL subroutines cannot execute i/o steps directly (i.e., cannot execute SMACK macro calls for i/o). Therefore, routines like `find.a.job.card`, `load.a.program`, and `interpret.program` must either be reached by GO T0s or be coded such that only the shell issues these macro calls. Rather than pursue this approach here (MIL-coding the shell), we will instead return to further consideration of the

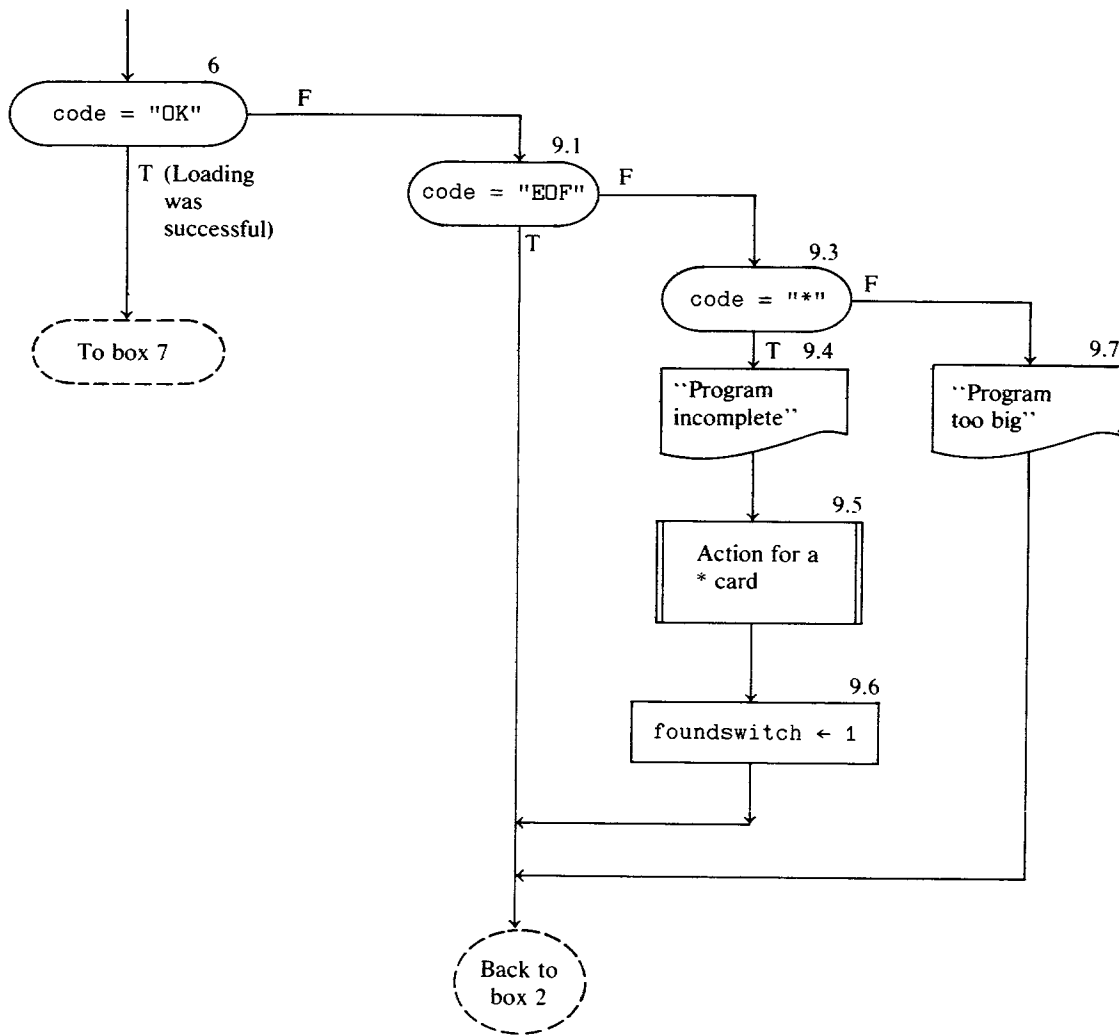


Figure 5.7. Details for boxes 6 and 9 of the shell.

interpret.program procedure, assuming it is called by a shell like the one in Figure 5.3.

5.4 MOVING TOP DOWN ON THE INTERPRETER STRUCTURE

To flesh out with further details the interpreter skeleton described in Section 5.1, we will now have to make some specific design choices. In particular, we will need to notice more and more of the properties of the target machine as we descend to levels of greater detail. In the context of the typical simulation project, the target machine and its full definition is known at the outset. We will therefore assume this context and select the SAMOS machine as our target from here on, although we will try to keep our discussion as general as possible. By way of review, we gather up some loose ends and present in Figure 5.8 an updated version of Figure 5.1 based on the discussions in Section 5.1.

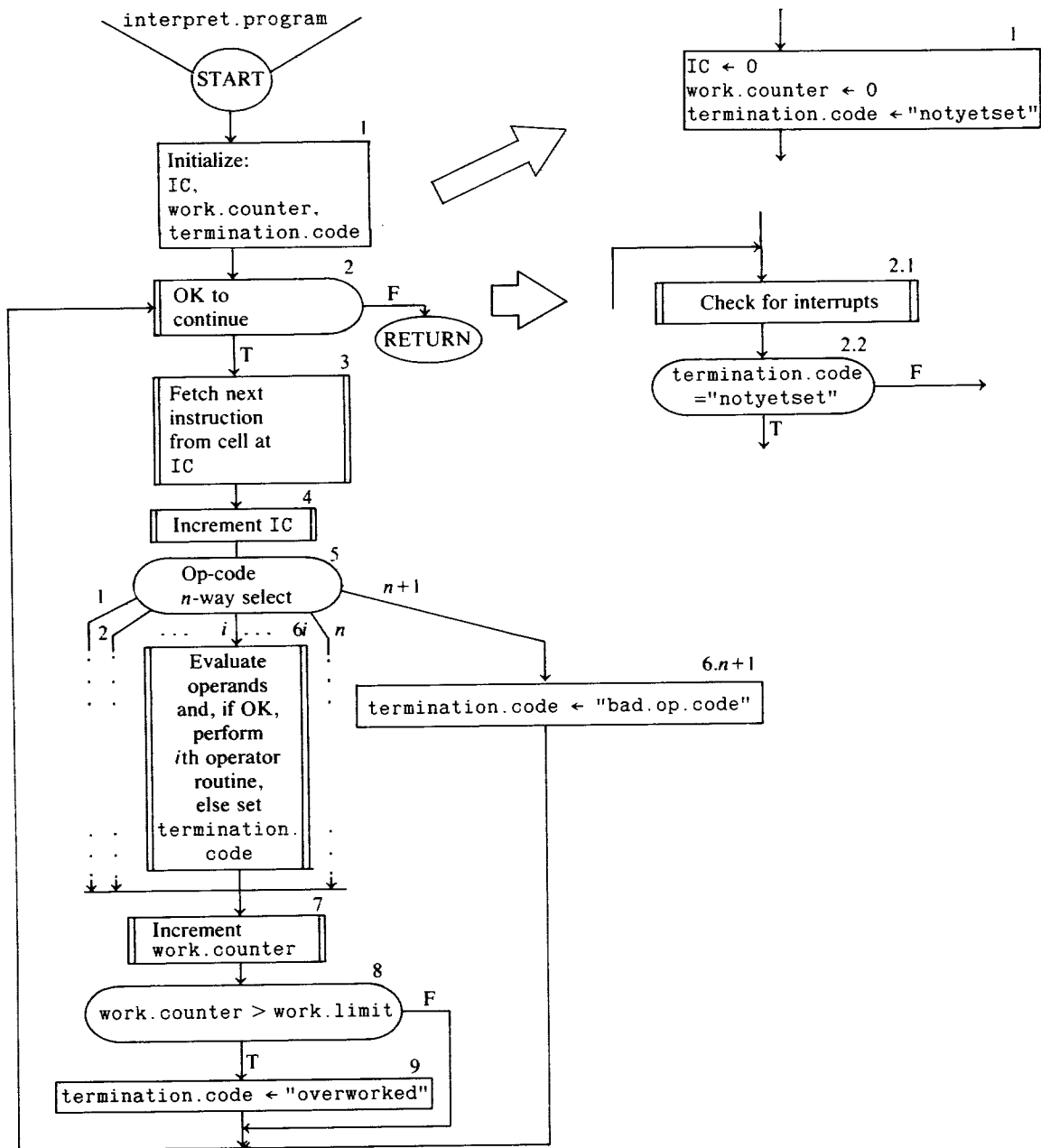


Figure 5.8. Updated interpreter skeleton.

5.4.1 Storage representation for the target machine

To focus on the details of boxes 3, 5, and 6 we must decide how to represent the SAMOS registers and storage in G-store. Recall that each SAMOS word consists of a sign position followed by 10 character positions. The SAMOS machine can be emulated from its original description which specified 61-bit words (1 bit for sign and 6 bits for each character.) We would therefore prefer to emulate each SAMOS word as a 61-bit field in G-store; but if we do this, we will miss the opportunity to exploit certain hardware features of the B1700's micro

processor which explicitly support the processing of 8-bit EBCDIC character codes and do not support the processing of (say) 6-bit BCD codes or 7-bit ASCII codes, etc. (Despite advertisements to the contrary, 8-bit EBCDIC characters are favored over other character coding on the “protean” B1700.) We are then faced with this tradeoff (dilemma): If we want to take advantage of the character processing potential of our B1700 microprocessor, we will have to waste G-store by representing each SAMOS word as 11 EBCDIC characters ($8 \times 11 = 88$ bits, rather than 61 bits as originally specified). Either way we go will be instructive here. It isn’t critical that we make a choice between these two options at this point in our top-down approach, because only the details of certain declarations and utility subroutines are affected. Nevertheless it will be convenient for this exercise if we assume we are going to opt for the second approach (8-bit EBCDIC character codes), so we can eventually illustrate some of the character-processing features of the B1700.

The first implication for the above choice is that the SAMOS store of SIZE words may be declared as

```
DEFINE SIZE = <a value chosen by the designer, e.g. 100> #
DECLARE 01 SAMOS.STORE(SIZE),
        02 WORD BIT(88),
        03 SIGN CHARACTER(1),
        03 OPCODE CHARACTER(3),
        03 INDEXES CHARACTER(3), % SEE
                                     % FOOTNOTE
        03 ADDRESS CHARACTER(4);
```

To refer to each of the individual index-register subfields by a unique name, we may redeclare SAMOS.STORE using the REMAPS feature, e.g.,

```
DECLARE 01 DUMMY REMAPS SAMOS.STORE, BIT(88),
        02 FILLER BIT(88),
        03 FILLER CHARACTER(4),
        03 INDEX1 CHARACTER(1),
        03 INDEX2 CHARACTER(1),
        03 INDEX3 CHARACTER(1);
```

How should the registers of the SAMOS processor be represented in G-store? As separately named fields? Perhaps, but an attractive alternative is to treat each register the same as an ordinary word of SAMOS storage, letting these registers constitute an extension to the SAMOS

store with registers having *negative* addresses, e.g.,

- 1 for the accumulator, ACC
 - 2 for the instruction counter, IC
 - 3 for the index registers, IX1,
 - 4 IX2,
 - 5 IX3,
- etc. (any pseudo registers we need can go here)

Presently, we will see how this alternative way of addressing these registers can prove useful. Storage allocation for these registers can be made contiguous with the base of SAMOS.STORE by having the DECLAREs for these registers immediately precede the DECLARE for SAMOS.STORE, e.g.,

```
DECLARE (IX3, IX2, IX1, IC, ACC) BIT(88);
DECLARE 01 SAMOS.STORE(SIZE),
etc. as before.
```

Letting all SAMOS registers and storage words have the same G-storage structure means that all arithmetic operations on them can be performed by the same set of decimal arithmetic operations. Binary arithmetic will be used mainly to convert SAMOS locations (decimal numbers) to the absolute binary G-store addresses needed to access SAMOS registers and storage words.

In short, we may use (B1700 4-bit) decimal arithmetic for simulating SAMOS address calculations that involve the instruction counter, address fields, and index registers. [We can of course also use B1700 decimal arithmetic for simulating the decimal arithmetic used by SAMOS for calculations involving the accumulator.]

One mapping rule suffices to compute the absolute G-store address of a SAMOS storage word or register. That rule is

$$\text{map}(s) = s \times 88 + \text{SAMOS.ZERO}$$

Here s is the SAMOS location and SAMOS.ZERO is the absolute G-store address of SAMOS location zero (0000). The value of s will be a small negative integer if it represents a SAMOS register (or pseudo register) and will be a nonnegative integer less than SIZE if it represents a SAMOS storage location. Note the following points.

1. SAMOS.ZERO may be computed as the sum of SAMOS.STORE and BR, i.e.,

$$\text{map}(s) = s \times 88 + (\text{SAMOS.STORE} + \text{BR})$$

Since the value of BR can change whenever control is handed over to the MCP via GISMO (as during an interrupt check), it is necessary to recompute the sum `SAMOS.STORE + BR` during each transit of the main loop of interpret program (Figure 5.8). But this sum can then be kept in a scratchpad for the duration of that transit.

2. Since `map(s)` is an absolute G-store address, which is a binary number, it makes sense to compute it using binary arithmetic. Hence `s` in the above formula should be the binary equivalent of the (decimal) SAMOS storage location.

SAMOS register locations, unlike SAMOS *storage* locations, are represented only implicitly in the SAMOS instruction, so it is never necessary to represent the location `s` of a SAMOS register as a decimal character string. This means, for example, that when we want to access the accumulator, we need only use the declared binary literal `(-1)` equivalent of the accumulator. There is no need to convert a decimal character number to binary integer before computing `map(s)` by the above formula. We can declare these literals by using `DEFINEs` such as

```
DEFINE ACC.ADDR = -1#      % AS 2'S COMPLEMENT
DEFINE IC.ADDR  = -2#      % AS 2'S COMPLEMENT
DEFINE IX1.ADDR = -3#      % AS 2'S COMPLEMENT
etc.
```

or

```
DEFINE ACC.ADDR = @800001@#
                                     %-1 AS SIGNED MAGNITUDE
DEFINE  IC.ADDR = @800002@#
                                     %-2 AS SIGNED MAGNITUDE
DEFINE IX1.ADDR = @800003@#
                                     %-3 AS SIGNED MAGNITUDE
```

But let us take a closer look at the problem we encounter when we need to access a SAMOS storage word whose location is determined from its *explicit representation* in an *instruction*, for example, such as `STO 000 0051`. Here the address 0051 is represented as a string of decimal characters whose G-store representation is

```
1111 0000 : 1111 0000 : 1111 0101 : 1111 0001 ← Bit string
  F  0  :   F  0  :   F  5  :   F  1  ← Hex char string
```

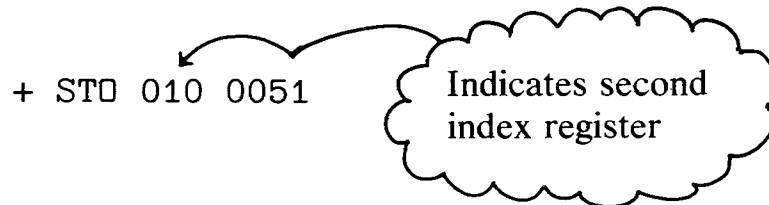
We need to convert this value to the binary integer, $s \equiv 110011$, so we'll

TABLE 5.2 Possible Utility Routines Useful in the SAMOS Interpreter

NAME AND FUNCTION	SPECIFICATIONS
VALIDATE.DECIMAL checks a SAMOS storage location for +, -, and decimal characters.	<i>Parameter:</i> Binary absolute G-store address, <i>s</i> , of SAMOS location containing word to be tested is in FA. <i>Return:</i> Flag telling if input parameter points to a valid 11-character decimal number. The first character should be "+" or "-". Each of the remaining 10 characters should be a decimal digit.
ADDRESS.TO.BINARY converts a 4-character decimal address field to binary and checks that the result is a valid SAMOS storage location, i.e., that $0 \leq \text{result} < \text{SIZE}$.	<i>Parameter:</i> Binary address, <i>s</i> , of SAMOS location for a storage word or register containing a decimal character address. <i>Returns:</i> a. The binary value, <i>s</i> , equivalent to decimal value pointed to by the parameter. The value <i>s</i> is left in an agreed upon register. b. Flag telling if the location specified by the parameter is a valid SAMOS storage address in the range 0 to SIZE.
BINARY.TO.FA converts the binary value, <i>s</i> , representing a SAMOS storage location to the absolute G-store address of that storage location.	<i>Parameter:</i> Binary address, <i>s</i> , of a SAMOS location. <i>Returns:</i> The corresponding absolute G-store address of that location in FA.
EFFECTIVE.ADDR computes the effective address as a decimal character string of a SAMOS operand or instruction.	<i>Parameter:</i> FA points to the index register subfield, INDEX, of the SAMOS instruction being interpreted. <i>Returns:</i> The effective address as a decimal value is left in the pseudo SAMOS register (e.g., EA) equivalent to location -6.
ADD adds two decimal character values.	<i>Parameters:</i> Two binary addresses, <i>s1</i> and <i>s2</i> , of SAMOS locations holding the operands. <i>Returns:</i> The sum (as a decimal character value) stored in the location indicated by second parameter.
SUB subtracts two decimal character values.	<i>Parameters:</i> Same as for ADD <i>Returns:</i> The difference (as a decimal character value) stored in the location indicated by the second parameter.
COPY.WORD copies a word from one SAMOS location to another.	<i>Parameters:</i> Two binary addresses, source and sink, of SAMOS locations. <i>Returns:</i> Nothing.

certainly need a subroutine to convert decimal character strings to integers.

Of course, each SAMOS operand fetch or store is more complex than this. In general what is wanted is an *effective address* for the operand. Thus in the instruction



we want the sum of 0051 and the contents of IX2. So before converting the character string 0051 to binary, it might be worth while to compute the required sum using a decimal-arithmetic addition routine. We anticipate the need for a subroutine (we might name it *EFFECTIVE.ADDR*) which would draw on several more primitive utility routines to deliver the required binary value *s*. Table 5.2 offers a possible set of these utility routines and their functional descriptions. Take a few minutes to study these specifications. You may wish to modify or improve them before seriously beginning to develop flowcharts and MIL code for them.

We can now begin detailing boxes 3, 5, and 6 of Figure 5.8. Consider box 3 first and the suggested expansion of its detail as seen in Figure 5.9.

We note from the plan in Figure 5.8 that there has as yet been no check made that the IC, incremented in box 4 while executing in the preceding loop transit, is within bounds. So this should be the first step in the interior of box 3; if the answer is no, we assign "out of bounds IC" as the new value for *termination.code*, to force a return to the shell. These details are given in boxes 3.1 and 3.2 of Figure 5.9. Next, the absolute G-store address of the instruction pointed to by IC must be computed as an FA pointer for fetching parts of the next instruction from G-store. To mimic SAMOS as it ignores the sign position of an instruction word, we simply "advance" FA by one character (bump FA by 8 bits). This repositions FA at the low-order address of the 3-character op-code field, which is then moved to a receiver register (X) within the microprocessor.

By taking advantage of the utility routines suggested in Table 5.2, it is easy to see how we could write the MIL code corresponding to boxes 3.1 through 3.4 of Figure 5.8. We give such code in Figure 5.10. This

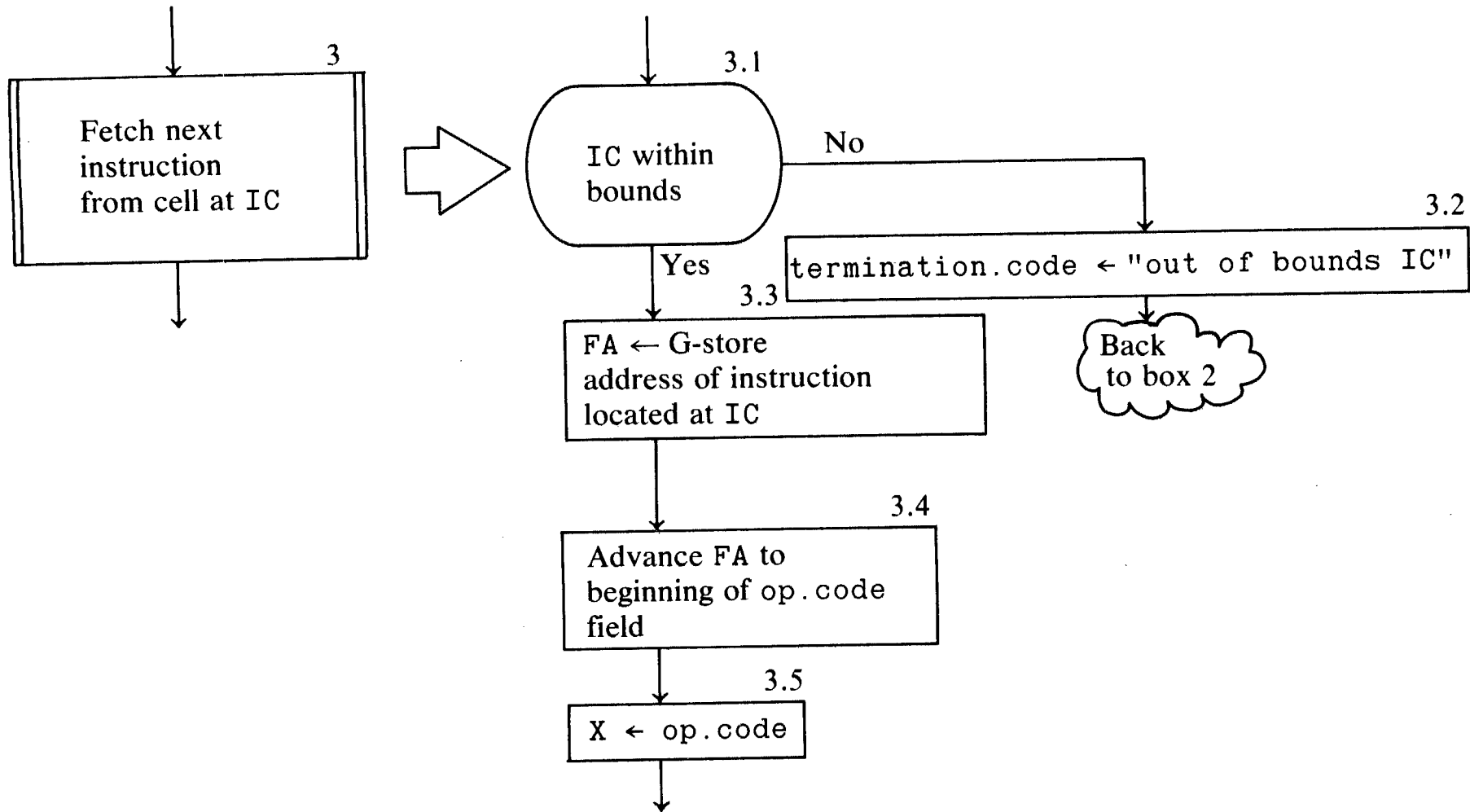


Figure 5.9. First level of detail for instruction fetch.

```

FETCH.NEXT.INSTRUCTION
  MOVE IC.ADDR TO T           % WHERE ARGUMENT FOR
  CALL ADDRESS.TO.BINARY     % THIS ROUTINE IS EXPECTED
  IF FLAG = 0 THEN
    BEGIN
      MOVE OUT.OF.BOUNDS TO TERMINATION.CODE
      GO TO -BOX2
    END
BOX3.3
  CALL BINARY.TO.FA          % CONVERTS VALUE POINTED AT
                             % BY IC.ADDR TO AN ABSOLUTE
                             % BIT ADDRESS IN G-STORE.
  COUNT FA UP BY 8          % BUMP FA WHICH NOW POINTS
                             % TO BEGINNING OF OPCODE
                             %
  READ 24 BITS TO X INC FA  % GETS OPCODE

```

Figure 5.10. Possible MIL code for box-3 details of Figure 5.9

code is assumed to be within the scope of declarations such as

```

DEFINE IC.ADDR = @800002@ #   % AS MENTIONED
                             % EARLIER
DEFINE FLAG = Y #           % ANY REGISTER OR
                             % BIT THAT CAN BE
                             % TESTED FOR ZERO
DEFINE OUT.OF.BOUNDS = 5 #  % THIS VALUE IS
                             % ARBITRARY
DEFINE TERMINATION.CODE = S10B % ANY SCRATCHPAD
                             % WILL DO

```

Note how useful the proposed ADDRESS.TO.BINARY subroutine turns out to be for us. Nearly every time we need a binary equivalent of a decimal address field at some SAMOS location, we also want a bounds check made. The routine ADDRESS.TO.BINARY does both, leaving a zero value in some flag register which can be checked for zero using a simple IF test. (Zero means out of bounds.) To check that the IC is in bounds, we simply call ADDRESS.TO.BINARY, giving as an argument the binary SAMOS location of the IC, in this case the literal IC.ADDR. The subroutine then converts the address field of the IC from a decimal string to the binary equivalent, checks that this binary value is in bounds, and sets the flag.

Now we can begin to see the advantage of representing the IC as a full SAMOS (11-character field). Had we chosen to represent IC as a 4-character field, we would have needed a separate routine to check the IC for a bounds error.

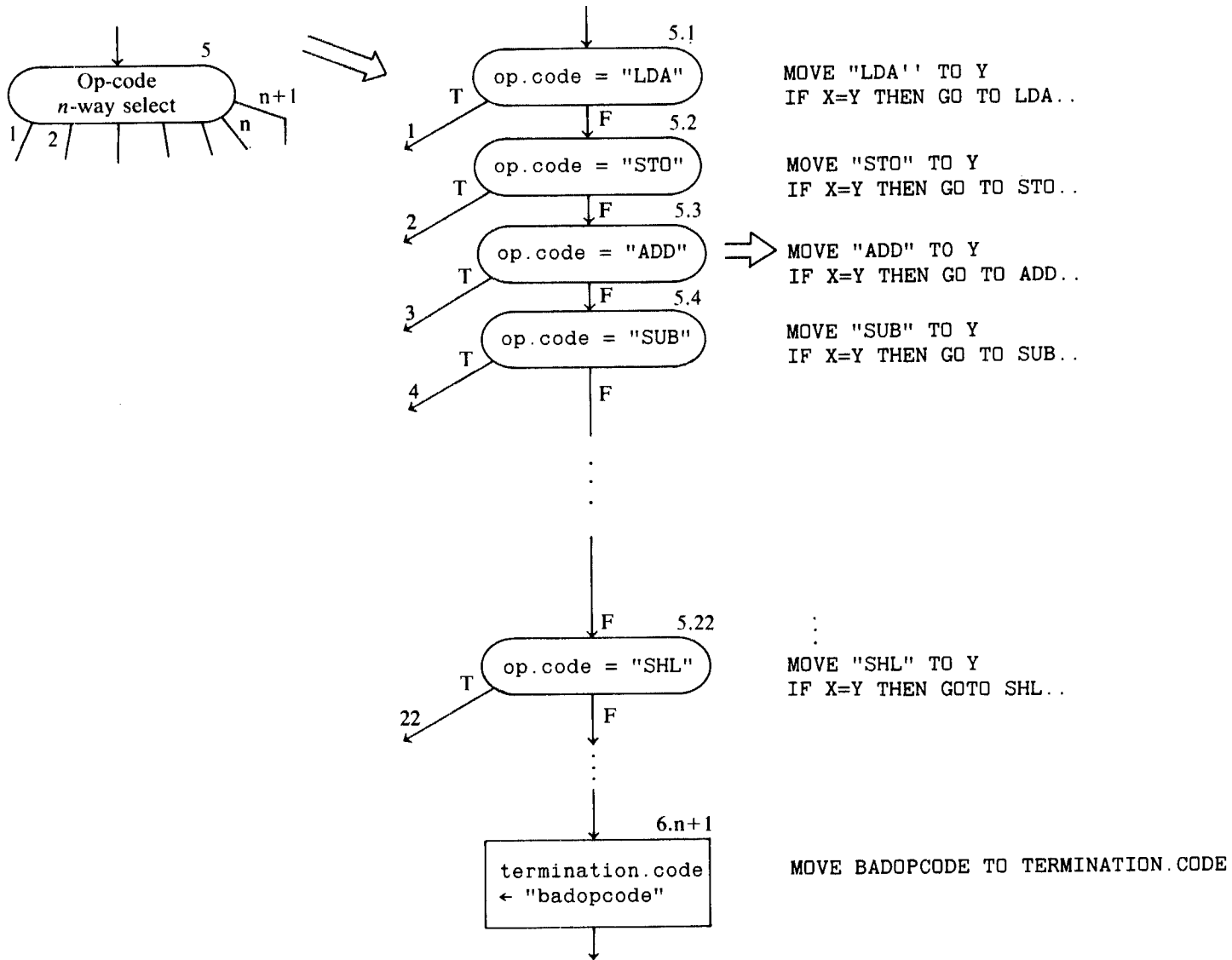


Figure 5.11. Multiway branch to n different opcode routines.

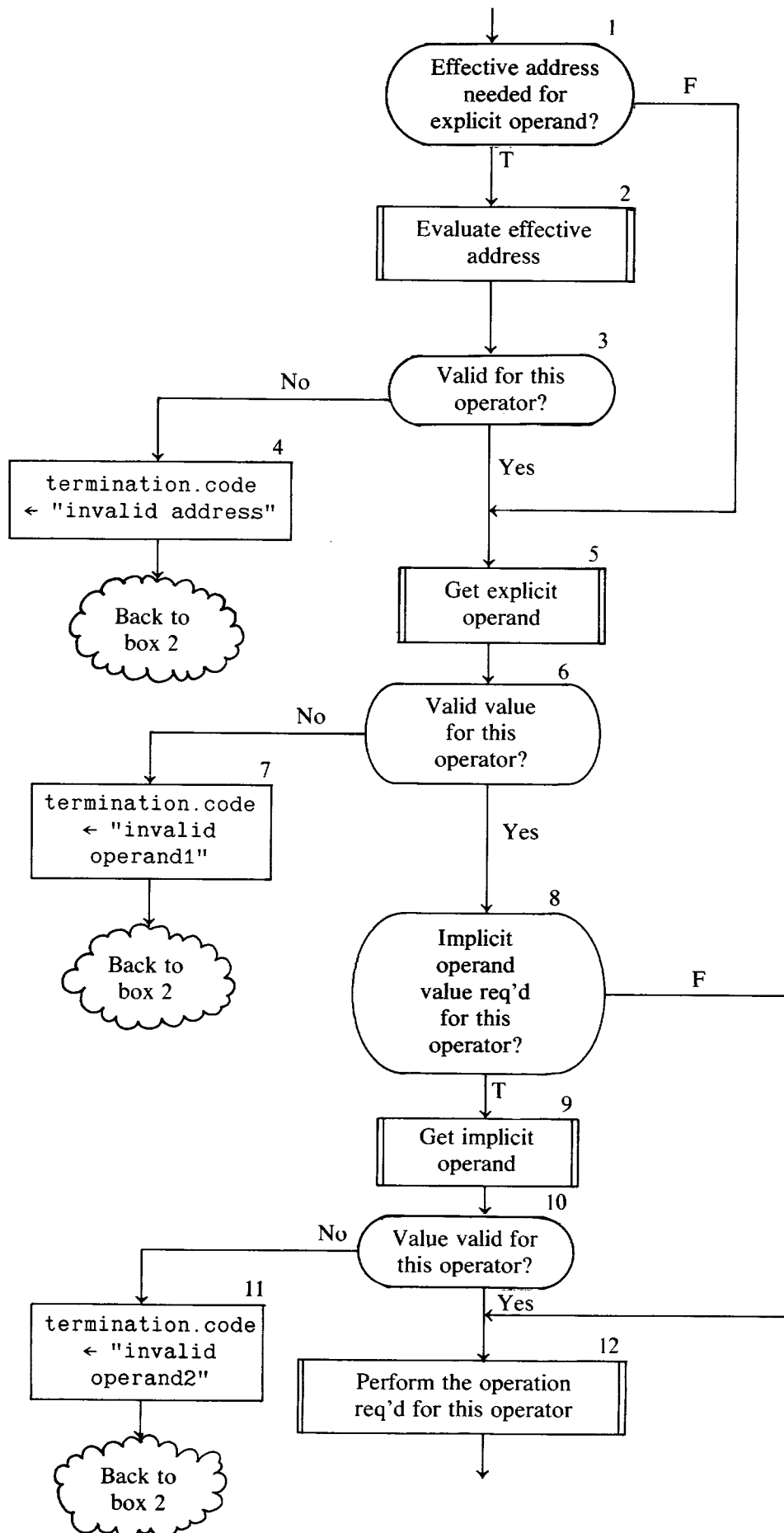


Figure 5.12. Guide for structuring details of box 6.i, the *i*th operator routine for interpret.program

Box 5 of `interpret.program` is an n -way select, where n is the number of distinct SAMOS opcodes. This multiway selection must be coded the “hard way”, i.e., as a sequence of n 2-way tests, as suggested in Figure 5.11. (Unfortunately, we can think of no simpler method for achieving a quick jump to the required operator routine. The indexed jump discussed in Section 3.4 is unfortunately not applicable here, because SAMOS op-codes are not small binary numbers but 3-character—i.e., 24-bit—fields.)

Each op-code routine requires that an explicit operand be “evaluated”. In SAMOS instructions, only one operand is given explicitly; the other, if present, is implicitly designated by the op-code. An index register indicator can be thought of as an explicit operand, but we prefer to regard it as a modifier for the one explicit operand. For example, in the instruction `LDA 010 0051`, the explicit operand is `0051`. Its modifier is `IX2`, and the implicit operand is the accumulator.

For some op-codes only one operand (the explicit operand) need be checked for validity after it is “evaluated”. For others both operands must be checked before the operations indicated by the op-code can be performed. In interpreting `LDA 010 0051`, for example, first the effective address must be found to be valid. Then the value of the operand at the effective address must be checked to be sure it is a decimal character string. Likewise, the accumulator (implicit operand) must be checked to be sure that it too contains a decimal character string. On the other hand, in interpreting `BRU 001 0016`, only the effective address (explicit operand) need be checked for validity (within bounds). The implicit operand, which is the instruction counter, requires no check for validity, since we use it here as a destination rather than a source.

Because of the wide variations in logic required in coding each operator routine, it is difficult to arrive at a prototypical one. The best we can do is offer Figure 5.12, which is intended as a guide (rather than a template) and is intended to be helpful in constructing each of the specific operator-routine flowcharts.

With Figure 5.12 as a guide, we next show in Figure 5.13 how the flowchart would look for an `ADD` instruction. Figure 5.14 shows an equivalent MIL coding for this flowchart.

5.4.2 Discussion of MIL code for `ADD` routine (Figure 5.13)

This figure merits study. Observe that the code is remarkably compact—hardly more than one line of MIL code per flowchart box. Why is this so? Largely because of our choice of the “utility” routines, which do most of the work (and which perhaps do too much work.)

Even more compact code might be obtained using macros. We could

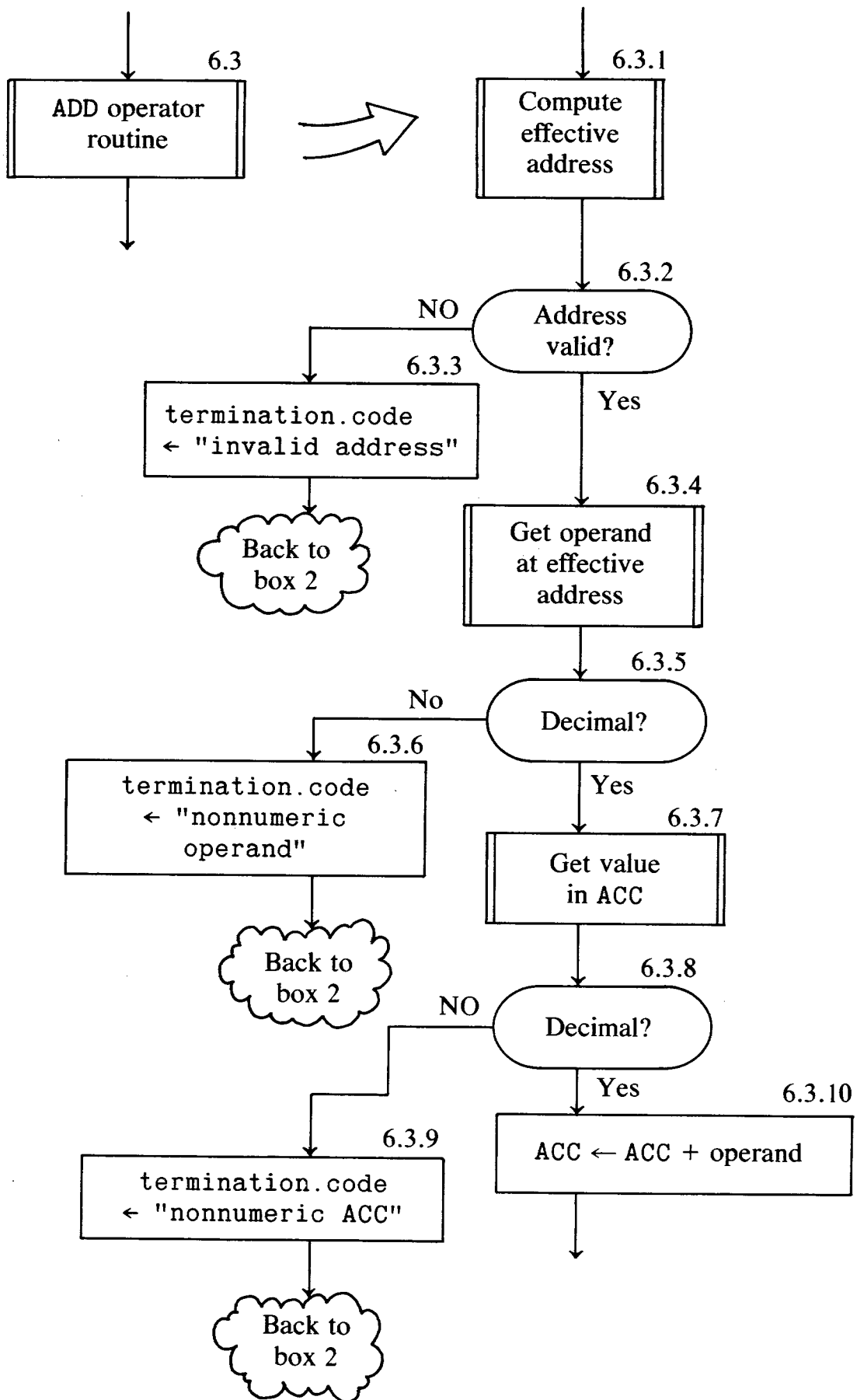


Figure 5.13. First-level details for ADD routine.

```

Line No.
1  ADD.  % ADD ROUTINE BEGINS HERE
2      CALL EFFECTIVE.ADDR  % ARGUMENT POINTED TO BY FA.
3      % RESULT IS LEFT IN PSEUDO-
4      % REGISTER EA
5      MOVE EA.ADDR TO T
6      %
7  [    CALL ADDRESS.TO.BINARY  % GETS (LOGICAL) POINTER ARGUMENT
8      % FROM T.  SEE TEXT DISCUSSION
9      IF FLAG = 0 THEN
10     BEGIN
11     MOVE INVALID.ADDRESS TO TERMINATION.CODE  % BOX 6.3.3
12     GO TO -BOX2
13     END
14     CALL BINARY.TO.FA  % PICKS UP ARGUMENT FROM REGISTER
15     % WHERE ADDRESS.TO.BINARY
16     % LEAVES ITS RESULT.
17  [    CALL VALIDATE.DECIMAL  % ARGUMENT POINTED TO FROM FA
18     IF FLAG = 0 THEN
19     BEGIN
20     MOVE NONNUMERIC.OPERAND TO TERMINATION.CODE % BOX 6.3.6
21     GO TO -BOX2
22     END
23     MOVE FA TO SOA  % SAVE ADDRESS OF 1ST OPERAND
24     MOVE ACC.ADDR TO T  % GET SECOND OPERAND AND VERIFY.
25     % ACC.ADDR IS A BINARY NUMBER
26  [    CALL BINARY.TO.FA
27     CALL VALIDATE.DECIMAL  % ARGUMENT POINTED TO FROM FA
28     IF FLAG = 0 THEN
29     BEGIN
30     MOVE NONNUMERIC.ACC TO TERMINATION.CODE % BOX 6.3.9
31     GO TO -BOX2
32     END
33     CALL ADD  % ARGUMENTS ARE THE POINTERS
34     % CURRENTLY IN SOA AND FA;
35     % RESULT LEFT IN SAMOS CELL
36     % POINTED TO BY FA.
37     GO TO INC.WORK.COUNTER  % GO TO BOX 7

```

Figure 5.14. Possible MIL code for the ADD operator routine flowcharted in Figure 5.13.

replace some of the repeating patterns by macro calls if the macro facility in the current MIL assembler were improved. Notice the similarity in form between the code in each of the three bracketed sequences (lines 7-13, 17-22, and 26-31). We could define the macro:

```

MACRO VALIDATE (UTILITY.NAME, ERROR.CODE, INDICATOR)=
CALL UTILITY.NAME
IF FLAG = 0 THEN
BEGIN
MOVE ERROR.CODE TO INDICATOR
GOTO -BOX2
END #

```


provided label arguments and parameters like `UTILITY.NAME` were allowed. Were this the case, we could then replace each bracketed code sequence in Figure 5.14 as follows:

```
VALIDATE (ADDRESS.TO.BINARY, INVALID.ADDRESS,  
          TERMINATION.CODE)
```

in place of lines 7 through 13,

```
VALIDATE (VALIDATE.DECIMAL, NONNUMERIC.OPERAND,  
          TERMINATION.CODE)
```

in place of lines 17 through 22, and

```
VALIDATE (VALIDATE.DECIMAL, NONNUMERIC.ACC,  
          TERMINATION.CODE)
```

in place of lines 26 through 31.

In the next chapter we will examine the details of these utilities. Undoubtedly there is some tradeoff between compactness of code (related to choice of utilities) and efficiency as measured in execution time. For example, the utility `EFFECTIVE.ADDRESS` must itself call on `VALIDATE.DECIMAL` and then on `ADD`, or else accomplish the equivalent operations in a more specialized manner. Recall that to compute an effective address, one forms the sum of the contents of the instruction's address field and the contents of the indicated index register. It is not necessary to check that the index register has a valid decimal number (because that value will have been previously validated), but it is necessary to decimal-validate the address field.

Notice also that the arguments of `ADD` are pointers to the actual operands in G-store and not to registers of the H-processor. These same arguments, however, had to be brought to the processor by `VALIDATE.DECIMAL`. They could have been left there, say in the scratchpads, but the code in Figure 5.14 implies that `ADD` doesn't know this.

In short, the coding plan suggested in Figure 5.14 is attractively compact, but its efficiency is apt to be relatively poor. It may be possible to establish more effective (but more implicit) communication among the utility routines to increase efficiency, though this may lead to code that is harder to understand or modify. These issues will be considered in the next chapter. The point to be made here is that if we decide efficiency is not important, then we can now proceed to code all the other operator routines (e.g., `STO..`, `LDA..`, `SUB..`), postponing until later the coding of the utility routines. However, if we suspect we will want to redesign the utilities and their interfaces, then we had better look into this matter before continuing to code the operator routines.

This is because redesign of the utility interfaces will directly affect the coding of almost every operator routine.

A few more remarks are in order before concluding this chapter.

1. The overall design of our interpreter and a shell surrounding it is now substantially complete. True, we have not yet flowcharted the individual operator routines (except for ADD), but this task is a relatively simple one though tedious.
2. We have not yet developed the details of the routines used by the operator routines, nor the utility routines needed by the shell (e.g., dumps and other displays). Details for the operator-routine utilities depend heavily on the representation of SAMOS registers and storage and on the degree of mutual interaction we wish these utilities to have for the sake of efficiency.
3. Although at the outset we said we would opt for the 8-bit EBCDIC representation of SAMOS characters in preference to other forms, such as the originally specified 6-bit BCD characters, very little of our design so far has really depended on this choice. We can still go either way without much undoing. This flexibility is now at an end. To detail the operator-routine utilities we must now bind this choice into our design.
4. Thus far we have been tacitly assuming that MIL coding should be accomplished only as a direct mapping from a higher-level language, such as our flowchart language. Moreover we have been careful to assure that each of our suggested flowcharts is well structured (in the Dijkstra sense). Since a primary justification for coding programs at as low a level as MIL is to take advantage of a particular microprocessor's architecture (that of the B1726), we can expect that to gain maximum efficiency in speed and/or space it will often be desirable to use MIL statement sequences that do not exhibit the same clean structure as the flowcharts do. Departure from clean structure for the sake of efficiency may, for example, occur in coding escapes from loops, or in treating exception conditions.

Many programmers whose aim is to achieve optimal coding become impatient with the apparent discrepancies between the flowchart structure and that of their optimal code. They tend to abandon the flowchart rather than update it to reflect at a higher level the compromises or changes they make at the lower level. Instead, they rely on comments inserted in the MIL code to provide adequate documentation. Indeed, some programmers don't even start at the flowchart level, but code directly in MIL.

These notes are not intended to be a style manual, but we are attempting to stress the importance of first looking at what we are trying to achieve at each stage *before* considering how to code the MIL equivalent. Occasionally the way we express an objective at the higher level will tend to restrict our imagination to suboptimal ways of implementing higher-level ideas. This often occurs when we have decomposed a process into too much detail (and consequently with the wrong machine model in mind) before beginning the MIL-coding process. We cannot always succeed in achieving the best balance. In the next chapter we will show a number of coding examples, and in a few cases show, for those interested in efficiency issues, how different ways of implementing the intent of certain utility subroutines can significantly influence the efficiency of the entire interpreter.

Chapter 6

MIL coding for data manipulation

Most of the MIL coding seen so far has been related to the control structure and decoding logic of an interpreter. We are now ready to become familiar with the coding techniques associated with data manipulation needed, for example, in utility routines such as those suggested in Table 5.2.

Several of the utility routines involve addition, subtraction, and multiplication for address computation and for conversion of decimal to binary values. Moreover, a basic decimal addition routine is needed to implement the SAMOS operators ADD, SUB, MPY, and DIV. Successful design of these utility routines will therefore depend on gaining a more complete understanding of the B1726 24-bit function box for addition and subtraction, and especially of the base-10 (decimal) feature. Section 6.1 explains the structure of the addition and subtraction mechanisms, and each subsequent section then develops the design of one of the needed utilities.

6.1 ARITHMETIC OF THE 24-BIT FUNCTION BOX

Recall that addition and subtraction are achieved under control of the CP register, as suggested in Figure 6.1. The inputs to the function box are X, Y, and CYF, where CYF is the one-bit *carry-in* register. The arithmetic outputs are SUM, DIFF, CYL, and CYD. The last two one-bit “registers” are explained later.

The results, SUM and DIFF, are governed by CPU and CPL. When CPU = 01_2 , values in X and Y are regarded as unsigned decimal integers up to 6 digits in length, where each digit is in 4-bit binary code. We shall refer to such coding as “packed decimal”. Therefore, when CPU = 01_2 , the results in SUM and DIFF are the packed decimal sum and difference of the packed decimal inputs X and Y augmented by CYF.

When a one-bit carry-out results for SUM, that value goes to CYL. Likewise, a one-bit borrow into DIFF sets CYD. [For other values of CPU, namely when CPU = 10_2 or 11_2 , the values of SUM, DIFF, CYD, CYL are undefined; when CPU = 00_2 , addition and subtraction are binary, but carries out and borrows in are registered in CYL and CYD in a similar fashion.]

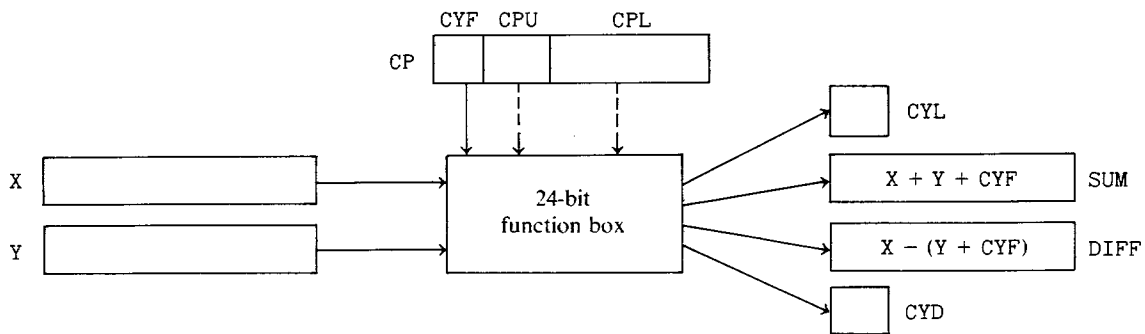


Figure 6.1. Control flow (----->) and data flow (————>) in the function box for addition and subtraction.

The value in CPL controls the length of SUM and DIFF. Thus the carry-out bit for a sum value that would exceed CPL bits is registered in CYL. We therefore say that the values in SUM, DIFF, and CYL are *conditioned by CPL*. For some (unknown) reason, the B1726 designers did not also condition CYD by CPL. Hence CYD is set to 1 only when a borrow into the 24th bit of DIFF has occurred.

Examples. Suppose $CP = 00110000_2$ (i.e., $CYF = 0_2$, $CPU = 01_2$, $CPL = 10000_2$), which specifies no carry in, 4-bit decimal mode, and results 16 bits (4 decimal digits) wide. Let $X = @123456@$, $Y = @654321@$ (i.e., decimal numbers 123456 and 654321). The results in SUM and DIFF are $SUM = @007777@$, $DIFF = @009135@$ (decimal quantities 7777 and 9135); $CYL = 0_2$ and $CYD = 1_2$.

Now suppose that $CP = 00110100_2$ (i.e., $CYF = 0_2$, $CPU = 01_2$, $CPL = 10100_2$), which specifies no carry in, 4-bit decimal mode, and 20-bit (5 decimal digits) width. The same quantities in X and Y will produce $SUM = @077777@$, $DIFF = @069135@$, $CYL = 0_2$, and $CYD = 1_2$.

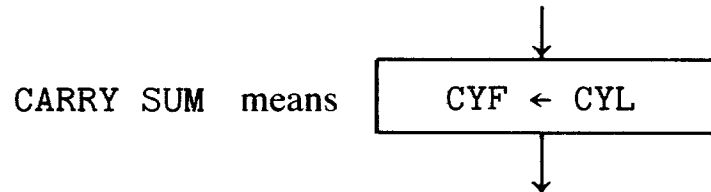
Here are two points to remember when $CPU = 01_2$;

1. The 4-bit quantities corresponding to hex digits A, B, C, D, E, and F cannot be evaluated by the 24-bit function box.
2. The only valid width settings (valid values of CPL) are 0, 4, 8, 12, 16, 20, and 24; other settings of CPL yield undefined results.

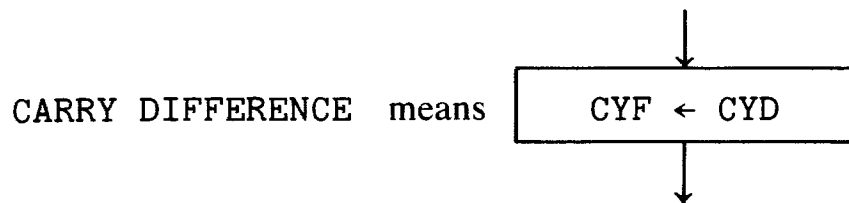
When the arithmetic fields to be added or subtracted *exceed* 24 bits in width, two or more separate loadings of X and Y must be made. The SUM and DIFF values for each loading of X and Y (and CYF) must be moved to some other registers (and if necessary to G-store) before reloading X and Y to obtain the higher-order portions of the results. Of course, the carry-out or borrow-in indicator must be *recycled* into CYF for the next

respective cycle of addition or subtraction. Recycling is accomplished by executing the MIL statements CARRY SUM and CARRY DIFFERENCE.

The MIL statement



and the MIL statement



Understanding how the function box works for decimal arithmetic now allows us to decide between two alternative strategies for implementing SAMOS arithmetic with 10-decimal operands A and B. The first strategy might have us perform addition by extracting the digits from the decimal characters A and B and adding pairwise right to left in G-store, until all ten digit pairs are summed or differenced, with suitably recycled carries. Of course it is essential, prior to addition or subtraction, to check that the characters of A and B represent valid decimal digits.

The alternative strategy might have us input from G-store all ten characters of each SAMOS operand, check each character for valid decimal, pack the characters into 4-bit decimal form, and then add (or subtract) the two 10-digit operands in no more than two successive loadings of X and Y—say 4 digits in the first load, and the remaining 6 digits in the second load. Further examination of this issue in the next section leads us to select this second strategy as the more efficient one.

6.2 VALIDATE.DECIMAL: CASE STUDY FOR A UTILITY ROUTINE

Figure 6.2 shows a top-level view of the details for VALIDATE.DECIMAL. The parameter PTR is a pointer to the sign position of an 11-character SAMOS word held in G-store. The procedure sets the variable FLAG to "Nogood" when the sign character is not a valid one ("+" or "-") or when one of the subsequent characters is not a decimal digit.

Since VALIDATE.DECIMAL is usually called prior to using its validated result as an arithmetic operand, we will reconsider later the

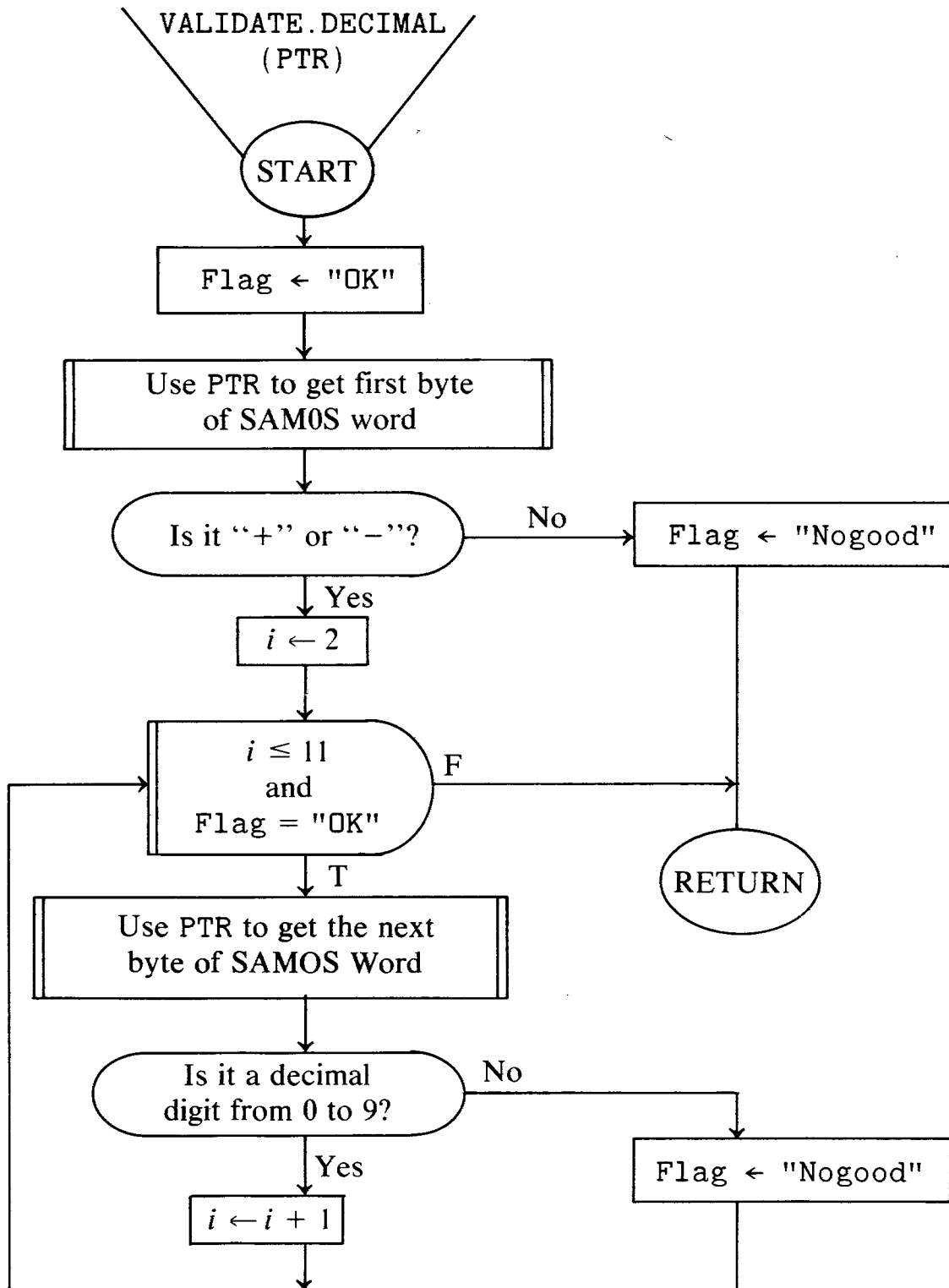


Figure 6.2. First overview of VALIDATE.DECIMAL.

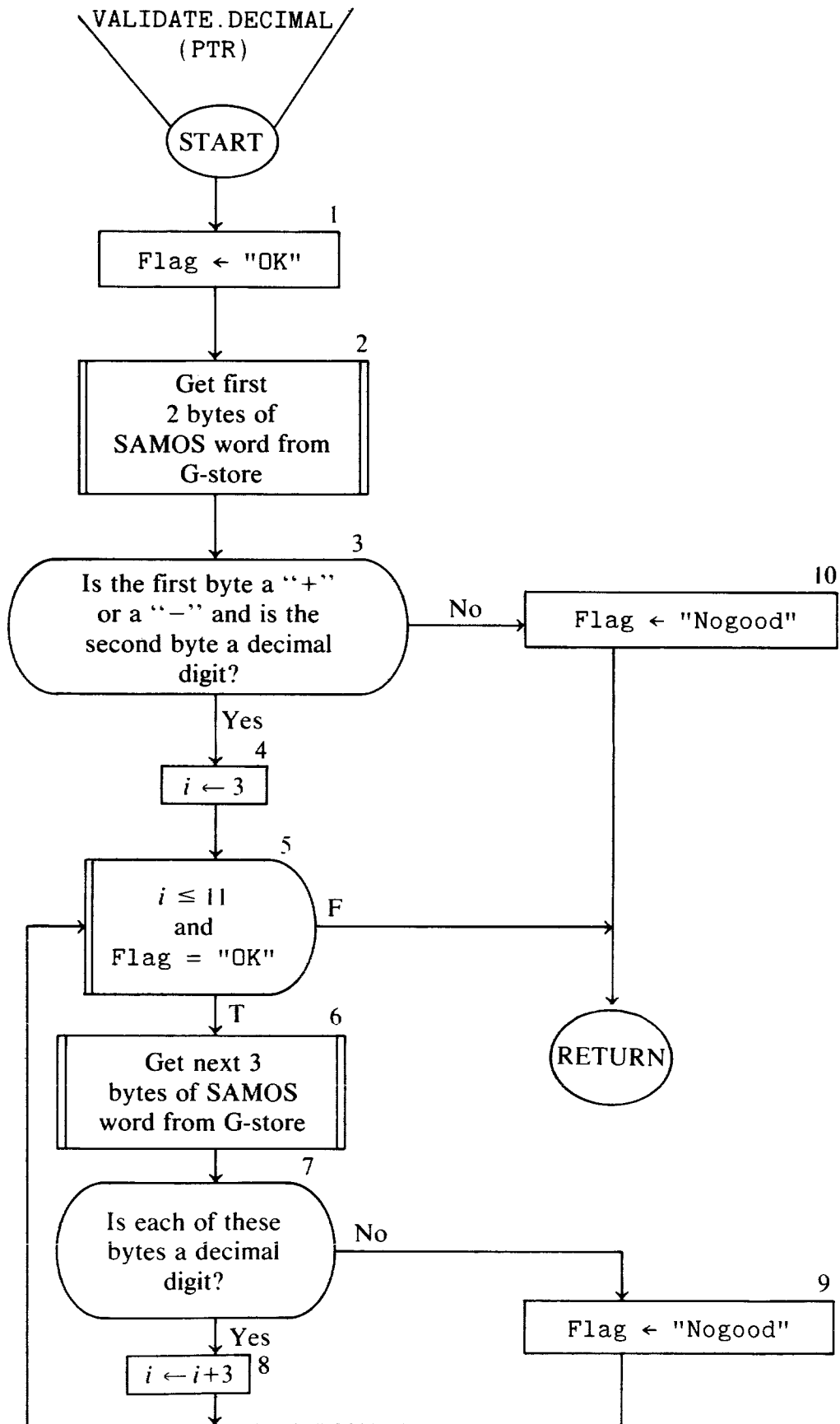
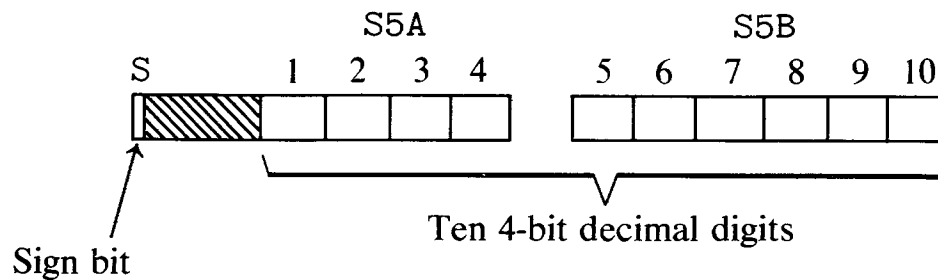


Figure 6.3. Second overview of VALIDATE.DECIMAL.

possibility of extending the objectives of this procedure so it not only validates a SAMOS word (fetched from G-store) as numeric, but also constructs and caches a 4-bit packed decimal representation of the SAMOS word in a more accessible scratchpad, such as S5



For the moment, however, we shall concentrate only on the functions implied by Figure 6.2.

If we bear in mind that READs from G-store take up to six times as long as most other microinstructions in the B1726, it seems worthwhile to minimize the number of READs. Since the maximum number of bytes per READ is 3, and since 11 bytes must be transferred, no fewer than 4 READs are needed. One way to transfer 11 bytes in 4 READs is to use chunks of 2, 3, 3, and 3 bytes. This is the plan used in Figure 6.3.

The structure shown in Figure 6.4 is even more B1726-specific. The parameter PTR is now represented as the register FA. The scanning control is achieved by letting the register FL serve as the counter. FL is decremented by 24 after each fetch of 3 bytes.

Assuming we are happy with the control structure exhibited in Figure 6.4, we can now consider ideas for implementing the details, especially those of boxes 3 and 7:

1. Let us assume that bytes from G-store are transferred to the T-register.
2. The sign byte can then be moved to the X-register and compared for equality against the literals “+” and “-” moved to Y.
3. The remaining bytes may be tested as follows: We observe that the EBCDIC decimal characters “0”, “1”, “2”, . . . , “9” are represented as the ordered (and dense) set of 8-bit binary integers, @(1)11110000@, @(1)11110001@, @(1)11110010@, . . . , @(1)11111001@, or, if you like, @F0@, @F1@, . . . , @F9@. Hence, any 8-bit integer less than @F0@ or greater than @F9@ is not a valid decimal character. Therefore, the byte to be tested can be placed in X and compared with bounds values “0” and “9” placed successively in Y.
4. The Flag variable takes only two values, so any available 1-bit

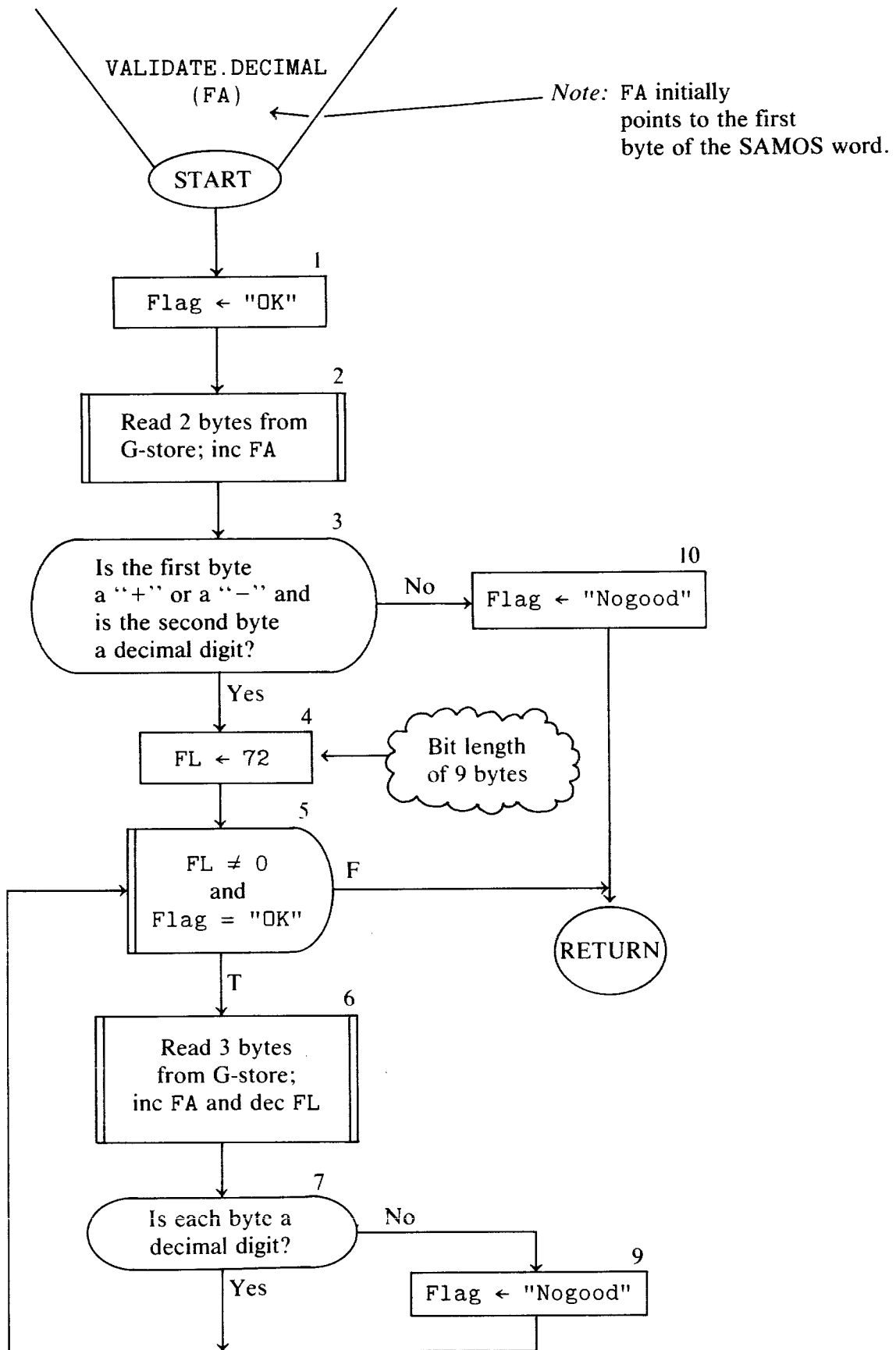


Figure 6.4. Third overview of VALIDATE.DECIMAL.

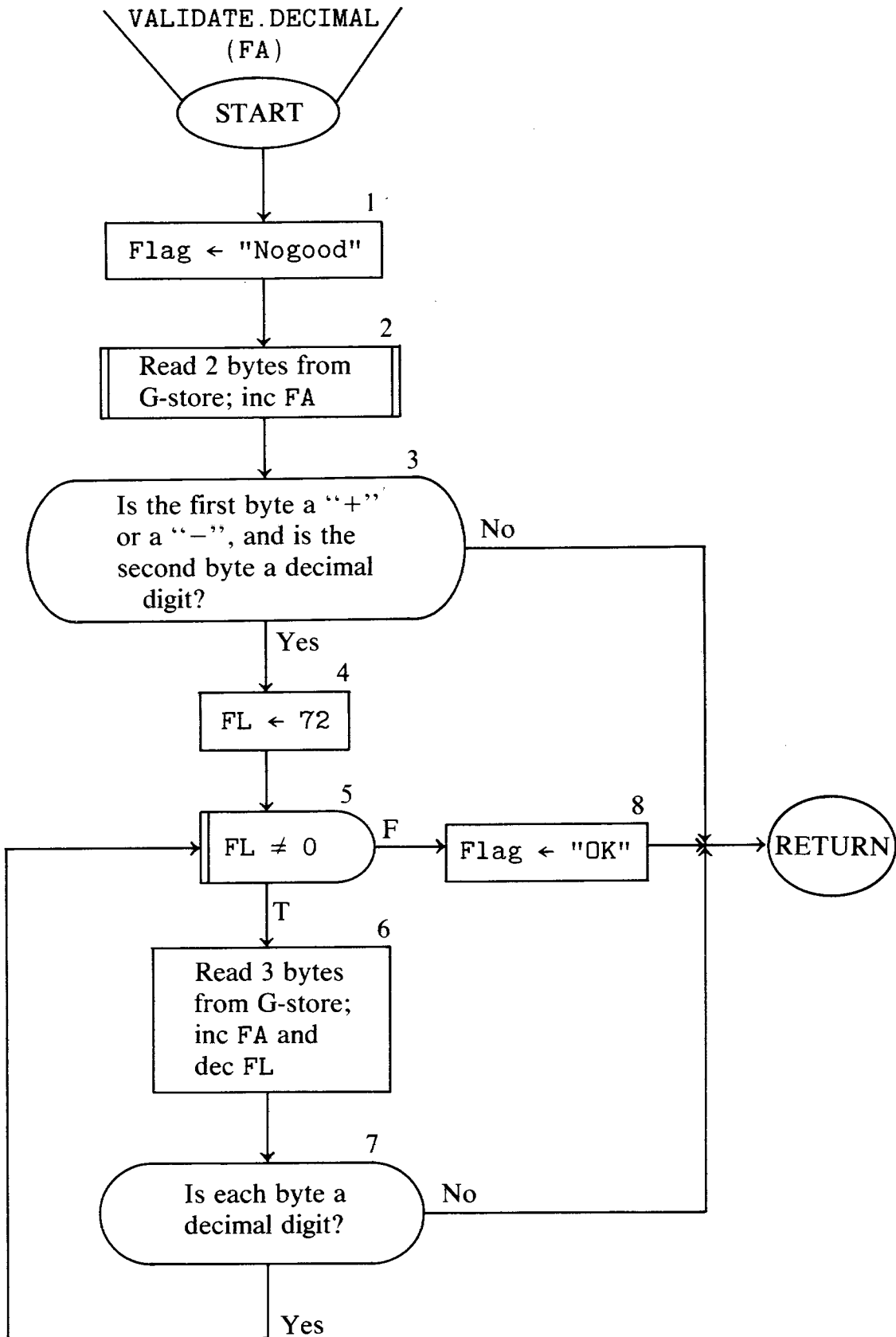


Figure 6.5. Fourth overview of VALIDATE.DECIMAL.

subregister, e.g. CB(0), can serve as the flag (1 for "OK" and 0 for "Nogood".)

One final observation is in order in anticipation of mapping a control structure like Figure 6.4 into MIL code. A number of low-level tests must be made on the characters of the SAMOS word. (Boxes 3 and 7 each decompose into several low-level tests.) We want to avoid coding the resetting of Flag to "Nogood" each time one of these tests fails. More compact code will be obtained if Flag is initially set to "Nogood". In this way Flag need only be reset once (to "OK") when and if the success exit is reached. The control structure in Figure 6.5 reflects this observation and is the one we will use for conversion to the MIL code we now show in Figure 6.6. Note that the flowchart in Figure 6.5 no longer exhibits the good structure we would like were efficiency not an important consideration.

Taking stock, we have now developed a method for validating a decimal SAMOS word so it can then be used as an operand in a SAMOS ADD, SUB, MPY, or DIV instruction. Unfortunately, the method does not also *save a copy* of the validated word where it would be highly accessible for the subsequent arithmetic operation.

What changes are needed so VALIDATE.DECIMAL caches in the processor registers a packed decimal representation of the validated word? A possible form for the packed decimal representation (sign and ten 4-bit decimal digits), reflecting the SAMOS signed-magnitude representation in G-store, is suggested in Figure 6.7, using a double scratchpad, in this case S5, as the cache.

Clearly, boxes 3, 6, and 7 of Figure 6.5 will require modification to include steps for saving the sign and decimal digits in the cache. But how will we decide whether to leave the result in T CAT L or in a scratchpad, and if the latter, which one? One approach would be to pick the same cache each time, e.g., S5. Or, we could "gild the lily" by specifying one more parameter, a 4-bit integer, and let the matching arguments serve as an index for a wanted scratchpad. If we choose the second approach, then each reference to the scratchpad will have to be modified by ORing into M the integer argument. A 4-bit register like CA or CB may be used to transmit the argument. Let us take the "easier", first approach. Later we can consider the more general alternate approach.

One possible strategy for the revised box-3 details is shown in Figure 6.8. First we clear L, and then, if the sign is "-", we set L(0) to 1 (box 3.5). Later, when we determine that the second byte of the SAMOS word is a valid decimal character, we copy TF, which holds the decimal

```

VALIDATE.DECIMAL                                     % VALIDATES A SAMOS WORD POINTED TO
                                                    % BY THE CONTENTS OF FA AS A DECIMAL
                                                    % INTEGER. IF NOT,CB(0) IS SET TO 1, ELSE
                                                    % IT IS SET TO 0. THIS ROUTINE USES FL,
                                                    % X, Y, AND T. CPL ASSUMED TO BE ≥ 8.

BEGIN
LOCAL.DEFINES
DEFINE FLAG = CB(0) #

MACRO BYTE.TEST (N) =                               % TESTS ONE BYTE FROM T FOR
    1                                                % DECIMAL DIGIT
    EXTRACT 8 BITS FROM T(N) TO X
    MOVE "0" TO Y
    IF X LSS Y THEN EXIT
    MOVE "9" TO Y
    IF X GTR Y THEN EXIT #

SET FLAG                                             % SET FLAG TO NOGOOD
READ 16 BITS TO T INC FA                            % OBTAIN SIGN AND FIRST DIGIT
EXTRACT 8 BITS FROM T(8) TO X                       % SIGN BYTE TO X
MOVE "+" TO Y
IF X NEQ Y THEN
BEGIN                                               % TRY "-"
    MOVE "-" TO Y
    IF X NEQ Y THEN EXIT
    MOVE @(1)1000@ TO LA
END
BYTE.TEST (16)                                     % TEST SECOND BYTE
MOVE 72 TO FL                                       % SET LOOP COUNTER
.LOOP IF FL EQL 0 GO TO +OK.EXIT
READ 24 BITS TO T INC FA AND DEC FL                % GET NEXT 3 BYTES
                                                    % TEST FIRST BYTE OF GROUP
BYTE.TEST (0)                                     % TEST SECOND BYTE OF GROUP
BYTE.TEST (8)                                     % TEST THIRD BYTE OF GROUP
GO TO -LOOP
.OK.EXIT
RESET FLAG                                         % SET FLAG TO OK
EXIT
END
    
```

Figure 6.6. MIL code for the Figure 6.5 flowchart. Note the use of a locally defined MACRO called BYTE.TEST which tests a byte found in the T-register and EXITS if the byte is not a decimal character.

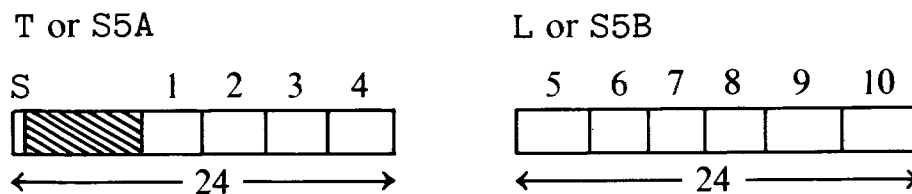


Figure 6.7

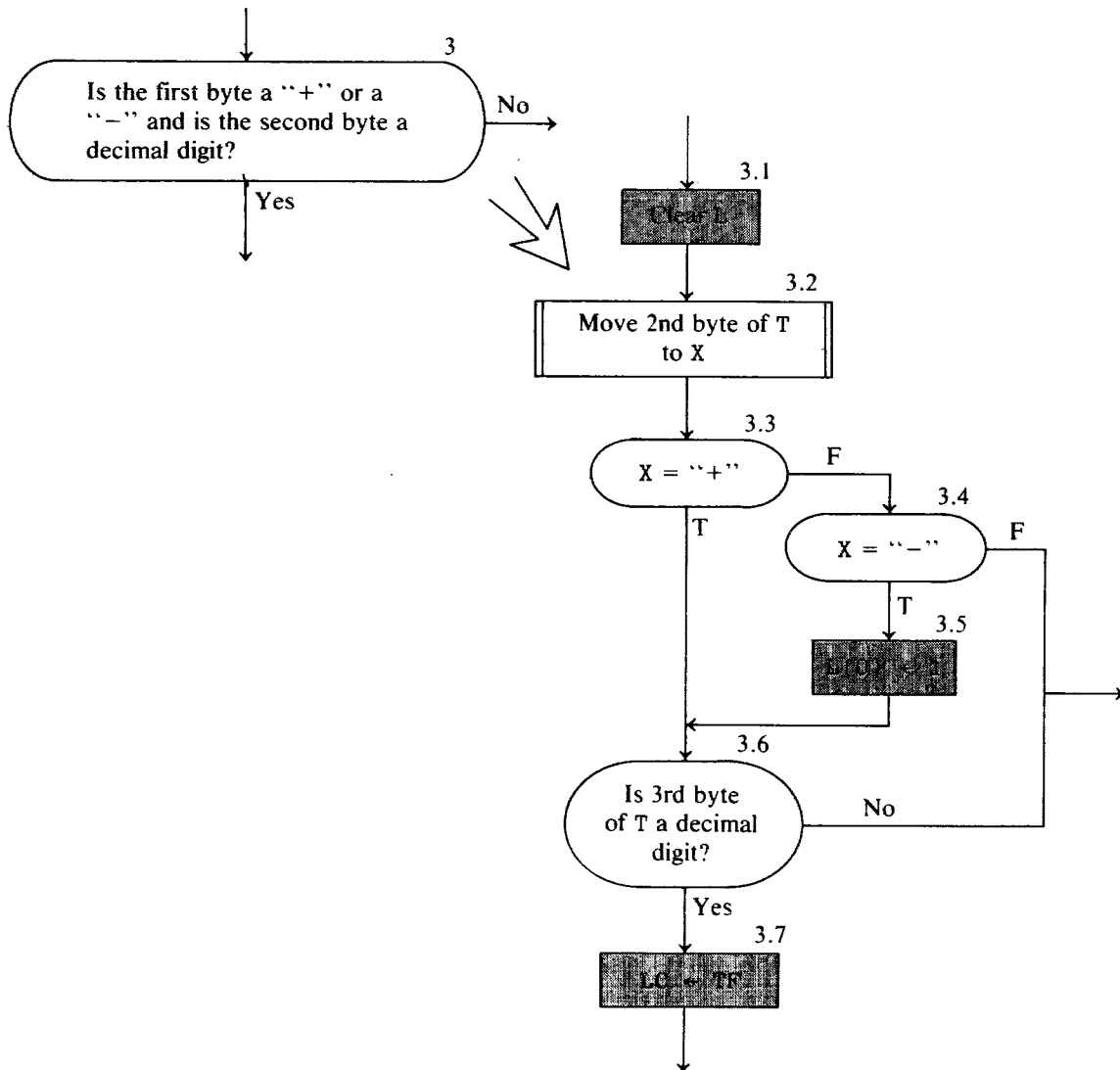
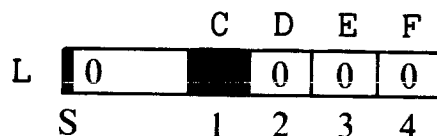


Figure 6.8. Shaded boxes show the changes to the details of box 3 necessary for caching a 4-bit decimal representation of a validated decimal SAMOS word into a scratchpad word.

code, into LC. At this point L contains the first 2 codes (sign and one digit) of the 11 required.

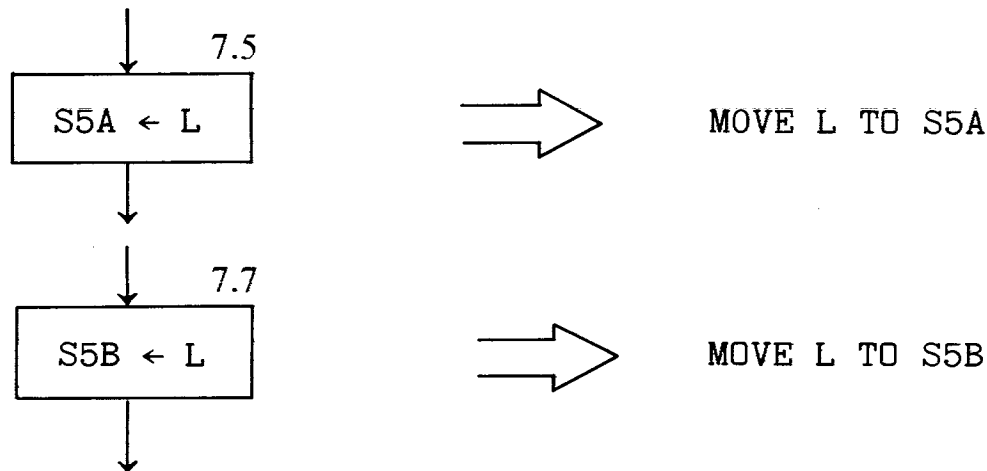


The shaded portions are now properly filled. Positions LD, LE, and LF in L will be filled with appropriate decimal code after the next 3 bytes of the SAMOS word are brought to T, and the remainder of L will be ignored.

In Figure 6.9 we see the details for extracting and caching digits for positions 2 through 10 of the SAMOS word in S5.

Exercise. Based on the flowchart details given in Figures 6.5, 6.8, and 6.9, recode the VALIDATE.DECIMAL routine in MIL so it saves a packed decimal representation of the validated SAMOS word in scratchpad S5.

We may wish to generalize these changes so the packed decimal representation is saved in a *specified* scratchpad rather than always in S5. It is very easy to do this, because, as we can see from Figure 6.9, only two flowchart steps refer to the scratchpads. These are boxes 7.5 and 7.7.



Only these two steps require modification.

We suppose that register CA holds the integer argument (0 through 15) specifying the scratchpad. Then Figure 6.10 shows the needed changes to the two instructions.

If speed is of paramount importance, additional improvements can be made to reduce execution time of VALIDATE.DECIMAL.

1. We might remove the loop by straight-line coding, thus eliminating the loop-control steps, but at the cost of inserting more lines of code in the routine. For example, the two control steps in Figure 6.6,

```
.LOOP IF FL EQL 0 GO TO +OK.EXIT
```

and

```
GO TO -LOOP
```

are executed on each of the three loop transits. Hence, the time to

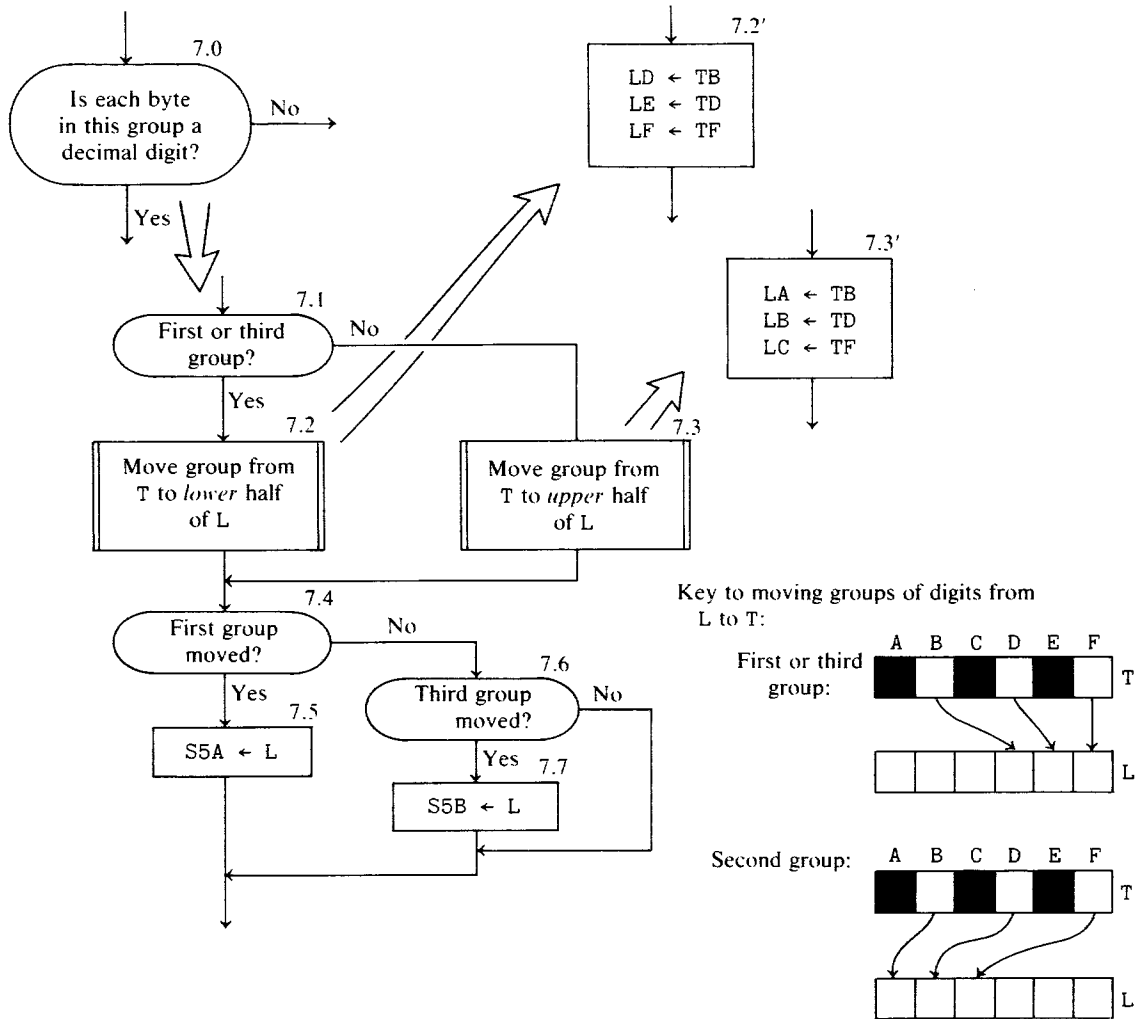


Figure 6.9. How box 7 might be augmented to cache additional decimal digits in L and then save them in S5A or S5B.

BEFORE	AFTER
MOVE L TO S5A	MOVE CA TO M MOVE L TO SOA % OR THE INDEX INTO M % TO COMPUTE THE % DESIGNATED PAD A
MOVE L TO S5B	MOVE CA TO M MOVE L TO SOB % OR THE INDEX INTO M % TO COMPUTE THE % DESIGNATED PAD B

Figure 6.10. Code for generalizing the cache.

execute six microinstructions can be saved with straight-line coding. But notice how many more lines of code will be needed if no other improvements are made. The loop body (Figure 6.6) contains one READ instruction and three macro calls, each of which expands to five microinstructions. So straight-line coding would result in 16×3 or 48 lines of code, a net increase of 28 lines. If the increase were smaller, straight-line coding would be more attractive.

2. We might take advantage of more special knowledge on how the B1700 function box works. Only a B1700 specialist would normally know that the decimal adder of the 24-bit function box is built so it is *guaranteed* to malfunction if nondecimal digits are given as inputs in X or Y. Thus, if $Y = 0$ and if X contains at least one invalid decimal digit, the decimal adder is guaranteed to form a value in SUM that is *not* equal to X. For example, with $CPU = 01_2$,

$$\begin{array}{ccccccc}
 0 & + & 0 & + & \text{garbage} & \neq & \text{garbage} \\
 \uparrow & & \uparrow & & \uparrow & & \uparrow \\
 \text{Carry} & & \text{Value} & & \text{Value} & & \text{Value} \\
 \text{in} & & \text{in Y} & & \text{in X} & & \text{in SUM}
 \end{array}$$

when garbage contains at least one invalid decimal digit.

We can exploit this special feature of the B1700 by testing up to 6 of the SAMOS digits *at one time*, but to do this will require a major restructuring of the algorithm. Here is one idea for the new structure:

The sign byte is tested first, as before, but the next 10 characters are not *fully* tested before packing into L. We will only test the high-order 4 bits of each decimal character before packing the lower half into L. After the packing is completed, each 24-bit portion of the packed representation is then used as an addend for the X-register in the test to determine if $X = X + 0$.

In particular, let us once again suppose that S5 contains the packed decimal representation of the SAMOS word now possibly invalid. Then only two tests are needed to validate all 10 decimal digits, as shown in the logic of Figure 6.11. Notice that for the sake of this test, the sign bit at the left end of S5A will be treated as part of a valid 4-bit decimal digit (either 0000 or 1000).

Since the expensive part of checking the decimal digits can be delayed until after packing, the new lines of code needed for straight-line coding of the loop body (boxes 6 and 7 of Figure 6.5) can now be significantly

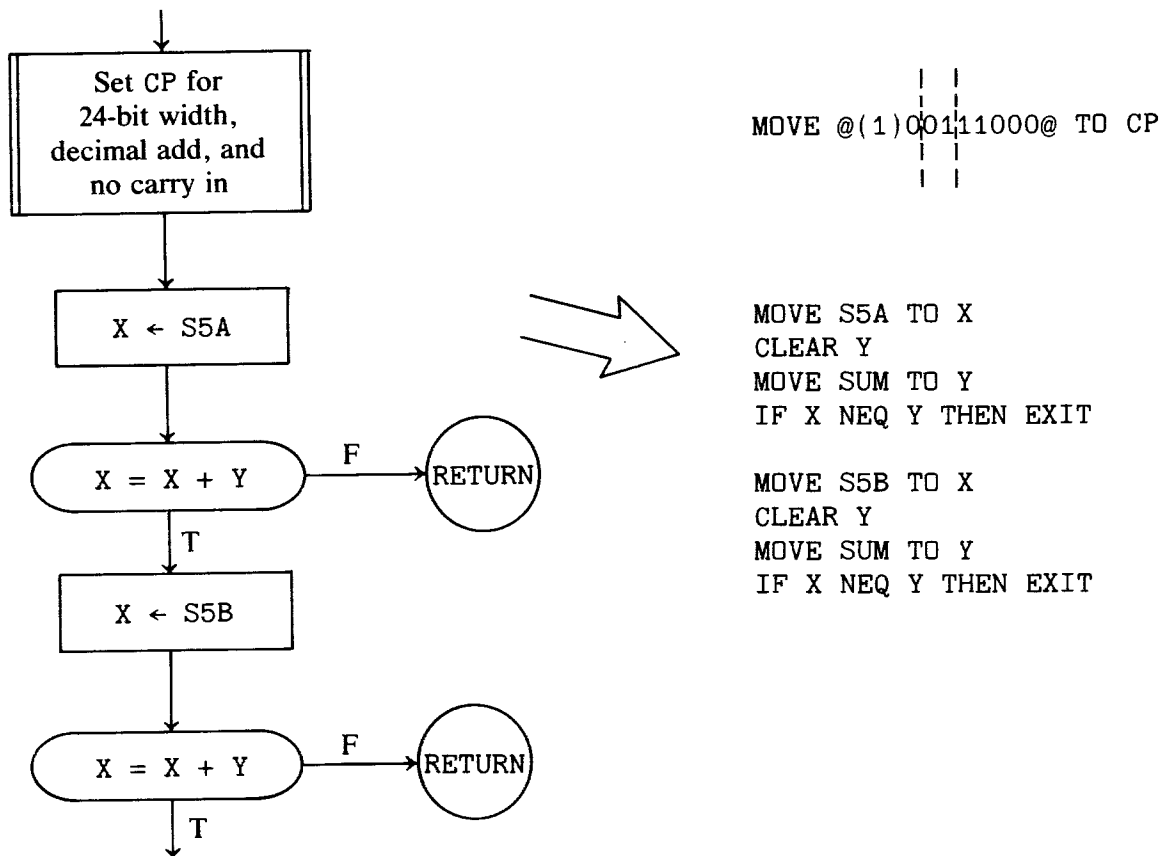


Figure 6.11. Logic for checking validity of 10 decimal digits in only 8 microinstructions.

reduced. Figure 6.12 shows the new strategy combining the best ideas in Figures 6.4, 6.8, 6.9, and 6.11. The MIL code for Figure 6.12 is shown in Figure 6.13.

Now we can see the savings effected by the combination of straight-line coding for the loop and use of our special knowledge of the B1700 circuitry for checking up to six decimal digits at one time. The Figure 6.13 code requires execution of 44 microinstructions for validating and packing a valid SAMOS number, as compared with 70 microinstructions for the Figure 6.6 code, where validity checking but no packing was achieved. There are only 11 more lines of code in Figure 6.13 than in Figure 6.6

Exercise

1. Can you see a way to “shave off” any more instructions from the code in Figure 6.13? Explain.

2. Revise the flowchart in Figure 6.12 and the MIL code in Figure 6.13 so VALIDATE.DECIMAL leaves its packed decimal result in T CAT L instead of S5. Choose a method that minimizes or eliminates the use of scratchpad registers as temporary storage.

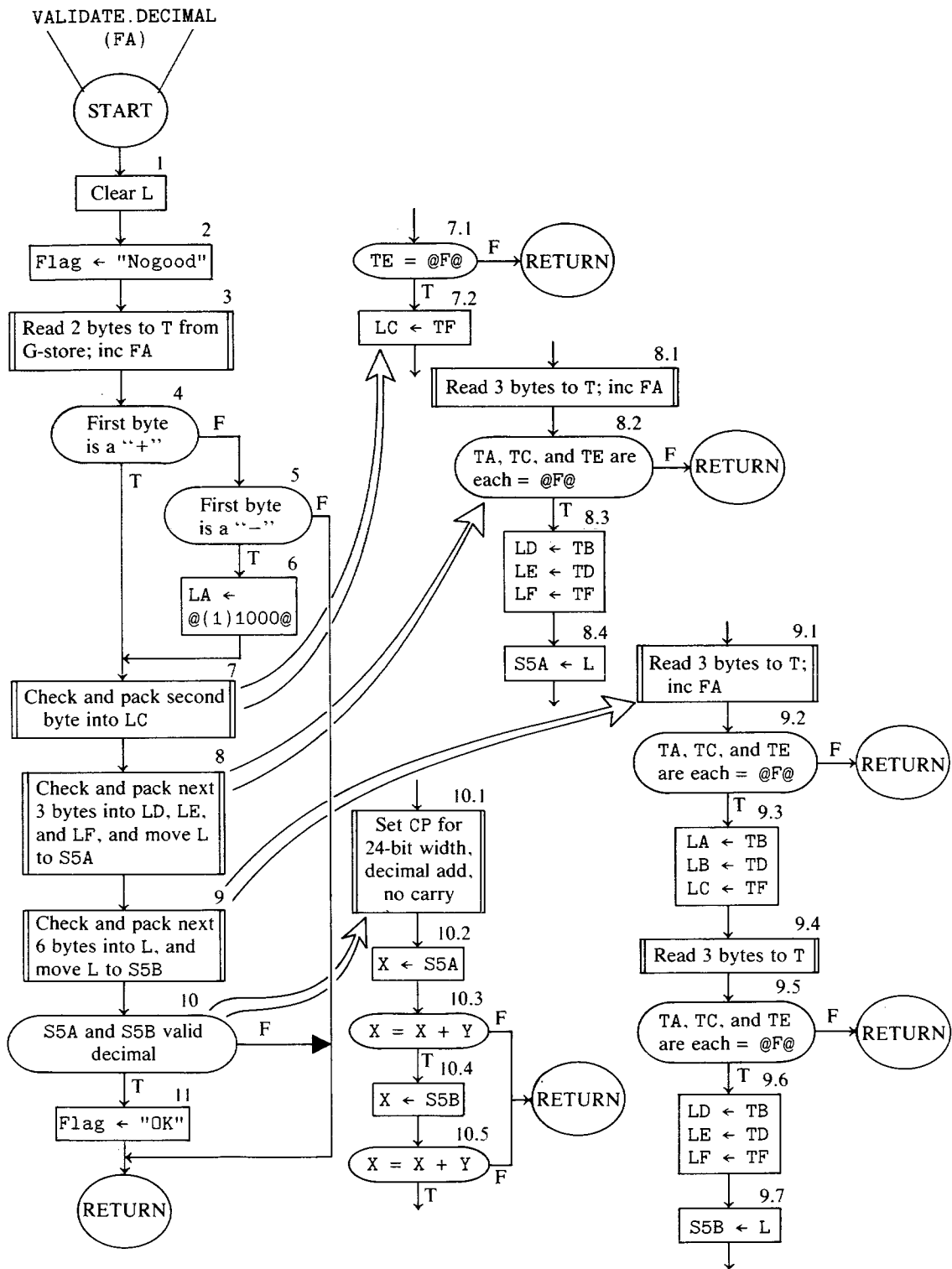


Figure 6.12. Fifth view of VALIDATE.DECIMAL.

```

VALIDATE.DECIMAL    % VALIDATE A SAMOS WORD POINTED TO BY THE
                   % CONTENTS OF FA AS A DECIMAL INTEGER AND
                   % PACKS A 4-BIT DECIMAL REPRESENTATION IN S5.
                   % IF NOT VALID, CB(0) IS SET TO 1, ELSE IT IS SET
                   % TO 0. THIS ROUTINE USES X, Y, T, L, AND CP.

BEGIN
LOCAL.DEFINES
DEFINE FLAG        = CB(0) #
MACRO CHECK.F(TK) =
    IF TK NEQ @F@ THEN EXIT #
CLEAR L
SET FLAG           % FLAG = NOGOOD
READ 16 BITS TO T INC FA    % GET FIRST 2 BYTES IN TC THRU TF
EXTRACT 8 BITS FROM T(8) TO X % SIGN BYTE TO X
MOVE "+" TO Y
IF X NEQ Y THEN
    BEGIN          % TRY "-"
        MOVE "-" TO Y
        IF X NEQ Y THEN EXIT
        MOVE @(1)1000@ TO LA
    END
    CHECK.F(TE)
    MOVE TF TO LC          % BOX 7
READ 24 BITS TO T INC FA  % BOX 8
    CHECK.F(TA)           % CHECK AND PACK INTO L
    CHECK.F(TC)
    CHECK.F(TE)
    MOVE TB TO LD
    MOVE TD TO LE
    MOVE TF TO LF
MOVE L TO S5A
READ 24 BITS TO T INC FA  % BOX 9
    CHECK.F(TA)           % CHECK AND PACK INTO L
    CHECK.F(TC)
    CHECK.F(TE)
    MOVE TB TO LA
    MOVE TB TO LB
    MOVE TF TO LC
READ 24 BITS TO T        % CHECK AND PACK INTO L
    CHECK.F(TA)
    CHECK.F(TC)
    CHECK.F(TE)
    MOVE TB TO LD
    MOVE TD TO LE
    MOVE TF TO LF
MOVE L TO S5B
MOVE @(1)00111000@ TO CP  % BOX 10
MOVE S5A TO X
CLEAR Y                  % BOX 10.3
MOVE SUM TO Y
IF X NEQ Y THEN EXIT
MOVE S5B TO X
CLEAR Y                  % BOX 10.5
MOVE SUM TO Y
IF X NEQ Y THEN EXIT
RESET FLAG              % BOX 11    SET FLAG TO OK
EXIT
END

```

Figure 6.13. MIL code for Figure 6.12 flowchart.

This concludes our discussion of VALIDATE.DECIMAL as a case study of how one might descend through levels of abstraction from high-level, machine-independent constructs to very low-level, B1726-dependent constructs through a succession of binding decisions. We will now examine much more briefly the code for the other important utility routines.

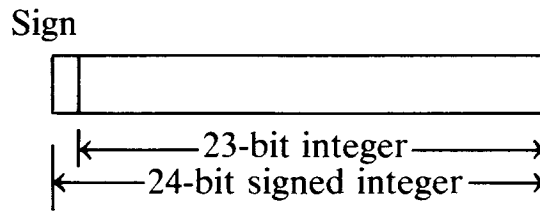
6.3 BINARY.TO.FA

The BINARY.TO.FA utility routine is used to convert the binary integer SAMOS address into an absolute address in G-store for the desired SAMOS word. The input parameter S ranges from small negative values (indicating SAMOS registers and pseudo registers as explained in Section 5.4) to positive values, 0 through SIZE (indicating ordinary SAMOS storage words). The procedure must produce and leave in FA the value of map(S), defined in Section 5.4, as

$$\text{map}(S) = S \times 88 + (\text{SAMOS.STORE} + \text{BR})$$

The function box of the B1726 cannot perform multiplication directly; it can only add or subtract positive integers. But the argument S may be negative. How will S be represented? We have three choices.

1. Signed magnitude, e.g.,



2. Twos complement (24 bits)
3. Ones complement (24 bits)

The flowchart logic in Figure 6.14 assumes signed-magnitude representation for S. MIL actually caters to the programmer who favors 2s-complement representation in the sense that negative literals are always mapped to 2s-complement representation. For example, the MIL statement MOVE -5 TO X is equivalent to

```
MOVE @FFFFFFB@ TO X @ 2'S COMPLEMENT OF -5
                    @ IN 24-BIT BINARY TO X
```

One can, of course always specify a signed-magnitude representation by

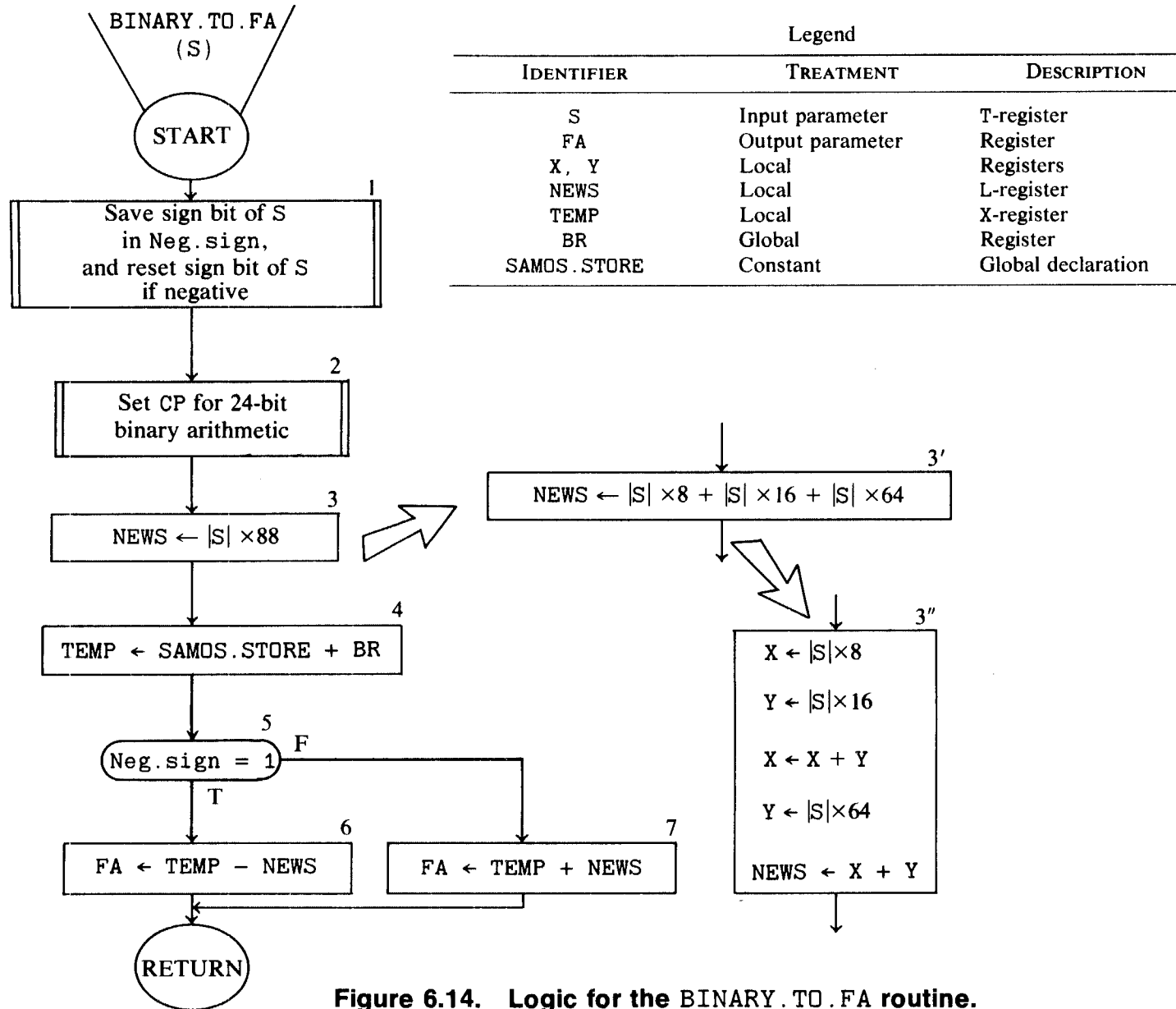


Figure 6.14. Logic for the BINARY.TO.FA routine.

giving the coding for the literal explicitly—for example,

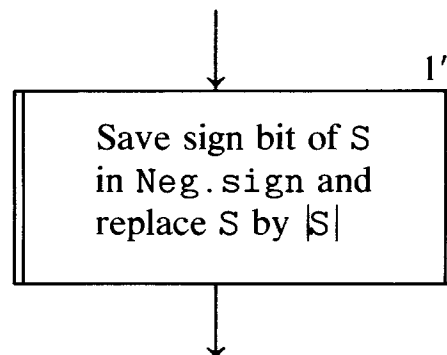
```
MOVE @800005@ TO X  % SIGNED MAGNITUDE REPRESENTATION
                   % OF -5 TO X
```

See Section 6.7 for further discussion of complement arithmetic on the B1726.

To obtain the product $S \times 88$ we would then first strip off the sign bit from S and get $|S| \times 88$. The box-3 details of Figure 6.14 suggest a way to perform this multiplication by summing products of $|S|$ and powers of 2. Once the product $|S| \times 88$ is formed, it is either added to or subtracted from the sum $SAMOS.STORE + BR$, depending on the saved sign of S . The legend of the flowchart indicates the B1726 registers that may (should) be used for local storage in the MIL implementation.

The MIL code shown in Figure 6.15 is straightforward. But those interested in efficiency should note the following points.

1. Had it first occurred to us to express flowchart box 1 of Figure 6.14 as



the following more efficient and more compact code might have occurred to us for box 1 (This code assumes that $CB(1)$, $CB(2)$ and $CB(3)$ can be destroyed.)

```
MOVE TA TO CB  % SAVE SIGN OF S IN NEG.SIGN
RESET T(0)    % REPLACE S BY |S|
```

2. In any case, the instruction to reset the sign bit, $T(0)$, is really not needed. We can take advantage of the fact that S is an even number (88 in this case). Hence the powers of two multipliers, 2^n , that sum to S are such that $n \geq 1$. Multiplication of S by these powers of 2 is accomplished by left shift of the T-register (at least one bit to the left). Only left-shifted copies of T are moved to X or to Y . The sign bit is lost in the process. (Remember, too, that a left shift of T to some other register leaves T unchanged.)

```

BINARY.TO.FA          % INPUT VALUE, S, IS IN T REGISTER
                     % OUTPUT RESULT IS IN FA
                     % X, Y, CB(0), AND L ARE USED AS LOCAL STORAGE

    BEGIN
    LOCAL.DEFINES
    DEFINE NEG.SIGN = CB(0) #
    DEFINE NEWS     = L     #
%BOX1
    IF T(0) THEN          % SAVE SIGN OF S
        BEGIN            % AND REPLACE S
            SET NEG.SIGN % BY |S|
            RESET T(0)   % NECESSARY INSTRUCTION ?
        END ELSE
        BEGIN
            RESET NEG.SIGN
        END
    MOVE 24 TO CP          % SETUP FOR ARITHMETIC
    SHIFT T LEFT BY 3 BITS TO X % 8×S TO X (SIGN BIT SHIFTED OFF)
    SHIFT T LEFT BY 4 BITS TO Y % 16×S TO Y
    MOVE SUM TO X          % 24×S IN X
    SHIFT T LEFT BY 6 BITS TO Y % 64×S TO Y
    MOVE SUM TO NEWS      % 88×S IN NEWS (L)
    MOVE BR TO X
    MOVE SAMOS.STORE(0) TO Y
    MOVE SUM TO X          % BR + SAMOS.STORE IN X
    MOVE NEWS TO Y
%BOX5
    IF NEG.SIGN THEN
        BEGIN
            MOVE DIFF TO FA
        END ELSE
        BEGIN
            MOVE SUM TO FA
        END
    EXIT
    END

```

Figure 6.15.

- By analogy with point 1 above, the code for box 5 may be coded much more compactly as

```

    MOVE SUM TO FA
    IF NEG.SIGN THEN MOVE DIFF TO FA

```

6.4 ADDRESS.TO.BINARY

The ADDRESS.TO.BINARY routine is used in computing effective addresses in SAMOS. Figure 6.16 shows the logic defining this utility routine, which converts a 4-character SAMOS address field, known in

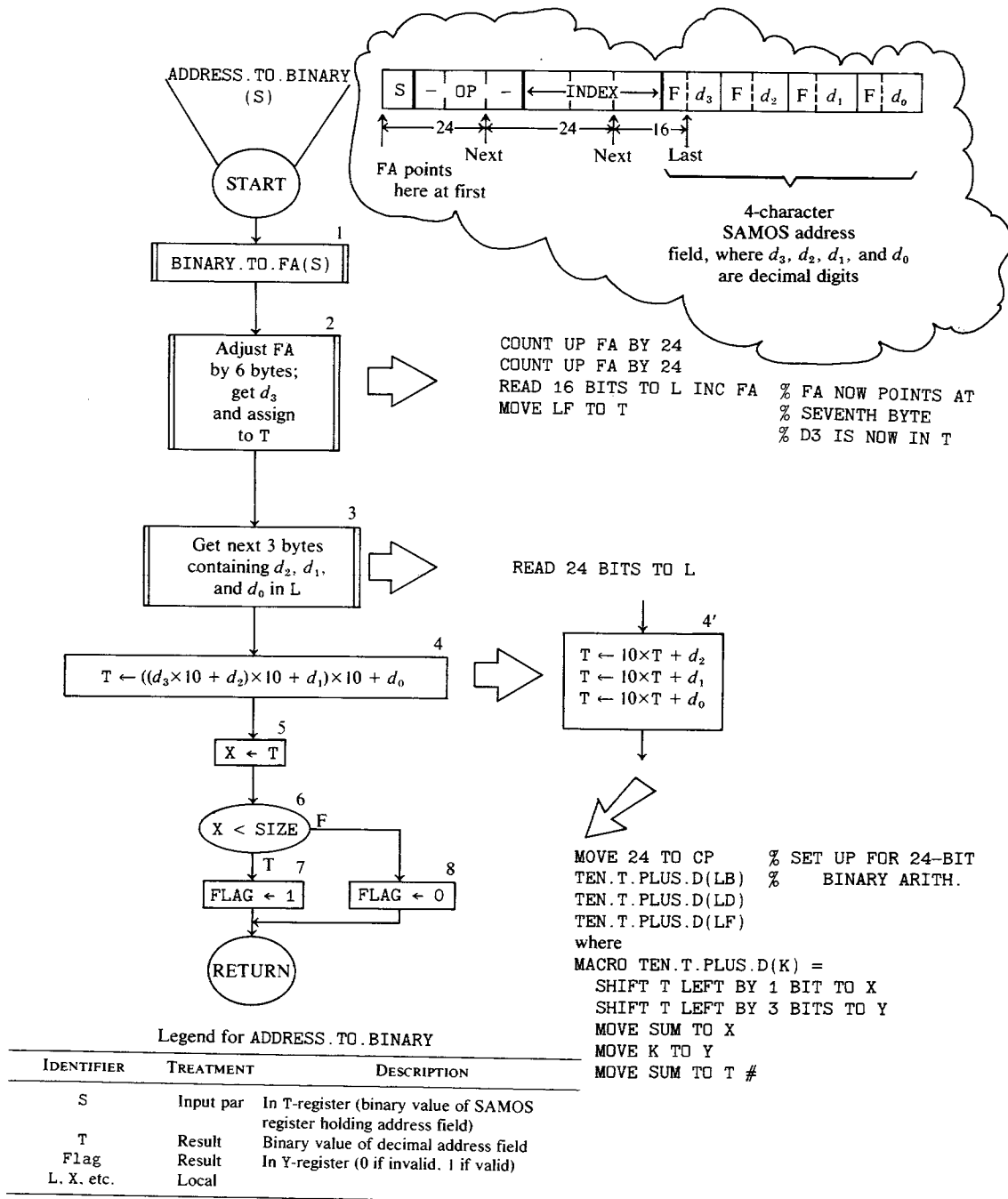
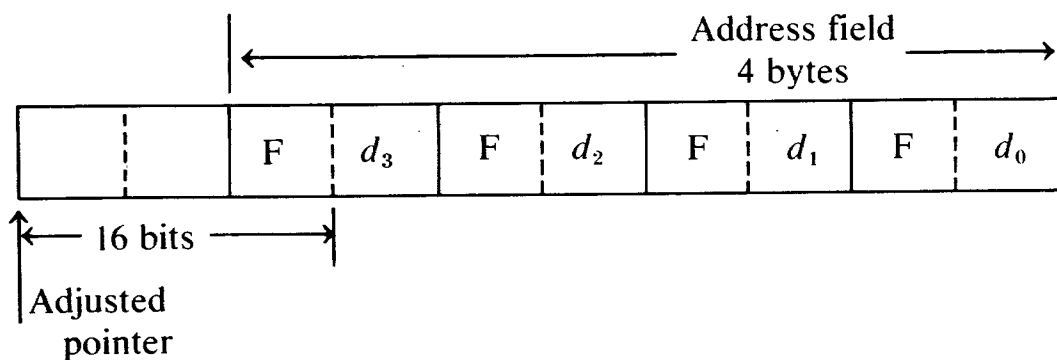


Figure 6.16.

advance to represent a valid decimal address, to a binary integer. The input parameter, S, is a (binary) address of the SAMOS storage word that contains the address field. The legend of the flowchart suggests that the input parameter and output result are assumed to be taken from and deposited in the T-register.

The first step (box 1) converts S to an absolute pointer into G-store via a call to BINARY . TO . FA. The resulting value is left in FA (see Section 6.3). The second step (box 2) adjusts this pointer within

“striking distance” of digit d_3 , fetches the next 2 bytes to L, and then transfers d_3 to T.



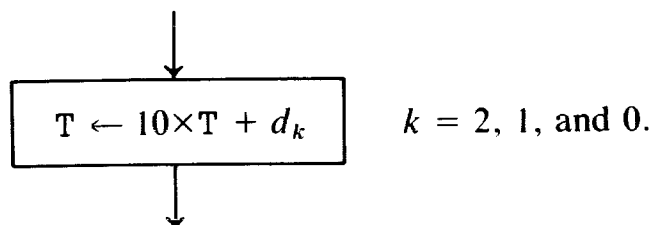
Here F is the hexadecimal digit @F@, and d_3 , d_2 , d_1 , and d_0 are the binary-coded decimal digits of the address. It is only necessary to extract these digits and evaluate the polynomial

$$d_3 \times 10^3 + d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0$$

or, in the more efficient factored form,

$$((d_3 \times 10 + d_2) \times 10 + d_1) + 10 \times d_0,$$

as suggested in the details shown in boxes 3, 4, and 5 of the flowchart. The computation can be performed in binary arithmetic using only registers T, L, X, and Y. Multiplication by 10 is accomplished by shifting multiples of T out of T to X and to Y and adding them ($2 \times T + 8 \times T$). Examination of box 4' shows repeated use of the same multiply-add step,



This suggests the use of a macro for the purpose, named TEN.T.PLUS.D, whose definition and use is also illustrated in Figure 6.16. The final steps of the routine (boxes 5, 6, 7, and 8) check that the polynomial evaluation results in a valid SAMOS storage address, i.e., a non-negative integer that is less than SIZE.

Figure 6.17 shows the complete MIL code for ADDRESS.TO.BINARY.

```

ADDRESS.TO.BINARY                                % ROUTINE BEGINS HERE
                                                  % INPUT VALUE, S, IS IN T REGISTER
                                                  % OUTPUT RESULTS:  BINARY ADDRESS IN T
                                                  %                               FLAG IN Y (0 IF INVALID
                                                  %                               1 IF VALID)
                                                  % USES X, Y, L, FA, S1A AS LOCAL STORAGE

      BEGIN
      LOCAL.DEFINES
      MACRO TEN.T.PLUS.D (LX) =
          SHIFT T LEFT BY 1 BIT TO X
          SHIFT T LEFT BY 3 BITS TO Y
          MOVE SUM TO X                        % T×10 IN X
          MOVE LX TO Y
          MOVE SUM TO T      #                % T×10 + D IN T

% BOX1
      CALL BINARY.TO.FA                        % WITH VALUE IN T AS ARGUMENT

% BOX2
      COUNT FA UP BY 24                        % COUNT FA UP BY
      COUNT FA UP BY 24                        % 6 BYTES
      READ 16 BITS TO L INC FA                 % D3 NOW IN LF
      MOVE LF TO T

% BOX3
      READ 24 BITS TO L                        % D2 IN LB, D1 IN LD, AND D0 IN LF

% BOX4
      MOVE 24 TO CP
      TEN.T.PLUS.D (LB)                        % 10×T + D2   GOES TO T (SEE MACRO DEF.)
      TEN.T.PLUS.D (LD)                        % 10×T + D1   GOES TO T (SEE MACRO DEF.)
      TEN.T.PLUS.D (LF)                        % 10×T + D0   GOES TO T (SEE MACRO DEF.)

% BOX5
      MOVE T TO X

% BOX6
      MOVE SIZE TO Y                            % SIZE IS A GLOBAL CONSTANT
      IF X LSS Y THEN
          BEGIN
              MOVE 1 TO Y                        % VALID SAMOS ADDRESS
          END ELSE
              MOVE 0 TO Y                        % INVALID SAMOS ADDRESS
          END
      EXIT % END OF ADDRESS.TO.BINARY ROUTINE

      END

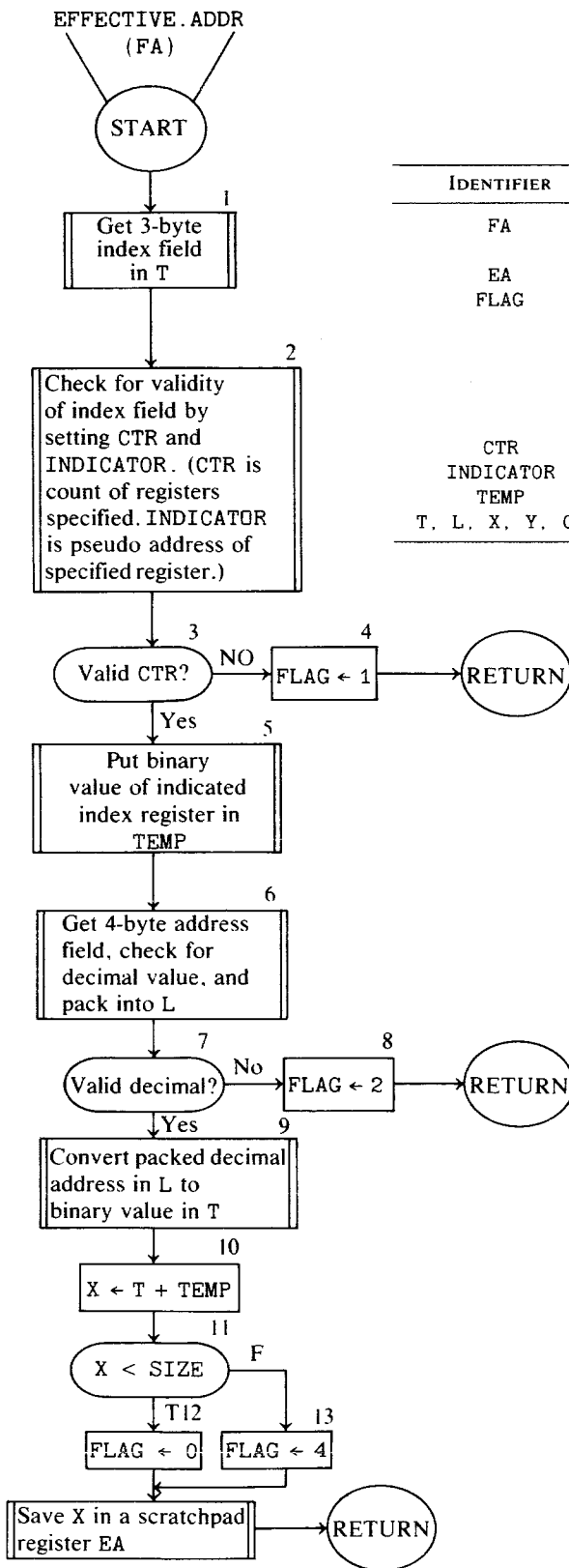
```

Figure 6.17. MIL code for the ADDRESS.TO.BINARY routine flowcharted in Figure 6.16.

The next section describes the use of this routine in computing an effective address.

6.5 EFFECTIVE . ADDRESS

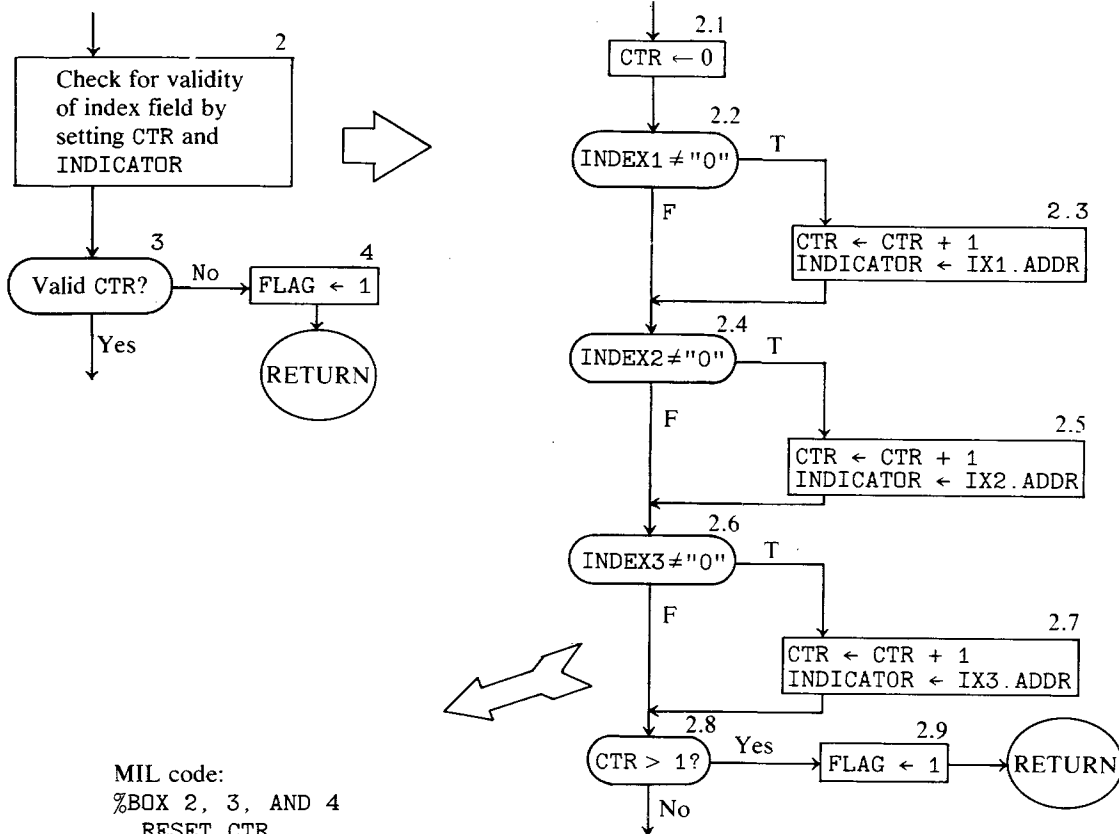
This section describes the rather powerful utility routine needed for computing an effective address. Figure 6.18 gives the top-level logic.



Legend for EFFECTIVE . ADDR

IDENTIFIER	TREATMENT	DESCRIPTION
FA	Input parameter	Points to index field of SAMOS instruction
EA	Result	Scratchpad register
FLAG	Result	FLF register
		0 = OK
		1 = too many index registers specified
		2 = address field not decimal
		4 = effective address too big
CTR	Local	FLE register
INDICATOR	Local	S0B "
TEMP	Local	S1B "
T, L, X, Y, CP	Locals	Registers as needed

Figure 6.18.



MIL code:
 %BOX 2, 3, AND 4
 RESET CTR
 MOVE "0" TO Y

```

EXTRACT 8 BITS FROM T(0) TO X
IF X ≠ Y THEN
  BEGIN
    INC CTR BY 1
    MOVE IX1.ADDR TO INDICATOR
  END
EXTRACT 8 BITS FROM T(8) TO X
BEGIN
  INC CTR BY 1
  MOVE IX2.ADDR TO INDICATOR
END
EXTRACT 8 BITS FROM T(16) TO X
IF X ≠ Y THEN
  BEGIN
    INC CTR BY 1
    MOVE IX3.ADDR TO INDICATOR
  END
IF CTR(2) THEN
  BEGIN
    SET FLAG TO 1
  END
  
```

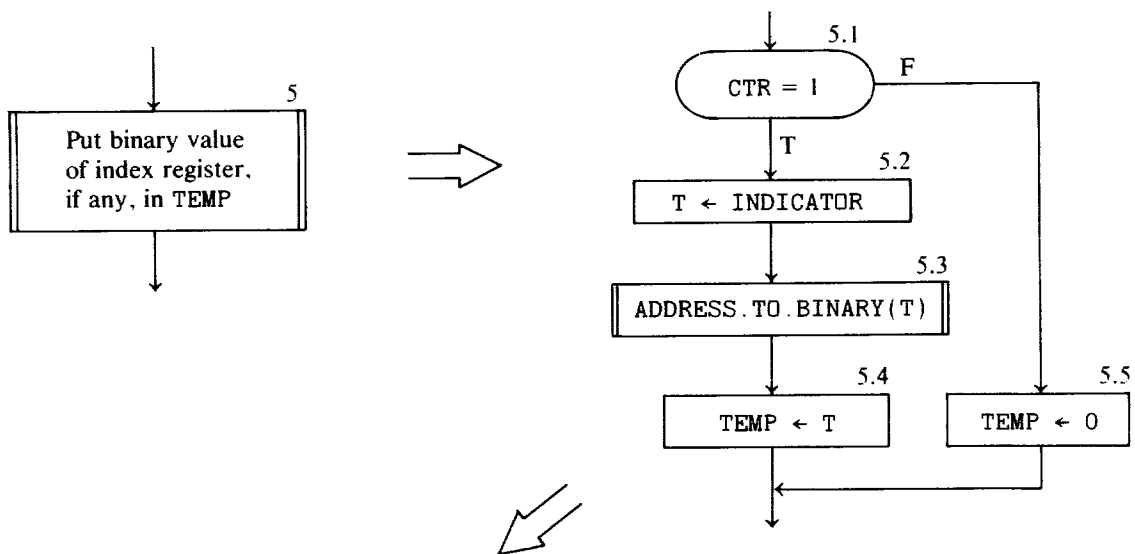
```

% MOVE INDEX1 TO X
% MOVE INDEX2 TO X
% MOVE INDEX3 TO X
% IS CTR ≥ 2
  
```

Figure 6.19.

The input parameter, assumed to be left in FA, is an absolute G-store address pointer to the index field (5th byte) of the SAMOS instruction being interpreted.

The first steps (boxes 1, 2, and 3) fetch the index field and determine if no more than one index register is marked. If more than one is, a flag is set for an error return (box 4). Figure 6.19 shows one possible implementation for this validity check. If the index field is valid and indicates an index register, then the value of that index register must be fetched from the register in G-store. Recall that index registers are represented as 11-character SAMOS storage words with negative addresses. It is further assumed that an index register is found in the address-field position of the storage word, i.e., in the rightmost four character positions.



```
MIL code:
% BOX5 DETAIL
IF CTR(3) THEN
  BEGIN
    MOVE INDICATOR TO T
    CALL ADDRESS.TO.BINARY
    MOVE T TO TEMP
  END ELSE
  BEGIN
    MOVE NULL TO TEMP    % MOVE 0 TO TEMP
  END
```

Figure 6.20. Details for box 5. Note that there is a special NULL register on the B1726 which always contains zero and which is always available. Alternatively we could have coded box 5.5 as MOVE 0 TO TEMP. Those interested in efficiency should observe, however, that if a literal is moved to a scratchpad, as in this case (since TEMP is a scratchpad), two instructions will be compiled by the MIL assembler, e.g., MOVE 0 TO TAS and MOVE TAS TO TEMP.

The negative address of the indicated index register is determined by the logic of Figure 6.19 and left in a scratchpad called INDICATOR. This value is then used as in the next step (box 5), whose detail is seen in Figure 6.20. The indicator value, after being moved to T, becomes the argument in a call on ADDRESS.TO.BINARY, which fetches the value from the index register and converts the decimal value to a binary integer. (See Section 6.4.)

We can now see why ADDRESS.TO.BINARY is coded assuming its argument points to a valid decimal address. We can always ensure in our interpreter that any value *stored* in the address field of an *index register* storage word will consist of only valid decimal characters. If the value is valid when it is stored, it will be valid when next retrieved so we need no further check. In any case, the (index register) value returned by ADDRESS.TO.BINARY is saved in a scratchpad named TEMP for use as a summand when the address part of the same instruction is obtained from G-store. That address part is obtained and “decimally validated” in the logic of boxes 6 and 7, whose details are shown in Figure 6.21. This logic is similar to that of the VALIDATE.DECIMAL routine, except here we check only 4 bytes instead of 10. If invalid, the flag is set to an appropriate value (box 8). If valid, the four decimal digits are extracted and packed into L.

The next step (box 9) converts the packed decimal integer in L to a binary value, leaving this value in T (see details of box 9 in Figure 6.22). Here again, use may be made of the macro named TEN.T.PLUS.D first described in Figure 6.16.

The last step is to form the sum of the (binary) index-register value saved in TEMP and the (binary) address value just left in T. The sum must be a nonnegative integer less than SIZE (the size of our SAMOS store). The flag must be set to an appropriate value (0 or 4) to indicate a valid or out-of-bounds effective address.

Whether valid or out of bounds, the computed effective address is deposited in a scratchpad register representing the EA pseudo register. The MIL coding for these last steps, boxes 10 through 14, is shown in Figure 6.23. Note that we have now discarded the idea of using the pseudo register EA, as first suggested in the specifications for EFFECTIVE.ADDR in Table 5.2. [The use of a pseudo register requiring conversion to and reversion from character representation now seems wasteful.]

Exercises

1. Put all the pieces together that were suggested in Figures 6.18 through 6.23 to make one complete MIL subroutine. The following is a

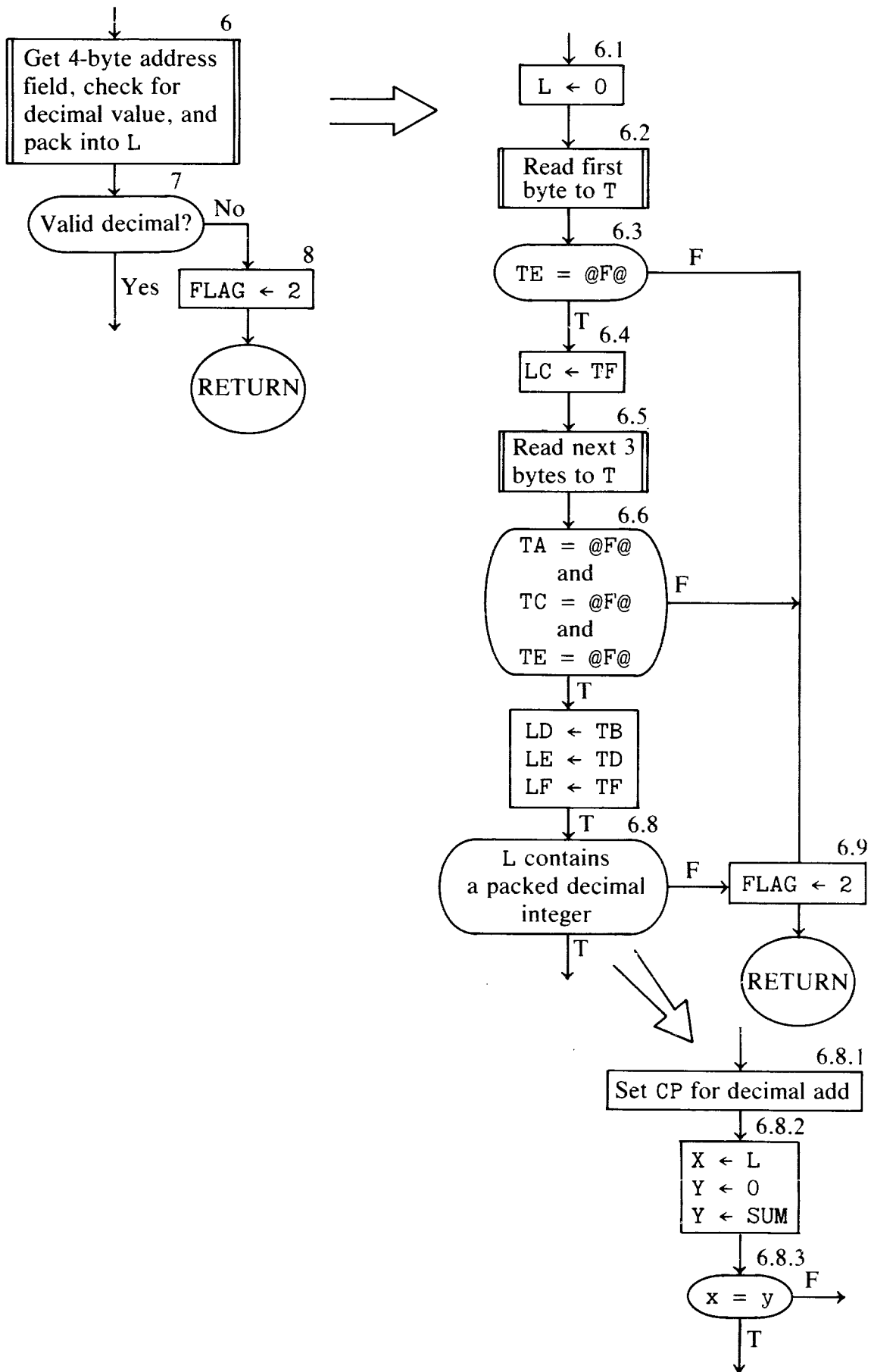


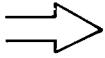
Figure 6.21.

MIL code:

```
CLEAR L AND Y
READ 8 BITS TO T INC FA
IF TE = @F@ FALSE THEN GO TO +SET.FLAG.EXIT
MOVE TF TO LC
READ 24 BITS TO T
IF TA = @F@ FALSE THEN GO TO +SET.FLAG.EXIT
IF TC = @F@ FALSE THEN GO TO +SET.FLAG.EXIT
IF TE = @F@ FALSE THEN GO TO +SET.FLAG.EXIT
```

```
MOVE TB TO LD
MOVE TD TO LE
MOVE TF TO LF
```

```
MOVE @(1)00111000@TO CP      % SET CP FOR PACKED DECIMAL
                               % ADD (24 BITS)
```



```
MOVE L TO X
% CLEAR Y ALREADY ACCOMPLISHED
```

```
MOVE SUM TO Y
IF X ≠ Y THEN
  BEGIN
```

```
% TESTS L + 0 = L
% IF SO, L MUST HAVE BEEN A
% VALID PACKED DECIMAL INTEGER
```

```
.SET.FLAG.EXIT SET FLAG TO 2
  EXIT
  END
```

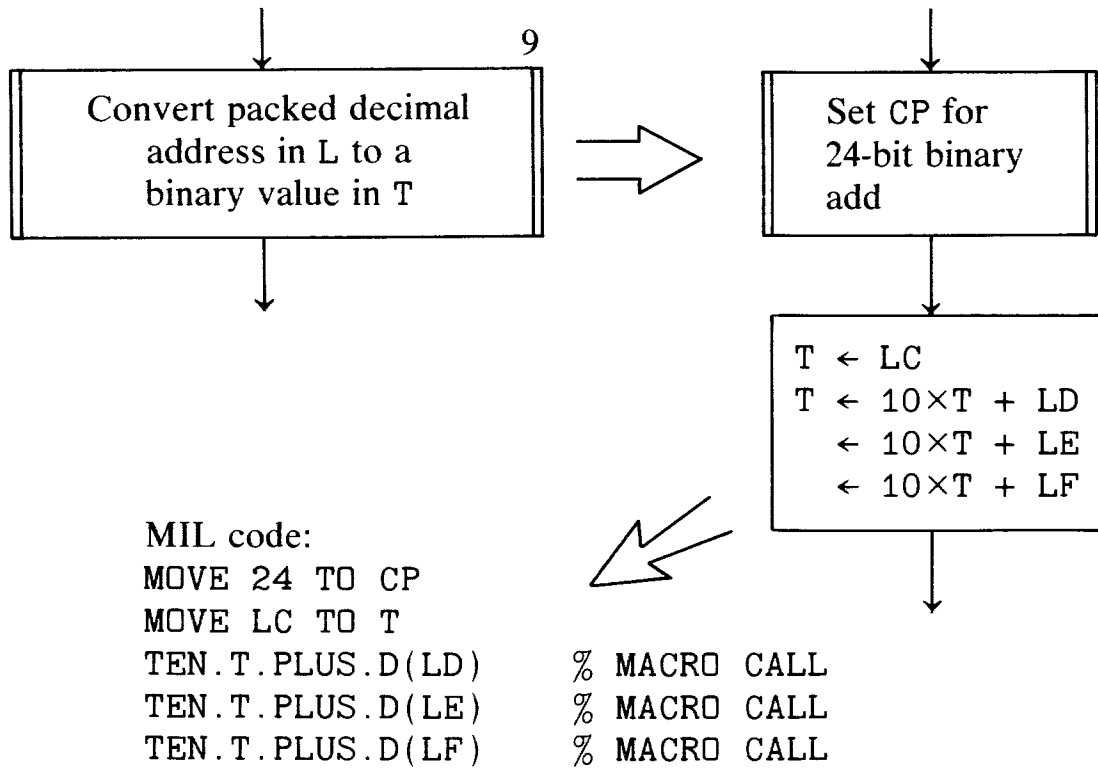


Figure 6.22.

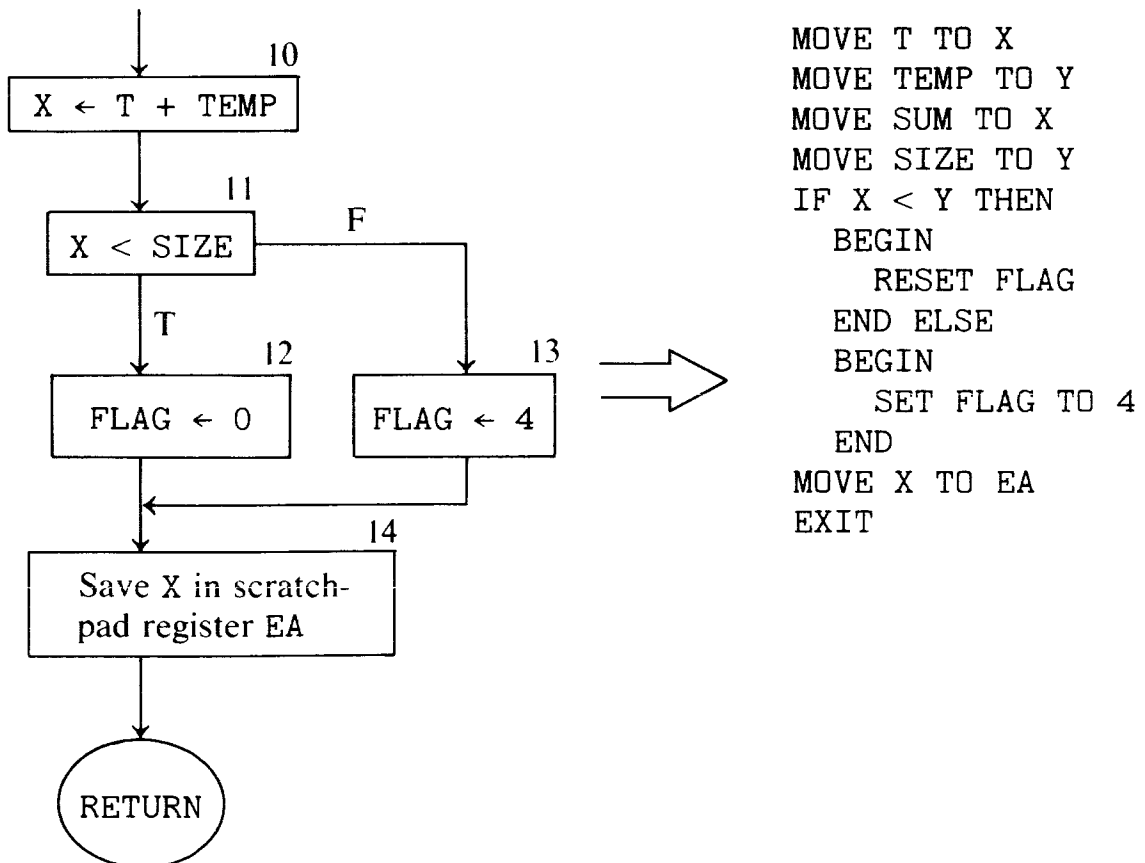


Figure 6.23.

possible beginning:

```

EFFECTIVE . ADDR  % ROUTINE BEGINS HERE
                  % INPUT IS POINTER IN FA TO INDEX
                  % FIELD
                  % OUTPUT IS FLAG (FLF REGISTER)
                  %   0 = OK
                  %   1 = TOO MANY INDEXES SPECIFIED
                  %   2 = NON DECIMAL ADDRESS FIELD
                  %   4 = EFFECTIVE ADDRESS OUT OF
                  %       BOUNDS

                  % ROUTINE USES X, Y, T, L, CP, FL, SOB
                  %       AND S1B AS LOCALS

```

BEGIN

```

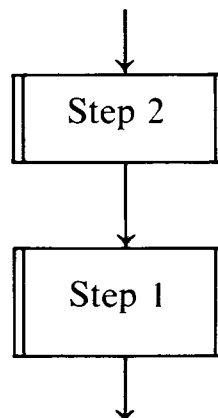
LOCAL . DEFINES
DEFINE FLAG      = FLF #
DEFINE CTR       = FLE #
DEFINE INDICATOR = SOB #
DEFINE TEMP      = S1B #

```

% MACRO TEN . T . PLUS . D GOES HERE?

2. Recode the logic of Figure 6.19 as a loop of the form shown in Figure 6.24. How many fewer instructions, if any, are required? Comment on the relative merits of the loop approach versus straight-line coding in this case.

3. *An exercise for those interested in efficient MIL coding.* The straightforward way to code a two-way selection step is to start with the flowchart structure shown in Figure 6.25. However, in the special case when step 1 and step 2 are such that executing the sequence



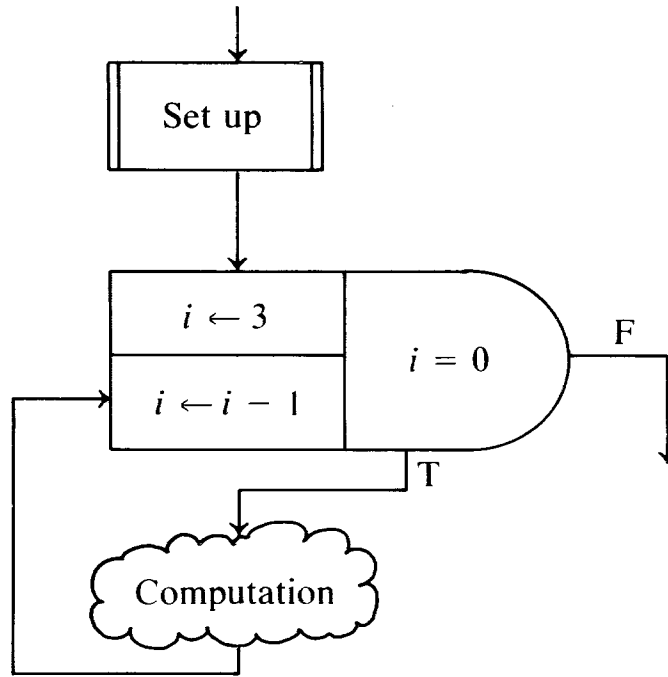
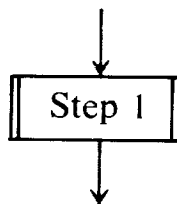


Figure 6.24.

has precisely the same net effect as executing just



alone, the selection step can be restructured in the form shown in Figure 6.26. Note that this special, but less obvious 2-way selection structure leads to slightly more efficient and compact MIL code. (One GO TO

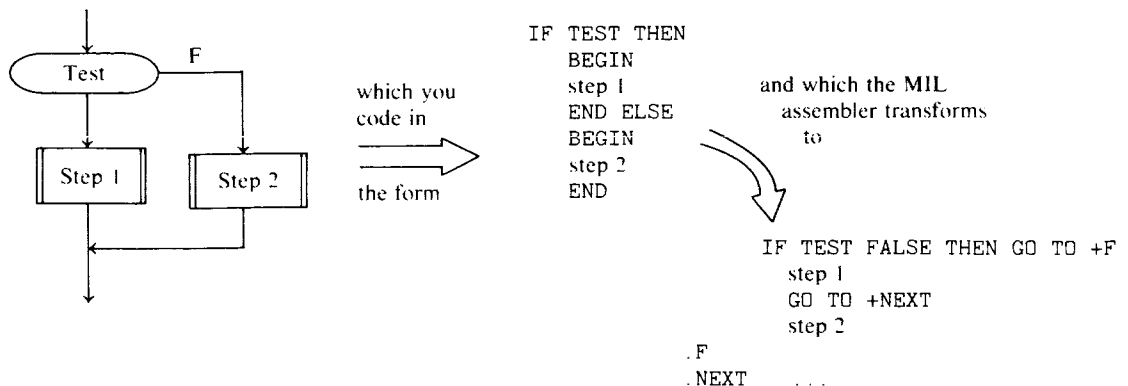


Figure 6.25.

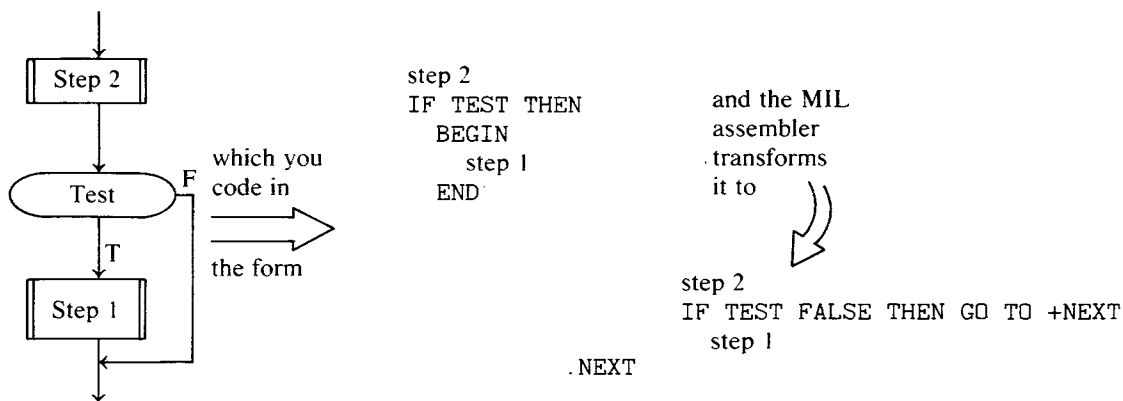
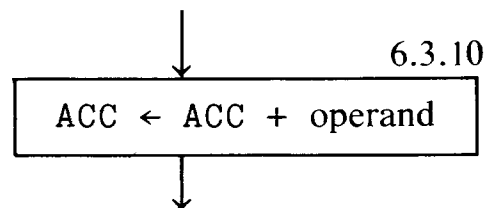


Figure 6.26.

instruction is saved.) Apply this principle to shave off one instruction from the code shown in Figure 6.20.

6.6 THE ADD ROUTINE

Recall that our principal objective in this chapter is to show the development of the utility routines useful for interpretation of SAMOS op-codes such as ADD. Back in Figure 5.11 we showed the (tentative) first-level details of that routine. At this point we seem to have developed all the utility routines except those needed to perform the actual decimal addition on two signed 10-decimal-digit SAMOS numbers. But in Chapter 5 we assumed that the operands of the addition routine would be found in G-store, so the arguments for addition would no doubt be pointers into G-store to these values. Since then, we have learned to convert operands into packed decimal representation and save them in double scratchpads. It will be much more efficient to perform addition (or for that matter, subtraction, multiplication, etc.) from validated packed decimal values. Thus, in coding step 6.3.10 of Figure 5.13,



we should assume that values of both ACC and operand are already represented in packed decimal form. For this to be a realistic assumption, we will also need one more utility routine. This one must *unpack* the result of the arithmetic operation (addition, subtraction, etc.) and move it to a designated 11-byte field of G-store. We will examine this

new routine, named UNPACK.AND.WRITE, after we consider the details for the routines needed to perform the actual addition.

Observe that the operand values stored in double scratch pads are of *signed magnitude* form. The leading bit is a sign, and the low-order 40 bits constitutes the magnitude. (See Figure 6.7.) If we only had to add (or subtract) the magnitudes, the addition (or subtraction) would be comparatively simple, as suggested in Figures 6.27 and 6.28, which show the utility routines, PLUS and MINUS, that form the sum $OP1 + OP2$ and difference $OP1 - OP2$, and assign the results to OP1.

The logic of addition (or subtraction) is more complex when the integer operands may be negative or positive. But the particular coding depends on whether we continue to represent the operands in signed-magnitude form and perform signed-magnitude arithmetic or convert negative operands to complement form, perform the addition, and then reconvert complement results to signed-magnitude results.

It turns out that 10s-complement arithmetic is quite convenient and efficient on the B1726, and we shall have a look at this approach at the end of this section. We examine first the signed-magnitude method, since it seems natural to simulate the signed-magnitude arithmetic of SAMOS via signed-magnitude logic on the B1726 (this reasoning does not necessarily lead to the most efficient simulation, however).

With signed operands, we must perform subtraction if the operands are of *unlike* sign, and moreover, the sign of that result depends on which of the two operands has the greater magnitude. The following table suggests the sign control logic we require, where the asterisk in the table signifies the sign of whichever operand had the larger magnitude.

Sign of OP1	+	+	-	-
Sign of OP2	+	-	+	-
Sign of addition result	+	*	*	-

A further complication arises in the event we are adding equal magnitudes of unlike sign. The result must be a positive zero, not a negative one.

Figure 6.29 illustrates the logic that implements the above sign control for signed-magnitude addition. Box 16 in this figure is a call on the utility routine PLUS which was illustrated in Figure 6.27. PLUS is called when the operands are found to be of like sign. The routine MINUS (Figure 6.28) is called (box 7) when the operands are of unlike sign but different in magnitude. Note that since MINUS is called only when the first operand exceeds the second one in magnitude, there is no possibility for

overflow as there was in the PLUS routine. The Flag parameter of ADD is reset in box 10 of the Figure 6.28 flowchart to reflect the fact that if control reaches this point there can be no overflow.

Exercises

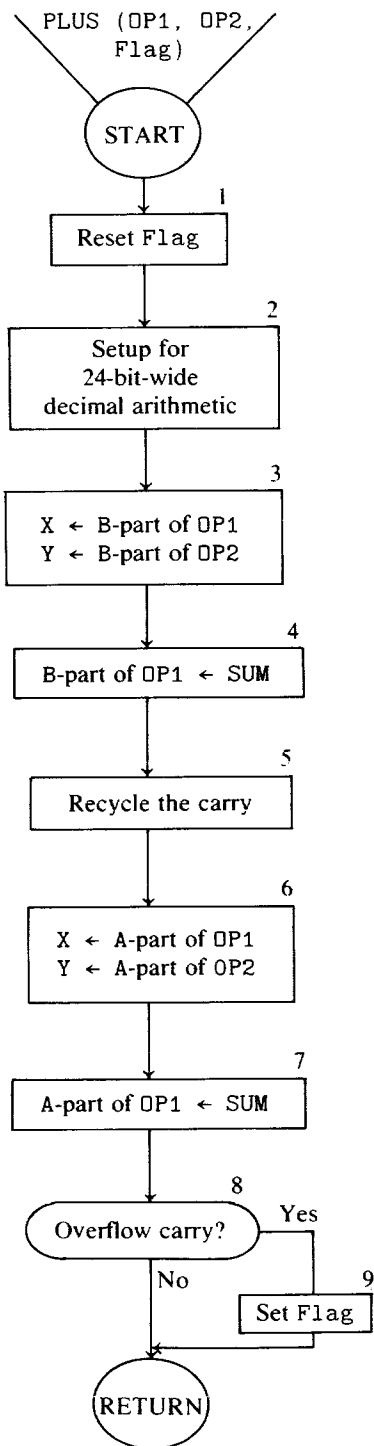
1. Code in MIL the ADD routine that is flowcharted in Figure 6.29.
2. The following table suggests the sign control logic for forming the *difference* between two signed magnitude integers, OP1 and OP2.

Sign of OP1	+	+	-	-
Sign of OP2	+	-	+	-
Sign of result for $ OP2 > OP1 $	-	+	-	+
Sign of result for $ OP1 > OP2 $	+	+	-	-

Construct a flowchart for a procedure SUBTRACT which computes the difference, $OP1 - OP2$, and assigns this result to OP1. One of two approaches might be taken:

- (i) Make the same assumptions used in the procedure ADD that was flowcharted in Figure 6.29. SUBTRACT should call on the PLUS and MINUS utility routines defined in Figures 6.27 and 6.28, and should obey the logic of the above table for control over the sign of the result. When OP1 and OP2 are of like sign and equal magnitude, the result assigned to OP1 should be a positive zero. Overflow indication is also needed in SUBTRACT.
 - (ii) Let SUBTRACT reverse the sign of either the first or the second operand and then call ADD.
3. What are the relative merits of the two approaches for constructing SUBTRACT, as discussed in the preceding exercise?

As many logic designers know, if we convert negative decimal operands to a 10s-complement form, add (or subtract), and then recon-vert complement results to signed-magnitude representation, the logic is simpler. For one thing, we cannot generate a minus zero in 10s-complement arithmetic, so that hazard (peculiar to signed-magnitude and to 9s-complement arithmetic) is avoided. Figure 6.30 shows the new and simpler logic for addition in the routine ADD.10.COMPL. The structure for a routine to perform 10s-complement subtraction would be almost identical. Only the name of the routine need be changed, and box 3



(a)

IDENTIFIER	TREATMENT	DESCRIPTION
OP1, OP2	Input parameters	Double scratchpads (10-decimal-digit magnitude)
OP1	Output parameter	Same
Flag	Output parameter	1-bit register to indicate overflow
X, Y	Local	

(b)

Figure 6.27. Subroutine for decimal addition of two ten-digit unsigned operands OP1 and OP2. The sum is assigned to OP1 and a flag is set in case of overflow. (a) Flowchart; (b) legend for PLUS; (c) subroutine.


```

PLUS          % SUMS TWO UNSIGNED 10-DECIMAL INTEGERS
              % IN DOUBLE SCRATCHPADS OP1 AND OP2
              % AND LEAVES RESULT IN OP1.  IF OVERFLOW, THE
              % FLAG BIT IS SET ELSE RESET
              % THIS ROUTINE USES S5, S6 AND CB(0) AS PARAMETERS
              % AND T, X AND Y AS LOCAL STORAGE

BEGIN
LOCAL.DEFINES
DEFINE OP1A = S5A#
DEFINE OP1B = S5B#
DEFINE OP2A = S6A#
DEFINE OP2B = S6B#
DEFINE FLAG = CB(0)#          % OVERFLOW

RESET FLAG
MOVE @(1)00111000@ TO CP    % SETUP FOR 24-BIT DECIMAL
                            % ARITHMETIC WITH 0 CARRYIN

MOVE OP1B TO X
MOVE OP2B TO Y

MOVE SUM TO OP1B
CARRY SUM                  % RECYCLES CYL TO CYF

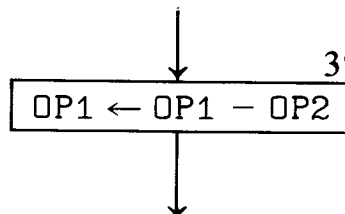
MOVE OP1A TO X
MOVE OP2A TO Y
MOVE SUM TO OP1A
% BOX 8
MOVE SUM TO T              % OVERFLOW DIGIT IS IN TB
IF TB(3) THEN SET FLAG    % OVERFLOW, THEN SET FLAG

EXIT
END

```

(c)

changed to



A simplifying feature of the Figure 6.30 flowchart is the suggestion that use of a procedure for complementing numbers can make the code more compact. Boxes 8, 9, and 10 show references to a function COMP. In boxes 8 and 9 the arguments to COMP are the magnitude parts of

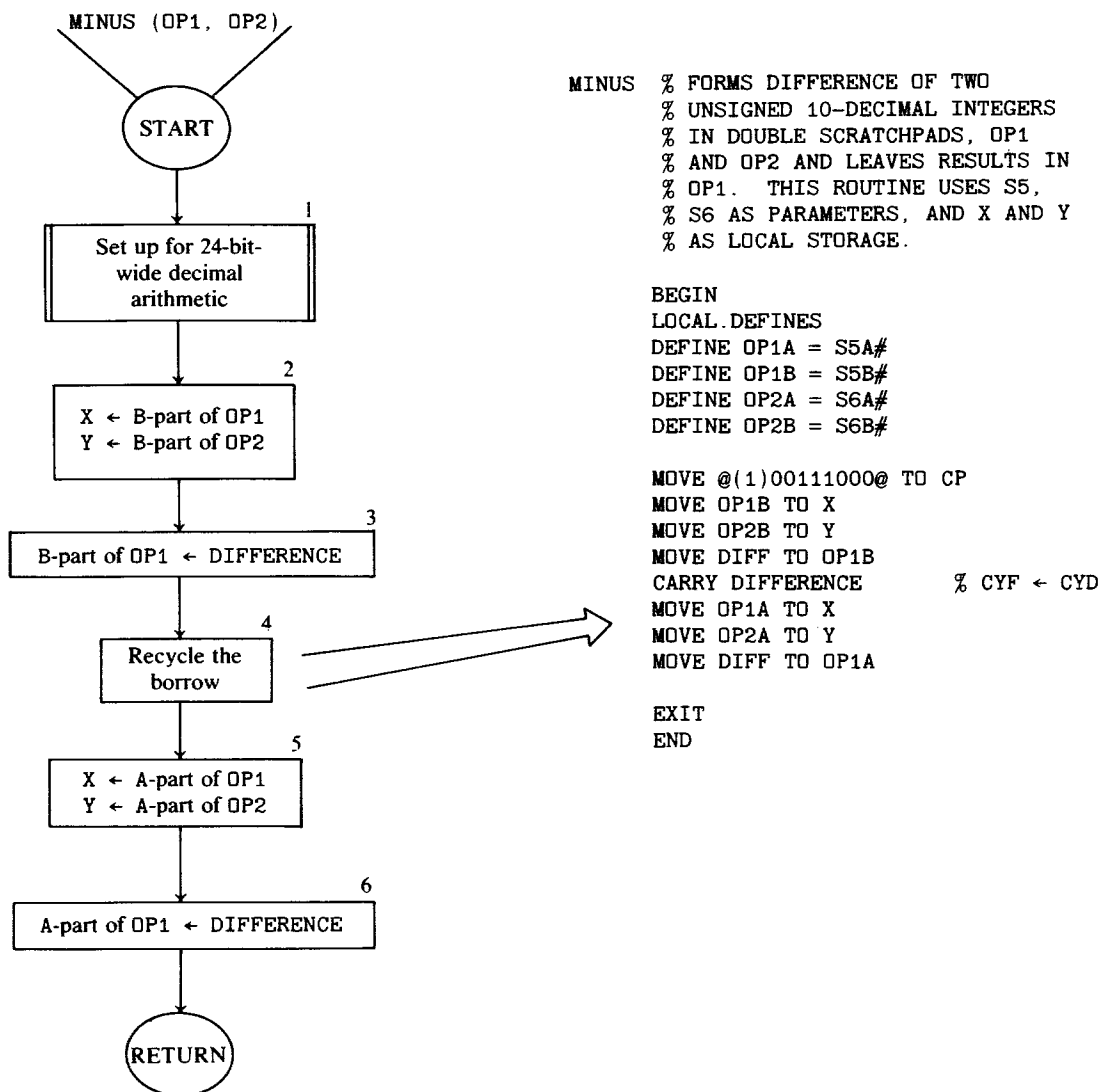


Figure 6.28. MINUS routine for subtracting two unsigned 10-decimal-digit integers. OP1 is assumed to be larger than OP2.

operands OP1 and OP2, respectively. If the result of the addition in box 3 is a number in complement form, then COMP is applied once again in box 10, this time to the result (which is regarded as an unsigned integer). A minus sign is then attached to the recomplemented result. Overflow, if any, is then detected at box 5.

Example Suppose we were dealing with signed, 2-decimal-digit integers. Table 6.1 shows traces of the Figure 6.30 algorithm for three different sets of operand values.

In applying the algorithm in Figure 6.30 to the task of simulating the SAMOS ADD operator, remember that OP1 would represent a copy of the accumulator and OP2 a copy of the operand fetched from the effective address location. So it does not matter that the value of OP2 is altered upon exit from the ADD. 10. COMPL procedure.

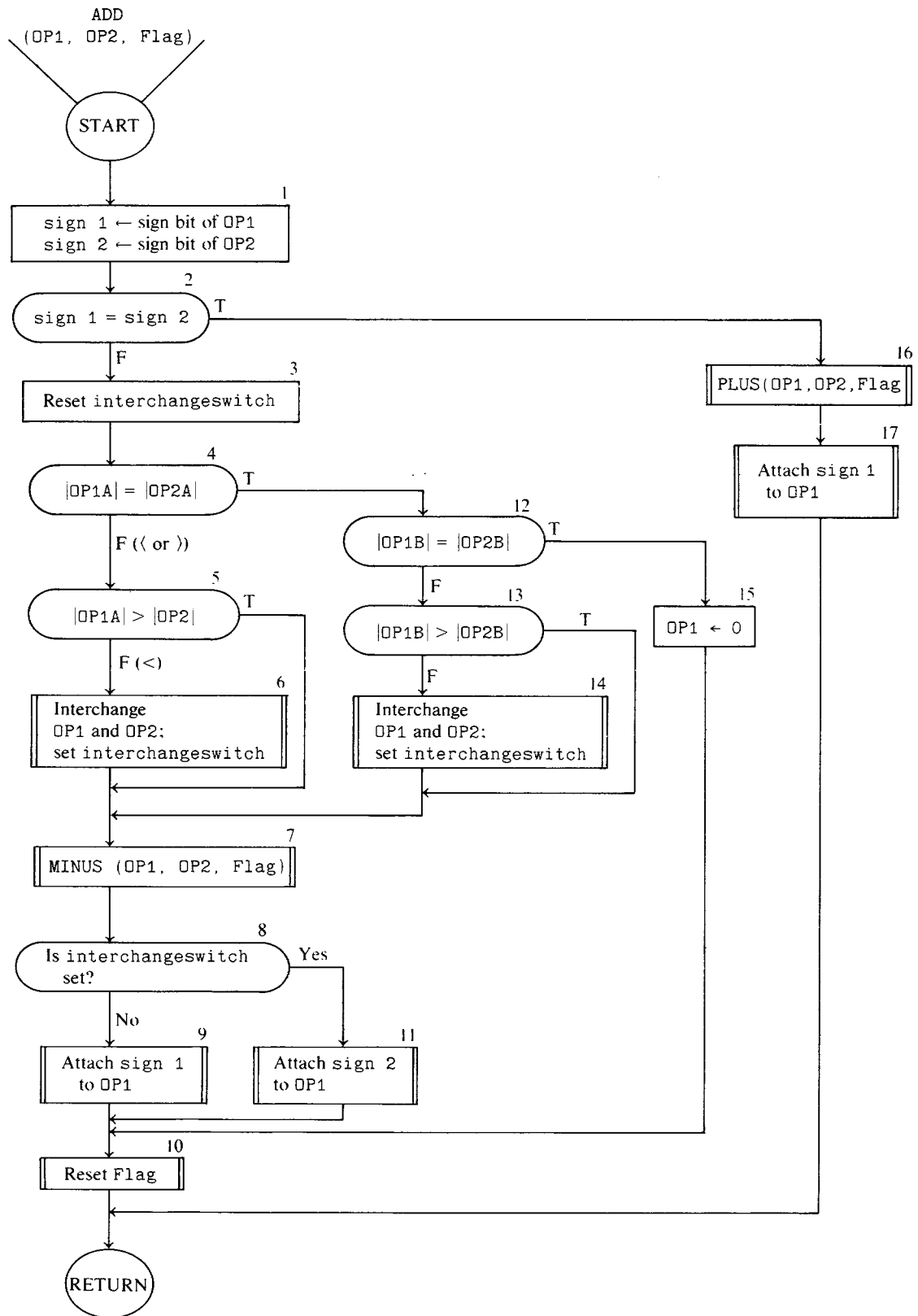


Figure 6.29. Logic for sign control in addition of signed 10-decimal-digit integers which may be of unlike sign. The integer operands OP1 and OP2 are assumed to be represented as packed decimal values in double scratchpads.

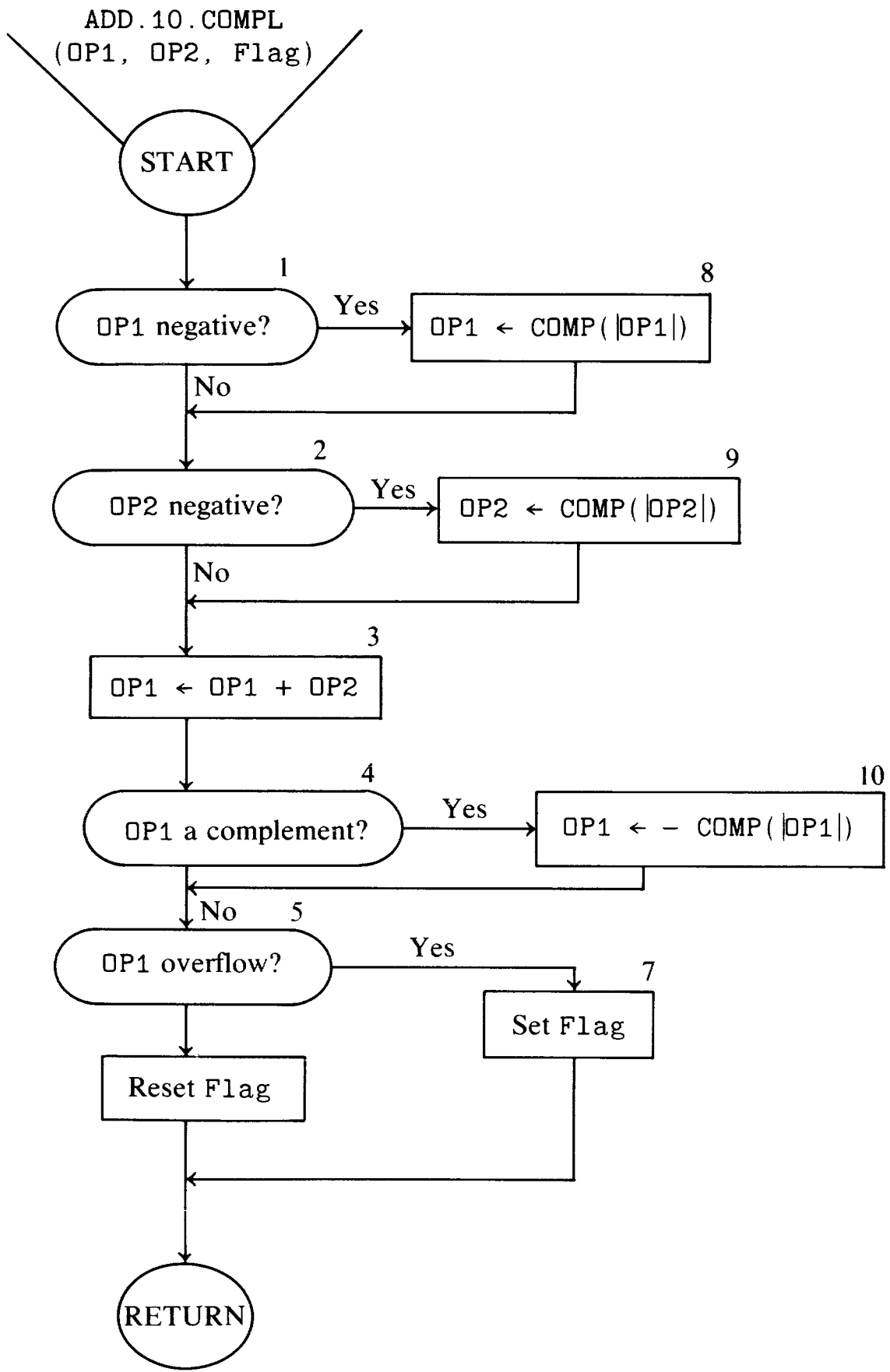


Figure 6.30. Logic for 10s-complement addition. The function procedure COMP returns the 10s complement of its argument.

```

ADD.10.COMP % ROUTINE BEGINS HERE. THE
             % ARGUMENTS ARE OP1, IN S5, OP2 IN S6
             % IN SIGNED MAGNITUDE FORM AS IN FIG. 6.7. RESULT IS
             % LEFT IN OP1, ,IE., S5, AND OVERFLOW FLAG IN CB(0).
             % THE STACK, X, Y, T, AND L ARE USED AS LOCAL STORAGE
             % THE PROCEDURE COMPL.T.L. CONVERTS T CAT L TO 10'S COMPLEMENT

```

```

BEGIN
LOCAL.DEFINES
DEFINE OP1A = S5A #
DEFINE OP1B = S5B #
DEFINE OP2A = S6A #
DEFINE OP2B = S6B #
DEFINE FLAG = CB(0) #

MOVE @(1)00111000@ TO CP % SETUP FOR 24-BIT DECIMAL ARITHMETIC
%BOX1
MOVE OP1A TO T
MOVE OP1B TO L
IF T(0) THEN % IF OP1 NEGATIVE
BEGIN % COMPLEMENT |OP1|.
RESET T(0)
CALL COMPL.T.L.
END
MOVE T TO TAS % SAVE OP1 ON STACK
MOVE L TO TAS % FOR LATER USE
%BOX2
MOVE OP2A TO T
MOVE OP2B TO L
IF T(0) THEN % IF OP2 IS NEGATIVE,
BEGIN % COMPLEMENT |OP2|.
RESET T(0)
CALL COMPL.T.L.
END
%BOX3
MOVE TAS TO X % LOW-ORDER PARTS OF OP1 AND
MOVE L TO Y % OP2 IN X AND Y, RESPECTIVELY
MOVE SUM TO L % LOW PART OF OP1 + OP2 IN L
CARRY SUM % RECYCLE CARRY DIGIT
MOVE TAS TO X % HIGH-ORDER PARTS OF OP1 AND
MOVE T TO Y % OP2 IN X AND Y, RESPECTIVELY.
MOVE SUM TO T % HIGH PART OF OP1 + OP2 IN T
%BOX4
IF T(0) THEN % IF RESULT IS A COMPLEMENT
BEGIN % IF HIGH-ORDER DIGIT IS AN 8 OR A 9
CALL COMPL.T.L. % COMPLEMENT AND
SET T(0) % MARK MINUS
END
%BOX 5
IF TB NEQ 0 THEN % IF 11TH DIGIT OF SUM NON ZERO,
BEGIN % THEN WE HAVE OVERFLOW
SET FLAG
END ELSE
BEGIN
RESET FLAG
END
MOVE T TO OP1A % NEW RESULT LEFT IN OP1
MOVE L TO OP1B
EXIT
END

```

Figure 6.31. MIL code corresponding to flowchart in Figure 6.30. See Figure 6.32 for COMPL.T.L. procedure code.

TABLE 6.1

JUST BEFORE EXECUTING	CASE:		2		3			
	1		OP1	OP2	OP1	OP2		
Box 1			77	35	-77	35	-29	-82
Box 2			77	35	923	35	971	-82
Box 3			77	35	923	35	971	918
Box 4			102	35	958	35	889	918
			(not a complement)		(complement)		(complement)	
Box 5			102		-42		-111	918
			(overflow)		(no overflow)		(overflow)	

Figures 6.31 and 6.32 show MIL coding for `ADD.10.COMPL`. The code in Figure 6.31 illustrates for the first time in this text how we may use the top of the hardware stack for fast-access temporary storage, avoiding the need to use scratchpads, which on some occasions may be in short supply.

The “double register” `T CAT L` is used as the principal working register. After moving `OP1` (from `S5`) into `T CAT L`, `T(0)` is tested for the presence of a sign bit. If *on*, then `COMPL.T.L` (Figure 6.32) is called to complement the contents of `T CAT L`. In any case, `T CAT L` is then saved on the top of the stack, freeing `T CAT L` to receive a copy of `OP2` (from `S6`). Later, when executing the addition step (box 3), the value of `OP1` (possibly complemented) is popped off from the stack and moved to `X` as input to the 24-bit function box. The code in Figure 6.31 can be shortened if `OP1` is assumed already to be in `T CAT L` upon entry to the subroutine and if the result can be left in `T CAT L`.

The subroutine `COMPL.T.L` is shown in Figure 6.32. A 10s complement is produced by subtracting that value from 0. The setup instructions initiate the 24-bit function-box controls and set `X` to 0. The last instruction, `CARRY 0`, “tidies up” the function box for subsequent use in 24-bit decimal arithmetic by resetting `CYF`.

Exercises

1. Write MIL code for the routine `SUB.10.COMPL` (subtract using 10s-complement arithmetic, following the logic of Figure 6.30, but with needed changes to reflect subtraction). Can we again make use of the subroutine `COMPL.T.L`?

2. Is there a simpler way to code the routine `SUB.10.COMPL`, described in the preceding exercise? *Hint*: How about coding

```

COMP.T.L      % PROCEDURE BEGINS HERE.
              % THIS PROCEDURE COMPUTES THE 10'S COMPLEMENT OF
              % T CAT L AND LEAVES THE RESULT IN T CAT L,
              % USING X AND Y AS LOCAL STORAGE
MOVE @(1)00111000@ TO CP      % TO BE SURE OF ARITH. SETUP
CLEAR X                      % MORE SET UP
MOVE L TO Y
MOVE DIFF TO L                % LOW ORDER PART OF COMPL. IN L
CARRY DIFFERENCE              % RECYCLE THE BORROW
MOVE T TO Y
MOVE DIFF TO T                % COMPLEMENT IN T CAT L
CARRY 0                        % LEAVE CARRY IN "CLEAN STATE"
EXIT

```

Figure 6.32

SUB.10.COMPL so it calls on ADD.10.COMPL after first reversing the sign of OP2? What are the relative merits of these two approaches for coding SUB.10.COMPL?

3. Compare the MIL coding needed for the signed magnitude ADD routine (Figure 6.29) with the MIL coding developed for 10s-complement addition (Figure 6.30). Which code has more lines? How many more? Which code executes in fewer instructions? How many fewer?

4. A student has studied the MIL code in Figures 6.31 and 6.32 and claims that a net decrease of 2 lines of code can be achieved. She says that the two instructions RESET T(0) in boxes 1 and 3 and the instruction SET T(0) in box 10 could be eliminated if the instruction RESET T(0) were inserted at the beginning of the code for COMPL.T.L. Verify whether or not she is correct, and if correct, explain why.

6.7 UNPACK.AND.WRITE

The last utility routine to be discussed in this chapter is one which will take the packed decimal result of the 10-digit decimal addition, subtraction, etc., unpack it, and store it in a SAMOS word in G-store. Only after this step is taken will the interpretation of a SAMOS ADD, SUB, MPY, or DIV instruction be completed.

The procedure UNPACK.AND.WRITE, shown in Figure 6.33, takes the signed decimal integer in S5 and stores it as an equivalent 11-byte character field pointed to by the parameter value in FA. The logic of this procedure is in essence the inverse of that used for packing in VALIDATE.DECIMAL.

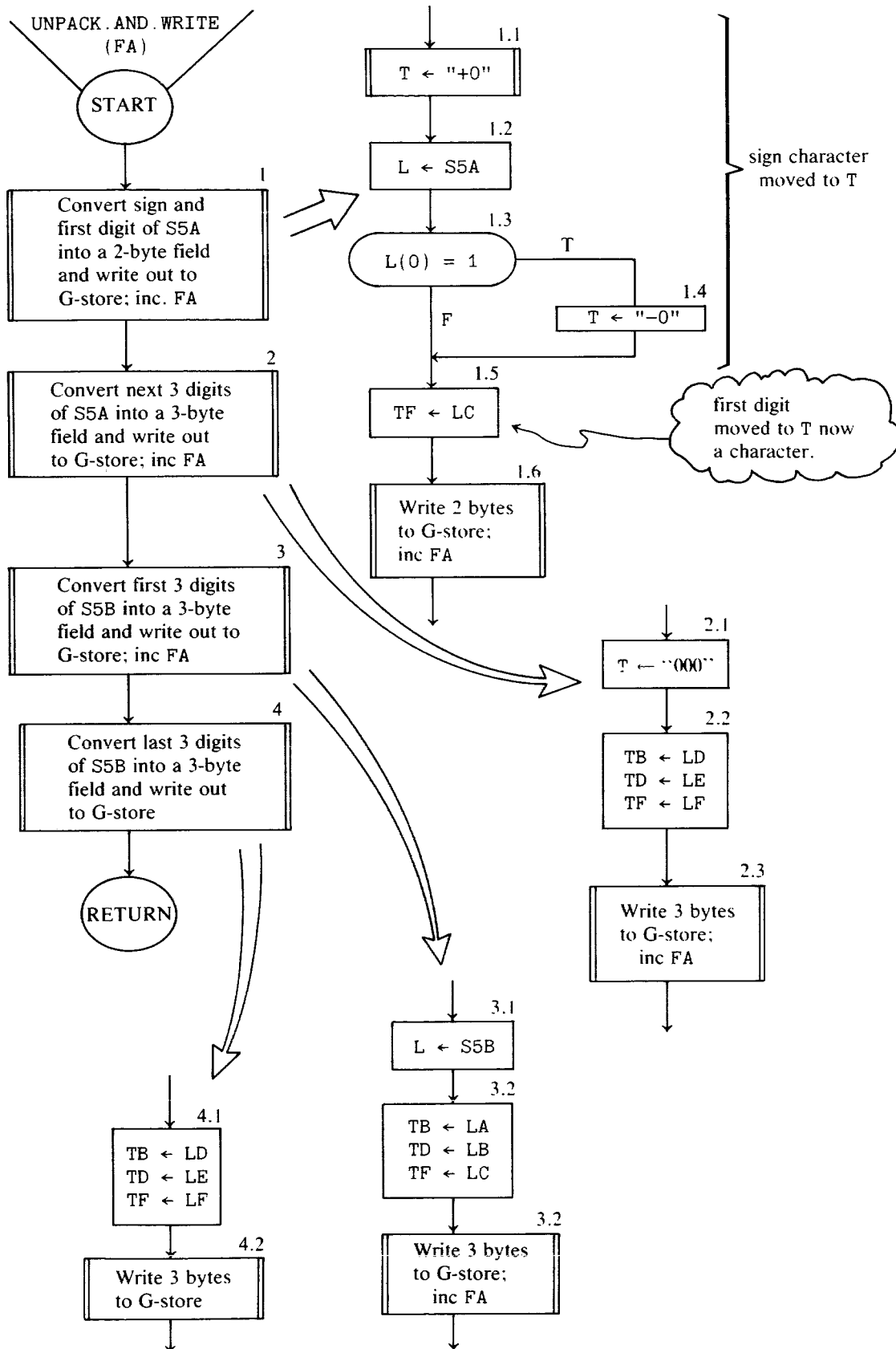


Figure 6.33. Procedure for unpacking a signed 10-decimal-digit integer in a double scratchpad (S5) and storing it as an 11-character field in G-store at the address given by the value in FA.

Exercise Produce the MIL-code equivalent of the flowchart given for UNPACK.AND.WRITE in Figure 6.33.

6.8 CHAPTER SUMMARY

We began this chapter with the task of designing all the routines needed for coding the details of the SAMOS ADD operator routine charted in Figure 5.13. We thought those utility routines listed in Table 5.2 were what we wanted. As often happens when one gets down to the details, new ideas crept in, and we departed from our earlier implementation concepts. It became clear in this chapter that we should input decimal character fields from G-store, but convert them to packed decimal representation for use as arithmetic operands and then upack the results and send them to G-store as decimal character byte strings.

The utility routine specifications listed in Table 5.2 now need to be updated to reflect these changes. We leave this task to the reader as a useful stock-taking exercise. Note that we did not change the specifications on two of the most basic routines, BINARY.TO.FA and ADDRESS.TO.BINARY, so the plans suggested in Figures 5.9, 5.10, and 5.11 for decoding are still quite valid. But the logic suggested for the top level of the ADD operator routine, as given in Figures 5.13 and 5.14, must be altered (and augmented) to reflect the new specifications for VALIDATE.DECIMAL, EFFECTIVE.ADDRESS, PLUS, MINUS, ADD (or ADD.10.COMPL, SUB.10.COMPL, COMPL, and COMPL.T.L), and UNPACK.AND.WRITE. We also leave to the reader the modification of Figures 5.13 and 5.14 as useful summarizing exercises.

For those not wishing to indulge in exercises, Appendix E presents solutions in the form of a tested McMIL version of the flowcharts in this and the preceding chapter. This appendix gives an abridged version of SAMOS (a basic set of eight instructions), an accompanying LOADER program for defining the needed workspace, and a sample data deck and execution.

As a final remark, we observe that our odyssey through the design exercises of this chapter has exposed us to nearly all the power and limitations of the B1726 microprocessor architecture and to many coding techniques. We have used nearly all the instructions in one way or another, and in doing so have begun to appreciate what is involved in achieving optimal or near-optimal MIL code. The astute reader should be able to apply many of these methods to the design of other interpreters.

Chapter 7

The split-level control store

One of the most interesting aspects of the B1726 architecture, so far only hinted at (end of Chapter 1), is the feature that allows microinstructions to be processed directly out of G-store as well as out of H-store. Thus, in addition to serving primarily as a store for guest-language code and data, G-store is also used as an “extension” of H-store. This feature is attractive because if an interpreter is too large to fit into the more expensive and faster H-store, then the less frequently used parts of the interpreter can be kept in the same physical storage medium as G-store and executed directly from this store,¹ perhaps without seriously degrading the performance of the interpreter. If necessary, it is still possible to use overlay methods and move a block of microinstructions from G-store to H-store whenever such a block needs, for reasons of efficiency, to be executed from H-store. In addition, in the extreme case where no space in H-store is available, an interpreter would be executable entirely from G-store.

We mentioned in Chapter 1 that the B1800 uses a cache store for holding most-recently fetched microinstructions. This approach creates the effect of having a large fast H-store without requiring the system

¹ In Chapter 1 we introduced H-store and G-store as conceptual stores (host and guest), but subsequently we have discussed them as if they are also actual physical storage devices. We could do this without blushing because, to a first approximation, H-store maps onto what Burroughs calls M-memory, and G-store maps onto what Burroughs calls S-memory, two physically different stores. The truth, however, is that microinstructions can be fetched and processed from either physical storage, which means that H-store maps in part onto M-memory and in part onto S-memory. There is a dilemma to be faced here with regard to the notation we should use in this chapter. Shall we be technically correct and refer to the physical stores by different names (M and S) to distinguish them from the conceptual names (H and G)? If so, we will have *four storage names* to keep straight, and this will become tedious. Or shall we risk confusion by continuing to apply to the physical storage devices and the conceptual storage devices the same names (H and G)? We opt for the latter, but hope the reader will realize that in the remainder of this chapter all references to H- and G-store are to the physical stores (M and S respectively). They are only coincidentally to be regarded as names for conceptual stores—and then, only if applicable.

programmer to manage it as a scarce resource. The B1800 approach is clearly simpler (for the programmer) than the solution used in the B1726. In fact, most of the programming problems discussed in this chapter are precisely the ones which have been eliminated using the cache microinstruction store of the B1800. By way of illustration, one of the technical problems whose solution is discussed fully in the body of this chapter is mentioned briefly here.

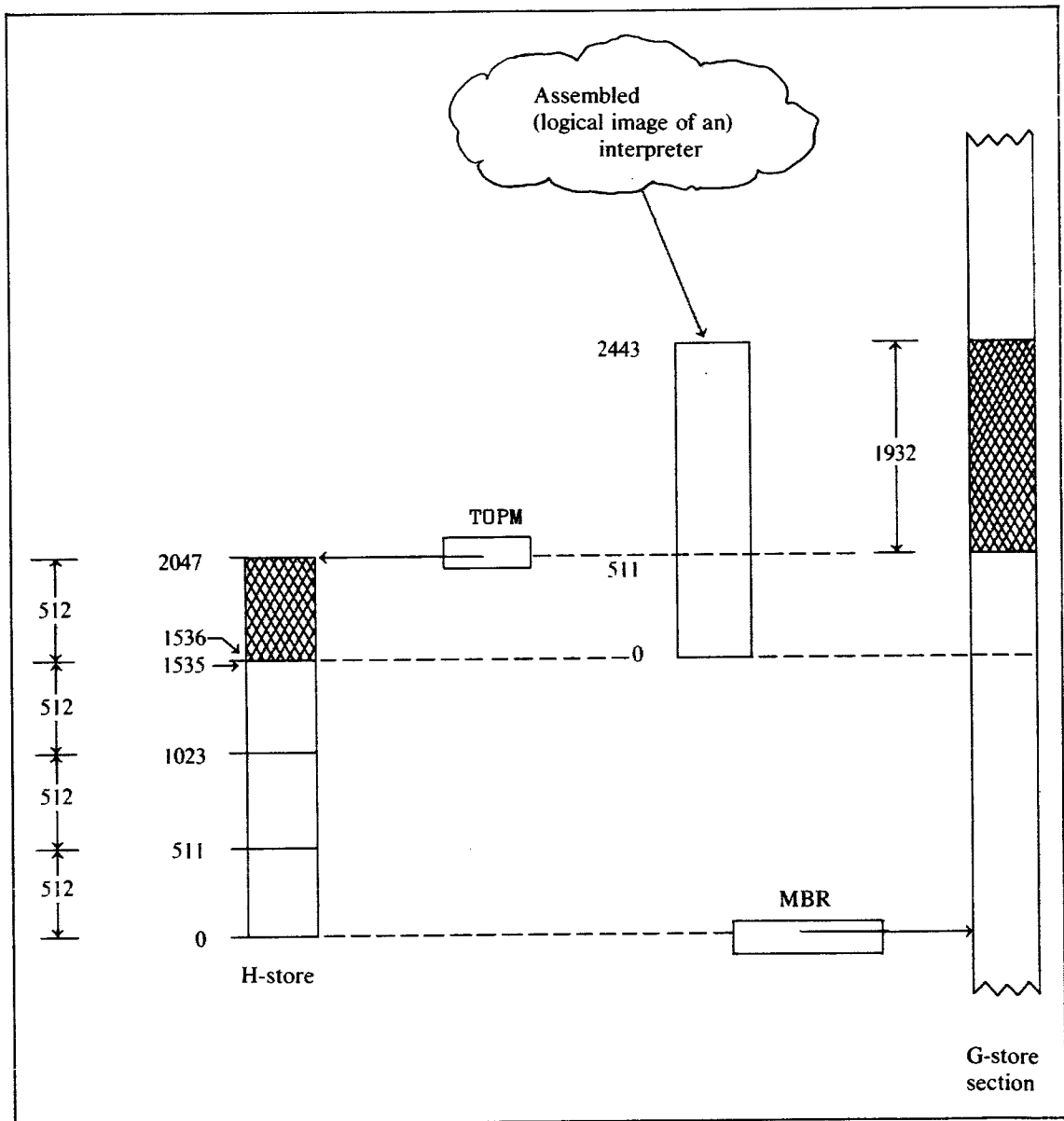
Often we have read-only tables and other long literals which are convenient to embed directly in the program part of an interpreter. However, H-store is organized to optimize the fetching of 16-bit microinstructions, and G-store is organized to optimize the fetching of data in chunks of up to 24 bits. Hence, different microinstructions must be used to read data to the processor, depending on which store the data reside in.² Given that the MCP has control over allocation of H-store to various interpreters and given that such allocation may vary according to the workload on the system, it is not practical for a MIL programmer to read data embedded in code that resides in H-store, since he has no assurance it will indeed reside in H-store when his interpreter is executed. Therefore as a practical matter, when data are to be read from the interpreter (apart from 8- or 24-bit literals transferred as part of a MOVE instruction), they must be read from the G-store-resident part of the interpreter. This can be done and is done, although the address arithmetic needed to calculate the absolute G-store address of such data is more awkward than we might wish (and more awkward than for fetching data from G-store workspace, which is simply based on the value of the base register, BR). As part of the solution to this problem, the system designers provide a MIL programmer the option to specify a part of an interpreter that *must* reside in G-store while the interpreter is being executed, and the system obeys and respects this specification. Use of this provision then offers the programmer the assurance of knowing precisely how to transfer data embedded in such code.

In this chapter we will explain in detail how the B1726 processes instructions from either store. Armed with this information, it will be easy to explain the precise way that control is switched from one interpreter to another, and thus to explain the nature of the interface between user interpreters and system service modules (the MCP and the central i/o-control, interrupt-handling, and process-switching module known as GISMO).

² On the B1726 the READ MSML TO X and WRITE MSML FROM X microinstructions, primarily used for diagnostic tests of H-store, allow the reading and writing of 16-bit fields from/to H-store and the X-register. We have not described this instruction elsewhere in this text, except to mention it in Appendices A and B.

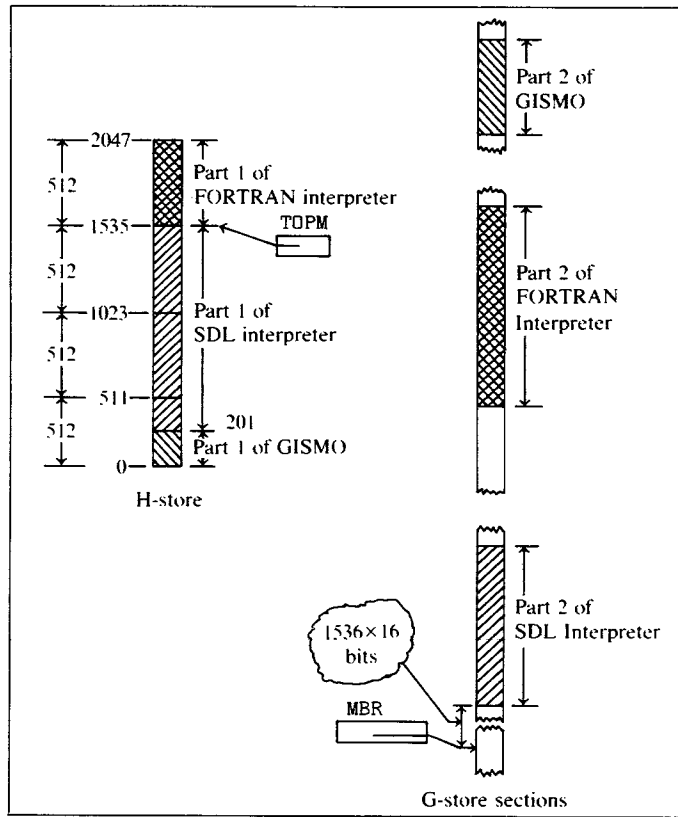
7.1 CONTROL OVER THE USE OF H-STORE

Since H-store is considered a scarce resource, interpreters that run under control of the MCP are allocated space in H-store by the MCP. The MCP manages H-store, in part by relying on software modules like the MIL assembler to adhere to certain agreed-upon conventions. In this section we will not look in detail at the MCP allocation strategies, but

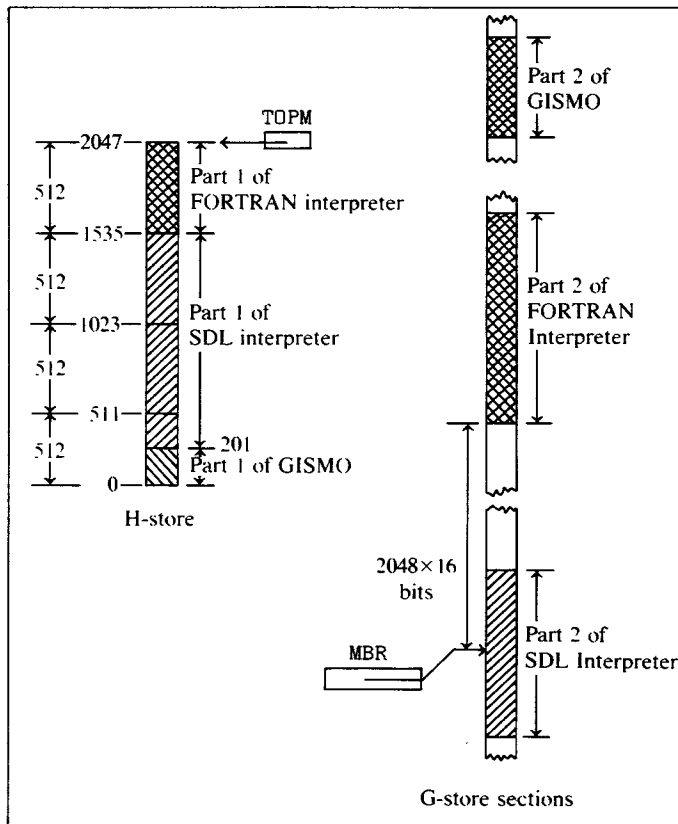


(a)

Figure 7.1. (a) Interpretation of part of G-store as an addressable extension of H-store. (b) Settings of MBR and TOPM for executing the SDL Interpreter. (c) Settings of MBR and TOPM for executing the FORTRAN Interpreter.



(b)



(c)

rather examine how the H-store *might* be managed, considering the hardware features provided.

Figure 7.1(a) suggests how a portion of G-store may be viewed as an addressable extension of H-store when executing an interpreter whose total length is 2444 microinstructions, whose first 512 microinstructions reside in a 512-word block of H-store, and whose remaining (1932) microinstructions reside in G-store. In this figure it is assumed that the H-store has a capacity for storing only 2048 microinstructions.

As pictured, microinstructions at logical addresses 0 through 511 should be fetched from H-store when the A-register has values 1536 through 2047 respectively. Instructions at higher logical addresses (512 through 2443) should be fetched from their locations in G-store when the A-register has values of 2048 and above (in particular, up to and including 3979, which is $1536 + 2444 - 1$). The use of two key registers, named TOPM and MBR, enables the B1726 processor to accomplish this feat. How it is done is explained in detail in the next sections.

What is important to note here is that the particular block of H-store where the first part of the interpreter is stored and the particular section of G-store where the remainder of the interpreter is kept can be chosen with some degree of freedom. It is only necessary to preset in a compatible way the values for TOPM and MBR. As we will see later, the TOPM register sets the bound on values of A for instructions to be fetched from H-store, and the MBR provides the base address such that the value $A + MBR$ becomes the effective G-store address for instructions in the G-store part of the interpreter.

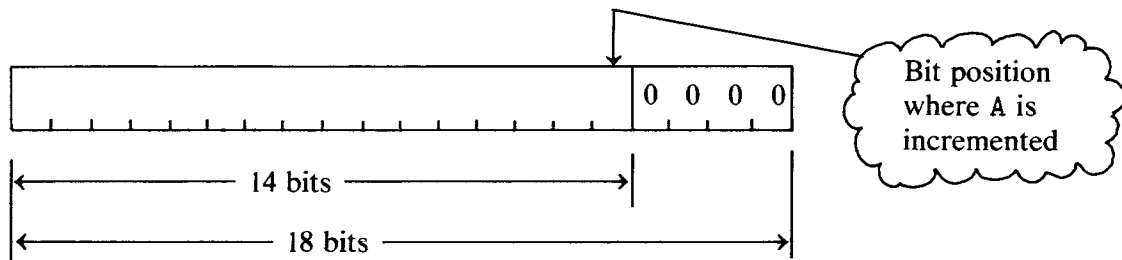
With these concepts in mind we can now picture that the first parts of two or more interpreters may be loaded into H-store and their respective second parts placed in G-store wherever adequate and available space can be found. Figure 7.1(b,c) shows a possible allocation of H-store which is feasible when only one user program (e.g., a FORTRAN program) is executing on the B1726. For comparison, part (b) of Figure 7.1 shows the MBR and TOPM settings when the SDL³ interpreter is executing, and part (c) shows the MBR and TOPM settings when the FORTRAN interpreter is executing. (Again an H-store size of 2048 16-bit words is assumed.) The second parts of each of these interpreters are shown in G-store. (We assume GISMO can also be regarded as an interpreter.) There is no particular relationship required between the

³ The SDL or Systems Development Language is an ALGOL-like language in which the MCP has been coded. See "B1700 Systems System Software Development Language (SDL) Reference Manual," Burroughs Corporation, Detroit, December 1973, Form 1072493.

relative positions of the part-1 portions in H-store and the relative positions of the part-2 portions in G-store.

7.2 MICROINSTRUCTION FETCH FROM H-STORE OR G-STORE

The address range of a single B1726 microprogram spans 2^{14} (or 16K) microinstructions, governed by the structure and function of the A-register which serves as the processor's instruction counter:



Because the low-order 4 bits of the 18-bit register A are always zero, the A-register can refer only to bit addresses at 16-bit “word boundaries”.

A limit register, known as TOPM, is set under program control to mark the upper-bound address in H-store. Instructions whose addresses are higher than the mark set by TOPM, but less than 2^{14} , are fetched from G-store at an offset from a G-store base address determined by the contents of the MBR register, as suggested in Figure 7.2. In this figure the

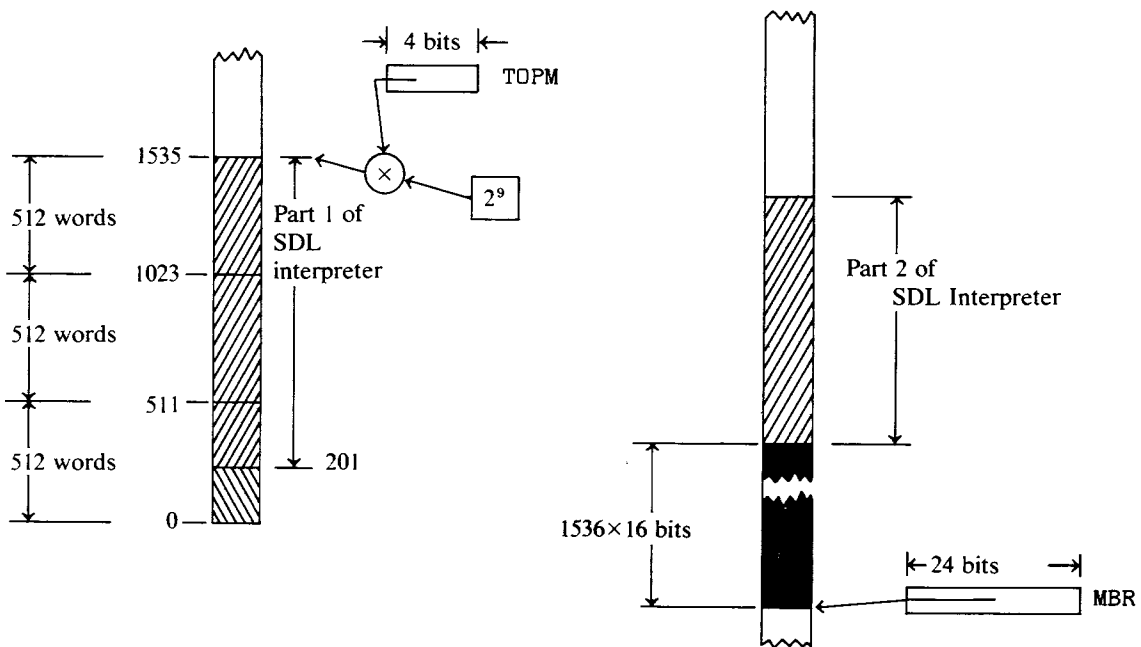


Figure 7.2.

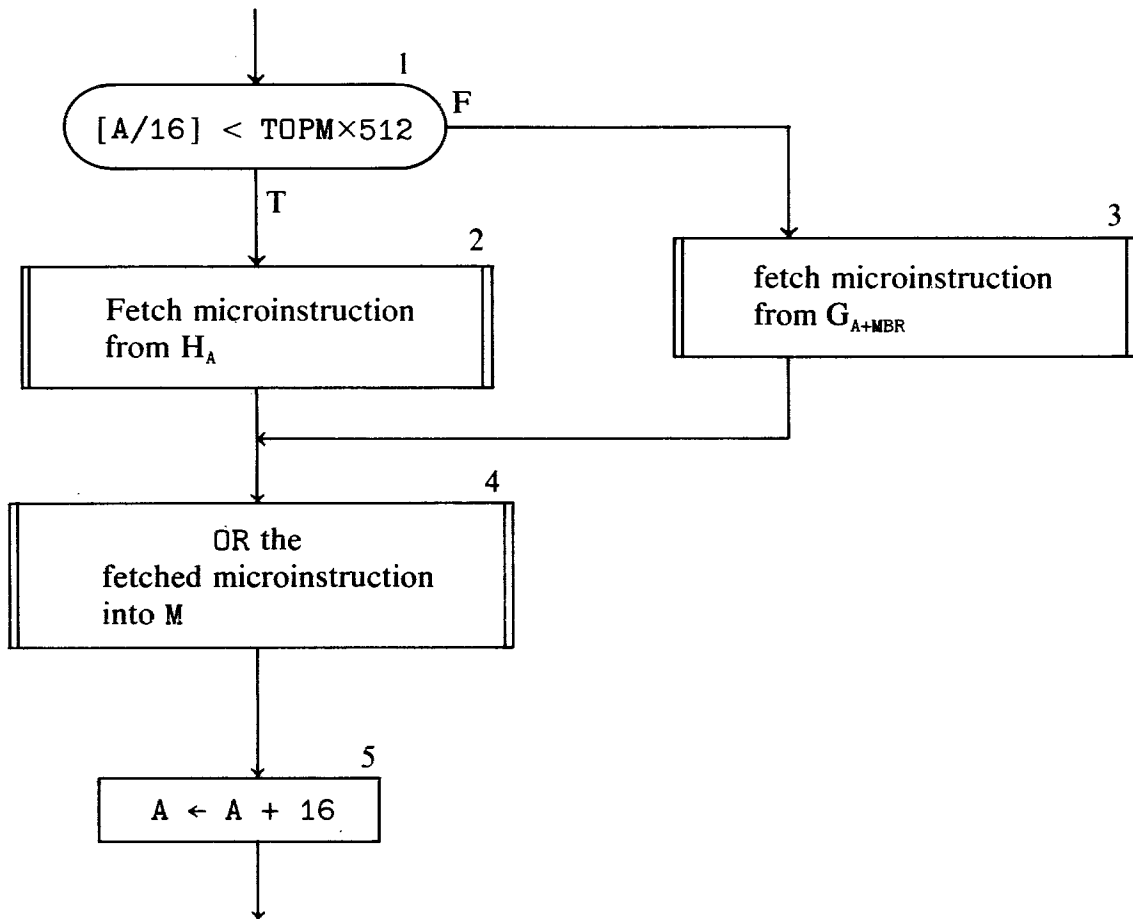


Figure 7.3. Fetch and increment logic details.

presumption is made that the processor is executing the SDL interpreter loaded in H- and G-store as first suggested in Figure 7.1(b,c).

Figure 7.3 shows the logic used in the hardware during each fetch cycle for selecting addresses from either H- or G-store, as the case may be. First the value of⁴ $[A/16]$ is compared with the product $TOPM \times 512$. If less than $TOPM \times 512$, then the microinstruction is fetched from H-store at the bit address whose value is A , or else the microinstruction is fetched from G-store at the bit address whose value is $A+MBR$. In either case the fetched 16-bit instruction is ORed into the M-register and the A-register incremented by 16.

In the example given in Figure 7.2, $TOPM$ would have the value 3; hence any value of $[A/16]$ that is greater than or equal to 3×512 (i.e., ≥ 1536) refers to a microinstruction located at bit address $A + MBR$ in G-store (a base-register value plus a single offset). We can now see why

⁴The square brackets represent the so-called *greatest integer* function. Thus $[A/16]$ is the integer quotient of A divided by 16.

the proper value for MBR is the G-store address at a distance of 1536×16 below the address of the first microinstruction in part 2 of the SDL interpreter. In Figure 7.2 this "image" of H-store from locations 0 through 1535 is shown as the black segment in G-store. Of course, the information stored in the black section need have no relation whatsoever to the SDL interpreter. The black section is shown in the diagram merely to indicate how the value of MBR must be preset before the fetching of SDL interpreter instructions from G-store can be done properly.

Two additional observations regarding these described hardware features are worth noting.

1. When TOPM is preset to zero before executing an interpreter, the test in box 1 of Figure 7.3 will always be false; hence all microinstructions will be fetched from G-store. This is the important special case we alluded to earlier, where no H-store space is available that can be allotted to an interpreter.
2. The maximum value of TOPM is 15, since TOPM is a 4-bit register. Hence, the largest size for H-store in a B1726 is 15×512 or 7680 microinstructions.

7.3 EMBEDDING TABULATED DATA IN MIL PROGRAMS

If tables of data are to be embedded in the interpreter, for reference during execution of the interpreter, it is essential that such tables reside in G-store. The programmer can ensure this eventuality by use of the special MIL pseudoinstruction `M.MEMORY.BOUNDARY MAXIMUM`. Any code or table that follows this instruction in a MIL program will be earmarked by the assembler so that at load time this section of the program will appear in G-store.

Before enlarging on this remark with an example, we digress here to describe the `TABLE` declaration available in MIL which has not been discussed earlier. A `TABLE` declaration has the format

```
TABLE label
  BEGIN
    first literal
    second literal
    :
    last literal
  END
```

For example,

```

TABLE MESSAGES
BEGIN
  "OVERFLOW"
  "UNDERFLOW"
  "_INVALID CHARACTER"
  "INFINITE LOOP"
  @(1)1111@
  @F2F3@
END

```

Such a declaration will cause the MIL assembler to generate and insert in line a sequence of bit strings representing the declared literals in the sequence given.

To read from the table into X, Y, T, or L it is necessary to compute the absolute address of the table so that that address can be placed in FA. The required computation can best be appreciated by a study of Figure 7.4.

In this figure we picture an interpreter which is partly resident in H-store (part 1) and partly resident in G-store (part 2). To compute the address of MESSAGES, it is necessary to add two offsets to MBR. The

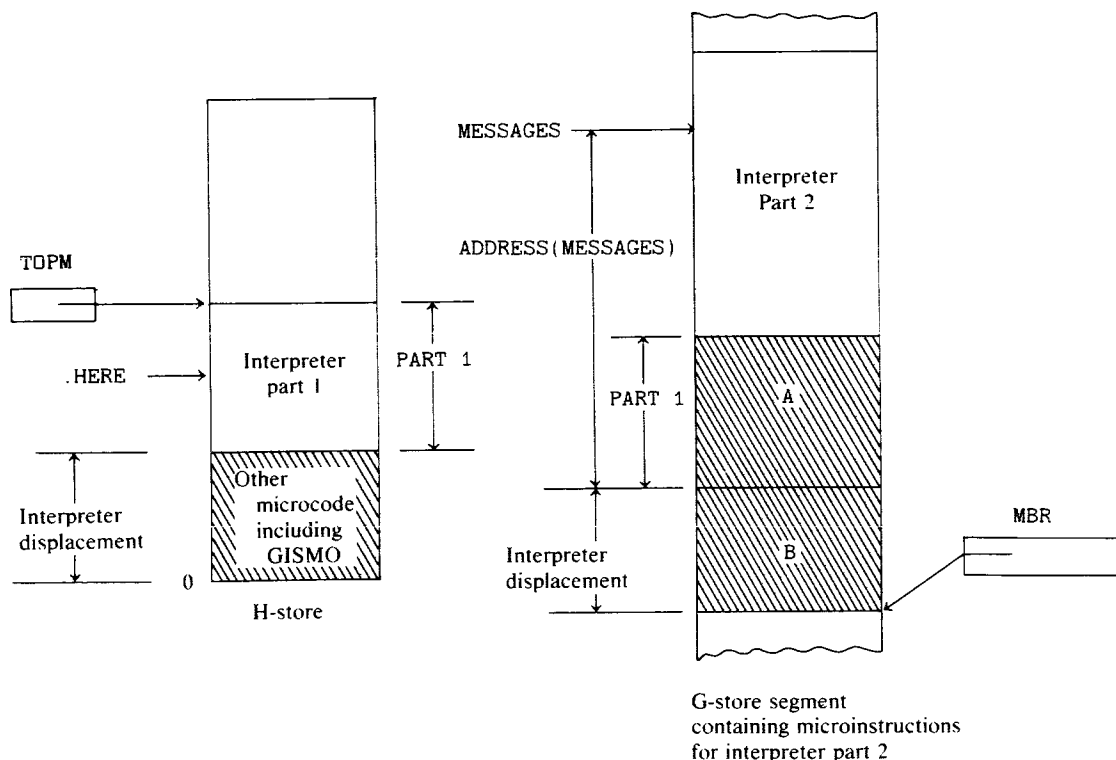


Figure 7.4.

first offset, called *interpreter displacement* in the diagram, is the offset from address 0 in H-store at which the first instruction of the interpreter (part 1) has been loaded. (Note that if indeed the interpreter is entirely resident in G-store, then interpreter displacement would be zero and Part 1 of the interpreter would reside in the shaded region marked A in G-store.) The second offset, ADDRESS(MESSAGES), is the location value generated by the MIL assembler for the label MESSAGES. (ADDRESS is a built-in MIL-language function which takes a label as an argument and returns a location relative to the beginning of the assembled microprogram.)

The code shown in Figure 7.5(a) suggests how a subroutine named LOCATE.TABLE.ADDRESS may be defined which computes the absolute G-store address of a named table. Figure 7.5(b) illustrates a call on this subroutine to compute the G-store address of MESSAGES to read the first 3 characters of this table to X.

The foregoing discussion, of course, assumes that the declared table resides in G-store, which can be guaranteed only if the declaration appears *after* the statement M.MEMORY.BOUNDARY MAXIMUM. A MIL program TABLE declaration is illustrated in Figure 7.6. (This piece of code appears in a Sequential Pascal interpreter developed by Mark

```

LOCATE.TABLE.ADDRESS                                %ROUTINE COMPUTES ABSOLUTE G-STORE
                                                    %ADDRESS OF A TABLE WHOSE
                                                    %MIL-ASSEMBLED RELATIVE ADDRESS IS
                                                    %IN L. THE RESULT IS PLACED IN FA.
                                                    %ROUTINE USES X AND Y.
                                                    %
MOVE 24 TO CP                                       %COMPUTE THE FIRST OFFSET AS FOLLOWS:
MOVE ADDRESS(+HERE) TO Y                           %RELATIVE ADDRESS OF .HERE IN Y
MOVE A TO X                                         %ACTUAL ADDRESS OF .HERE IN X
.HERE
MOVE DIFF TO X                                     %INTERPRETER DISPLACEMENT IN X
                                                    %NOW AUGMENT BY SECOND OFFSET WHICH IS
                                                    %THE ARGUMENT IN L.
MOVE L TO Y                                         %SUM OF TWO OFFSETS IN X
MOVE SUM TO X                                       %NOW AUGMENT BT MBR
MOVE MBR TO Y                                       %AND ASSIGN IT TO FA
MOVE SUM TO FA
EXIT
                                                    (a)
MOVE ADDRESS(MESSAGES) TO L
CALL LOCATE.TABLE.ADDRESS
READ 24 BITS TO X INC FA
                                                    (b)

```

Figure 7.5. (a) Subroutine for computing absolute G-store address of a table, given its label value. (b) Illustrative use of subroutine in (a) to read first 3 characters of the table labeled MESSAGES.

that `BASE.OF.INTERPRETER` (a scratchpad register) holds the current value for the interpreter displacement.

Another statement, `M.MEMORY.BOUNDARY MINIMUM`, may be inserted in a MIL program to indicate the minimum amount of the interpreter one would like loaded in H-store during execution. Normally one would insert such a statement immediately following the most frequently used portion of an interpreter. The statement is treated as *advice* to the system and not as a command. Hence there is no guarantee that during every execution of the interpreter the minimum amount of H-store requested by the programmer will actually be awarded. Award of H-store for this purpose is a function of the current work load on the system, the actual size of H-store, and the version of MCP being used.

7.4 TRANSFER OF MICROCODE FROM G-STORE TO H-STORE

Blocks of one or more microinstructions may be transferred from G-store to H-store by executing the `OVERLAY` microinstruction. This single microinstruction executes a hardware subroutine whose logic is given in Figure 7.7. The hardware subroutine has three parameters whose matching argument values are assumed to be present in the L, FL, and FA registers. L should contain the starting address of the overlay area in H-store. FL should contain the number of microinstructions to be copied, and FA should point to the starting address in G-store of the microinstructions to be copied.

As can be seen from the flowchart logic in Figure 7.7, at least one microinstruction will be overlaid as a result of executing this instruction. Each transit of the loop copies a 16-bit field from G-store *directly into* H-store at the address specified by the A-register, which serves as an auxiliary pointer into H-store during the overlay operation.

The value of A prior to the start of the `OVERLAY` instruction execution is safe-stored on the stack (box 1) and later restored (box 6). This action frees up A for use as an auxiliary register which is initialized to the value of the argument L (box 2) prior to entering the loop.

The elapsed time for each transit of the transfer loop is quite short (less than a microsecond on the B1726). Still, the cost in time for overlaying a large block of microcode is not insignificant, so the `OVERLAY` instruction is used sparingly. The `OVERLAY` instruction is used principally by GISMO to ensure the presence in H-store of an interpreter as decided by the MCP. That is, if, prior to transferring control to some interpreter, its H-store-resident portion as determined by MCP is not in place, then GISMO will perform the required `OVERLAY`.

To illustrate the use of `OVERLAY`, the following is a hypothetical

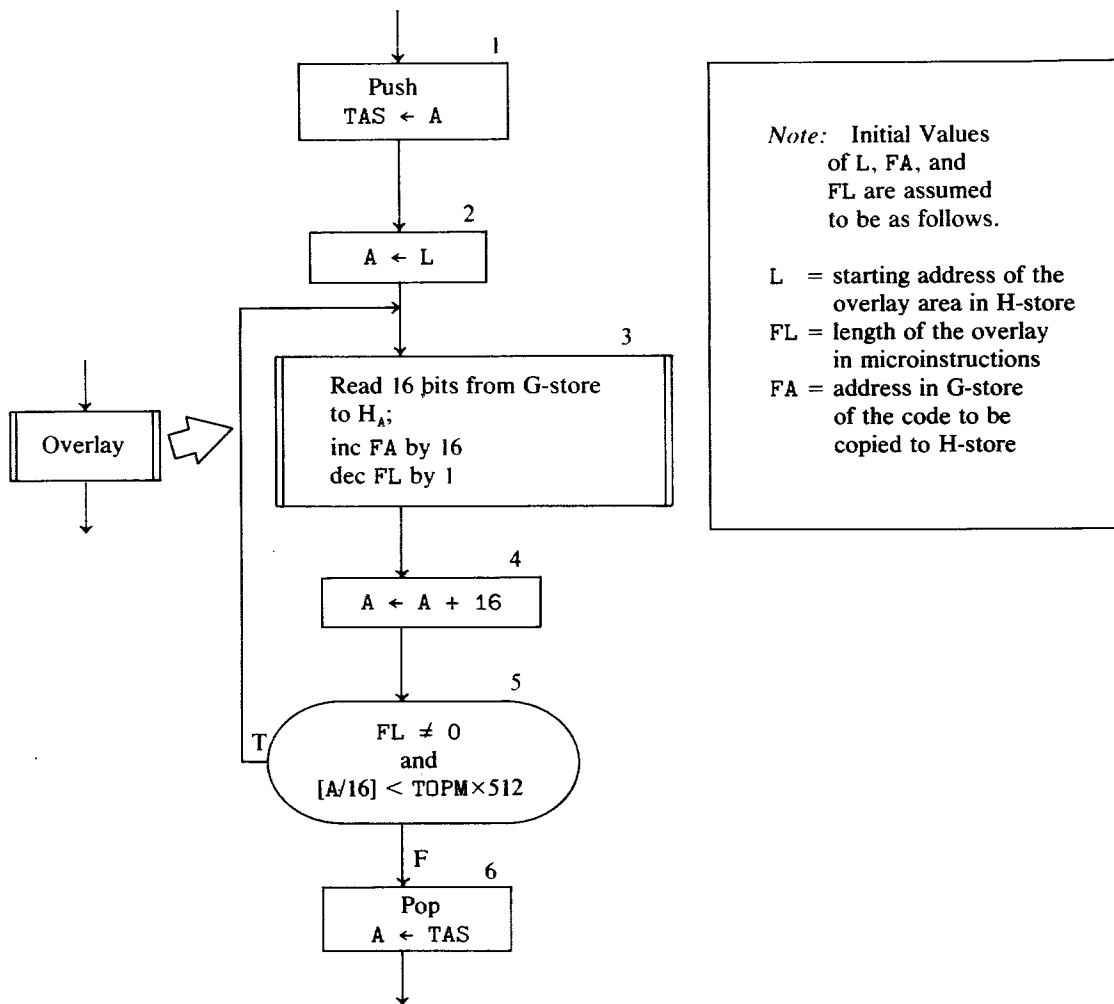


Figure 7.7.

```

:
OVERLAY.AREA
H.STORE.MULTIPLY          %AN ALIAS FOR OVERLAY.AREA
    ADJUST LOCATION TO LOCATION + 100    %INSERT 100 NO-OP
                                          %INSTRUCTIONS HERE
                                          %SEE APPENDIX A FOR FURTHER
                                          %EXPLANATION OF ADJUST

MAT.MULTIPLY
:
: } See text.
:
M.MEMORY.BOUNDARY MINIMUM
:
M.MEMORY.BOUNDARY MAXIMUM
:
MULTIPLY
: } 73 microinstructions
END
FINI
    
```

Figure 7.8.

application. Let there be a MIL subroutine called MAT.MULTIPLY. Upon entry to this routine we would like to transfer into H-store an otherwise infrequently used MULTIPLY routine which would appear in our MIL program *following* the M.MEMORY.BOUNDARY MAXIMUM statement. In this way we are assured that the MULTIPLY routine to be copied from G-store is indeed *in* G-store when the OVERLAY instruction is used. Assume that MULTIPLY is a routine known to comprise 73 microinstructions (by actual count). Further assume that we have coded near the beginning of our program an overlay area of 100 microinstructions, large enough to hold the MULTIPLY routine. The overlay area is labeled OVERLAY.AREA, and its initial contents is filled with no-op instructions. This program structure is illustrated in Figure 7.8.

The code at the entry point of MAT.MULTIPLY to transfer the MULTIPLY subroutine into H-store might be written as follows.

```

MAT.MULTIPLY
    MOVE ADDRESS(OVERLAY.AREA) TO L
    MOVE ADDRESS(+HERE) TO Y           %COMPUTE
                                       %INTERPRETER
    MOVE A TO X                       %DISPLACEMENT
.HERE
    MOVE DIFF TO X                    %INTERP.
                                       %DISPLACEMENT IN X

    MOVE MBR TO Y
    MOVE SUM TO X
    MOVE ADDRESS(MULTIPLY) TO Y       %ABSOLUTE G-STORE
                                       %ADDRESS
    MOVE SUM TO FA                    %NOW MOVED TO FA
    MOVE MULTIPLY.LENGTH TO FL        %MULTIPLY.LENGTH
                                       %DEFINED EQUAL
                                       %TO 73*16

OVERLAY
%NOW MULTIPLY IS IN H-STORE AND CAN BE CALLED
%WITHIN MAT.MULTIPLY BY A STATEMENT OF THE FORM:
    %CALL H.STORE.MULTIPLY
        %OR ALTERNATELY,
    %CALL OVERLAY.AREA

```

Note that the region labeled OVERLAY.AREA may have as many alias labels as the programmer may wish to give it. In our example, only one other, H.STORE.MULTIPLY, is shown in Figure 7.8. Another point to note is that the code illustrated above lacks adequate generality. It works fine, but only when the overlay area is certain to be in H-store

and only when the copied (“overlay”) version of MULTIPLY does not call or jump to any other routine. Any jumps within the copied version of MULTIPLY will be incorrect unless they are local jumps, i.e., jumps *within* MULTIPLY itself. A more advanced description of the MIL assembler would explain the special declarations needed to circumvent this problem.

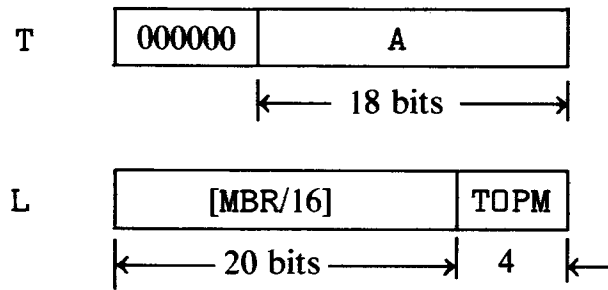
Misuse of the OVERLAY instruction can quickly destroy the integrity of the operating system, since sensitive code (GISMO and the MCP’s interpreter) usually occupy the lower half of H-store. For this reason use of OVERLAY, as suggested in the above example, is proscribed (forbidden as harmful) in the MCP environment. Such use of OVERLAY may be made only for programs executing as stand-alone code.

7.5 TRANSFERRING CONTROL TO ANOTHER INTERPRETER

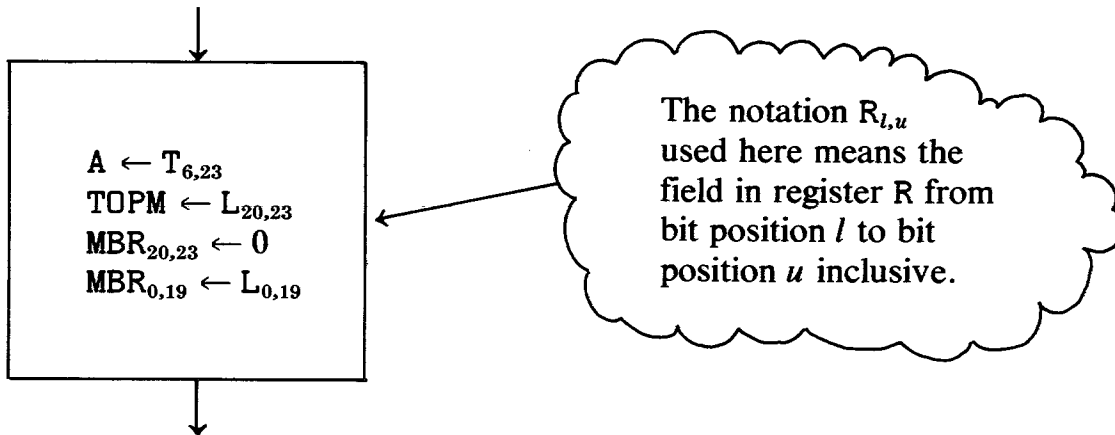
To transfer from one interpreter to another requires, in general, more than a simple jump or GO TO instruction. It is easy to see why this is so if we again consult Figures 7.1 and 7.2 for a case in point. Let us assume it is desired to transfer from some point in the SDL interpreter at say address 220 to the first instruction of the FORTRAN interpreter, located at address 1536 in H-store. Note that while executing in SDL, TOPM will in this case have the value 3, indicating that the effective top of H-store is currently one less than 3×512 , or 1535. A simple transfer by a GO TO will cause a jump to an instruction in the SDL interpreter of G-store *and not to location 1536 in H-store*. This result is simply a consequence of the logic shown in Figure 7.3.

A little thought should convince the reader that it is not possible to jump out of the SDL interpreter into another interpreter without changing values in TOPM and MBR simultaneously with the change in the A-register that results from a GO TO. In this particular case we need to change A from 220 to 1536, change TOPM from 3 to 4, and change MBR from its present value to one that refers to a point that is 2048×16 bits below the remainder of the FORTRAN interpreter held in G-store. These three changes must be accomplished in one indivisible operation, i.e., by one microinstruction, or else the job of transferring from one interpreter to another simply cannot be accomplished.

In fact, such a microinstruction is part of the B1726 repertoire, but because it is such a subtle and perhaps dangerous instruction for an “amateur” MIL programmer to use, we elected not to mention it until this point. Thus, the instruction takes the symbolic form TRANSFER.CONTROL, which is mapped to the hex string @0004@. A TRANSFER.CONTROL instruction expects its 3 argument values (for A, TOPM, and MBR) to be present in the T- and L-registers as follows.



Execution of TRANSFER . CONTROL then has the following semantics.



If interpreter I1 needs to transfer control to some other interpreter I2 (or to GISMO, which may in this context be regarded as another interpreter), then I1 must know how to restore the MBR and TOPM values of I2. Some system conventions must be established by which one interpreter knows where to find saved copies of the MBR, TOPM values of the other.

A user's interpreter normally only needs to transfer control to GISMO. The transfer to the MCP's interpreter (SDL) is always handled by GISMO. Accompanying a transfer to GISMO will be some message which indicates the purpose of the transfer. This message is a simple integer code left in the X-register. Thus, if the message indicates an intent to activate the MCP, then GISMO will send control to the appropriate point in the SDL interpreter with the limit register, LR, set properly for the MCP.

Transfer of control to GISMO is made easy by an established system convention that some fixed field in G-store will always hold the (MBR, TOPM) value pair for GISMO. Moreover there is the further convention established that the entry point into GISMO is always at $A = 0$. So a typical sequence for transfer of control to GISMO could be something like Figure 7.9.

Observe that such a sequence does not supply GISMO with any

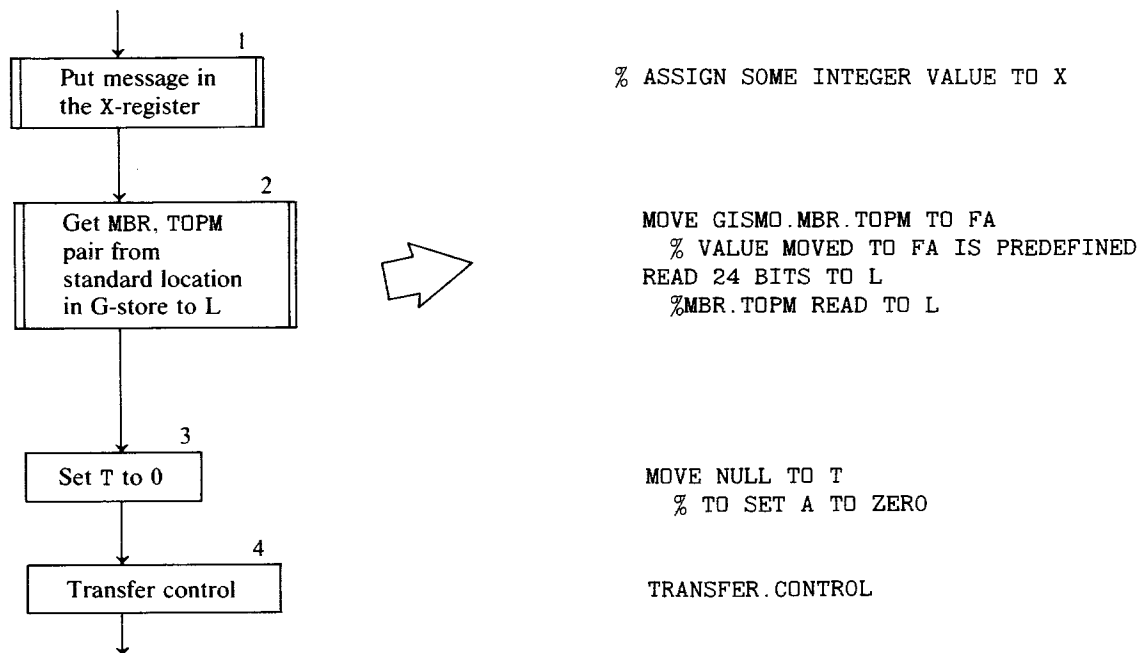


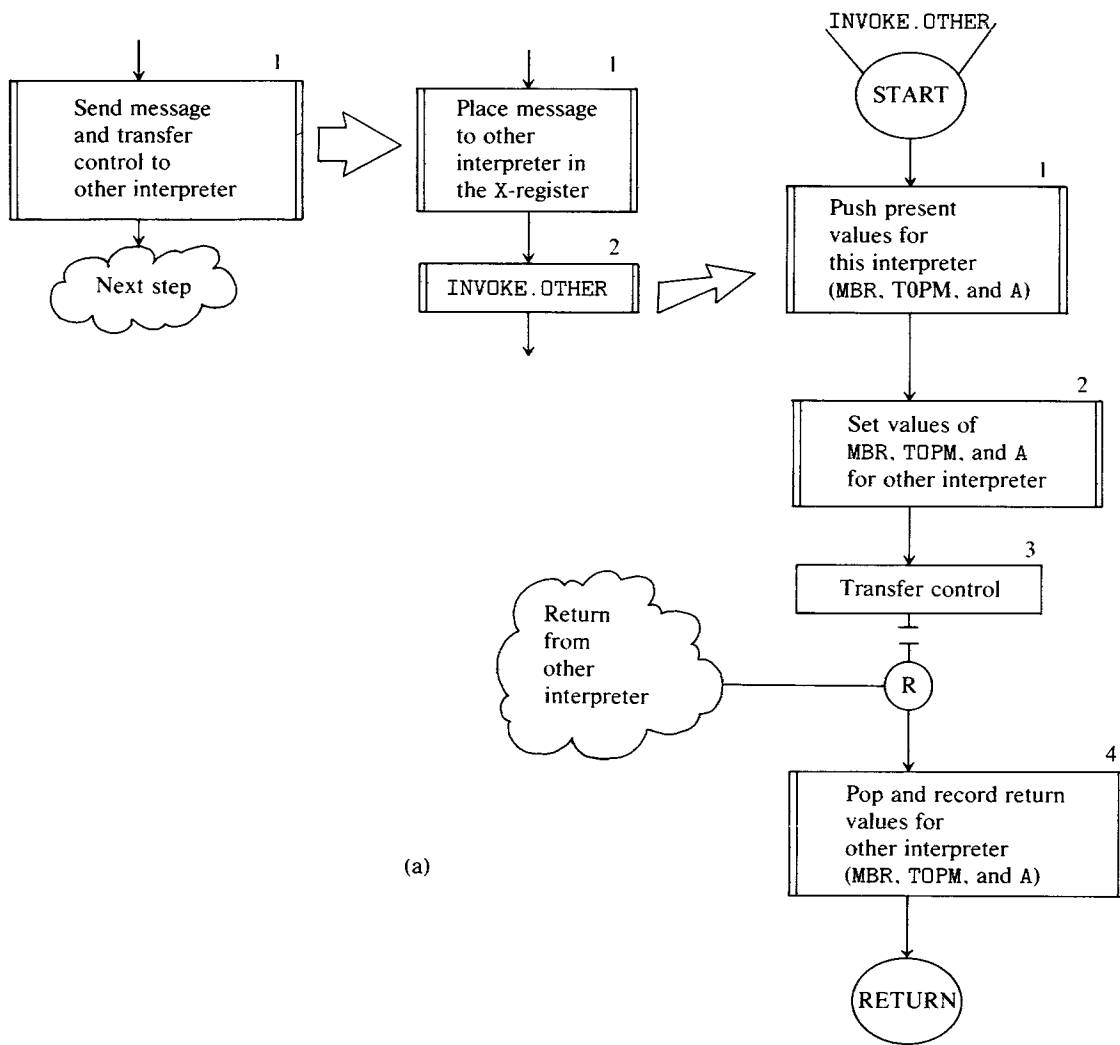
Figure 7.9.

explicit “return values” for A, MBR, and TOPM. In general, GISMO is called in the sense of a subroutine, and it is necessary to supply return values as arguments in the transfer of control.

Actually, it is not very difficult to supply return values for A, MBR, and TOPM. The coding scheme shown in Figure 7.10 illustrates how in principle one interpreter can transfer control to another interpreter to implement a “conversation” between two microprograms (i.e., back-and-forth corouting). This code is similar to the current conventions for invoking GISMO.

After placing an integer message in the X-register for the other interpreter (box 1), a general-purpose switching routine, INVOKE.OTHER, is called. This routine is almost straightforward.

- Box 1** composes and pushes as two stack words the present MBR, TOPM, and A register values for *this* interpreter. (The *other* interpreter will retrieve this information when it receives control).
- Box 2** is based on the assumption that *this* interpreter has previously recorded the values of MBR, TOPM, and A for the *other* interpreter (see Box 4 below). The action of box 2 then copies those recorded values into MBR, TOPM, and A.
- Box 3** The transfer of control is executed which will “pass the baton” to the *other* interpreter. When the other interpreter is thus activated, it can as its *next step* first examine X to decide



(a)

```

    MOVE MESSAGE.FOR.OTHER TO X
    CALL INVOKE.OTHER
    NEXT STEP
    %EXAMINE X TO DECIDE WHAT TO DO
    .....
    INVOKE.OTHER
    CALL +SAVE.MY.RETURN
    RETURN.POINT.R
    MOVE TAS TO OTHER.TOPM.MBR
    MOVE TAS TO OTHER.A.VAL
    EXIT
    .SAVE.MY.RETURN
    MOVE MBR TO L
    MOVE TOPM TO LF
    MOVE L TO TAS
    MOVE OTHER.MBR.TOPM TO L
    MOVE OTHER.A.VAL TO T
    TRANSFER.CONTROL
    %THIS IS THE EFFECTIVE
    %POINT OF RETURN
    %PUSH RETURN.POINT.R, I.E., THE VALUE OF A
    %
    %POP AND RECORD RETURN
    %INFO OF OTHER INTERPRETER.
    %"PACKAGE" MBR AND
    %TOPM INTO ONE 24-BIT
    %REGISTER AND PUSH IT.
    %SET L AND T FROM
    %PREVIOUSLY RECORDED VALUE
    
```

(b)

Figure 7.10. (a) Flowchart logic for orderly switch of control from one interpreter to another. (b) MIL code for flowcharts in (a).

why it was activated. Eventually the *other* interpreter will execute a call on INVOKE.OTHER, and when it does, and box 3 is again executed, control can return to *this* interpreter at the point marked with a circled R in the flowchart.

Box 4 The first action upon reactivation of *this* interpreter is taken here, which is to pop the top two words of the stack containing MBR, TOPM, and A values for the *other* interpreter and to record these values somewhere (e.g. scratchpads) for safekeeping.

A trivial extension to the switching code given in Figure 7.10 will allow efficient implementation of coroutines between two microcode modules. Complete symmetry is assumed, i.e., both modules would have essentially identical switching code.

Any interprocess or interinterpreter communication scheme, such as the one just discussed, requires a first and crucial step of initialization before the conversation mechanism can function properly. In the above illustration, if one module sends the first message, then it must by some special, explicit, or ad hoc means be *told* how to locate the other one, i.e., be told the other module's (MBR, TOPM) and A values. Ordinarily, some central data structure must be maintained which holds information about the states of active processes (or interpreters), and more often than not, some central agent (supervisory routine) is made responsible for the management of such a central data base. In Burroughs software the *location* of each interpreter is determined through an information structure managed by the MCP. Details of this information structure and the specific functions of the MCP and GISMO are not covered here.

7.6 SUMMARY

We have now reviewed all the hardware features of the B1726 related to execution of microprograms from two levels of store and to switching back and forth between independent microprograms. There is much more to know about the particular way the H-store is managed, and the particular implementation schemes for intercommunication between user-coded microprograms and the operating system. Such details however are strongly dependent on the system software architecture of the MCP and GISMO. We have regarded such details as a separate topic entirely, and for this reason have avoided bringing them to the reader's attention in this book. Other literature may be consulted for information on the MCP and GISMO.

Our justification in “suppressing” such information has been twofold:

1. To give beginning microprogrammers a chance to practice writing microcode as quickly as possible (minimum overhead).
2. To give the sophisticated computer professional a feeling for the architecture of the B1726 and its potential *independent* of the specific software products the Burroughs Corporation elected to implement on the B1726.

Appendix A

Abridged MIL reference guide¹

DIRECTORY

	Page
NOTATION	172
EXECUTABLE MIL STATEMENTS	173
ADD (scratchpad)	173
AND	173
BIAS	174
CALL	176
<hr/>	
CARRY	176
CASSETTE	— ²
CLEAR	177
COMPLEMENT	177
<hr/>	
COUNT	178
DEC	179
DISPATCH	—
EOR	181
<hr/>	
EXIT	181
EXTRACT	181
GO TO	183
HALT	—
<hr/>	
IF	183
INC	185
JUMP	186
LIT	—
<hr/>	
LOAD	187
LOAD .MSMA	—
LOAD .SMEM	—
MICRO (See Appendix B)	—

¹ See also "B1700 MICRO Implementation Language (MIL) Reference Manual", Burroughs Corporation, December 1973, Form 1072568.

² Dash denotes MIL statements not covered in these notes.

<i>Directory</i>	171
	Page
MONITOR	187
MOVE	187
NOP (See Appendix B)	—
NORMALIZE	190
<hr/>	
OR	191
OVERLAY	191
READ	192
READ MSML	193
<hr/>	
RESET	193
ROTATE OR SHIFT T	194
ROTATE OR SHIFT X, Y, AND XY	196
SET	196
<hr/>	
SKIP	197
STORE	199
SUBTRACT (scratchpad)	200
SWAP	200
<hr/>	
TRANSFER. CONTROL	202
WRITE	202
WRITE MSML	204
WRITE. STRING	—
<hr/>	
XCH	204
NONEXECUTABLE MIL STATEMENTS (DECLARATIONS)	205
ADJUST LOCATION	205
DEFINE	205
DEFINE. VALUE	—
DECLARE	206
<hr/>	
MACRO	208
SEGMENT	—
TABLE	209
SPECIAL MIL EXPRESSIONS	210
ADDRESS	210
DATA. LENGTH	210

1 NOTATION

Each description of a statement contains the following three subentries

Syntax of the MIL statement

Semantics

Page reference, if any, to portions of the text that illustrate (use) the statement, or discuss its syntax or semantics

Syntax notation

Terminals (or key words) are *capitalized*.

Nonterminals are given in *lower case* (without the angle brackets customarily used in BNF notation).

Optional words or phrases are enclosed in square brackets, [].

Choose one from a set of alternatives. The set of choices is enclosed in curly brackets, { }.

Example

$$\text{READ literal BITS [REVERSE] TO } \left\{ \begin{array}{c} \text{T} \\ \text{X} \\ \text{Y} \\ \text{L} \end{array} \right\} \left[\begin{array}{c} \{ \text{INC} \} \\ \{ \text{DEC} \} \end{array} \right] \left[\begin{array}{c} \{ \text{FA} \} \\ \{ \text{FL} \} \end{array} \right] \left[\text{AND } \left[\begin{array}{c} \{ \text{INC} \} \\ \{ \text{DEC} \} \end{array} \right] \left[\begin{array}{c} \{ \text{FL} \} \\ \{ \text{FA} \} \end{array} \right] \right]$$

Here the key word REVERSE is optional, as are the phrases


$$\left\{ \begin{array}{c} \text{INC} \\ \text{DEC} \end{array} \right\} \left\{ \begin{array}{c} \text{FA} \\ \text{FL} \end{array} \right\}$$

and

$$\text{AND } \left\{ \begin{array}{c} \text{INC} \\ \text{DEC} \end{array} \right\} \left\{ \begin{array}{c} \text{FL} \\ \text{FA} \end{array} \right\}.$$

Within the latter two optional phrases, choices may be made such as INC FL or AND DEC FA, but note that these choices must be consistent with one another. The syntax notation is not so precise as we might like. Thus, the semantics of READ would tell us that

READ 8 BITS TO X INC FA AND DEC FA


 Inconsistent choice of
optional phrases

incorporates a contradiction (or nonsense)

Semantics notation

Flowchart language is used wherever possible

Bit fields for registers or for G-store are notated as follows:

- (i) Single bits as simple subscripts or subscript expressions³, e.g.,

L_3 means $L(3)$

- (ii) Multibit fields as parenthesized subscript *spans*, e.g.,

$T_{(i, i+j)}$ means the subfield of T that spans from T_i to T_{i+j} (inclusive),

$G\text{-store}_{(FA, FA+k)}$ means the subfield of G-store that spans from address FA to $FA+k$ (inclusive).

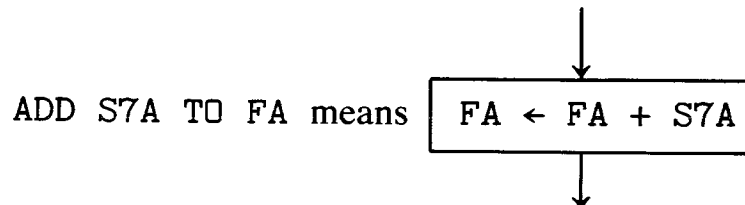
2 EXECUTABLE MIL STATEMENTS

ADD

Syntax

ADD $\left\{ \begin{array}{c} S0A \\ S1A \\ : \\ S14A \\ S15A \end{array} \right\}$ TO FA

Semantics



Page references: 35, 39, 54

AND

Syntax

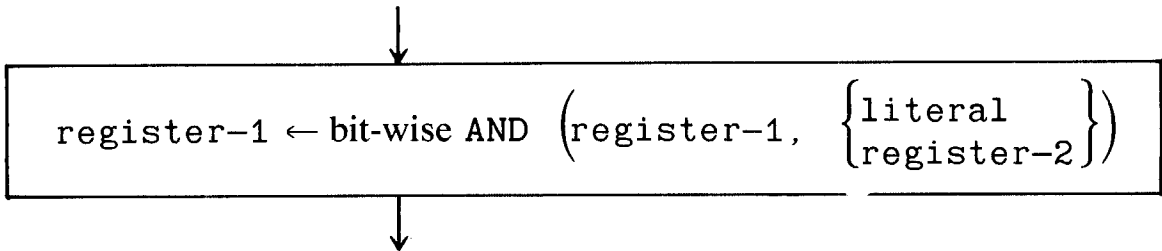
AND register-1 WITH $\left\{ \begin{array}{c} \text{literal} \\ \text{register-2} \end{array} \right\}$

where register-1 and
register-2 are 4-bit registers and
literal is any integer 0 thru 15.

³ In MIL notations, bit positions of all registers are indexed left to right starting at zero.

AND *continued*

Semantics



Page references: None

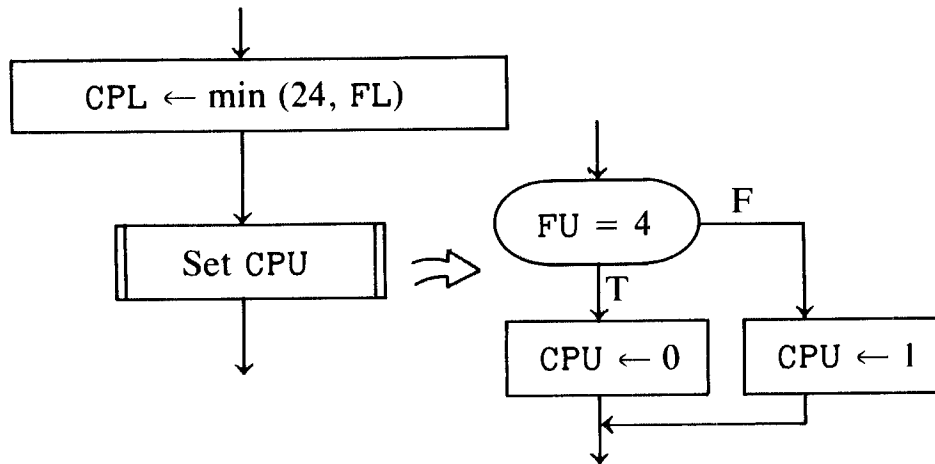
BIAS

Syntax

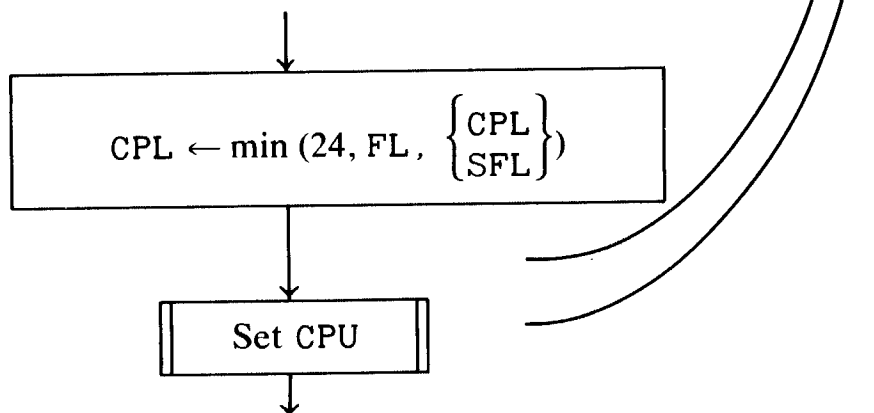
BIAS BY $\left\{ \begin{matrix} \text{UNIT} \\ \text{F} \end{matrix} \right\}$ $\left[\text{AND} \left\{ \begin{matrix} \text{CP} \\ \text{S} \end{matrix} \right\} \right]$ [TEST]

Semantics

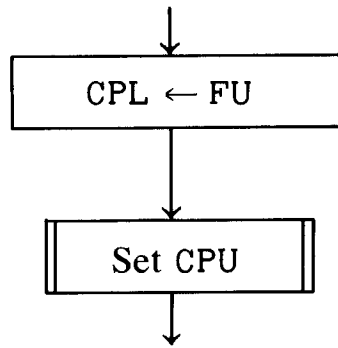
BIAS BY F means



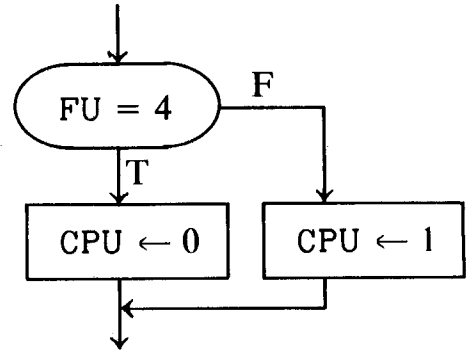
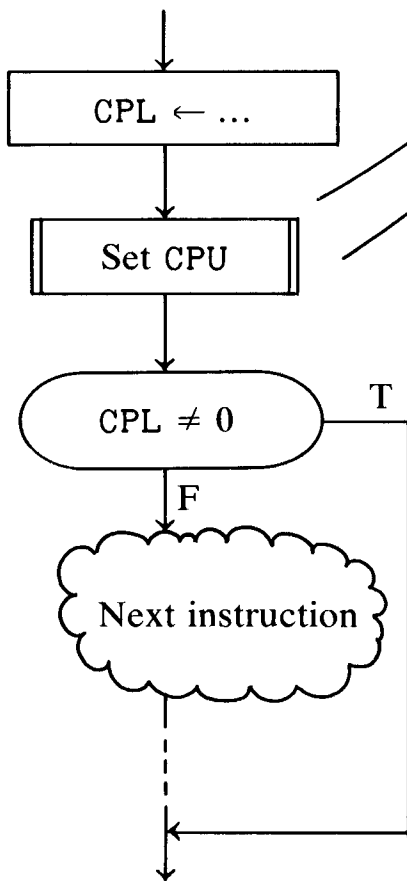
BIAS BY F AND $\left\{ \begin{matrix} \text{CP} \\ \text{S} \end{matrix} \right\}$ means



BIAS BY UNIT means



BIAS BY... TEST means

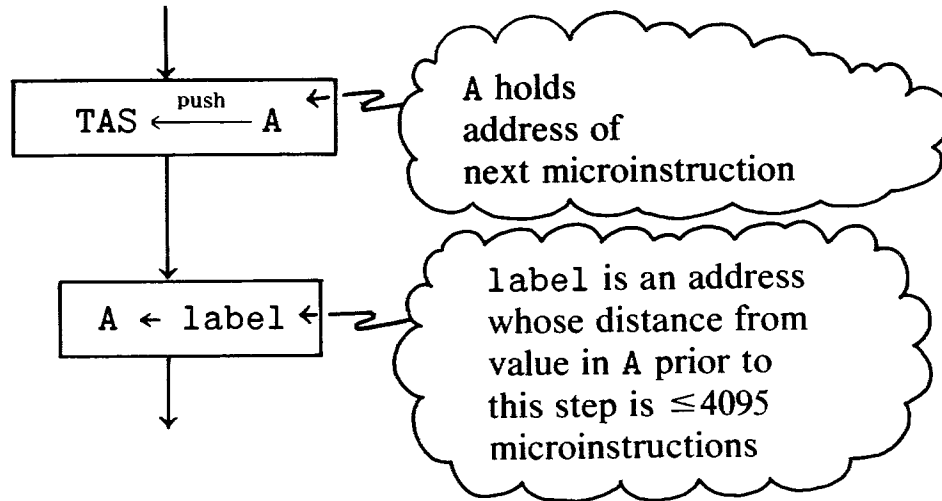


CALL

Syntax

CALL label

Semantics



Page references: 41, 95, 100

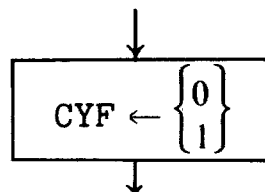
CARRY

Syntax

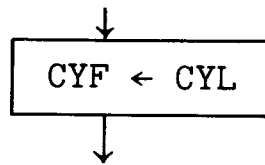
CARRY $\left\{ \begin{array}{l} 0 \\ 1 \\ \text{SUM} \\ \text{DIFFERENCE} \end{array} \right\}$

Semantics

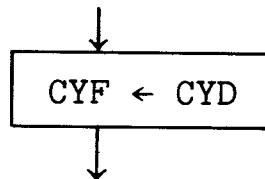
CARRY $\left\{ \begin{array}{l} 0 \\ 1 \end{array} \right\}$ means



CARRY SUM means



CARRY DIFFERENCE means



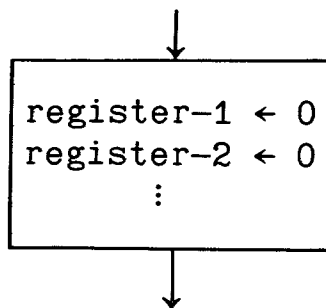
Page references: 106, 147

CLEAR

Syntax

CLEAR register-1 [register-2 [register-3
...[register-n]...]]

Semantics



same as MOVE 0 TO REGISTER-1
MOVE 0 TO REGISTER-2
⋮

Page references: 118, 120

COMPLEMENT

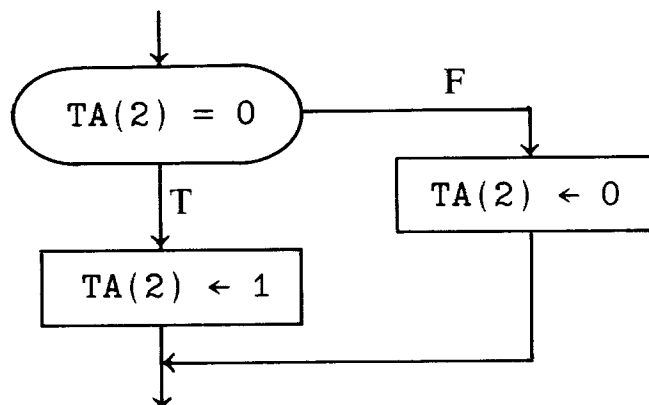
Syntax

COMPLEMENT register (literal-1)
[AND register (literal-2)
[AND register (literal-3)
[AND register (literal-4)]]]

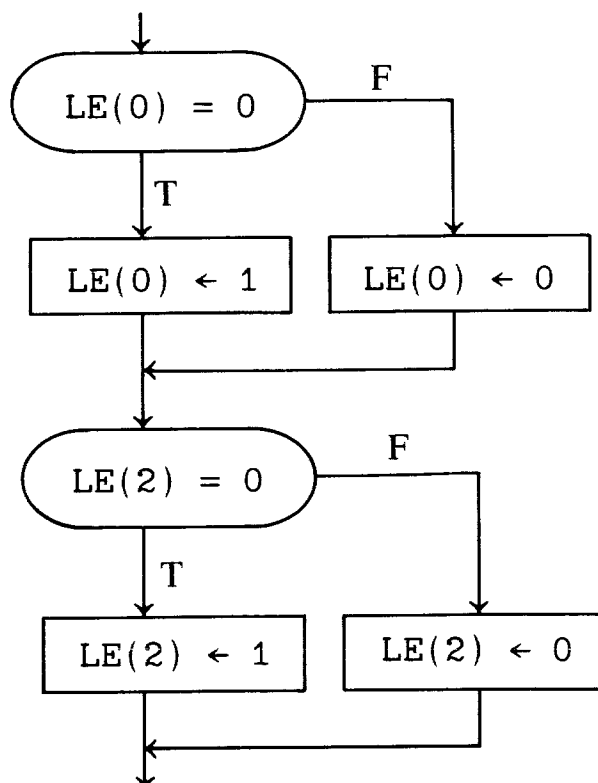
where register refers to a 4-bit register or a 4-bit subfield of FL, FB, L, or T

COMPLEMENT *continued***Semantics**

COMPLEMENT TA(2) means



COMPLEMENT LE(0) AND LE(2) means



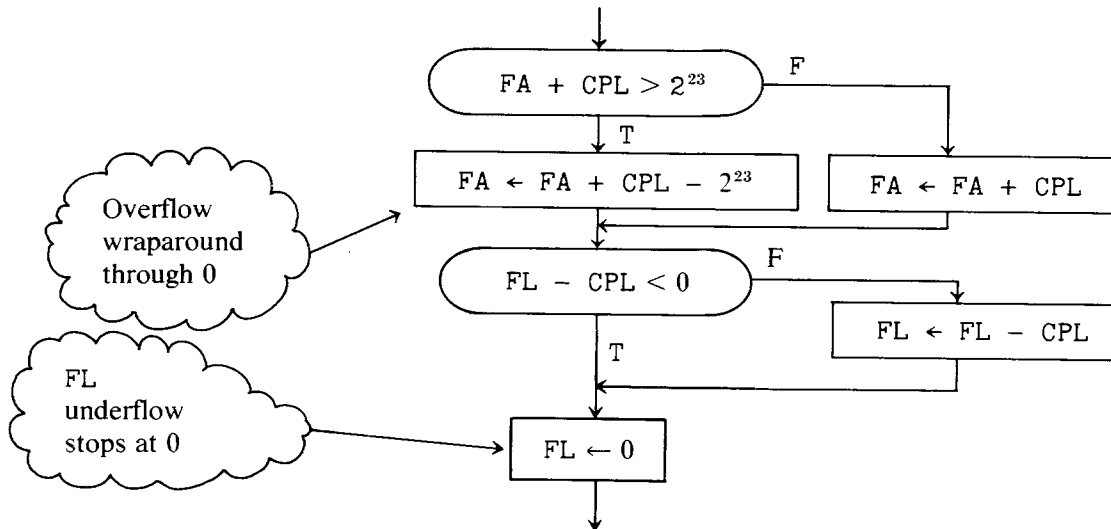
Page references: None

COUNT**Syntax**

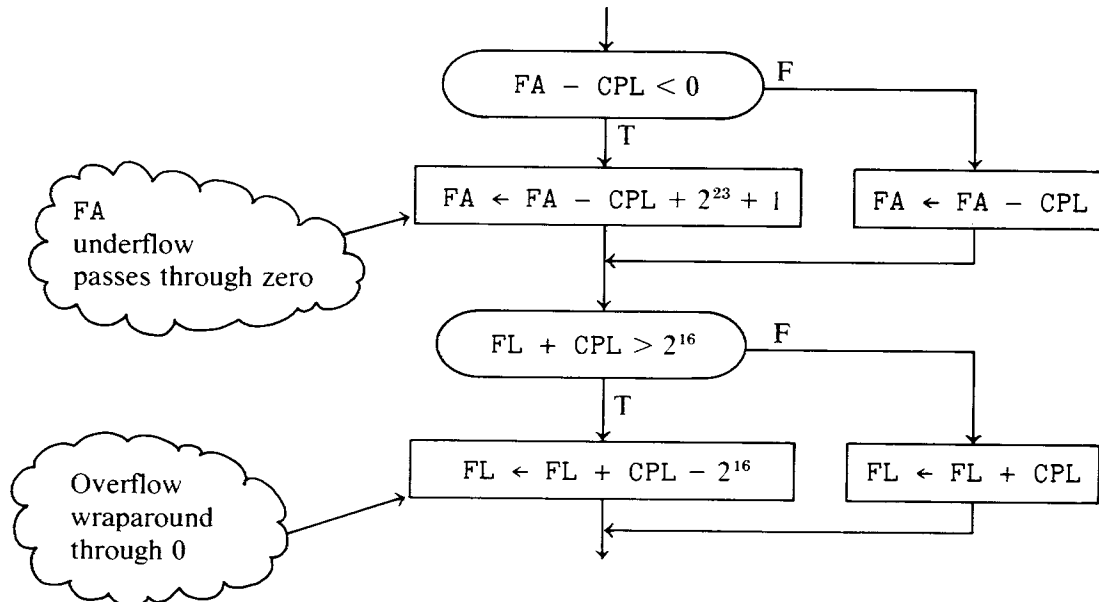
$$\text{COUNT } \left\{ \begin{array}{l} \text{FA} \\ \text{FL} \end{array} \right\} \left\{ \begin{array}{l} \text{UP} \\ \text{DOWN} \end{array} \right\} \left[\text{AND } \left\{ \begin{array}{l} \text{FL} \\ \text{FA} \end{array} \right\} \left\{ \begin{array}{l} \text{UP} \\ \text{DOWN} \end{array} \right\} \right] \left[\text{BY } \left\{ \begin{array}{l} \text{CPL} \\ \text{literal} \end{array} \right\} \right]$$
where *literal* is any integer in the range 0 to 24

Semantics

COUNT FA UP AND FL DOWN BY CPL means



COUNT FA DOWN AND FL UP BY CPL means



Page references: 95, 127

DEC

Syntax

DEC register-1 BY $\left\{ \begin{array}{l} \text{literal} \\ \text{register-2} \end{array} \right\}$ [TEST]

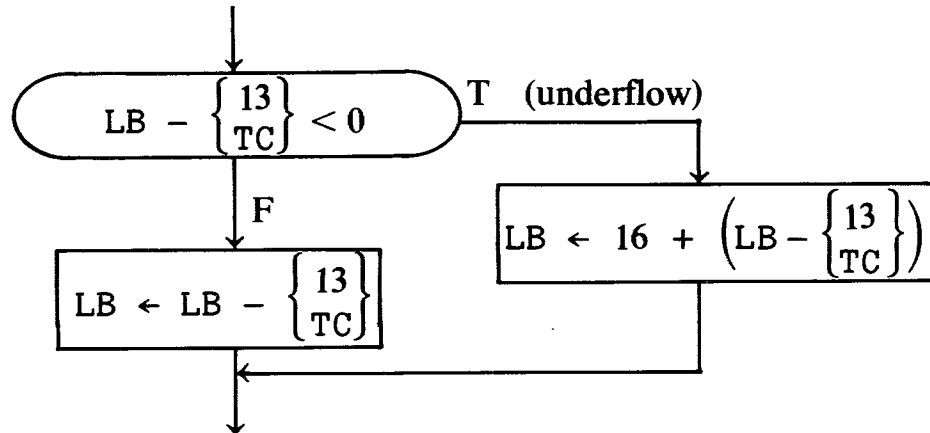
where

register-1 and register-2 are any 4-bit registers,
 literal is any integer 0 through 15

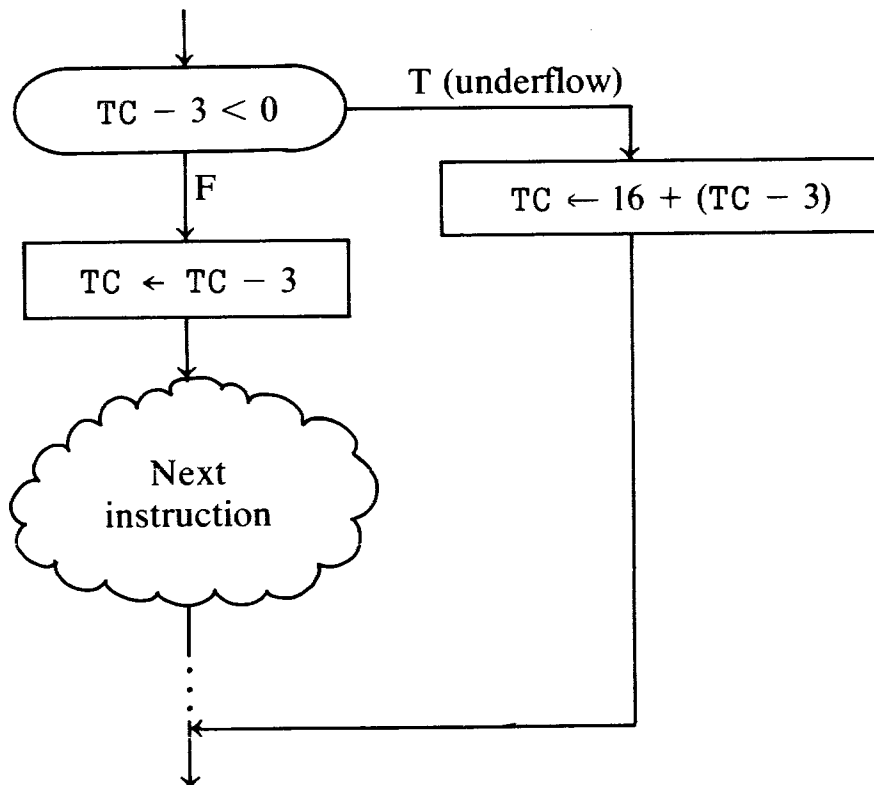
DEC *continued*

Semantics

DEC LB BY $\begin{Bmatrix} 13 \\ TC \end{Bmatrix}$ means



DEC TC BY 3 TEST means



Page references: None

EOR(exclusive OR)

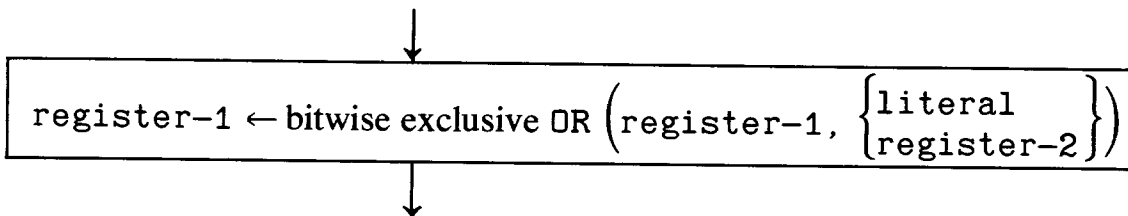
Syntax

EOR register-1 WITH $\left\{ \begin{array}{l} \text{literal} \\ \text{register-2} \end{array} \right\}$

where

register-1 and register-2 are any 4-bit registers,
literal is any integer 0 through 15

Semantics



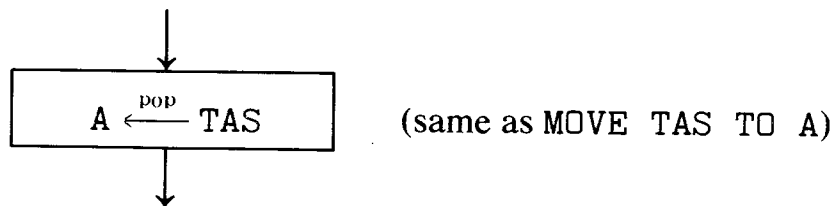
Page references: None

EXIT

Syntax

EXIT

Semantics



Page references: 41

EXTRACT

Syntax

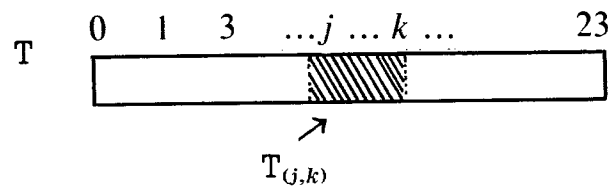
EXTRACT literal-1 BITS FROM T (literal-2) [TO register]

where literal-1 is any integer 0 through 24,
literal-2 is any integer 0 through 23,
register is T, L, X, or Y

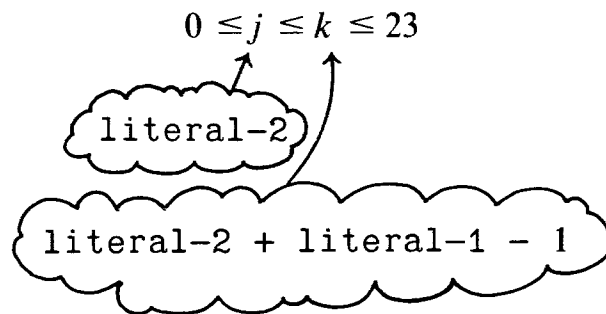
EXTRACT *continued*

Semantics

Given

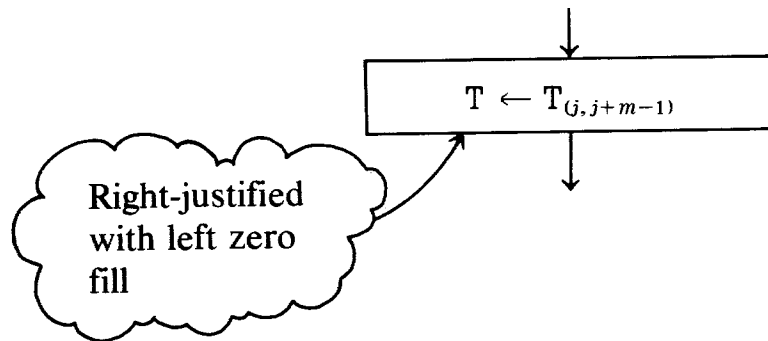


where

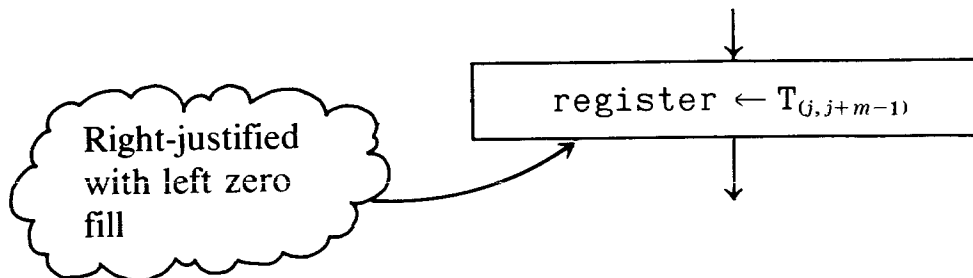


Then

EXTRACT M BITS FROM T(j) means



EXTRACT m BITS FROM T(j) TO register means



Page references: 29

GO TO

Syntax

$$\text{GO TO } \left\{ \begin{array}{l} \text{global-label} \\ +\text{point-label} \\ -\text{point-label} \end{array} \right\}$$

Semantics

global-label is any label whose run-time address has a displacement of less than 4096 microinstructions from the address of the GO TO. + point-label refers to the first *forward* instance of .point-label - point-label refers to the first *backward* instance of .point-label

Page reference: 39

IF

Syntax

$$\text{IF } \left\{ \begin{array}{l} \text{relation} \\ \text{bit-expression} \end{array} \right\} \left[\left[\begin{array}{l} \text{TRUE} \\ \text{FALSE} \end{array} \right] \right] \text{ THEN simple-MIL-statement}$$

$$\text{IF } \left\{ \begin{array}{l} \text{relation} \\ \text{bit-expression} \end{array} \right\} \left[\left[\begin{array}{l} \text{TRUE} \\ \text{FALSE} \end{array} \right] \right] \text{ THEN}$$

```
BEGIN
  :
END [ELSE
BEGIN
  :
END]
```

relation is any relational expression or bit designation listed under Condition Syntax in Section 2 of Appendix B.

bit-expression designates one or more bits of the *same* 4-bit register. Up to 2 bits may be designated on or off using AND, OR as logical operators, e.g.,

```
TC(1)
TB(0) AND TB(2)
LA(0) OR LA(3)
CA(1) FALSE AND CA(3) FALSE
CB(1) FALSE OR CB(2) FALSE
```

IF *continued*

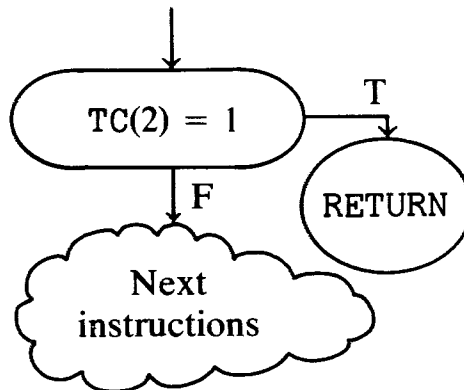
To test more than 2 bits, the 4-bit register must be equated to a literal, e.g.,

```
TC = 12
LA = @(1)0101@
```

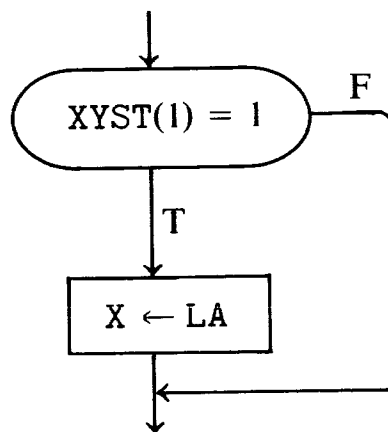
simple-MIL-statement is any executable MIL statement other than another IF statement.

Semantics

IF TC(2) THEN EXIT means

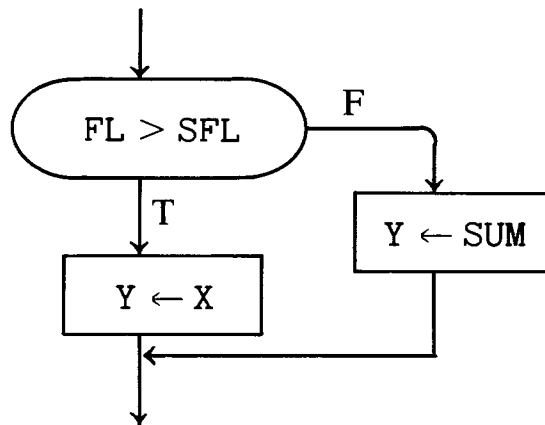


IF ANY.INTERRUPT THEN MOVE LA TO X means



```
IF FL > SFL THEN
  BEGIN
    MOVE X TO Y
  END ELSE
  BEGIN
    MOVE SUM TO Y
  END
```

means



Page references: 30, 31

INC

Syntax

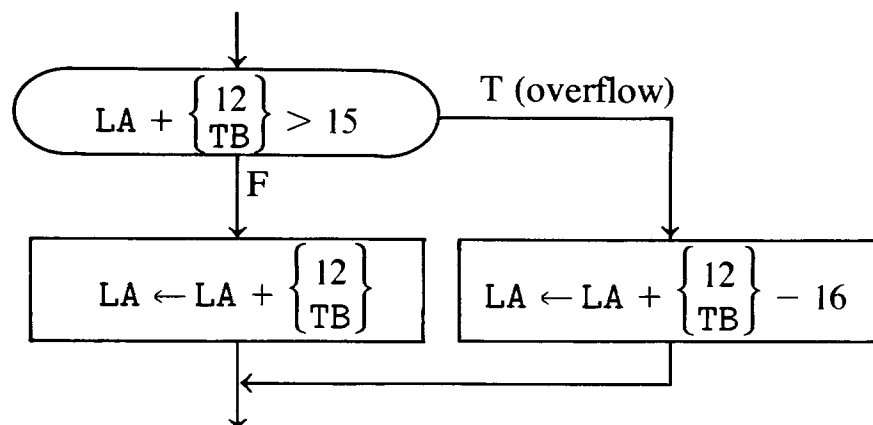
INC register-1 BY $\left\{ \begin{matrix} \text{literal} \\ \text{register-2} \end{matrix} \right\}$ [TEST]

where

register-1 and register-2 are 4-bit registers,
literal is any integer 0 through 15.

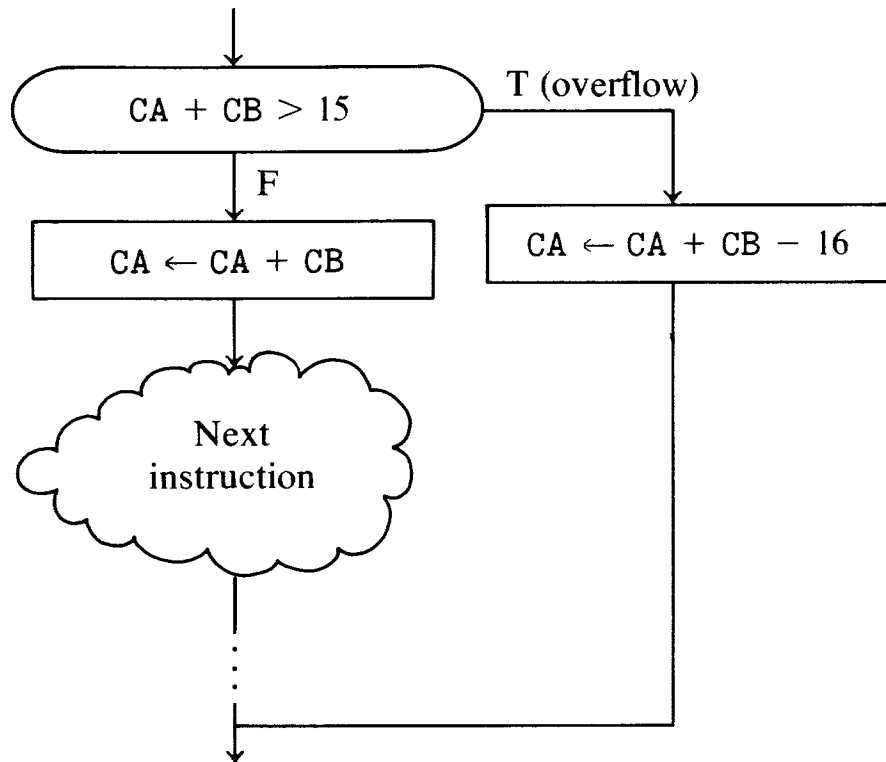
Semantics

INC LA BY $\left\{ \begin{matrix} 12 \\ \text{TB} \end{matrix} \right\}$ means



INC *continued*

INC CA BY CB TEST means



Page references: 129

JUMP**Syntax**

$$\text{JUMP } \left\{ \begin{array}{l} \text{FORWARD} \\ \text{TO label} \end{array} \right\}$$
Semantics

JUMP TO label means the same as GO TO label

JUMP FORWARD means JUMP to here + 0

This instruction is usually preceded by an instruction that ORs a displacement value into the M-register, e.g.,

```

MOVE L TO M
JUMP FORWARD
  
```

Page references: 39

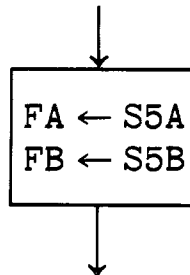
LOAD F

Syntax

LOAD F FROM $\left. \begin{array}{c} S0 \\ S1 \\ : \\ S14 \\ S15 \end{array} \right\}$

Semantics

LOAD F FROM S5 means



Page references: None

MONITOR

Syntax

MONITOR 8-bit literal

Semantics

The 8-bit literal is sent out as a set of 8 signals on a set of 8 lines. External connections may be made to the ends of these lines for sensing the value of the literal. A literal may be sensed whenever the MONITOR instruction is executed. Such literals may be recorded and/or displayed for purposes of performance measurement and evaluation.

Page references: None

MOVE

Syntax

MOVE source TO destination

MOVE *continued*

where

source is either a literal, a 24-bit scratchpad, or a register that can serve as a source,

destination is a 24-bit scratchpad, or a register that can serve as a destination,

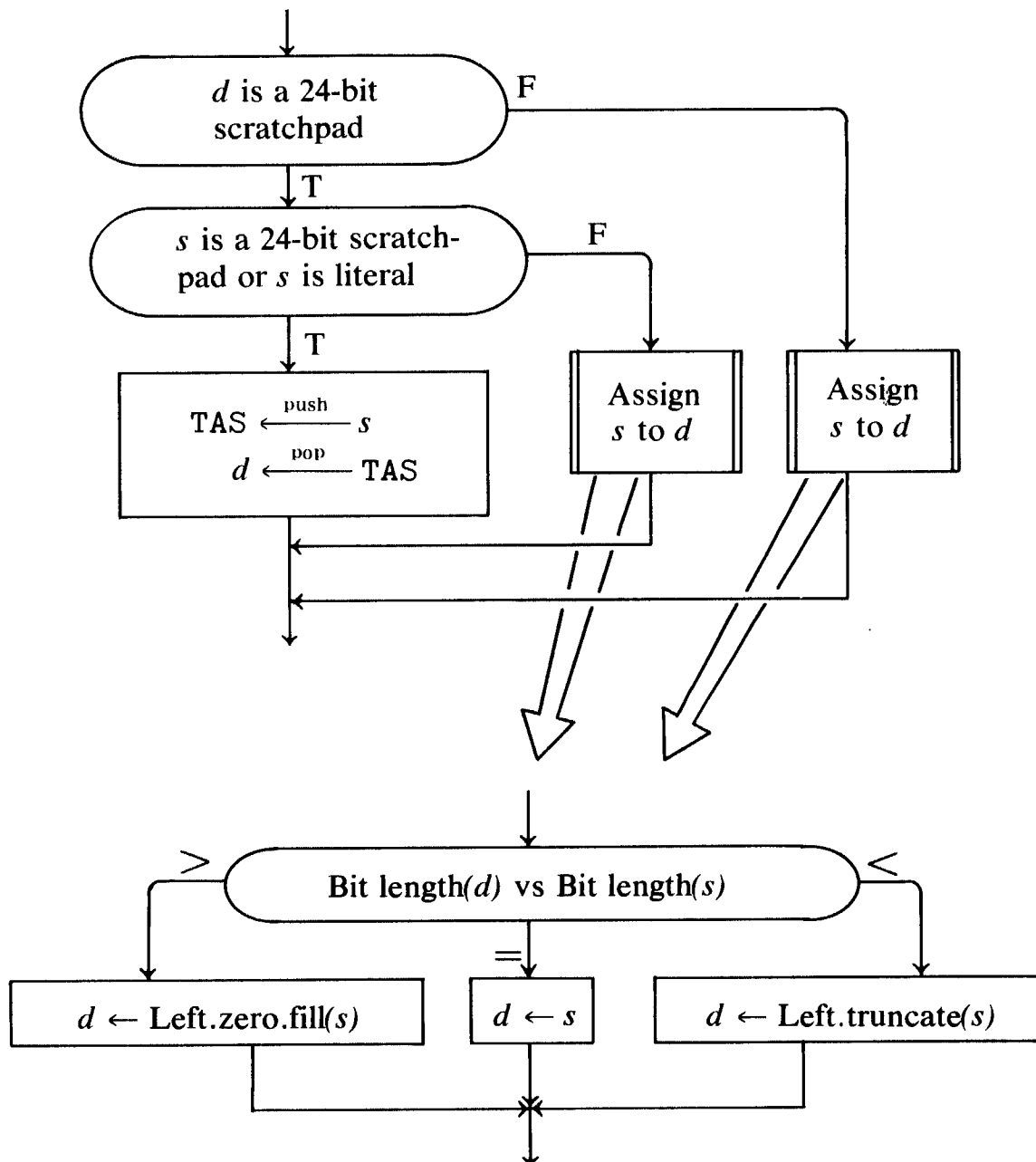
literal is any integer 0 through 2^{24} expressed either as a decimal number (0 to 16777215),

a binary number (@(1)0@ to @(1)1111...111@)

a hexadecimal number (@0@ to @FFFFFF@)

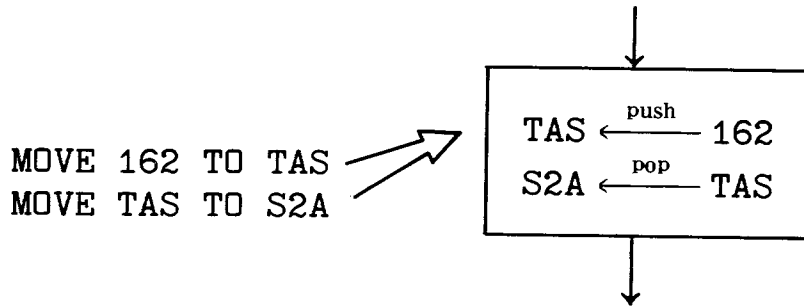
Semantics

MOVE *s* TO *d* means

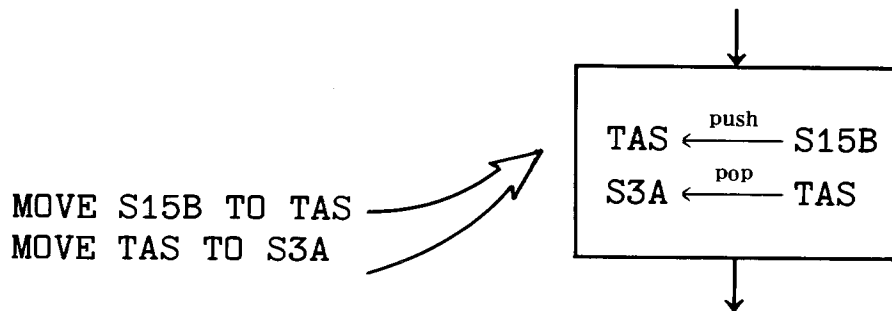


Examples

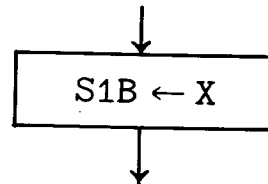
MOVE 162 TO S2A means



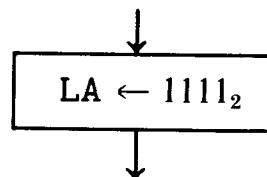
MOVE S15B TO S3A means



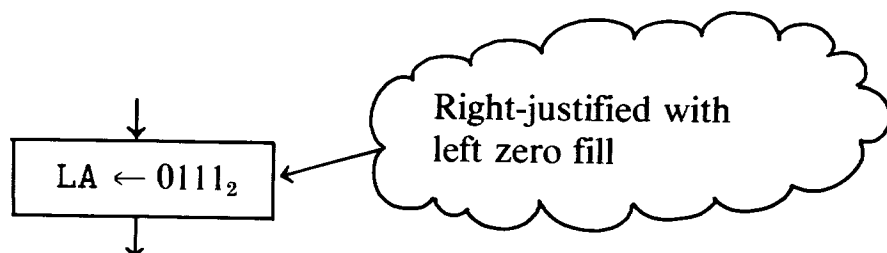
MOVE X TO S1B means



MOVE @(1)1111@ TO LA means

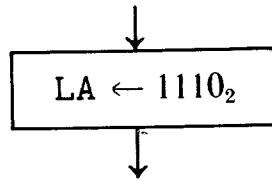


MOVE @(1)111@ TO LA means



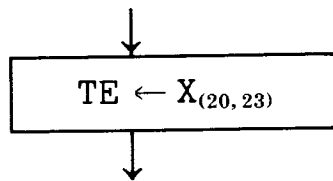
MOVE *continued*

MOVE @(1)11110@ TO LA means



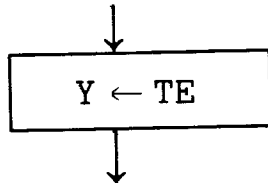
Right-justified with truncation on the left

MOVE X TO TE means



Low-order 4 bits of X copied into TE

MOVE TE TO Y means



Left zero fill of Y

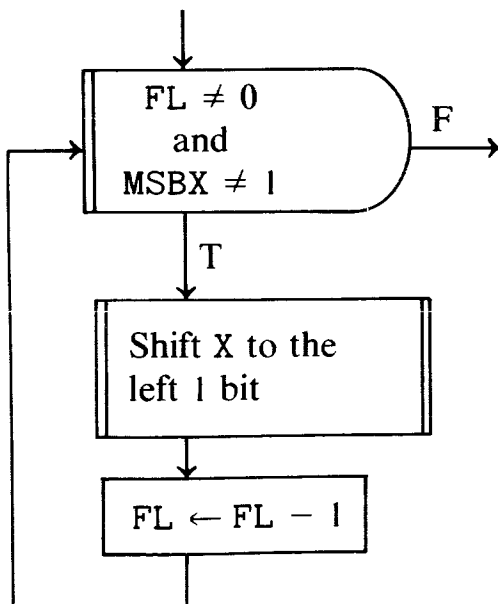
Page references: 18, 28, 35

NORMALIZE

Syntax

NORMALIZE

Semantics



i.e., shift X to the left until either FL = 0 or the most significant bit of X (MSBX), as conditioned by CPL, is 1.

Page references: None

OR

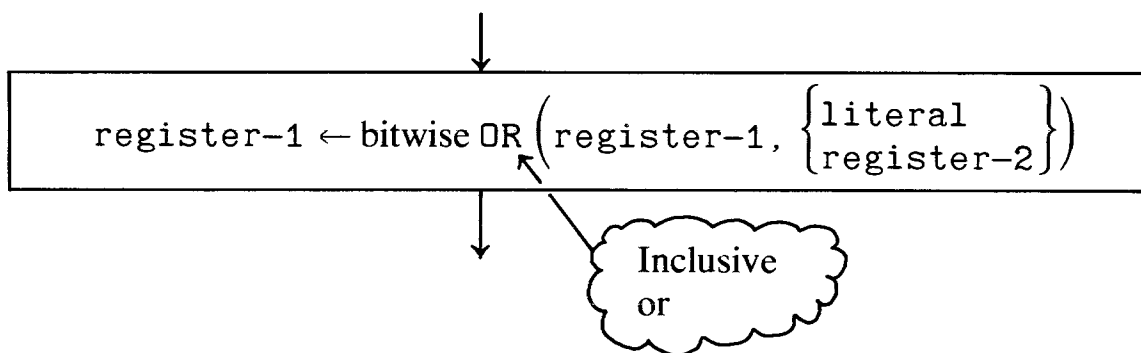
Syntax

OR register-1 WITH $\left\{ \begin{array}{l} \text{literal} \\ \text{register-2} \end{array} \right\}$

where

register-1 and register-2 are any 4-bit registers,
and literal is any integer 0 through 15.

Semantics



Page references: None

OVERLAY

Syntax

OVERLAY

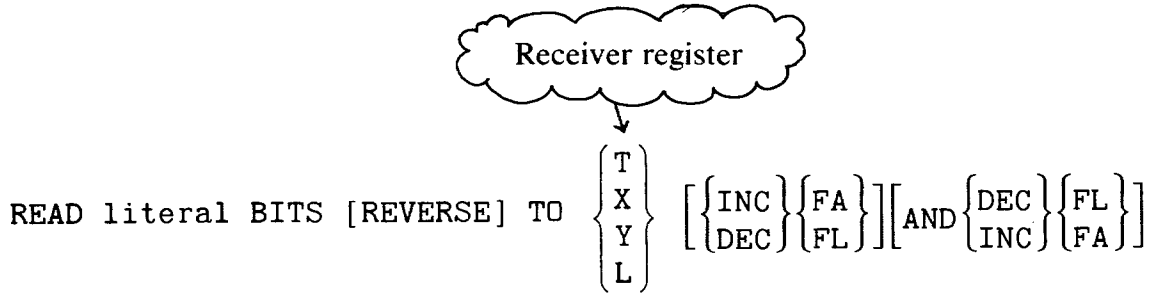
Semantics

After setting L to point at the overlay region in H-store,
FA to point to the region in G-store, and
FL to a count of the number of microinstructions to be copied,
OVERLAY causes FL microinstructions to be successively copied from G-
store beginning at FA to H-store beginning at L. The copying is
prematurely halted if the address in H-store of the next microinstruc-
tion overlay would exceed TOPM×512.

Page references: 161, 164

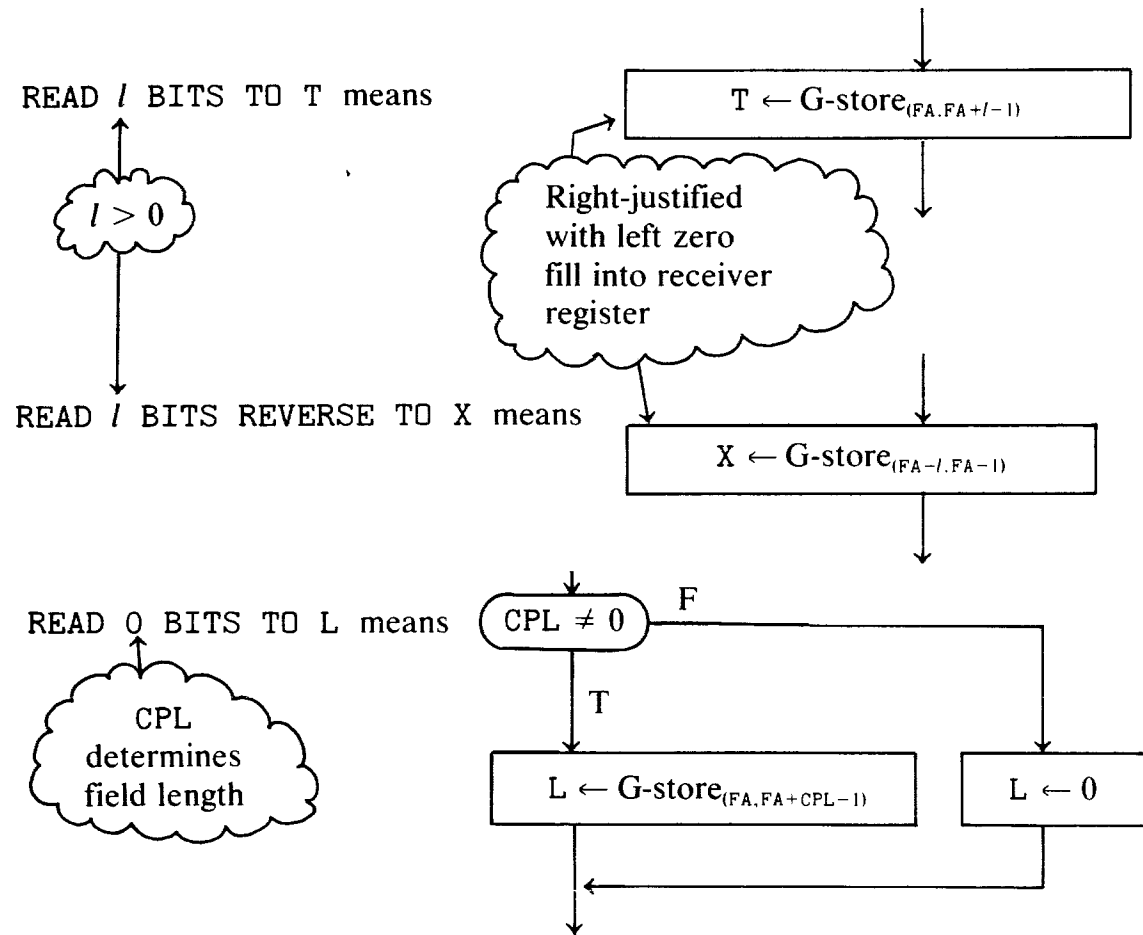
READ

Syntax

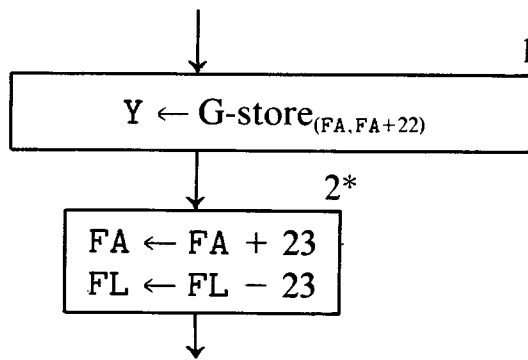


where literal is any integer 0 through 24

Semantics



READ 23 BITS TO Y INC FA AND DEC FL means



* Execution of box 2 overlaps execution of box 1.

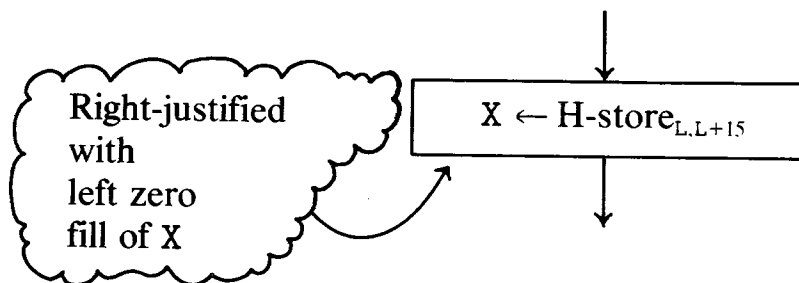
Page references: 14, 18, 25

READ MSML

Syntax

READ MSML TO X

Semantics



Page references: None

RESET

Syntax

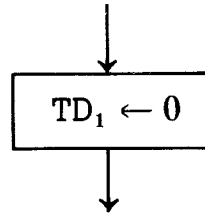
RESET register(literal-1)[AND register(literal-2)
[AND...register(literal-4)]]

where

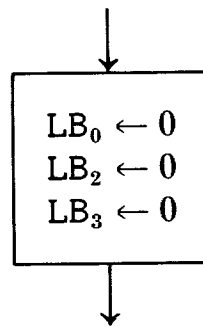
register is any 4-bit register or 4-bit subregister of FL, L, or T, or bit of FB, L, or T, that can serve as a destination,
literal-*i* is any integer, 0 through 3 for a 4-bit register, 0 through 15 for a subregister of FL, or 0 through 23 for a subregister of L or T.

RESET *continued***Semantics**

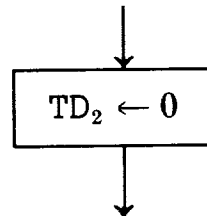
RESET TD(1) means



RESET LB(0) AND LB(2) AND LB(3) means



RESET T(14) means



Page references: 113, 120, 124

ROTATE OR SHIFT T**Syntax**

$\left\{ \begin{array}{l} \text{SHIFT} \\ \text{ROTATE} \end{array} \right\} T \text{ LEFT BY } \left\{ \begin{array}{l} \text{literal BITS} \\ \text{CPL} \end{array} \right\} [\text{TO register}]$

ROTATE T RIGHT BY literal BITS [TO register]

SHIFT T RIGHT BY literal BITS [TO $\left\{ \begin{array}{l} X \\ Y \\ T \\ L \end{array} \right\}]$

where

literal is any integer 0 through 23,

register is any register that can serve as a destination, including T itself.

ROTATE OR SHIFT X, Y, or XY

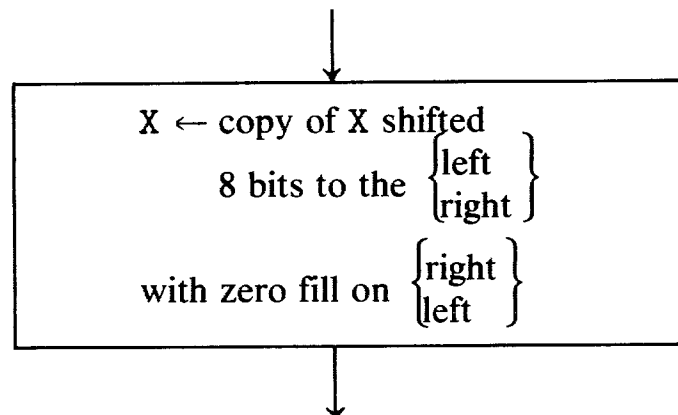
Syntax

$$\left\{ \begin{array}{l} \text{SHIFT} \\ \text{ROTATE} \end{array} \right\} \left\{ \begin{array}{l} X \\ Y \\ XY \end{array} \right\} \left\{ \begin{array}{l} \text{LEFT} \\ \text{RIGHT} \end{array} \right\} \text{ BY literal BITS}$$

where XY means X concatenate Y and
 literal is any integer 0 through 23, or, when XY is used, any
 integer 0 through 47.

Semantics

SHIFT X $\left\{ \begin{array}{l} \text{LEFT} \\ \text{RIGHT} \end{array} \right\}$ BY 8 BITS means



Page references: None

SET

Syntax

SET reg TO literal

or

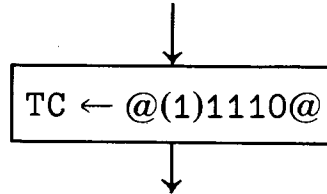
SET reg(literal-1)[AND REG(literal-2)[AND...
 [AND reg(literal-4)]]]]

where

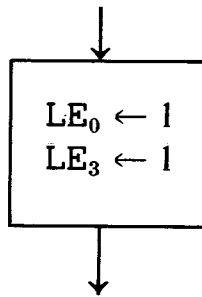
reg is any 4-bit register or any 4 bit subregister of FL, L, or T that can
 serve as a destination,
 literal-i is any integer, 0 through 3 for a 4-bit register, 0 through 15
 for a subregister of FL, or 0 through 23 for a subregister of L or T.

Semantics

SET TC TO 14 means



SET LE(0) and LE(3) means



Page references: 113, 120, 124, 129

SKIP

Syntax

SKIP WHEN register $\left\{ \begin{array}{l} \text{ALL} \text{ [CLEAR]} \\ \text{ANY} \\ \text{EQL} \end{array} \right\}$ mask [FALSE]

where register is any 4-bit register and mask is any integer 0 through 15 (represented as decimal, binary, or hexadecimal).

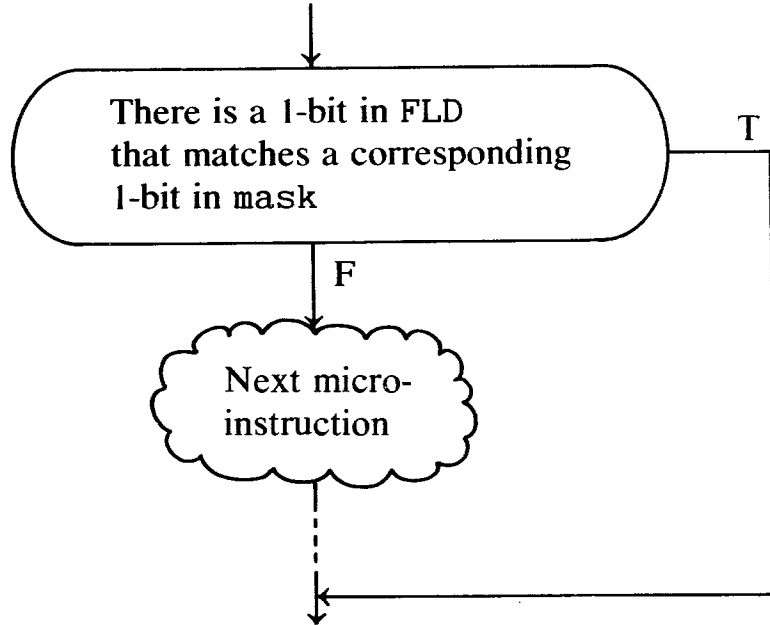
SKIP WHEN condition [FALSE]

where condition is any condition available from the condition register, BICN, XYCN, XYST, FLCN, or INCN. (See Condition Syntax in Section 2 of Appendix B.)

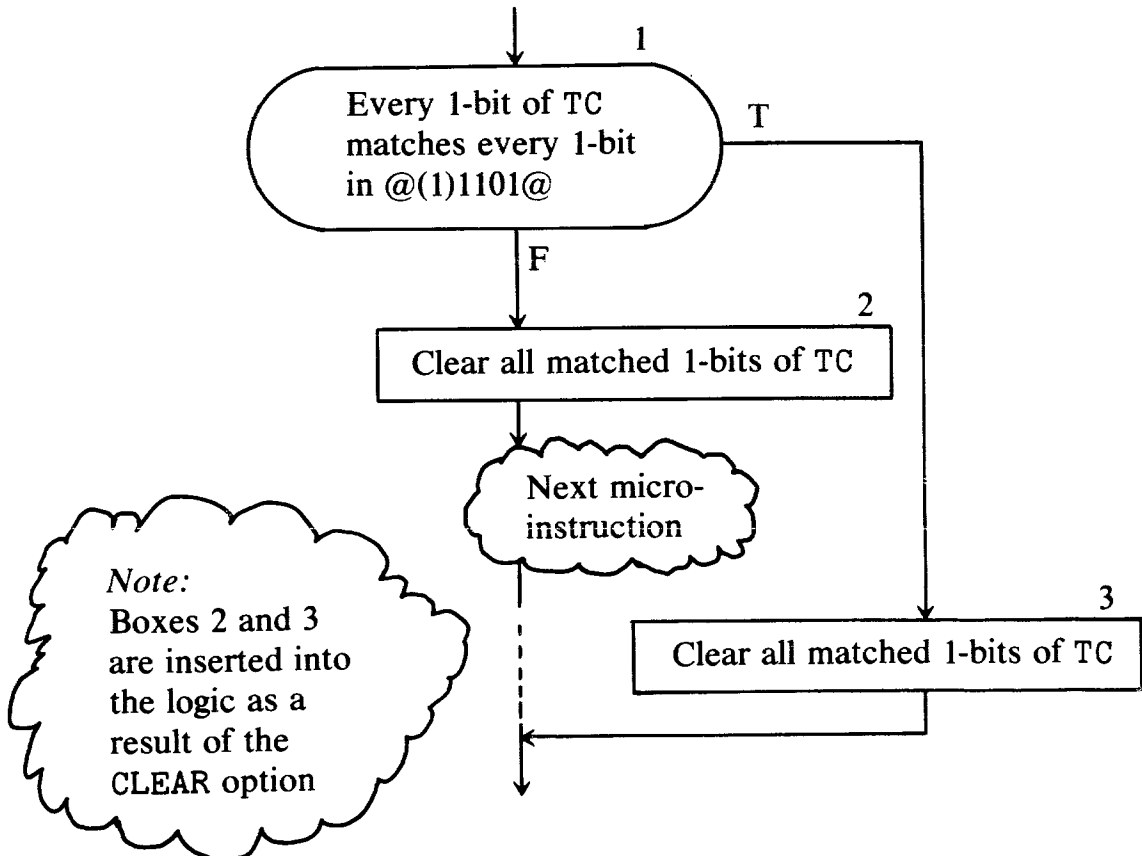
SKIP *continued*

Semantics

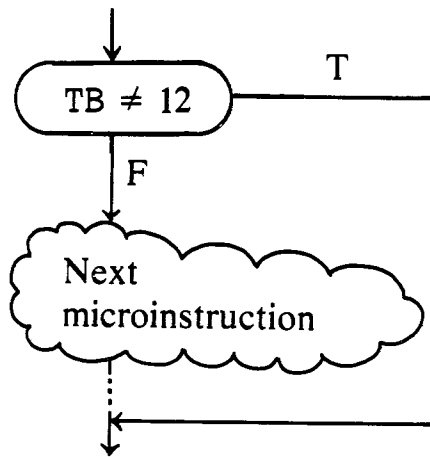
SKIP WHEN FLD ANY mask means



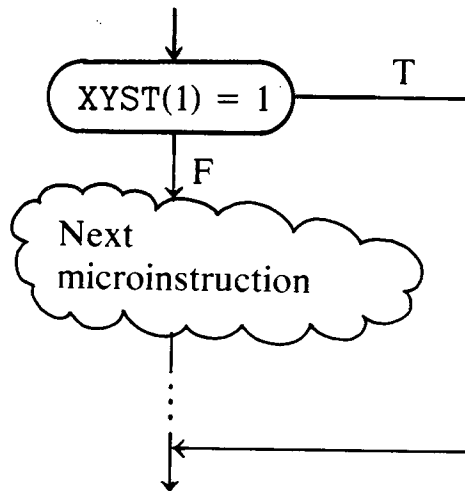
SKIP WHEN TC ALL CLEAR @(1)1101@ means



SKIP WHEN TB EQL 12 FALSE means



SKIP WHEN ANY.INTERRUPT means



Page references: 39

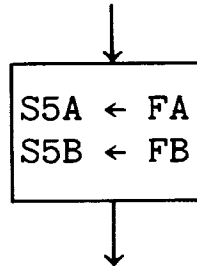
STORE F

Syntax

STORE F INTO $\left\{ \begin{array}{l} S0 \\ S1 \\ \vdots \\ S14 \\ S15 \end{array} \right\}$

STORE F *continued***Semantics**

STORE F INTO S5 means

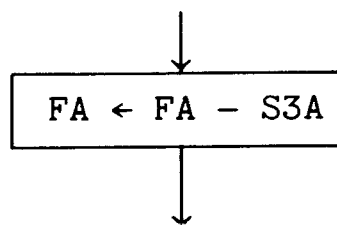


Page references: None

SUBTRACT**Syntax**

$$\text{SUBTRACT } \left\{ \begin{array}{c} \text{S0A} \\ \text{S1A} \\ \vdots \\ \text{S14A} \\ \text{S15A} \end{array} \right\} \text{ FROM FA}$$
Semantics

SUBTRACT S3A FROM FA means



Page references: 35

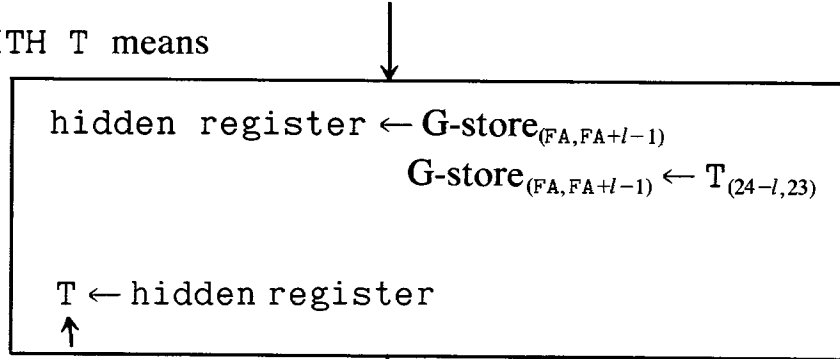
SWAP**Syntax**

$$\text{SWAP literal BITS [REVERSE] WITH } \left\{ \begin{array}{c} \text{T} \\ \text{X} \\ \text{Y} \\ \text{L} \end{array} \right\}$$

where literal is any integer, 0 through 24.

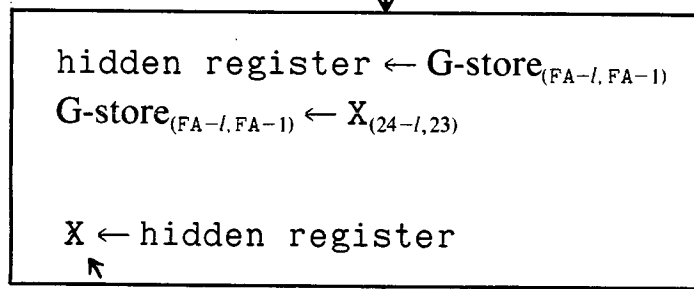
Semantics

SWAP / BITS WITH T means



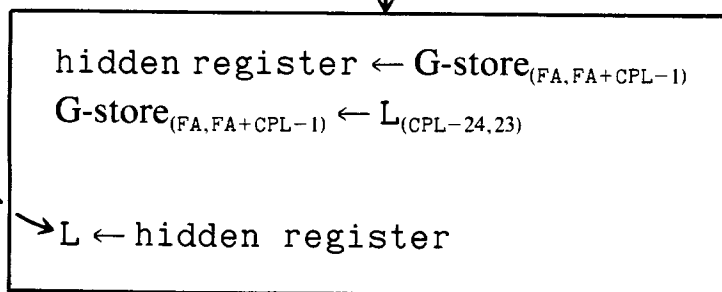
Right-justified with left zero fill

SWAP / BITS REVERSE WITH X means



Right-justified with left zero fill

SWAP 0 BITS WITH L means



CPL determines field length

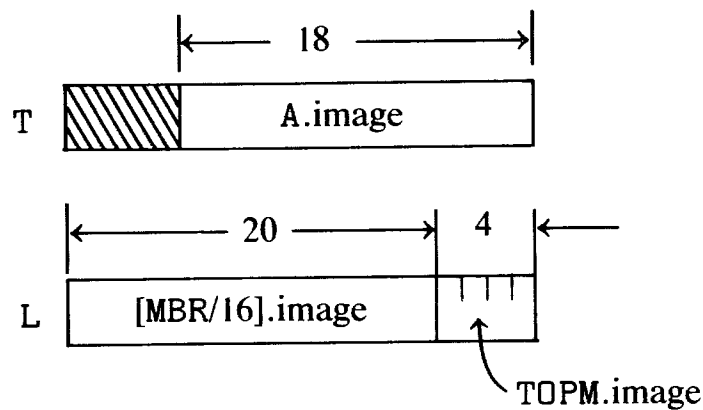
TRANSFER.CONTROL**Syntax**

TRANSFER . CONTROL

Semantics

See also the BIND instruction in Appendix B.

This instruction is to be issued after L and T have been preset as indicated



Execution of this instruction causes new values to be assigned to the A, TOPM, and MBR registers as follows

$$\begin{aligned} A &\leftarrow A . \text{image from T} \\ \text{TOPM} &\leftarrow \text{TOPM.image from L} \\ \text{MBR} &\leftarrow 16 \times [\text{MBR}/16].\text{image from L} \end{aligned}$$

Page References: 164–169

WRITE**Syntax**

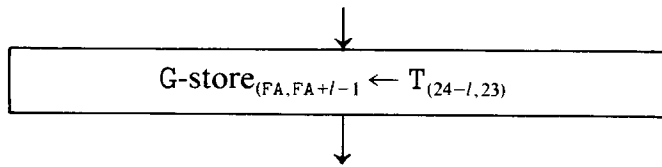
$$\text{WRITE literal BITS [REVERSE] FROM } \left\{ \begin{array}{c} T \\ X \\ Y \\ L \end{array} \right\}$$

$$\left[\left\{ \begin{array}{c} \text{INC} \\ \text{DEC} \end{array} \right\} \left\{ \begin{array}{c} \text{FA} \\ \text{FL} \end{array} \right\} \right] \left[\text{AND} \left\{ \begin{array}{c} \text{DEC} \\ \text{INC} \end{array} \right\} \left\{ \begin{array}{c} \text{FL} \\ \text{FA} \end{array} \right\} \right]$$

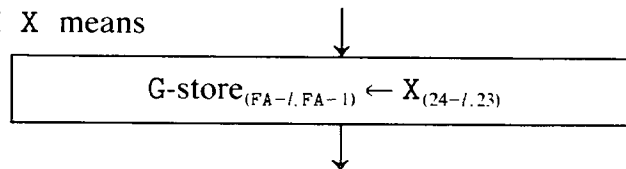
where *literal* is any integer 0 through 24

Semantics

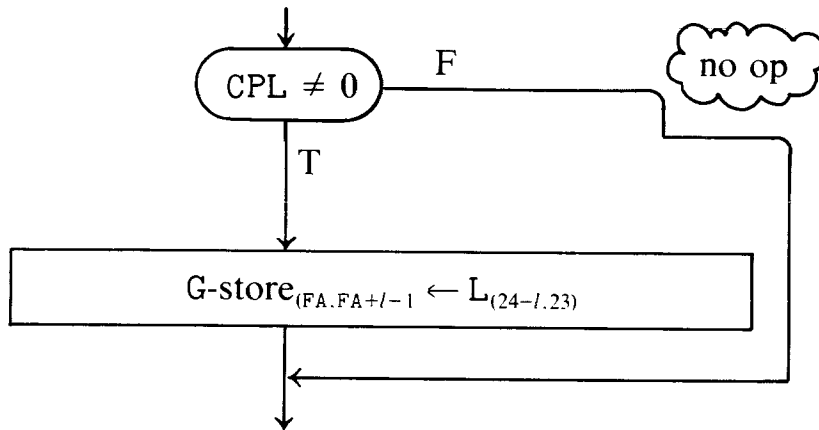
WRITE l BITS FROM T means



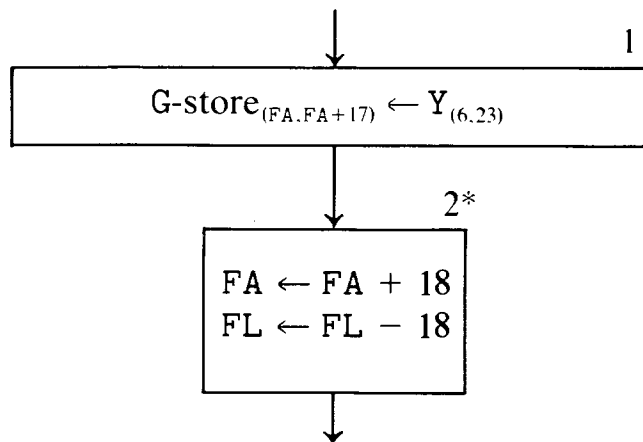
WRITE l BITS REVERSE FROM X means



WRITE 0 BITS FROM L means



WRITE 18 BITS FROM Y INC FA AND DEC FL means



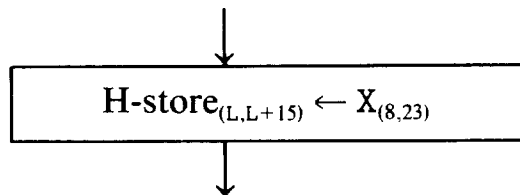
* Execution of box 2 overlaps execution of box 1.

WRITE MSML

Syntax

WRITE MSML FROM X

Semantics



Page reference: None

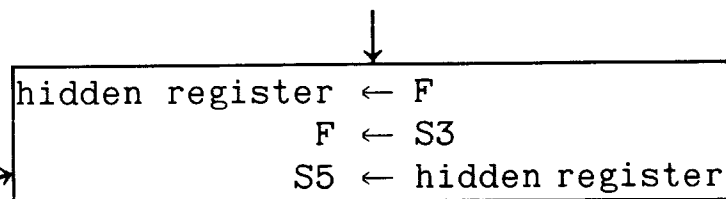
XCH

Syntax

$$XCH \begin{Bmatrix} S0 \\ S1 \\ \vdots \\ S14 \\ S15 \end{Bmatrix} F \begin{Bmatrix} S0 \\ S1 \\ \vdots \\ S14 \\ S15 \end{Bmatrix}$$

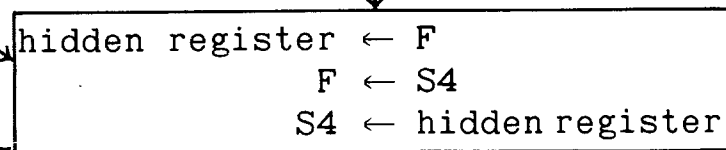
Semantics

XCH S3 F S5 means



48-bit transfers

XCH S4 F S4 means



Pure interchange of F and S4

Page reference: 25-27

3 Nonexecutable MIL STATEMENTS

ADJUST LOCATION

Syntax

$$\text{ADJUST LOCATION TO } \left\{ \begin{array}{l} \text{literal} \\ \text{location} \end{array} \left\{ \begin{array}{l} \text{PLUS} \\ + \end{array} \right\} \text{literal} \right\}$$

where *literal* must be a number $\equiv 0$ modulo 16

Example ADJUST LOCATION TO LOCATION + 1600

Semantics

The ADJUST declaration is a command to the MIL assembler to change the value of its location counter. The assembler initializes a location counter to zero at the beginning of its operation and increments this counter by 16 after each microinstruction is assembled, so the counter's value corresponds to the address of the next microinstruction to be assembled relative to an H-store base address of zero.

Examples

ADJUST LOCATION TO 160 forces the counter to be changed to 160.

ADJUST LOCATION TO LOCATION PLUS 256 forces the counter to be incremented by 256 and is equivalent to inserting a sequence of 16 NOP instructions (256 zero bits) into the generated code stream at this point.

Page references: 162

DEFINE

Syntax

DEFINE identifier = string #

Semantics

Any subsequent reference to *identifier* will be replaced by *string*.

Examples

```

DEFINE  BASE.OF.INTERPRETER = SOA   #
DEFINE  NUMBER.OF.TERMINALS = 3     #
DEFINE  I                          = CA   #
DEFINE  FLAG                        = CB(0) #

```

DEFINE *continued*

Ordinarily, the scope of a DEFINE is the entire MIL program. However DEFINE scopes may be nested, as are ALGOL declarations, using a similar blocking device:

```
BEGIN
    LOCAL DEFINES
    :
END
```

A given identifier may be DEFINEd or reDEFINEd within such a block, just as an ALGOL identifier may be declared or redeclared within a begin, end block. (See especially Figure 4.4 on p. 53.)

Page references: 51–53

DECLARE**Syntax**

```
DECLARE declare-element-1
    [, declare-element-2[, ... ]... ] ;
```

where the format of a declare element is either *simple* or *structured*.

A simple declare element has the syntax

$$\left\{ \begin{array}{l} \text{identifier-1}[(\text{array-size})] \\ (\text{identifier-and-array-size-list}) \end{array} \right\}$$

$$\left[\text{REMAPS} \left\{ \begin{array}{l} \text{identifier-2} \\ \text{BASE.ZERO} \end{array} \right\} \right] \left\{ \begin{array}{l} \text{FIXED} \\ \text{CHARACTER}(\text{length}) \\ \text{BIT}(\text{length}) \end{array} \right\} [\text{REVERSE}]$$

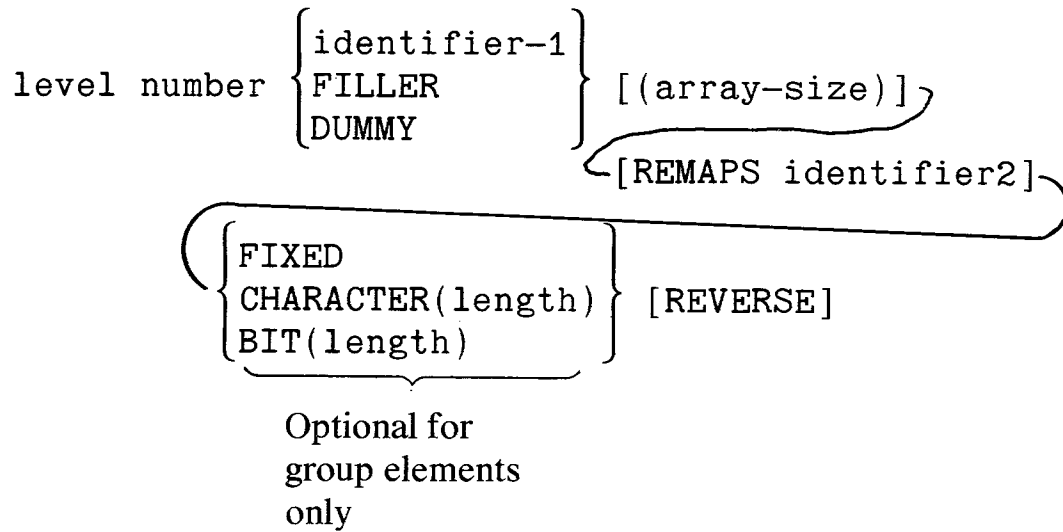
where identifier-and-array-size-list has the syntax

```
id-1[(array-size-1)][, id-2[(array-size-2)]
    [, id-2[(array-size-3)]...]
```

Examples using simple declare elements are

```
DECLARE A CHARACTER(20) REVERSE;
DECLARE A FIXED,
        B CHARACTER(3),
        C BIT(20),
        (D, E, F, (5)) BIT(4)
        G(20) FIXED,
        H(3) CHARACTER(6)
        AA REMAPS A CHARACTER(3),
        CC(4) REMAPS C BIT(5);
DECLARE P REMAPS BASE.ZERO BIT(200);
```

A structured declare element has the syntax



Example

```
DECLARE 01 MABEL REVERSE,
        02 BAKER,
        03 CHARLIE BIT(20);
        03 DOG BIT(30);
        02 ELLEN CHARACTER(5);
```

(Group elements, such as MABEL and BAKER, need not contain type-length attributes.)

Semantics

The DECLAREs of a MIL program define (but do not allocate) successive field addresses of G-store relative to BASE.ZERO, the value of the BR resiter. Each declare element carries with it an explicit or implicit length based on the given type-length attribute, so a G-store address is associated with each DECLARED identifier.

If an identifier is given the additional attribute REVERSE, then the G-store address associated with that identifier is the address that would be needed for a READ REVERSE or WRITE REVERSE of the corresponding object from G-store, i.e., the bit address that is one higher than the rightmost bit of the field corresponding to that identifier.

If a group item is declared REVERSE, then each of its subitems will be treated as if it were declared REVERSE.

DECLARED arrays may only be one-dimensional, so that if a group item of a structure is an array, then an array specification may not appear in any subordinate group item. Such subordinate group items are regarded as descriptions of array elements.

DECLARE *continued**Example*

```

01 ABEL(5) BIT(48)
02 BAKER FIXED,
02 CHARLY FIXED;

```

declares that ABEL is an array of 5 elements, each 48 bits long. Each element of ABEL is further described by declare items at level 02.

Any piece of G-store previously declared as either a simple or a structured item may be renamed as a REMAPS item. For example,

```

DECLARE ABEL1 REMAPS ABLE BIT(240),
01 ABEL2 (10) REMAPS ABEL BIT(24),
02 BAGEL BIT(3)
02 CABEL BIT(20);

```

renaps the declared structure ABEL in the following two ways

1. ABEL1 is a single field of 240 bits that exactly covers ABEL
2. ABEL2 is a 10-element array exactly covering ABEL, but each element of ABEL2 consists of a 3-bit field, BAGEL, a 20-bit field CABEL, and an implied 1-bit filler field.

If only the subfields of a REMAPS group item will ever be referred to, then it is not necessary to give a unique identifier for that group item. A DUMMY REMAPS item may be used, e.g.,

```

DECLARE 01 DUMMY(10) REMAPS ABEL BIT(24),
02 BAGEL BIT(3),
03 CABEL BIT(20);

```

Page references: 51–56

MACRO**SYNTAX**

```

MACRO macro-name [(fp-1 [, fp-2 [, ... [, fp-n] ... ]])] =

    statement-1
    [statement-2]
    :
    [statement-n] #

```

where $fp-i$ is the i th formal parameter.

Semantics

A macro-name is declared with no, one, or more formal parameters, which in turn may occur within the statement(s) of the macro's body. When the macro is referred to in a subsequent MIL statement, the text of the macro body is inserted, with string replacements of each occurrence of a formal parameter by its corresponding actual parameter. A formal parameter may not represent a label. All MACRO definitions must appear ahead of any executable statement.

Example

```
MACRO WRITE.ITEM (ITEM1, ITEM2, ITEM3) =
  XCH ITEM1 F ITEM1
  WRITE 24 BITS FROM ITEM2 ITEM3 FA AND DEC FL
  XCH ITEM1 F ITEM1 #
```

When later referenced as

```
WRITE.ITEM(S2, X, INC)
```

this reference will be replaced by the in-line MIL code

```
XCH S2 F S2
WRITE 24 BITS FROM X INC FA AND DEC FL
XCH S2 F S2
```

Page references: 56, 57, 100

TABLE**Syntax**

```
TABLE label
  BEGIN
    first literal
    second literal
    ...
    last literal
  END
```

Allowed literals include character strings, binary, and hexadecimal constants. Decimal constants are not allowed. The label following TABLE is treated by the MIL assembler as an addressable label.

TABLE *continued*

Example (See also Section 7.3)

```

TABLE MIL.OPCODES
  BEGIN
    "MOVE"
    "IF"
    "GO TO"
    "BIAS"
    "SET"
    "RESET"
  END

```

Semantics

The MIL assembler presets a space in the microcode beginning at an address corresponding to label with a sequence of values corresponding to the literals given between the BEGIN, END pair of the TABLE declaration.

Page references: 157–160

4 SPECIAL MIL EXPRESSIONS

4.1 ADDRESS

An expression of the form

ADDRESS(label)

may appear in a MIL statement in place of a literal. The ADDRESS value of a label is the value of the MIL assembler's location counter that corresponds to that label's occurrence in the MIL program. An address value of a label is necessarily congruent to zero modulo 16.

Examples

MOVE ADDRESS(MULTIPLY) TO S1A moves the address value of the label MULTIPLY to S1A. This address value is relative to the beginning of the assembled program.

MOVE ADDRESS(-HERE) TO X moves the address value of the point label .HERE (the first one that precedes this statement) to X.

4.2 DATA.LENGTH

An expression of the form

DATA.LENGTH(declared identifier)

may appear in a MIL statement in place of a literal. A declared identifier is any simple or array identifier that appears in a DECLARE statement. The DATA.LENGTH for that identifier is its length *in bits*.

Examples Given

```
DECLARE 01 MABEL(5) BIT(56),  
        02 BAKER FIXED,  
        02 CHARLY CHARACTER(4);
```

then

```
MOVE DATA.LENGTH(MABEL) TO X
```

would assign the value 5×56 or 280 to X,

```
MOVE DATA.LENGTH (CHARLY) TO FL
```

would assign 32 to FL, and

```
WRITE DATA.LENGTH(BAKER) BITS FROM Y
```

would write 24 bits from Y.

Appendix B

Abridged reference guide to the B1726

1 B1726 REGISTER SUMMARY

REGISTER NAME

X, Y

Each is a 24-bit register which can be used as a receiver register (source/sink) for G-store transfers.

Each is always one of the inputs to the 24-bit function box.

Neither is composed of 4-bit subregisters.

Inspection of the high and low bits of X is possible (MSBX, LSUX).

Inspection of only the low bit of Y (LSUY) is possible.

Each can be compared against zero or against the other ($X=0$, $X\neq 0$, $X > Y$, $X \leq Y$, etc., $Y=0$).

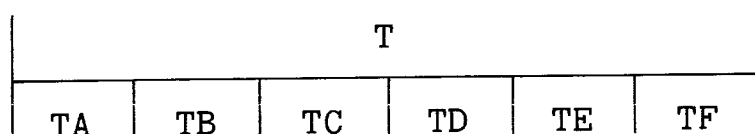
X and Y can each be shifted or rotated right or left.

The 48-bit field formed by concatenating X and Y may be shifted or rotated left or right.

T

A 24 bit register which can be used as a receiver register (source/sink) for G-store transfers.

T is composed of 4-bit subregisters in the following fashion:



This allows any bit of the T-register to be tested [bits are numbered from left (0) to right (23)].

Bits of any subregister may be tested [bits of a subregister are numbered from left (0) to right (3), e.g., TE(2) is the third bit from the left of TE and can also be referred to as T(18)].

T does not act as an input to the 24-bit function box.

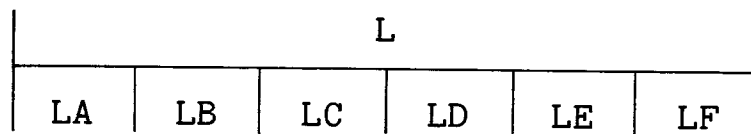
The T register may be shifted or rotated left and the result may be transferred to any other register.

A group of contiguous bits may be extracted from anywhere within T and the group transferred to X, Y, T, or L.

L

A 24-bit register which can be used as a receiver register (source/sink) for G-store transfers.

L is composed of 4-bit subregisters in the following fashion:

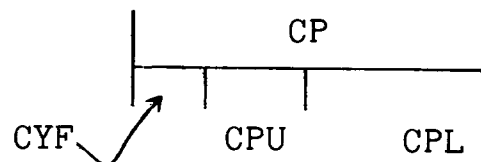


L may *not* be rotated or shifted.

L does not act as an input to the 24-bit function box.

CP

An 8-bit control register



The subfields of CP are

CYF (1 bit), the “carry-in” for the 24-bit function box (for SUM, DIFF)

CPU (2 bits), the arithmetic mode of the 24-bit function box:

- 00 binary
- 01 4-bit decimal
- 10 not defined
- 11 not defined

CPL (5 bits), the width of the 24-bit function box; any precision up to and including 24 bits may be specified.

If CP = 0, the 24-bit function box is in essence turned off, since the length (CPL) is zero.

CP cannot be used for general storage.

FA

A 24-bit register.

It is *not* composed of 4-bit registers.

The contents of FA specify the location of G-store to be accessed during a G-store transfer (READ, WRITE, or SWAP).

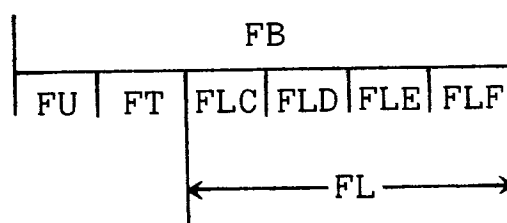
FA may not be shifted or rotated.

It is not possible to test a particular bit within FA or to compare the contents of FA with any register or value.

A fast 24-bit adder is attached to FA. The adder can be used to add or subtract a small constant (0–24) to FA or to add or subtract the 24-bit contents of the left half of a scratchpad to FA (S1A, for example).

FB

A 24-bit register composed of 4-bit registers in the following manner



This structure allows any bit of FB to be tested.

Some of the subfields of FB have special uses:

FU can alter the contents of CP when used with the BIAS BY UNIT instruction.

FLC, FLD, FLE, FLF comprise a 16-bit register called FL. The FL register has a fast adder attached which can increment or decrement FL by a small constant (0–24).

The 16-bit value of FL can be compared with zero or with the value represented by the low-order 16 bits of SOB. This 16-bit field of SOB is called SFL.

TAS (The stack)

A group of 32 registers (24 bits wide) of which only one (the top) is available (i.e., addressable) at any time.

The LIFO discipline is observed.

Any data may be placed on the stack and retrieved later.

The hardware will automatically place a microcode return address on the stack when entering a microsubroutine, facilitating the return from a subroutine.

Overflow or underflow (i.e., pushing too many values or popping too many values) is not detected, and care must be taken to prevent incorrect operation of a microcode subroutine.

The scratchpads

This is an array of 32 registers (each 24 bits wide) that is organized in the following fashion.

	A (left half)	B (right half)
S0		
S1		
:		
S15		

Access to any A or B half is allowed.

Access to a left, right pair (e.g., S3A, S3B) as a 48-bit register is also allowed when transferring to/from or exchanging with the FA, FB register pair (see instructions LOAD, STORE, XCH).

In addition, the A half of a scratchpad register may be added to or subtracted from the FA register.

4-bit registers

Any bit of a 4-bit register can be examined.

Up to two bits of a single 4-bit register can be tested in a single microinstruction.

Many of the four-bit registers have preassigned meanings and reflect the status of X, Y, FL, etc.

Most other four-bit registers are subfields of 24-bit registers (T, L, FB). There are only two four-bit registers that have no preassigned meanings and are not part of a larger register. These are CA and CB.

A

A 16-bit register which serves as the microinstruction address register. This is the program counter. (On many conventional machines this register is not addressable.)

A jump can be achieved by moving a value to A.

A return address can be generated by moving from A, e.g., to the stack (TAS).

While executing an instruction, the A-register reflects the address of the instruction that follows the current instruction.

M

A 16-bit register that contains the current microinstruction.

This register is not useful as a source of data. However, the next

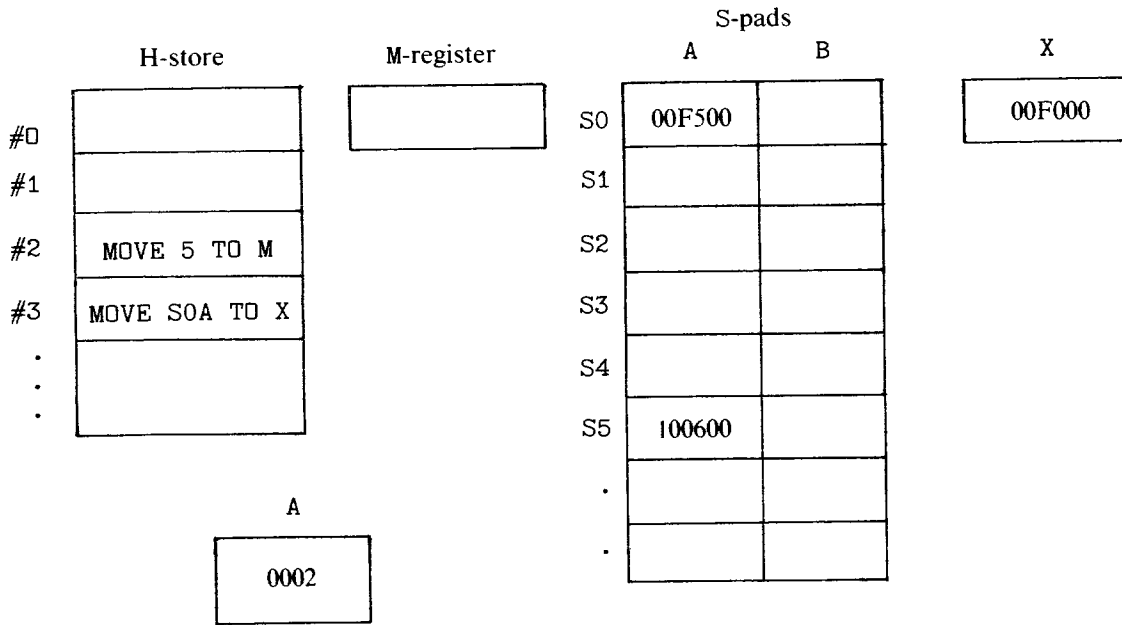


Figure B.1. Snapshot just before instruction #2 is executed.

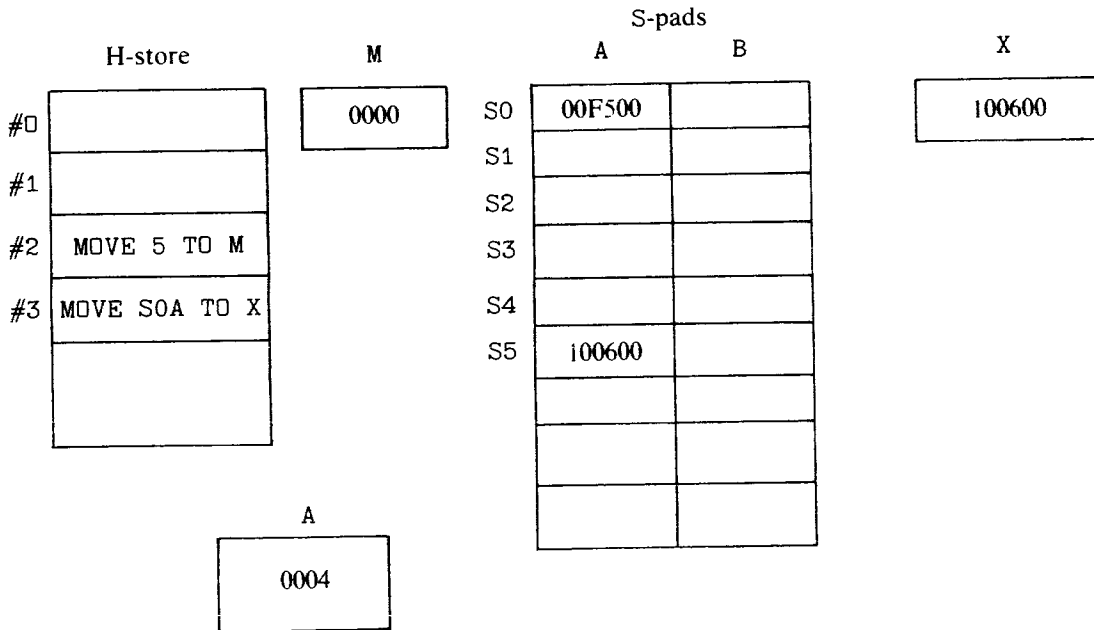


Figure B.2. Snapshot just after instruction #3 is executed.

instruction that the hardware executes can be modified by a value moved to M; for example,

```
MOVE 5 TO M
MOVE SOA TO X
```

will actually perform as follows. We begin as shown in Figure B.1.

Instruction cycle for A = 0002 The hardware

1. ORs microinstruction #2 to M, which is assumed to have been cleared to zero as a result of step 4 of the preceding instruction cycle
2. Increments A to 0003
3. Decodes the MOVE 5 TO M instruction
4. Clears M
5. Executes MOVE 5 TO M, which results in

M 0005

Instruction cycle for A = 0003 The hardware

1. ORs microinstruction #3 with M, forming MOVE S5A TO X in M
2. Increments A to 0004
3. Decodes the modified instruction
4. Clears M
5. Executes MOVE S5A TO X, yielding the snapshot in Figure B.2.

Note that the #3 microinstruction is not changed.

Also note that the M-register retains the “0005” (modification) for only one instruction cycle.

2 TESTABLE BITS FOR IF STATEMENTS

The following testable conditions all reside in 4-bit registers. The bit numbering is the software convention, starting with bit 0 on the extreme left (high-order position).

REGISTER WHERE BIT IS LOCATED	CONDITION SYNTAX		
	PRIMARY	ALTERNATE	NOTES
BICN			Binary conditions—Read only
	LSUY	BICN(0)	Low-order bit of Y ^a
	CYF	BICN(1)	Carry input for ALU
	CYD	BICN(2)	Borrow out from ALU ^b
XYCN	CYL	BICN(3)	Carry out from ALU ^c
			X-Y Conditions—Read only
	MSBX	XYCN(0)	High-order bit of X ^c
	X=Y	XYCN(1)	24-bit comparison ^b

REGISTER WHERE BIT IS LOCATED	CONDITION SYNTAX		
	PRIMARY	ALTERNATE	NOTES
XYST	X<Y	XYCN(2)	24-bit comparison ^b
	X>Y	XYCN(3)	24-bit comparison ^b
			X-Y states—Read only
	LSUX	XYST(0)	Low-order bit of X ^a
	ANY. INTERRUPT	XYST(1)	On if any interrupt bit is set
FLCN	Y≠0	XYST(2)	
	X≠0	XYST(3)	
			Field-length conditions—Read only
	FL=SFL	FLCN(0)	16 bit comparison
	FL>SFL	FLCN(1)	16-bit comparison
CC	FL<SFL	FLCN(2)	16-bit comparison
	FL≠0	FLCN(3)	
			External interrupts ^d —read and write
	CC(0)		State light
CD	CC(1)		Set by hardware timer every $\frac{1}{10}$ sec (no mnemonic) ^d
	CC(2)		Set by I/O controllers for service request ^d
	CC(3)		Set by switch labeled “INT” on front panel ^d
			Abnormal main memory conditions—read and write
T	CD(0)		Set by parity error detected in main memory ^d
	CD(1)		Set by program to allow writes to all of storage (<i>override</i>)
	CD(2)		Set when a read out of bounds is attempted
	CD(3)		Set when a write out of bounds is attempted ^d
L	Tx(i)	T(j)	Any bit of T or L or of a subregister of T or L, x::=A B C D E F i::=0 1 2 3 j::=0 1 2 3 . . . 21 22 23
	Lx(i)	L(j)	
CA	CA(i)		Any bit of CA or CB
CB	CB(i)		i::=0 1 2 3
FB	FU(i)		Any bit of FB or of a subregister of FB
	FT(i)		
	FLC(i)	FL(k)	i::=0 1 2 3

FLD(<i>i</i>)	FL(<i>k</i>)	$k ::= 0 1 2 3 \dots 14 15$
FLE(<i>i</i>)	FL(<i>k</i>)	
FLF(<i>i</i>)	FL(<i>k</i>)	

- ^a Conditioned by CPU.
- ^b Not conditioned by CPL.
- ^c Conditioned by CPL.
- ^d This interrupt also sets XYST(1).

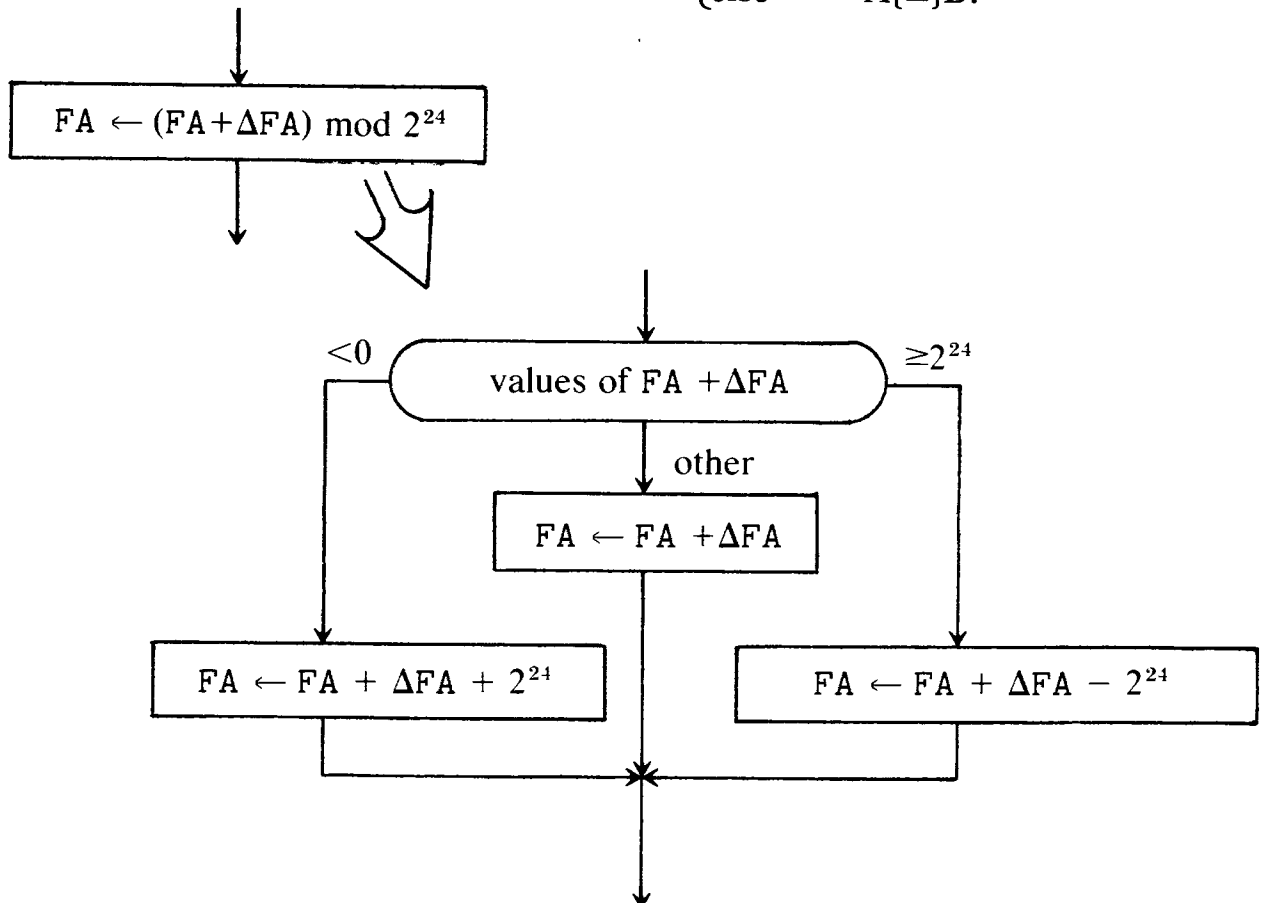
3 MICROINSTRUCTIONS: SYNTAX AND SEMANTICS²

This section lists the B1726 microinstructions alphabetically by their mnemonics. Summary charts are given at the end.

Notation

The syntax notation is self-explanatory. The semantics notation is the same as that used in Appendix A except that we add one new convention to express *modular arithmetic* as follows:

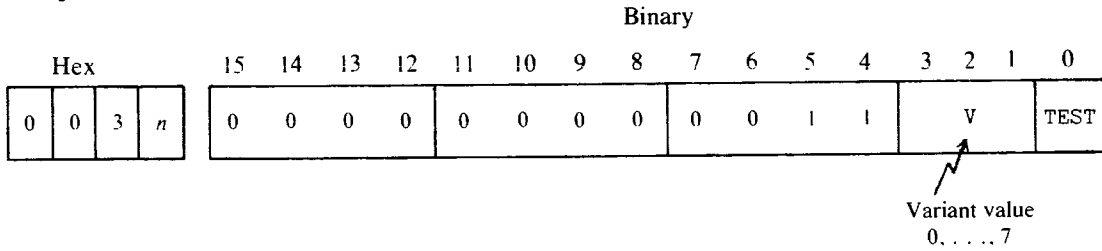
$$(A\{\pm\}B) \bmod C \text{ means if } A\{\pm\}B \equiv \begin{cases} <0, \text{ then } A\{\pm\}B+C, \\ \geq C, \text{ then } A\{\pm\}B-C, \\ \text{else } A\{\pm\}B. \end{cases}$$



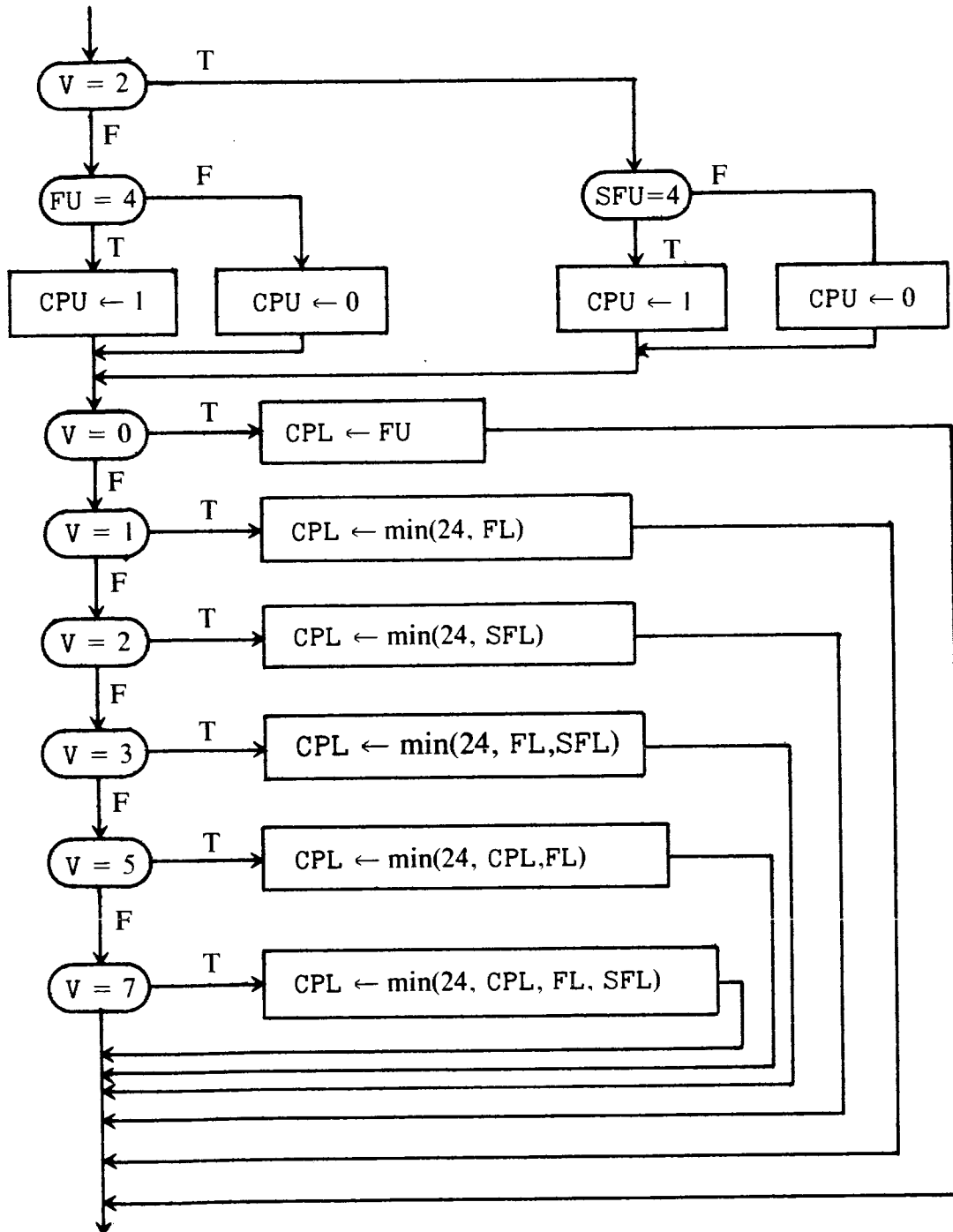
² Bit positions for microinstructions are indexed from *right* to *left* to conform with hardware conventions.

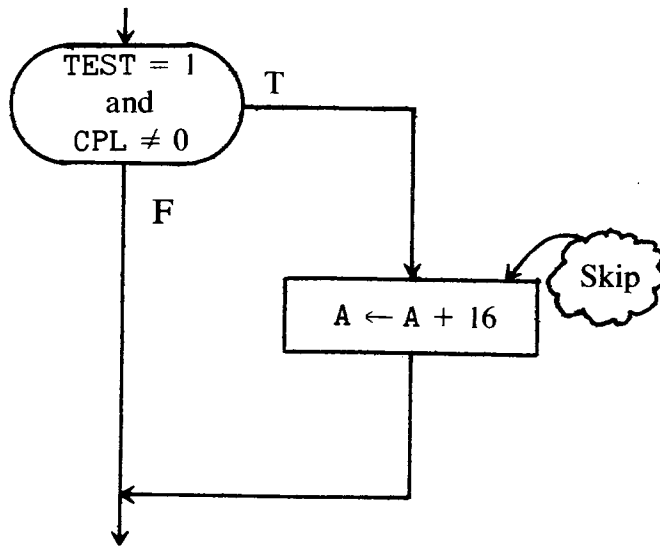
BIAS

Syntax



Semantics

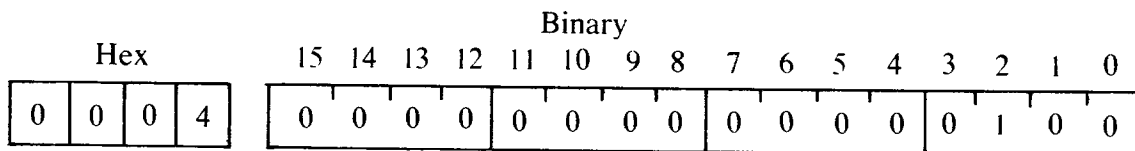




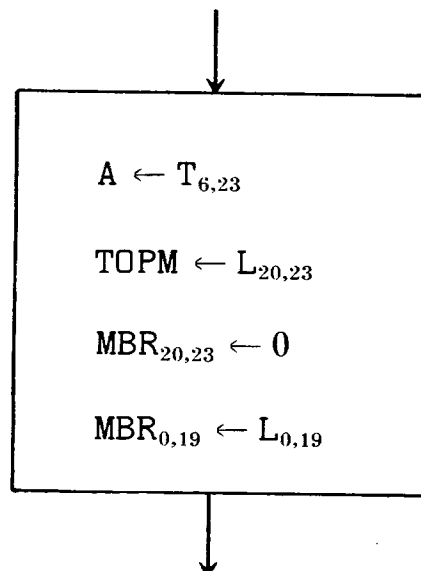
Note: Variants $V = 4$ and $V = 6$ have no effect on CPL.

BIND (Same as TRANSFER. CONTROL MIL statement)

Syntax

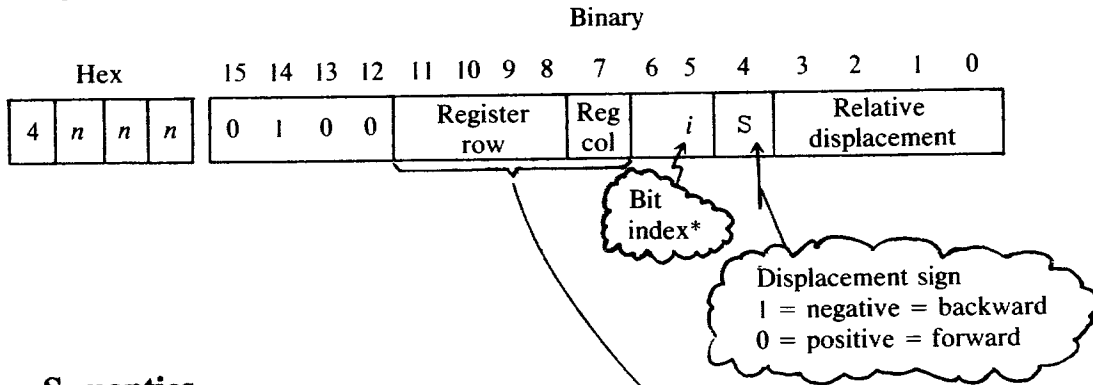


Semantics

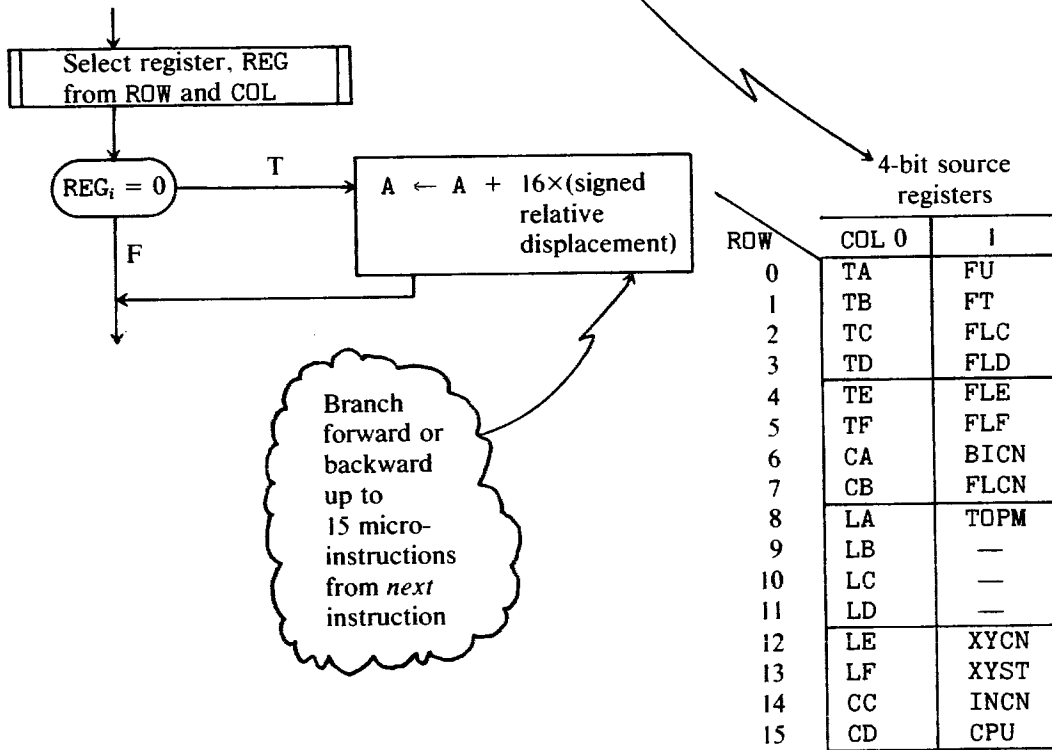


**BIT TEST
RELATIVE BRANCH FALSE**

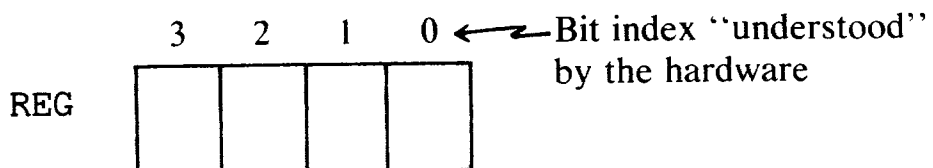
Syntax



Semantics

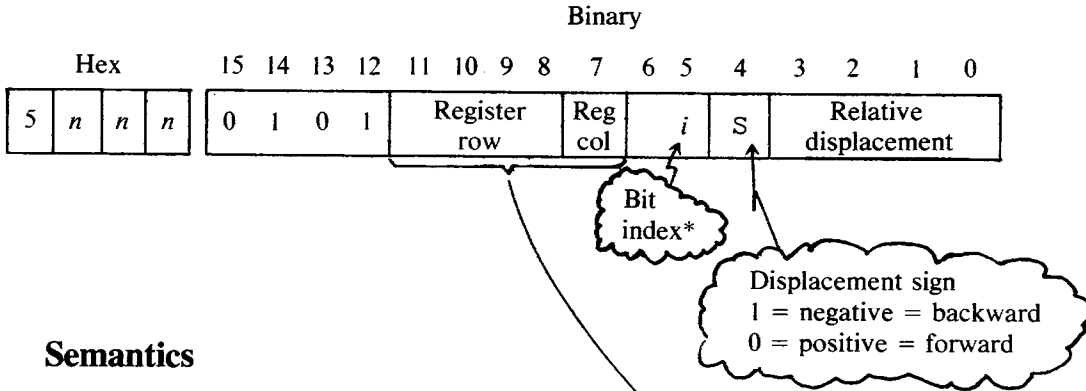


* The hardware assumes that register bits are indexed from *right* to *left*:

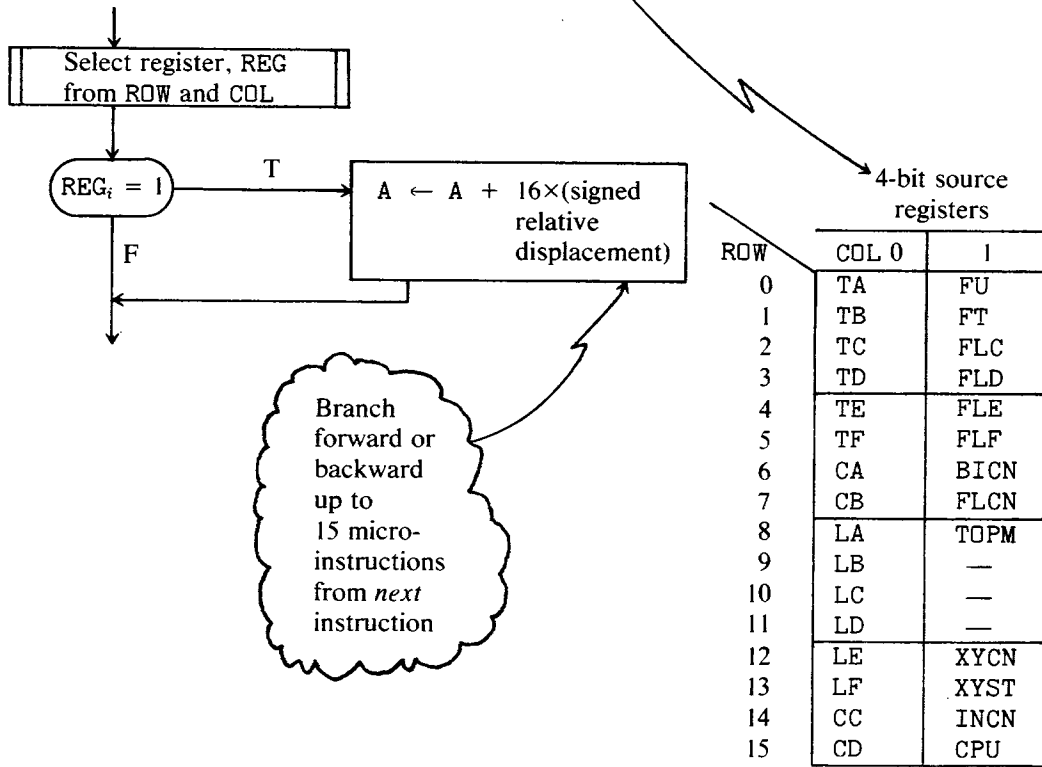


**BIT TEST
RELATIVE BRANCH TRUE**

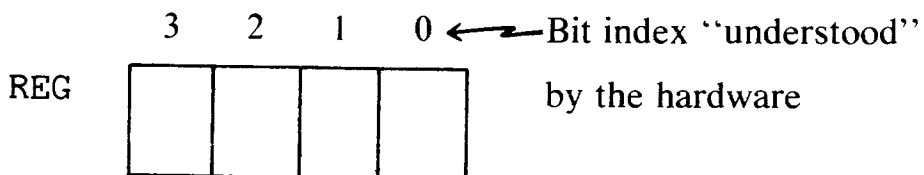
Syntax



Semantics

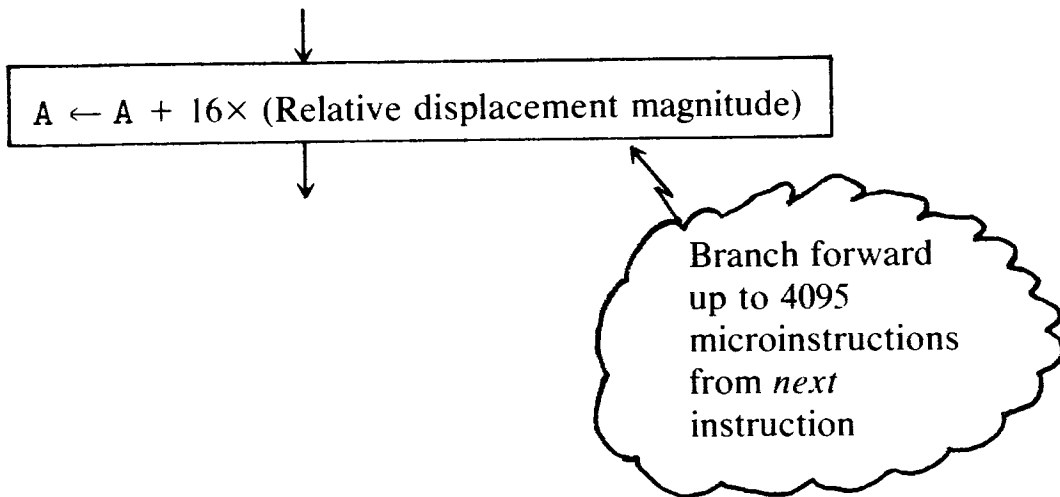


* The hardware assumes that register bits are indexed from *right* to *left*:



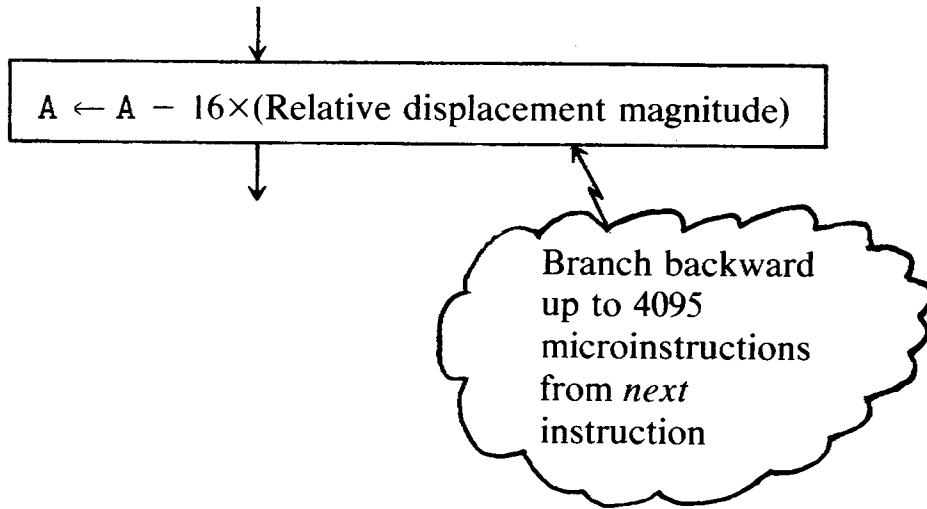
BRANCH RELATIVE FORWARD**Syntax****Binary**

Hex				15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	n	n	n	1	1	0	0	Relative displacement magnitude											

Semantics**BRANCH RELATIVE BACKWARD****Syntax****Binary**

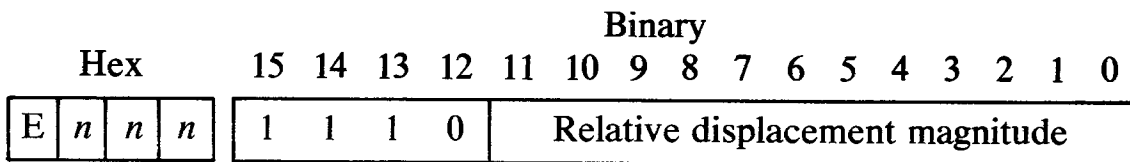
Hex				15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D	n	n	n	1	1	0	1	Relative displacement magnitude											

Semantics

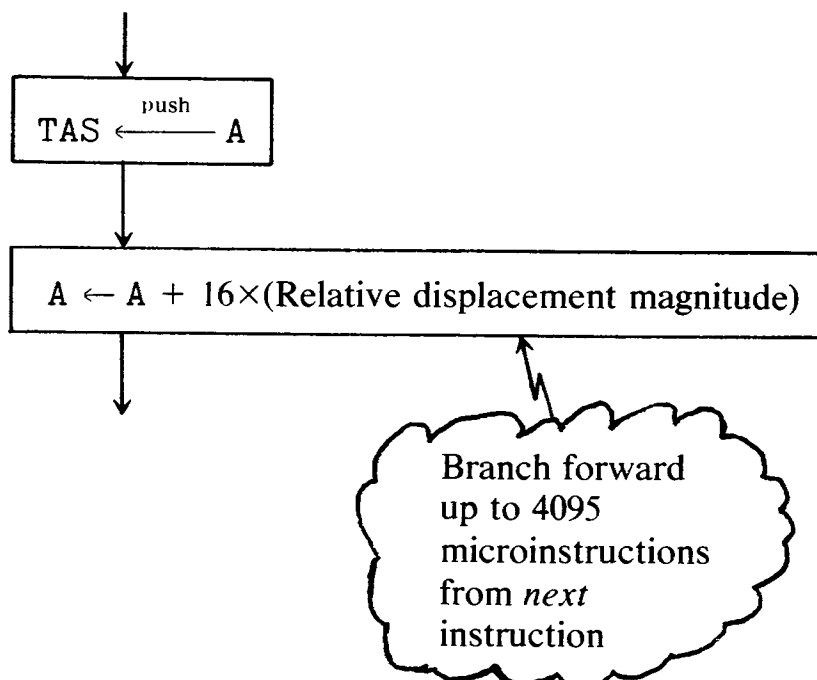


CALL REL FORWARD

Syntax

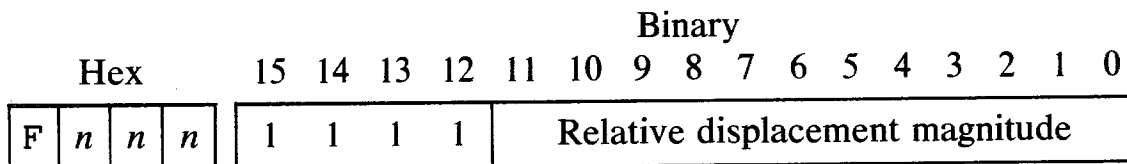


Semantics

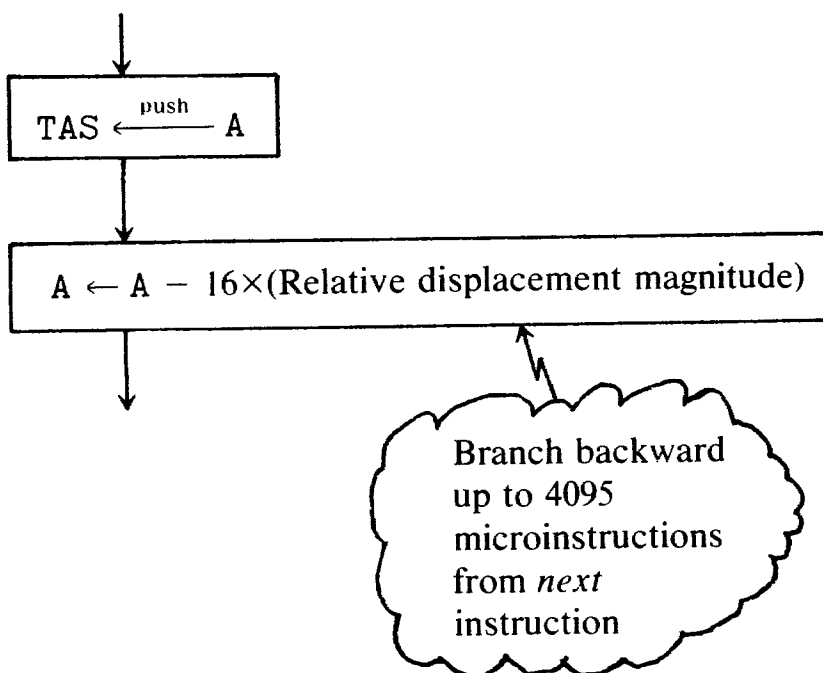


CALL REL BACKWARD

Syntax

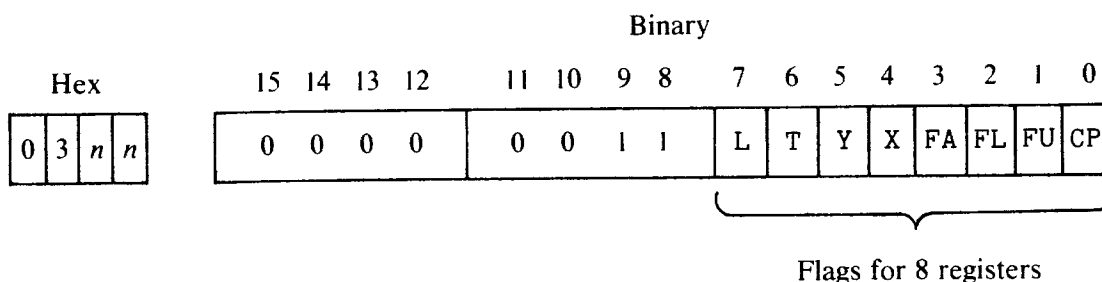


Semantics

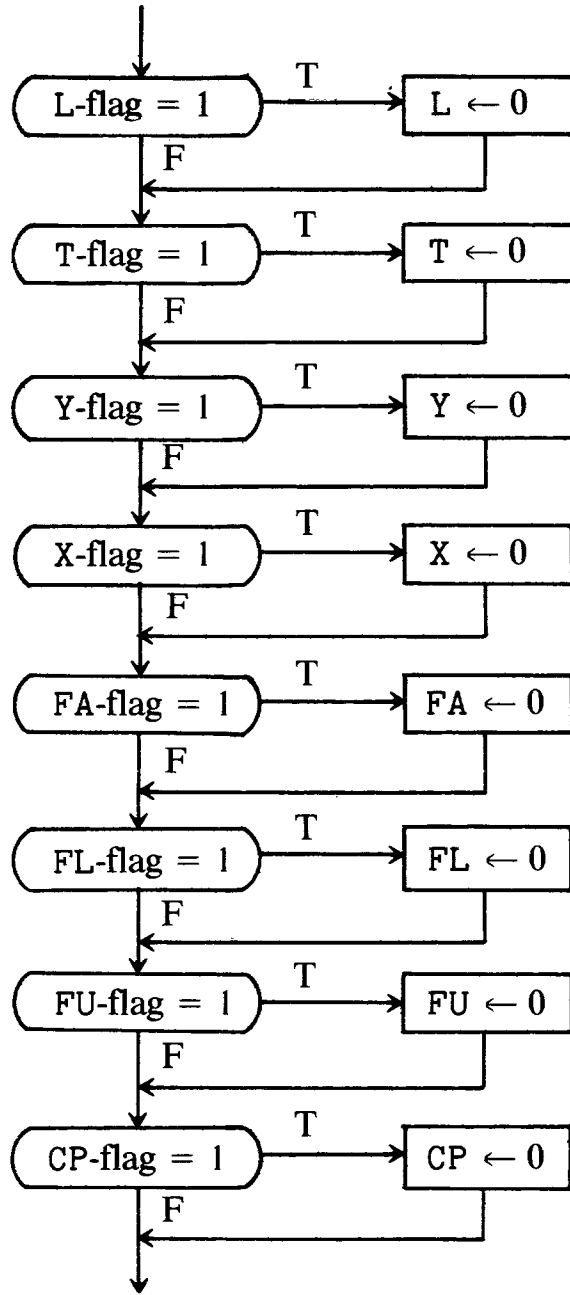


CLEAR REGISTERS

Syntax



Semantics



COUNT FA AND FL

Syntax

Hex

0	6	n	n
---	---	---	---

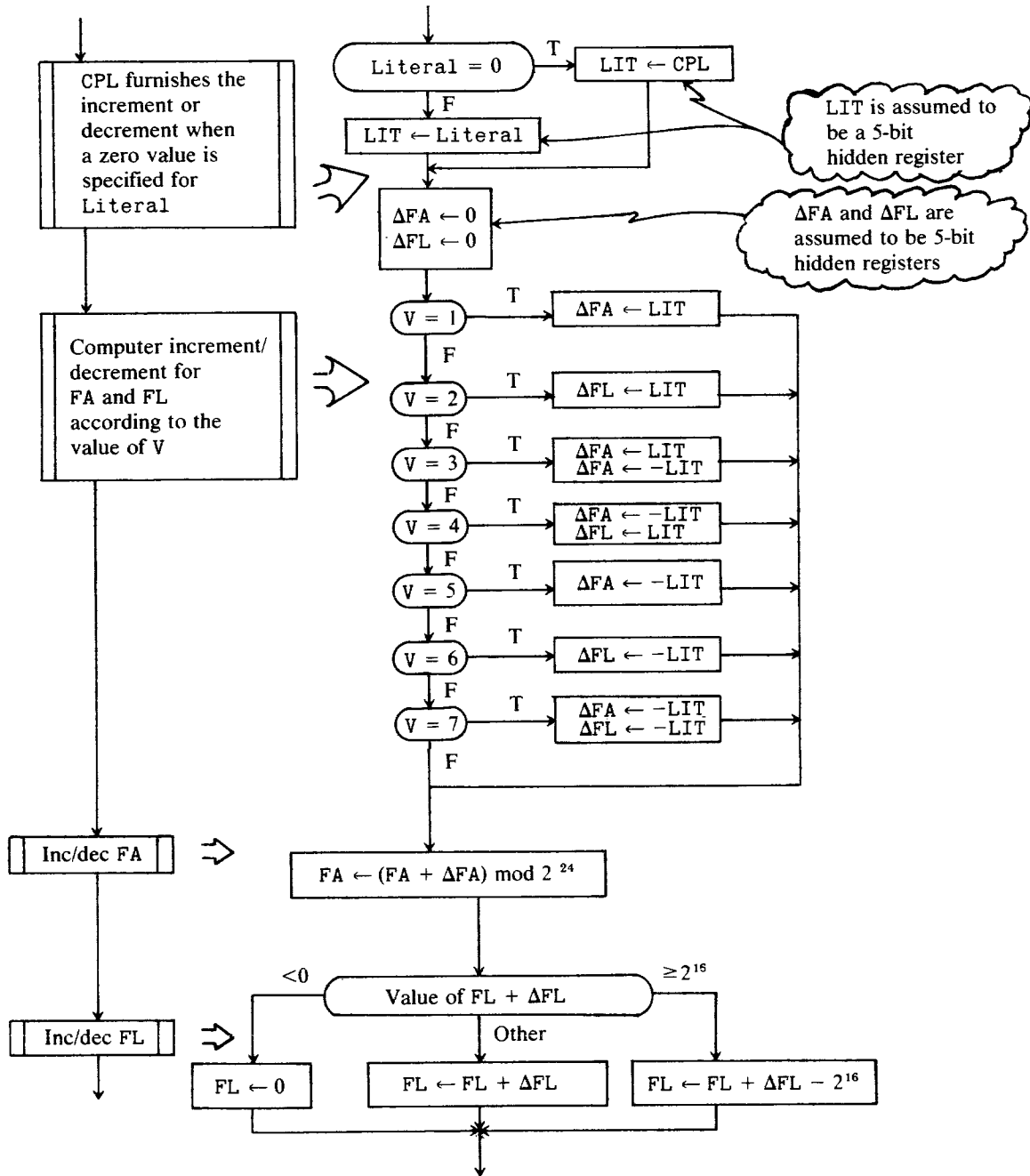
Binary

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	0		V				Literal	

Range is 0 to 24 inclusive

COUNT FA AND FL *continued*

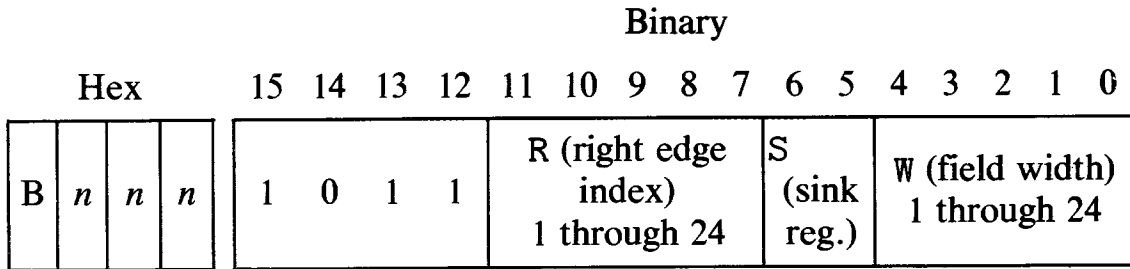
Semantics



Note: This instruction cannot count FA and FL up.

EXTRACT FROM T

Syntax



S	MEANS
00	X
01	Y
10	T
11	L

Semantics

An EXTRACT microinstruction specifies as its arguments

R, the rightmost index plus 1 of the field to be extracted

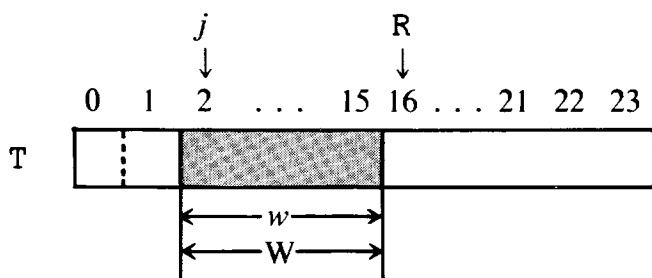
S, the sink register (X, Y, T, or L) to receive the extracted field

W, the width of the field to be extracted

A MIL instruction of the form

EXTRACT *w*-BITS FROM T(*j*). . .

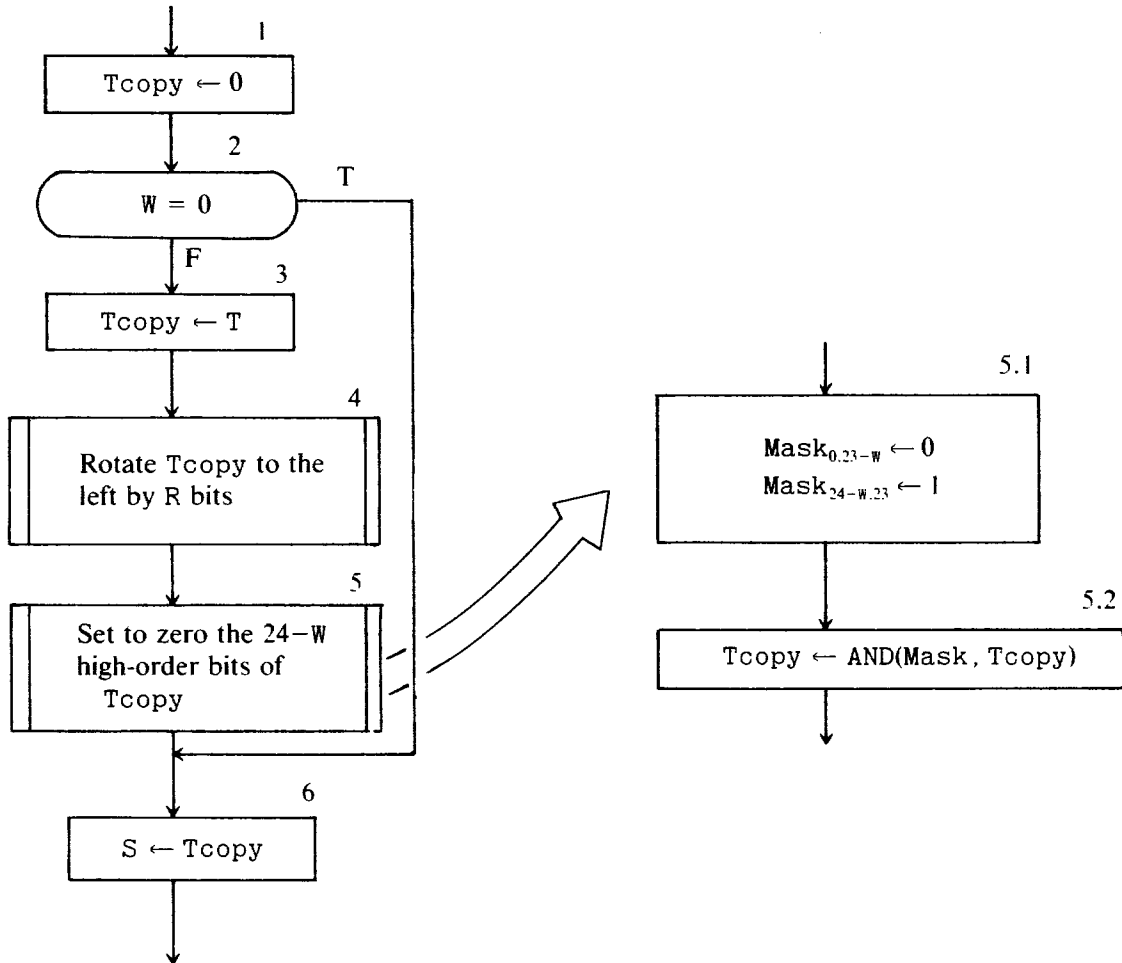
is mapped by the MIL assembler into an EXTRACT microinstruction by computing microargument R from the MIL arguments *j* and *w*:



MIL	=	MICRO
$j + w$	=	R
w	=	W

EXTRACT FROM T *continued*

Let Tcopy and Mask be 24-bit hidden registers as shown.



HALT

Syntax

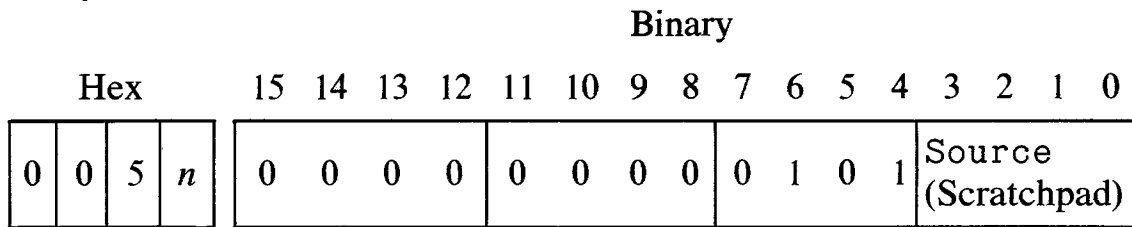
Hex				Binary															
				15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Semantics



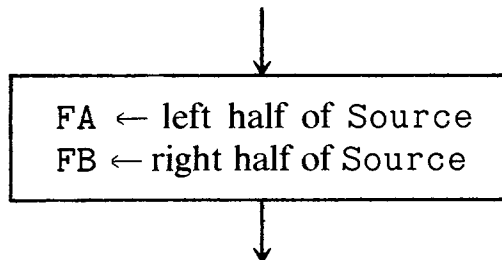
LOAD F

Syntax

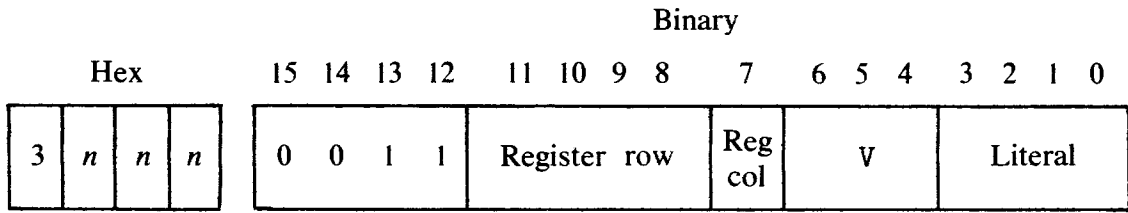


Source	MEANS
0	S0 (48 bits)
1	S1
⋮	⋮
15	S15

Semantics



MANIPULATE 4-BIT



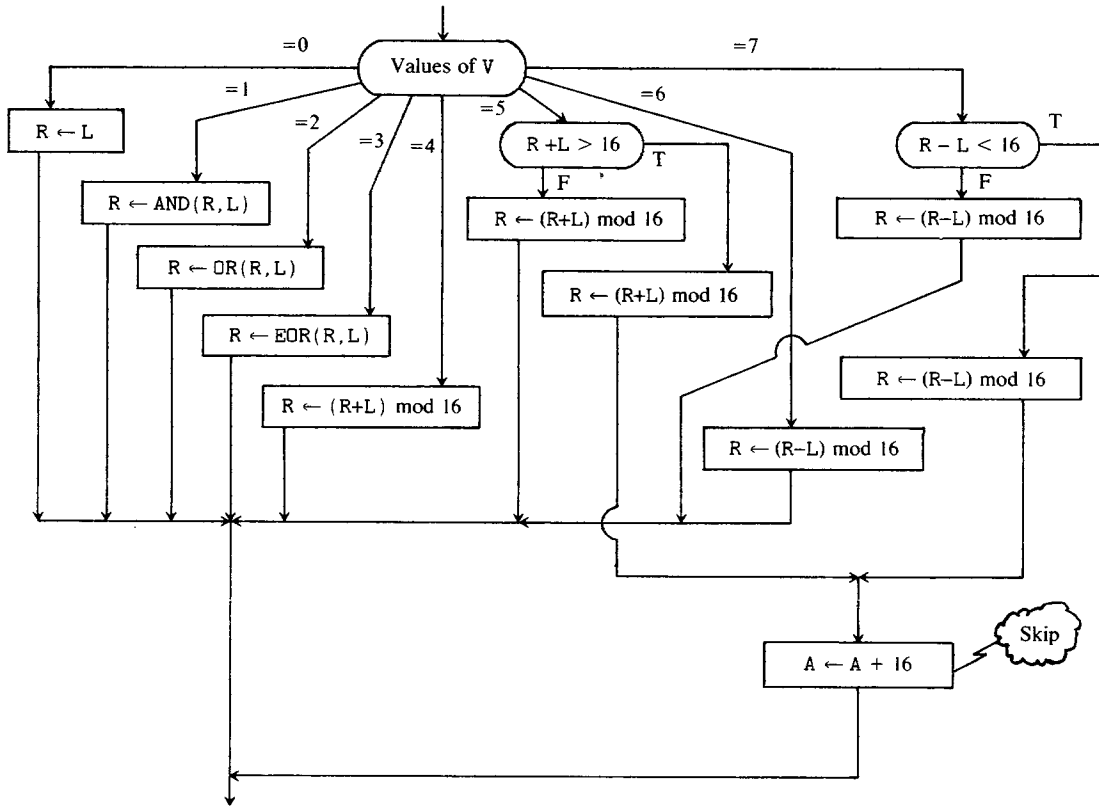
Affected 4-bit Register

ROW	COL 0	I
0	TA	FU
1	TB	FT
2	TC	FLC
3	TD	FLD
4	TE	FLE
5	TF	FLF
6	CA	—
7	CB	FLCN
8	LA	TOPM
9	LB	—
10	LC	—
11	LD	—
12	LE	
13	LF	
14	CC	
15	CD	

Semantics

Let

R = the specified register,
L = the specified literal.



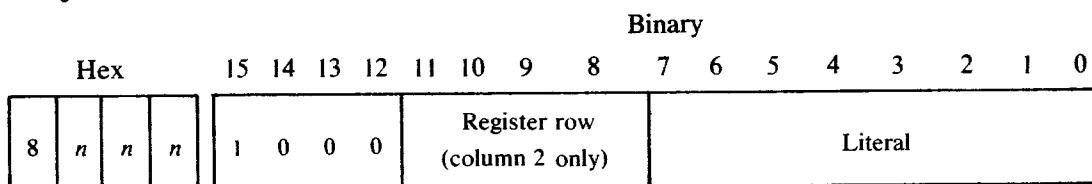
MONITOR

Syntax

Hex				Binary												
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	9	n	n	0	0	0	0	1	0	0	1	Literal				

Semantics

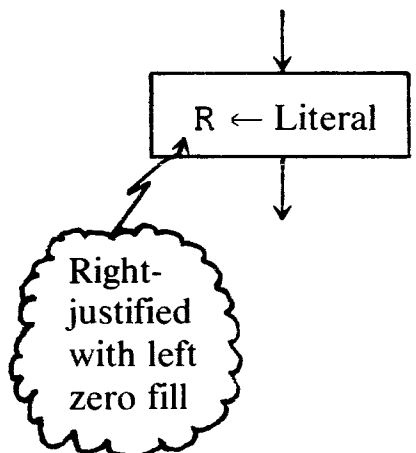
Literal is sent out on 8 monitor lines, one per bit of the literal, to be sensed by any device designed for the purpose.

MOVE 8-BIT LITERAL**Syntax**

ROW	Registers (col. 2)
0	X
1	Y
2	T
3	L
<hr/>	
4	A
5	M
6	BR
7	LR
<hr/>	
8	FA
9	FB
10	FL
11	TAS
<hr/>	
12	CP
13	—
14	—
15	—

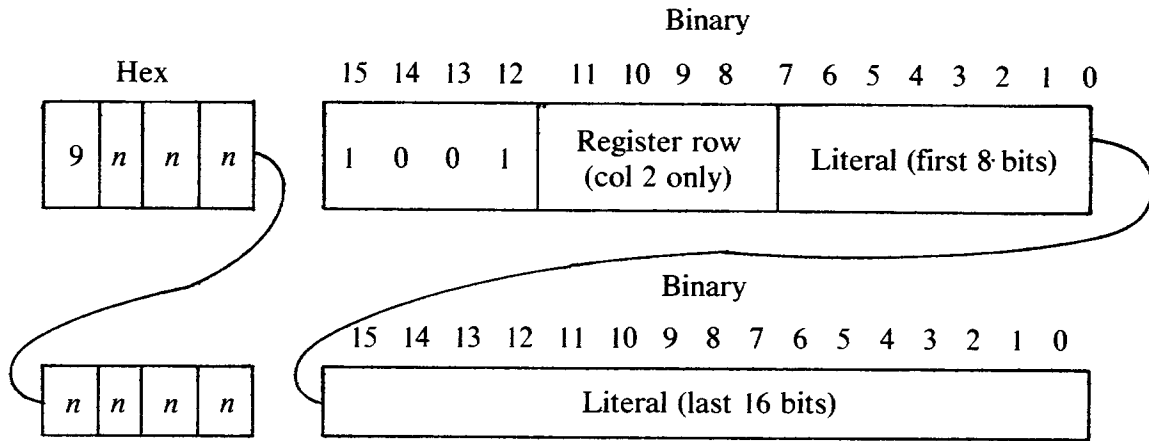
Semantics

Let R be the specified register.



MOVE 24-BIT LITERAL

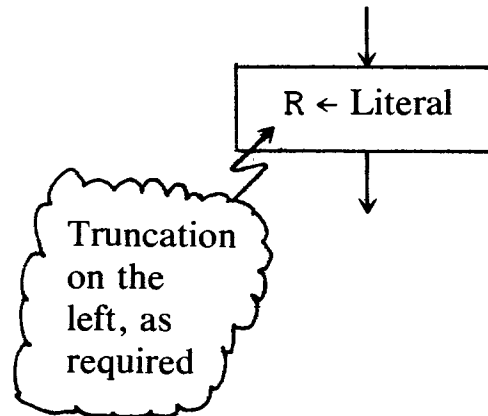
Syntax



Row	REGISTER (COL 2)
0	X
1	Y
2	T
3	L
4	A
5	—
6	BR
7	LR
8	FA
9	FB
10	FL
11	TAS
12	CP
13	—
14	—
15	—

MOVE 24-BIT LITERAL *continued***Semantics**

Let R be the specified register.

**NO OP****Syntax**

Hex				Binary															
				15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Semantics

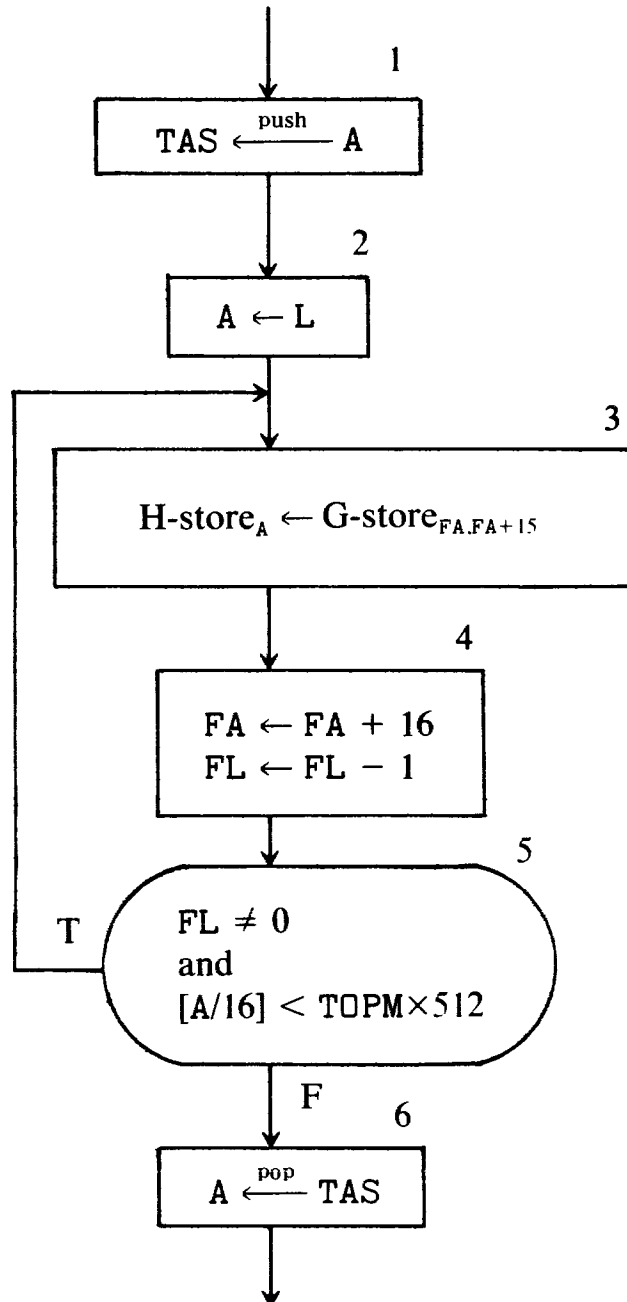
Operation is null.

NORMALIZE X**Syntax**

Hex				Binary															
				15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

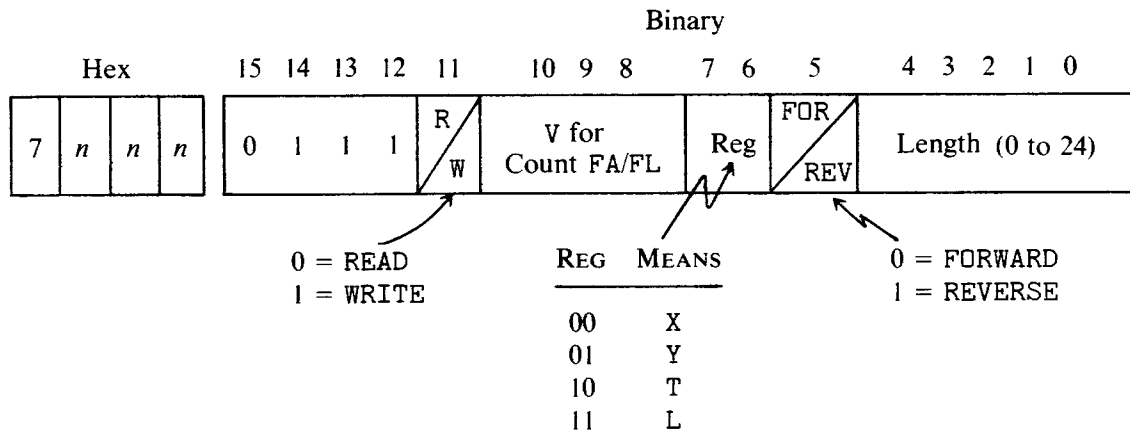
OVERLAY *continued*

Semantics



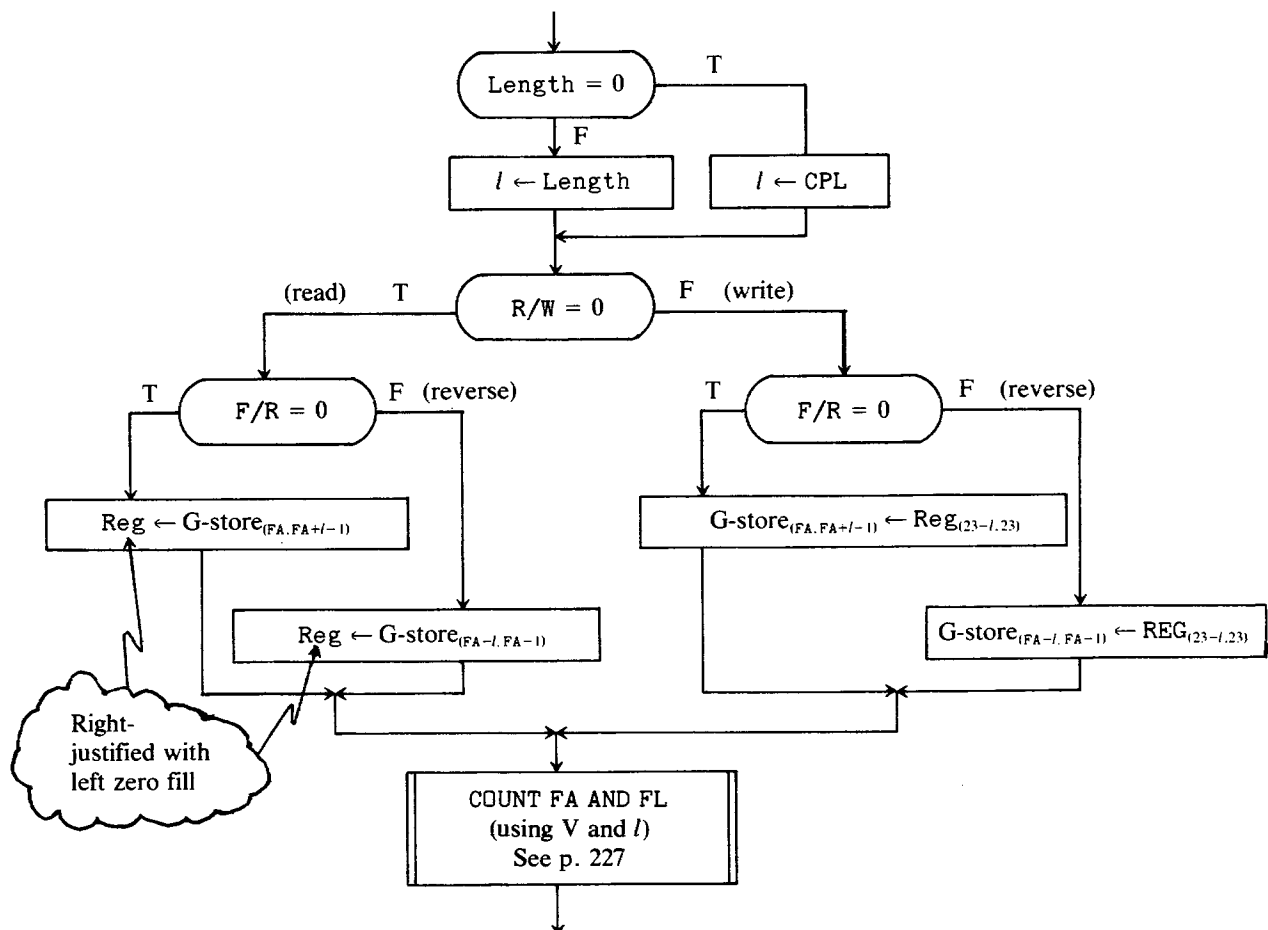
READ/WRITE G-store

Syntax



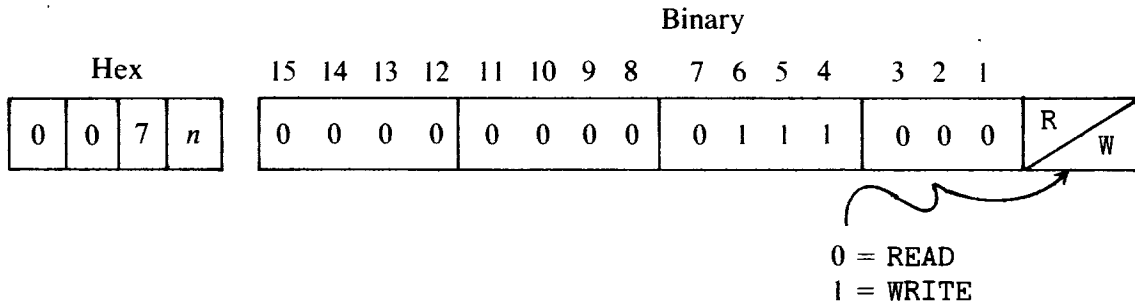
Semantics

Let l be a hidden register, and $F/R = \text{FOR}/\text{REV}$.

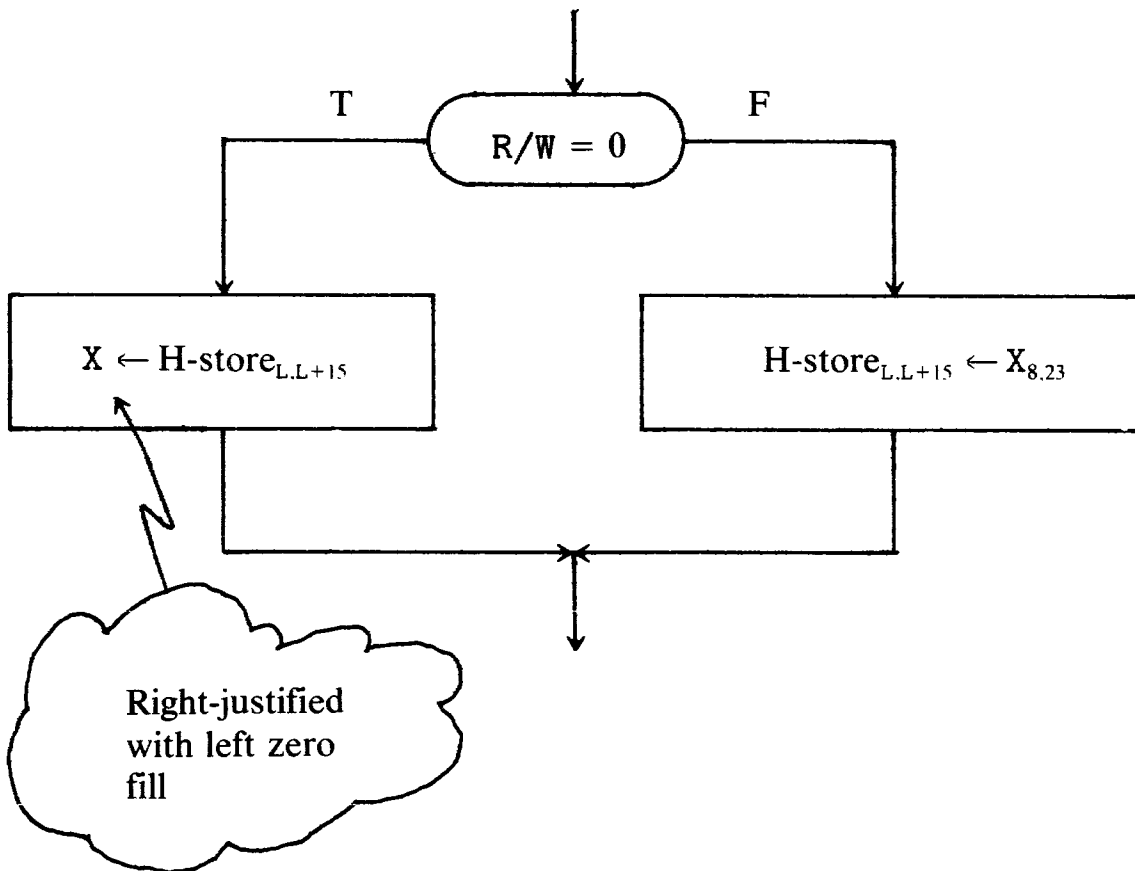


READ/WRITE H-store

Syntax

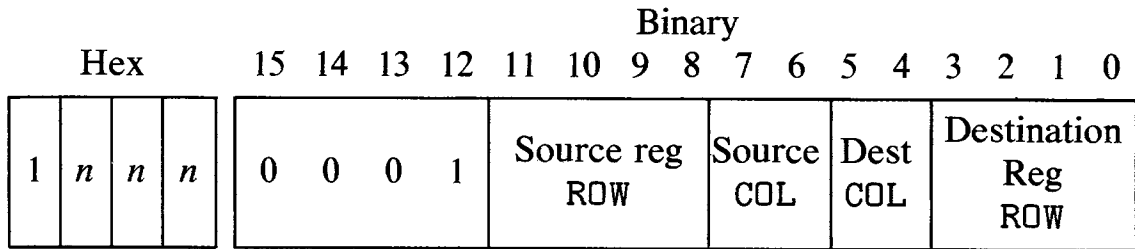


Semantics



REGISTER MOVE

Syntax

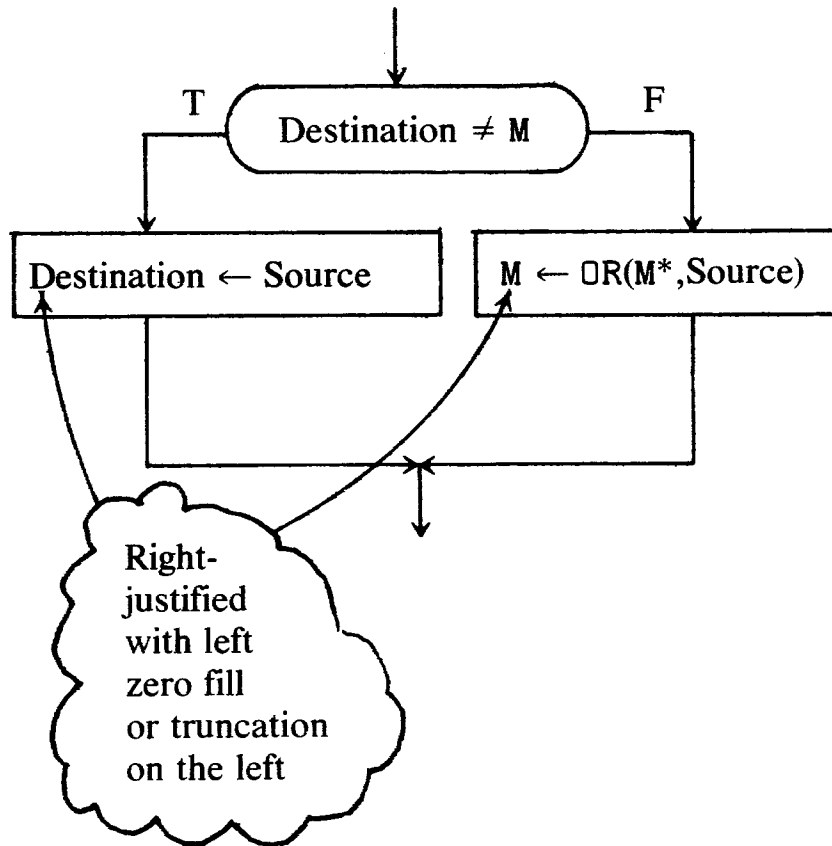


Registers

Row	COL 0	1	2	3 ^a
0	TA	FU	X	SUM
1	TB	FT	Y	CMPX
2	TC	FLC	T	CMPY
3	TD	FLD	L	XANY
4	TE	FLE	A	XEOY
5	TF	FLF	M	MSKX
6	CA	BICN ^a	BR	MSKY
7	CB	FLCN ^a	LR	XORY
8	LA	TOPM	FA	DIFF
9	LB	—	FB	MAXS
10	LC	—	FL	MAXM
11	LD	—	TAS	—
12	LE	XYCN ^a	CP	MBR
13	LF	XYST ^a	MSMA	DATA
14	CC	INCN ^a	—	CMND
15	CD	CPU ^b	—	NULL

^a Source only.

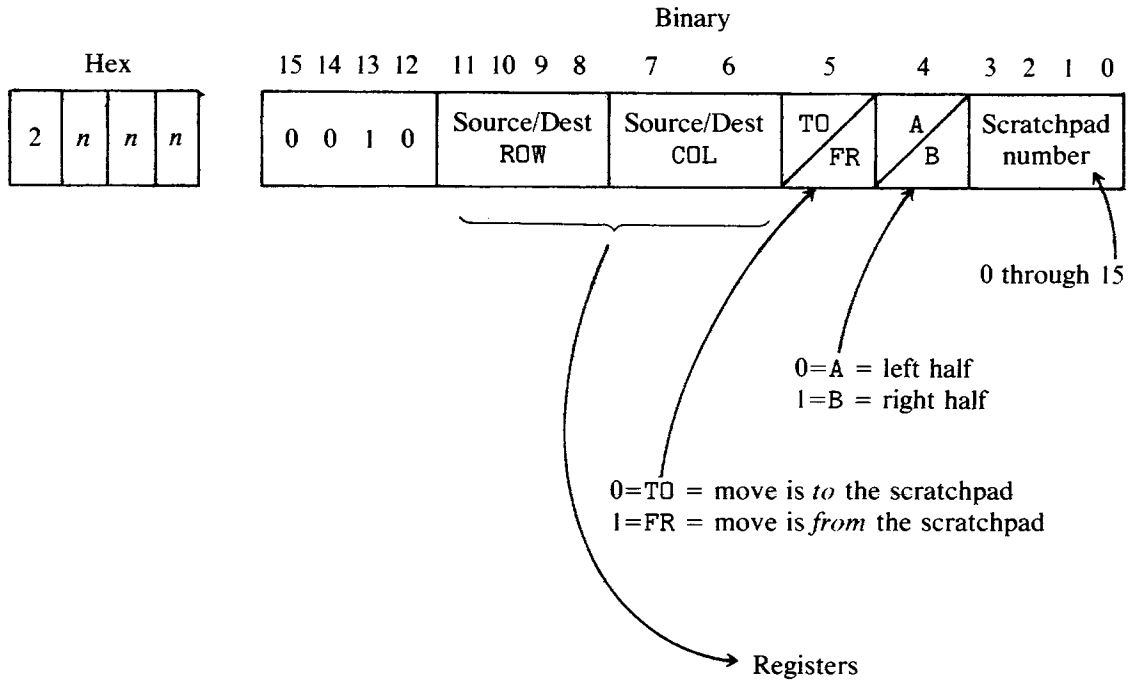
^b Destination only.

REGISTER MOVE *continued***Semantics**

where M^* is a copy of the next microinstruction.

SCRATCHPAD MOVE

Syntax



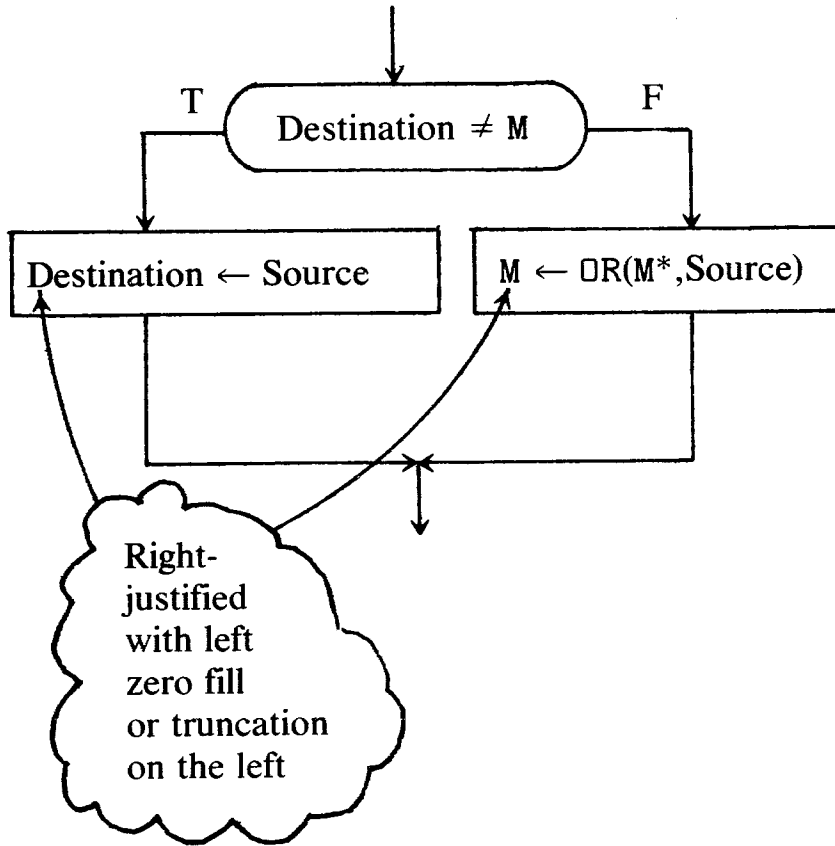
ROW	COL 0	1	2	3 ^a
0	TA	FU	X	SUM
1	TB	FT	Y	CMPX
2	TC	FLC	T	CMPY
3	TD	FLD	L	XANY
4	TE	FLE	A	XEOY
5	TF	FLF	M	MSKX
6	CA	BICN ^a	BR	MSKY
7	CB	FLCN ^a	LR	XORY
8	LA	TOPM	FA	DIFF
9	LB	—	FB	MAXS
10	LC	—	FL	MAXM
11	LD	—	TAS	—
12	LE	XYCN ^a	CP	MBR
13	LF	XYST ^a	MSMA	DATA
14	CC	INCN ^a	—	CMND
15	CD	CPU ^b	—	NULL

^a Source only.

^b Destination only.

SCRATCHPAD MOVE *continued*

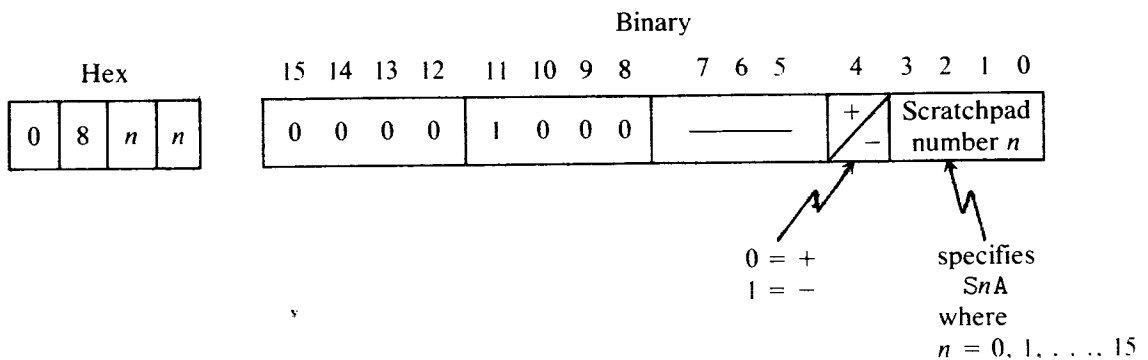
Semantics



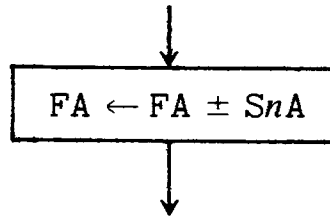
where M^* is a copy of the next microinstruction.

SCRATCHPAD RELATE FA

Syntax

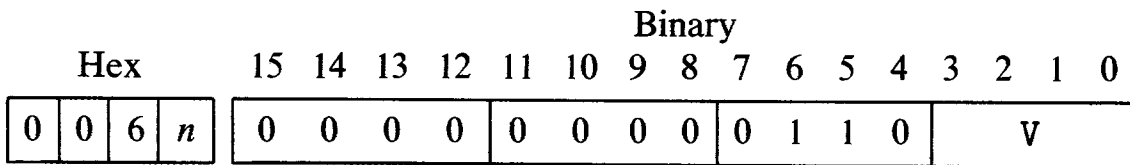


Semantics

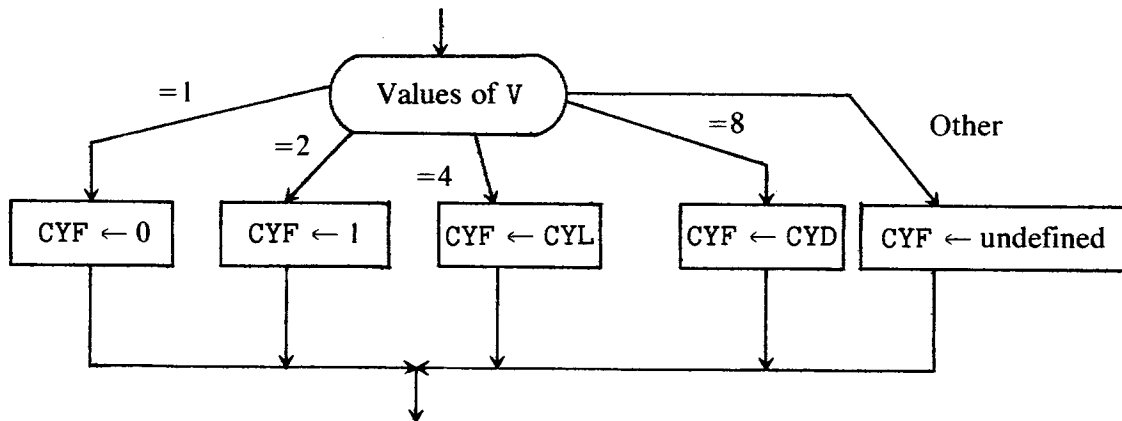


SET CYF

Syntax

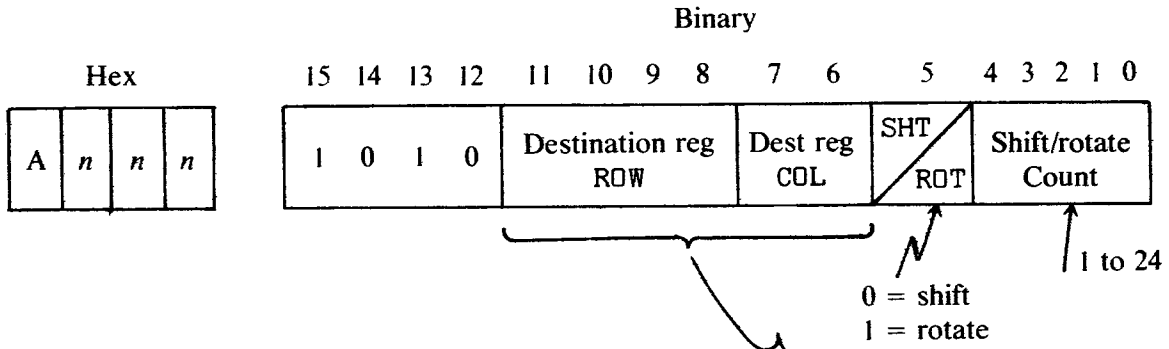


Semantics



SHIFT/ROTATE T

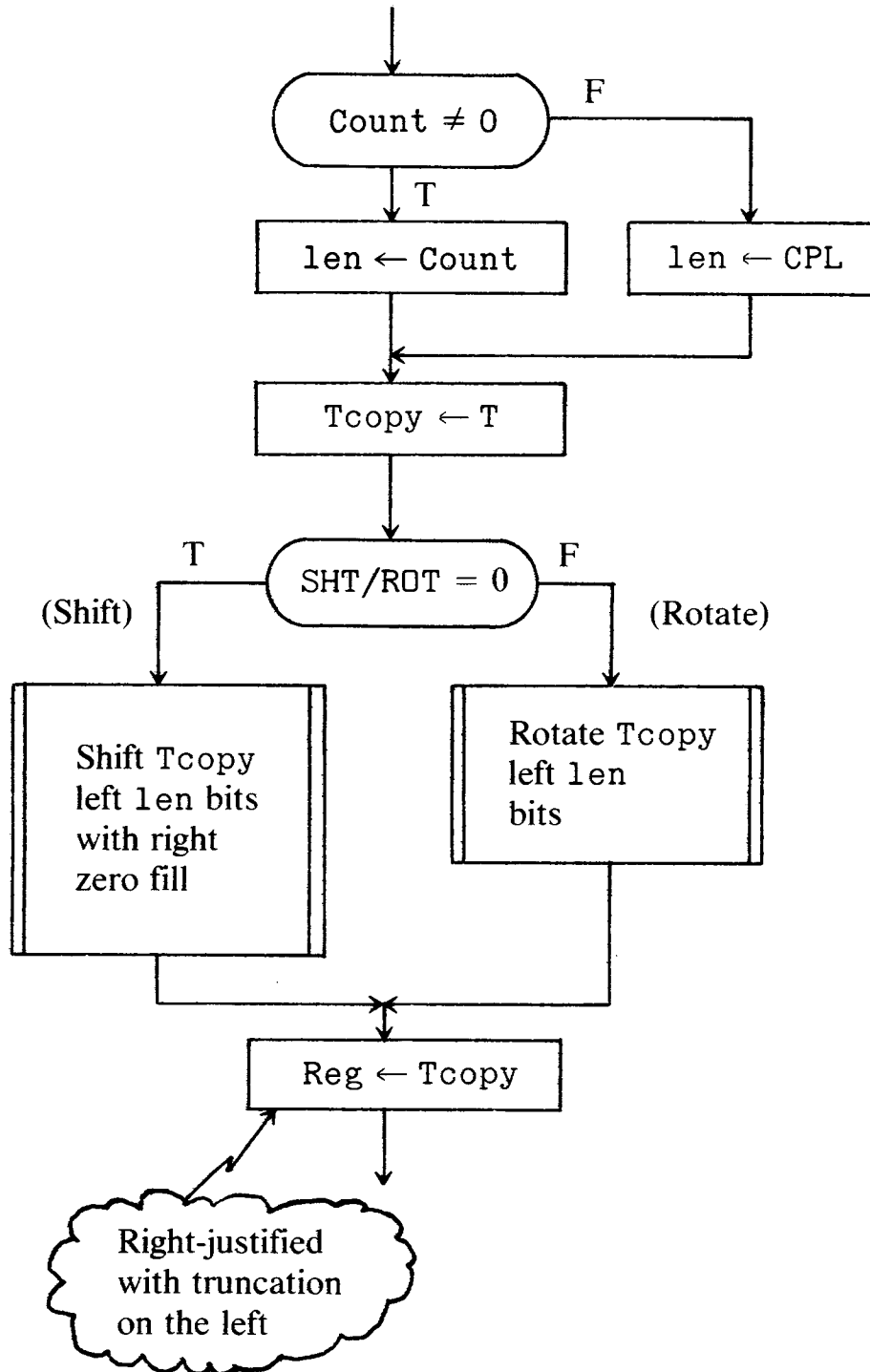
Syntax



ROW	COL 0	1	2
0	TA	FU	X
1	TB	FT	Y
2	TC	FLC	T
3	TD	FLD	L
4	TE	FLE	A
5	TF	FLF	M
6	CA	—	BR
7	CB	—	LR
8	LA	TOPM	FA
9	LB	—	FB
10	LC	—	FL
11	LD	—	TAS
12	LE	—	CP
13	LF	—	MSMA
14	CC	—	—
15	CD	CPU	—

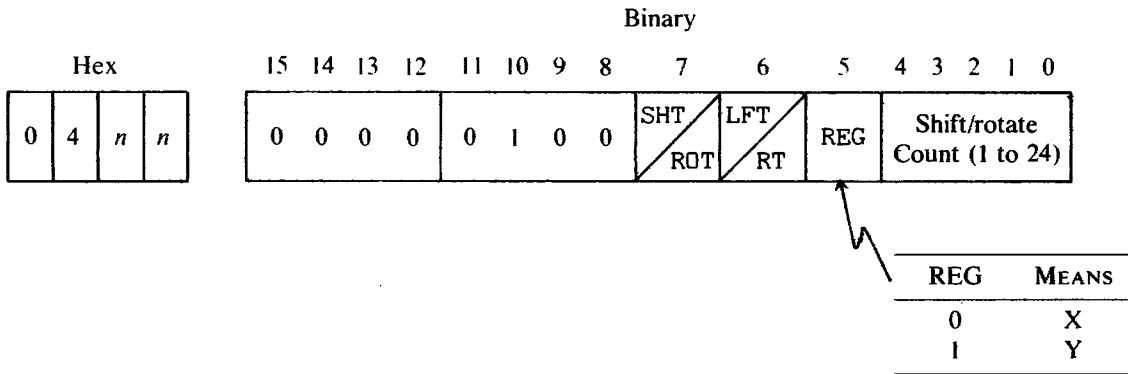
Semantics

Let *len* and *Tcopy* be hidden registers.



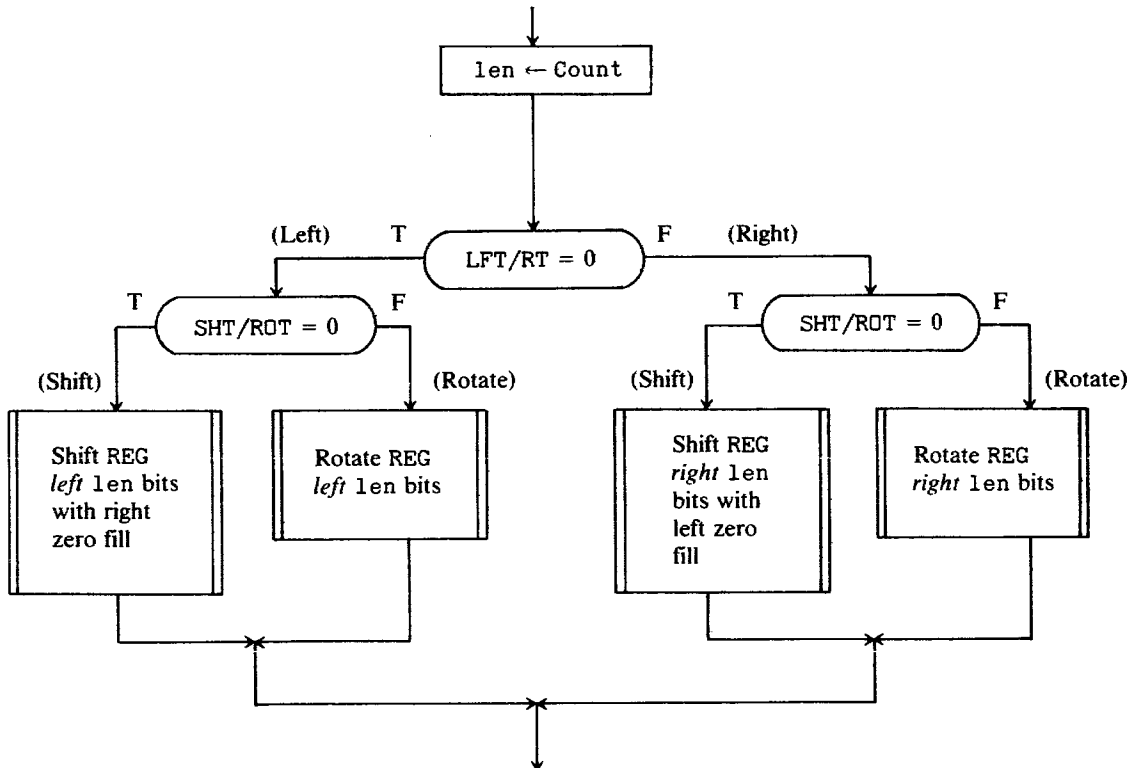
SHIFT/ROTATE X OR Y

Syntax



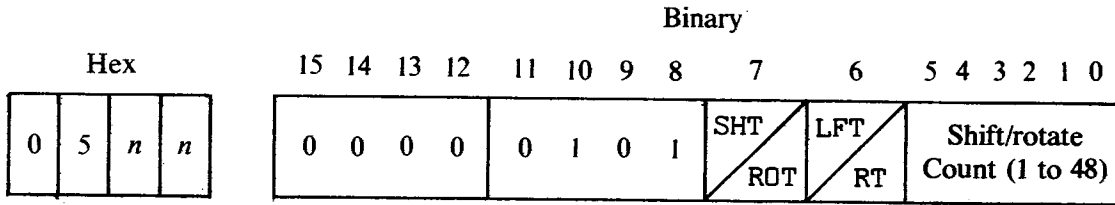
Semantics

Let len be a hidden register.



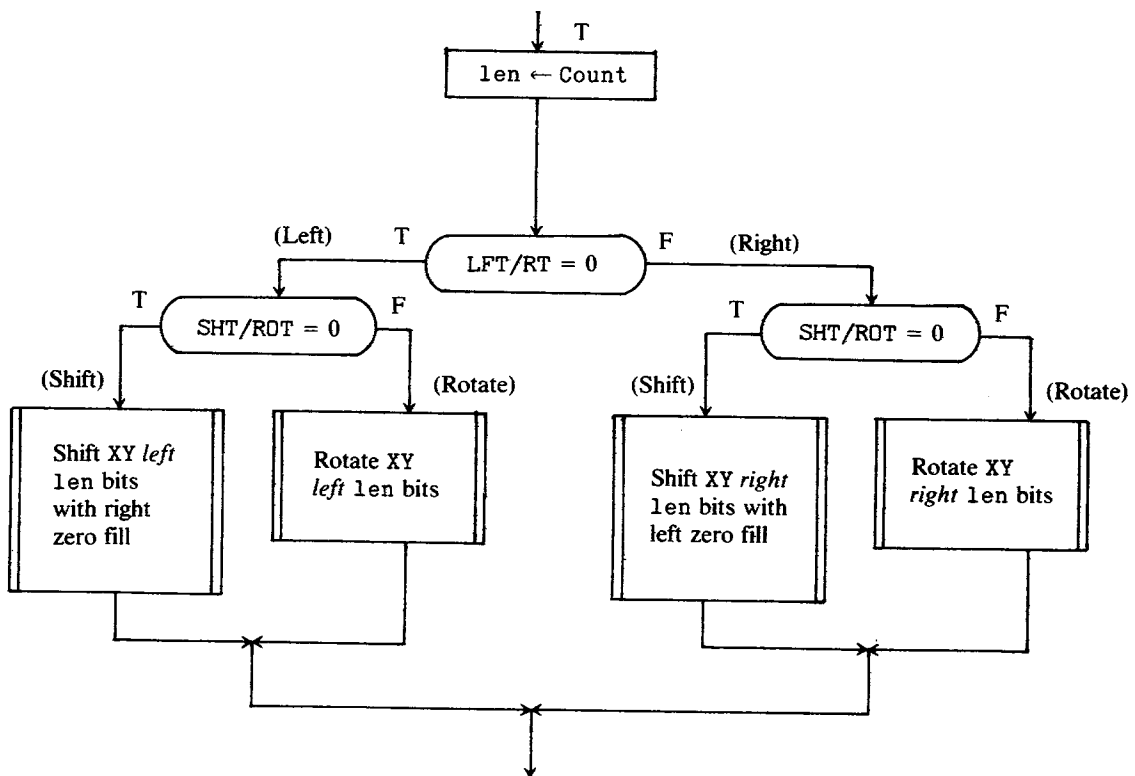
SHIFT/ROTATE XY

Syntax



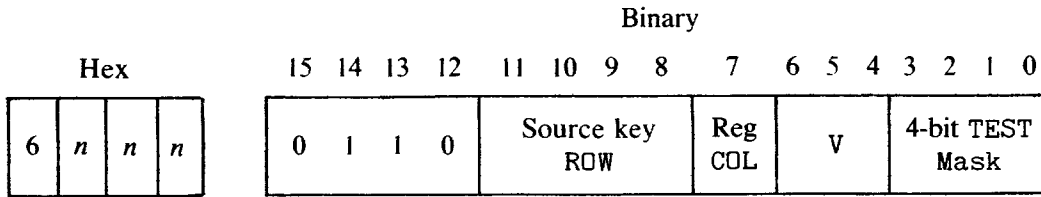
Semantics

Register XY is X cat Y (48 bits). Let len be a hidden register.



SKIP WHEN

Syntax



The MIL assembler maps the

$\left\{ \begin{array}{l} \text{ALL [CLEAR]} \\ \text{ANY} \\ \text{EQL} \end{array} \right\} \text{ [FALSE]}$

specification of the MIL SKIP instruction into a value of the variant V of the SKIP WHEN microinstruction as follows

MIL SPEC.	V
ANY	0
ALL	1
EQL	2
ALL CLEAR	3
ANY FALSE	4
ALL FALSE	5
EQL FALSE	6
ALL CLEAR FALSE	7

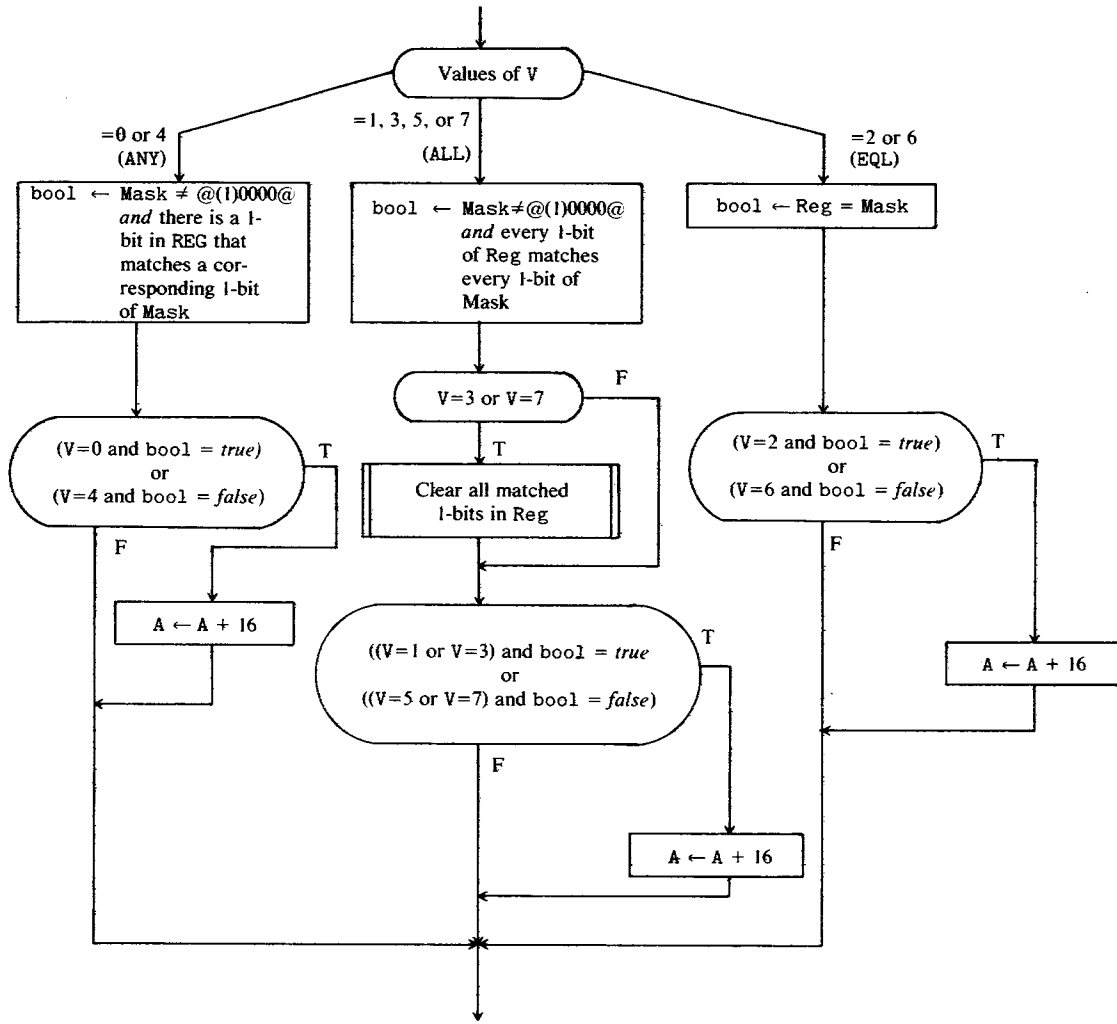
Registers

ROW	COL 0	1
0	TA	FU
1	TB	FT
2	TC	FLC
3	TD	FLD
4	TE	FLE
5	TF	FLF
6	CA	BICN ^a
7	CB	FLCN ^a
8	LA	TOPM ^a
9	LB	—
10	LC	—
11	LD	—
12	LE	XYCN ^a
13	LF	XYST ^a
14	CC	INCN ^a
15	CD	

^a May not be specified with V-values of 3 and 7.

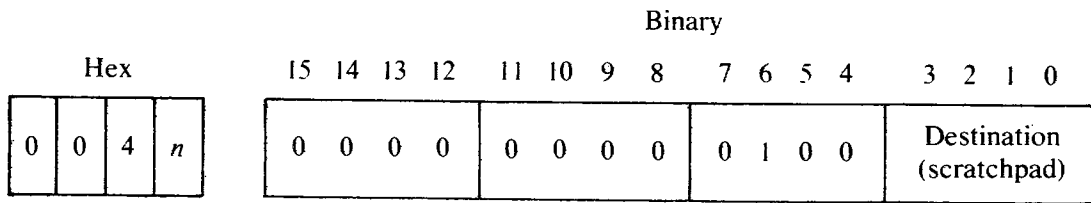
Semantics

Let bool be a hidden Boolean (1-bit) register.



STORE F INTO DOUBLE WORD

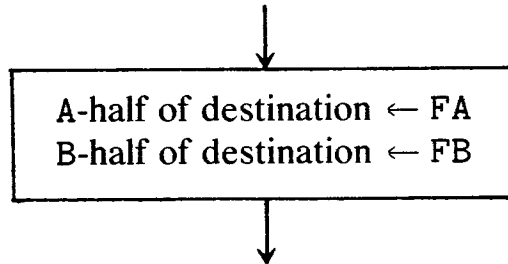
Syntax



Destination	MEANS
0	S0 (48 bits)
1	S1
⋮	⋮
15	S15

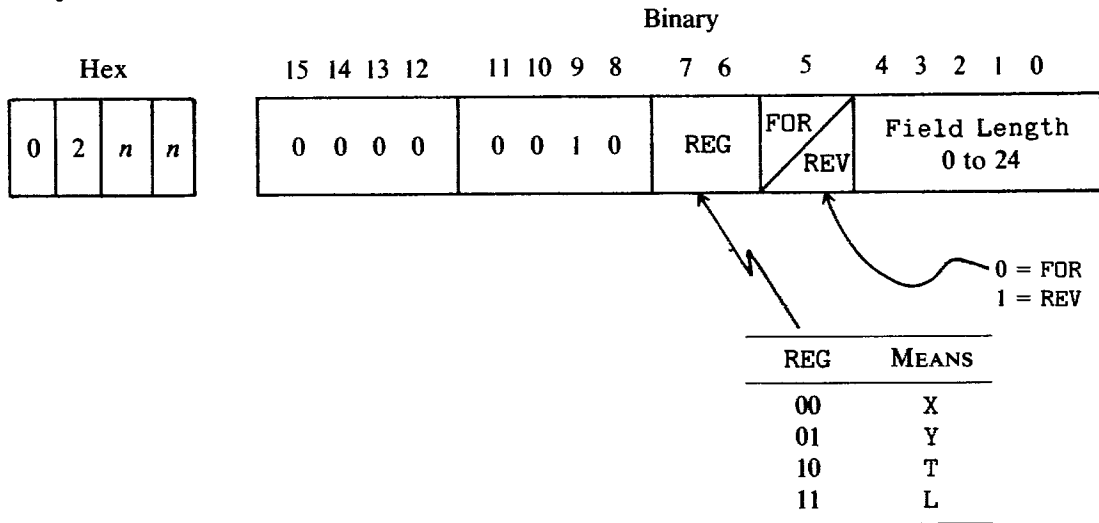
STORE F INTO DOUBLE WORD *continued*

Semantics



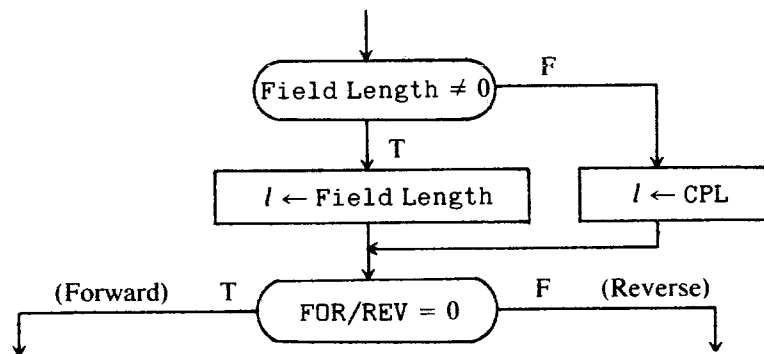
SWAP MEMORY

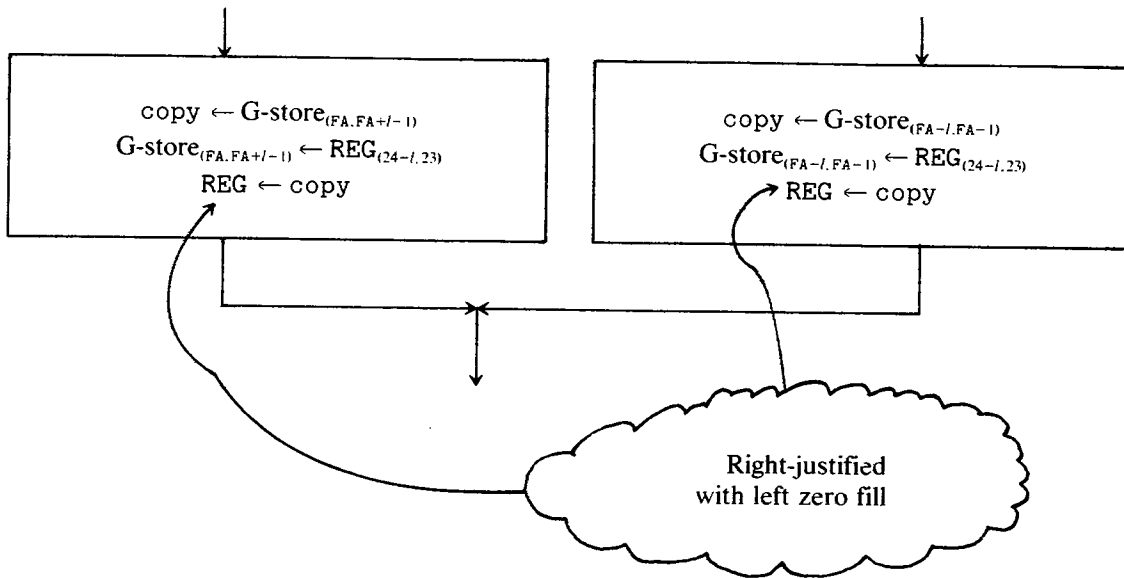
Syntax



Semantics

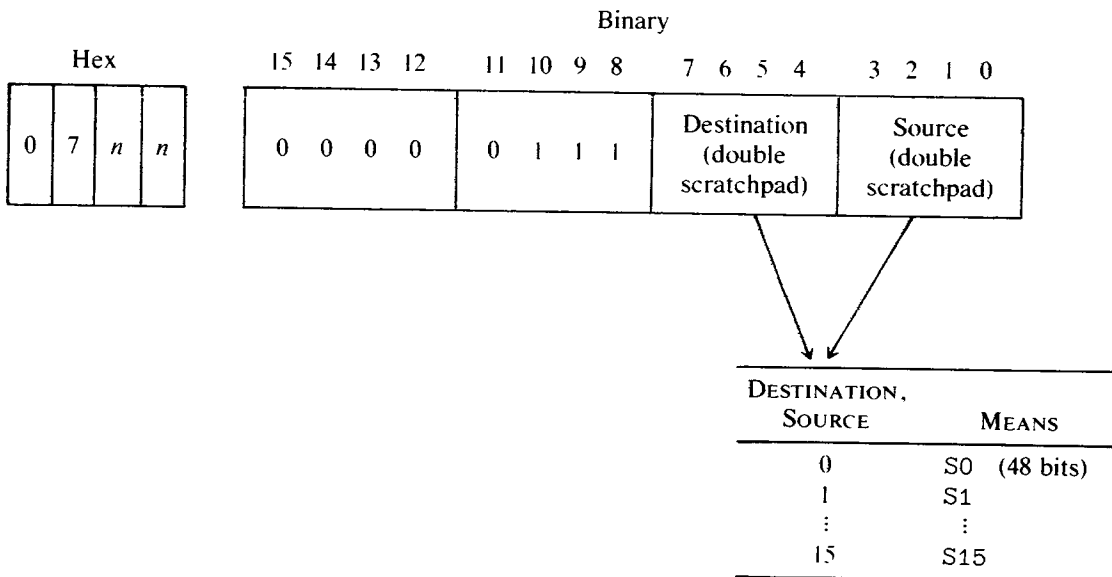
Let copy be a hidden register the same length as that specified in Field Length. Let *l* be a hidden register.





XCH DOUBLEPAD WORD WITH F

Syntax



Semantics

Let h be a 48-bit hidden register.

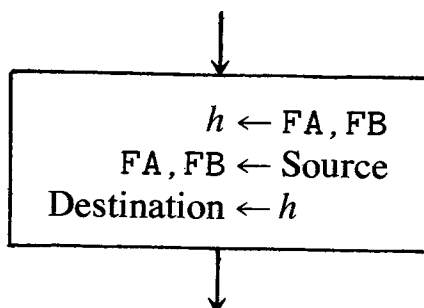


TABLE 1 B1726 Microinstructions—an Abridged Summary

					Binary															
Hex					15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<i>Register Move</i>	1	<i>n</i>	<i>n</i>	<i>n</i>	0	0	0	1	Source reg ROW	Source COL	Dest COL		Destination reg ROW							
<i>Scratchpad Move</i>	2	<i>n</i>	<i>n</i>	<i>n</i>	0	0	1	0	Source/Dest ROW	Source/Dest COL	TO FR	A B	Scratchpad number							
<i>Manipulate 4-bit</i>	3	<i>n</i>	<i>n</i>	<i>n</i>	0	0	1	1	REG ROW	REG COL	V^a				Literal					
<i>Bit Test False</i>	4	<i>n</i>	<i>n</i>	<i>n</i>	0	1	0	0	REC ROW	REG COL	Bit index		S	Relative displacement						
<i>Bit Test True</i>	5	<i>n</i>	<i>n</i>	<i>n</i>	0	1	0	1	REG ROW	REG COL	Bit Index		S	Relative displacement						
<i>Skip When</i>	6	<i>n</i>	<i>n</i>	<i>n</i>	0	1	1	0	Source reg ROW	REG COL	V^a				4-bit Test Mask					
<i>Read/Write</i>	7	<i>n</i>	<i>n</i>	<i>n</i>	0	1	1	1	R W	V^a	REG	FOR REV	Length							
<i>Move 8-bit literal</i>	8	<i>n</i>	<i>n</i>	<i>n</i>	1	0	0	0	REG ROW	Literal										
<i>Move 24-bit literal</i>	9	<i>n</i>	<i>n</i>	<i>n</i>	1	0	0	1	REG ROW (col 2 only)	Literal (first 8 bits)										
	<i>n</i>	<i>n</i>	<i>n</i>	<i>n</i>	Literal (last 16 bits)															
<i>Shift/Rotate T</i>	A	<i>n</i>	<i>n</i>	<i>n</i>	1	0	1	0	Dest. reg ROW	Dest reg COL	SHT ROT	Shift/rotate count								
<i>Extract from T</i>	B	<i>n</i>	<i>n</i>	<i>n</i>	1	0	1	1	R (right edge index)	S (sink)	W (field width)									
<i>Branch Relative Forward</i>	C	<i>n</i>	<i>n</i>	<i>n</i>	1	1	0	0	Relative displacement magnitude											
<i>Branch Relative Backward</i>	D	<i>n</i>	<i>n</i>	<i>n</i>	1	1	0	1	Relative displacement magnitude											
<i>Call Relative Forward</i>	E	<i>n</i>	<i>n</i>	<i>n</i>	1	1	1	0	Relative displacement magnitude											
<i>Call Relative Backward</i>	F	<i>n</i>	<i>n</i>	<i>n</i>	1	1	1	1	Relative displacement magnitude											

<i>Swap memory</i>	0	2	<i>n</i>	<i>n</i>	0 0 0 0	0 0 1 0	REG		FOR REV	Field Length				
<i>Clear registers</i>	0	3	<i>n</i>	<i>n</i>	0 0 0 0	0 0 1 1	L	T	Y	X	FA	FL	FU	CP
<i>Shift/Rotate X or Y</i>	0	4	<i>n</i>	<i>n</i>	0 0 0 0	0 1 0 0	SHT ROT	LFT RT	REG	Shift/rotate Count				
<i>Shift/Rotate XY</i>	0	5	<i>n</i>	<i>n</i>	0 0 0 0	0 1 0 1	SHT ROT	LFT RT	Shift/rotate count					
<i>Count FA and FL</i>	0	6	<i>n</i>	<i>n</i>	0 0 0 0	0 1 1 0	V^a			Literal				
<i>XCH (exchange)</i>	0	7	<i>n</i>	<i>n</i>	0 0 0 0	0 1 1 1	Destination (scratchpad)				Source (scratchpad)			
<i>Scratchpad Relate</i>	0	8	<i>n</i>	<i>n</i>	0 0 0 0	1 0 0 0	—			+ -	Scratchpad number <i>n</i>			
<i>Monitor</i>	0	9	<i>n</i>	<i>n</i>	0 0 0 0	1 0 0 1	Literal							
<i>Bias</i>	0	0	3	<i>n</i>	0 0 0 0	0 0 0 0	0	0	1	1	V^a		TEST	
<i>Store F</i>	0	0	4	<i>n</i>	0 0 0 0	0 0 0 0	0	1	0	0	Destination (scratchpad)			
<i>Load F</i>	0	0	5	<i>n</i>	0 0 0 0	0 0 0 0	0	1	0	1	Source (scratchpad)			
<i>Set CYF</i>	0	0	6	<i>n</i>	0 0 0 0	0 0 0 0	0	1	1	0	V^a			
<i>Read/Write (H-store)</i>	0	0	7	<i>n</i>	0 0 0 0	0 0 0 0	0	1	1	1	0	0	0	R W
<i>Halt</i>	0	0	0	1	0 0 0 0	0 0 0 0	0	0	0	0	0	0	0	1
<i>Overlay</i>	0	0	0	2	0 0 0 0	0 0 0 0	0	0	0	0	0	0	1	0
<i>Normalize X</i>	0	0	0	3	0 0 0 0	0 0 0 0	0	0	0	0	0	0	1	1
<i>Bind</i>	0	0	0	4	0 0 0 0	0 0 0 0	0	0	0	0	0	1	0	0
<i>No-Op</i>	0	0	0	0	0 0 0 0	0 0 0 0	0	0	0	0	0	0	0	0

^a For explanation of variant field V, see Table 2.

TABLE 2 Explanation of Microinstruction Variants V

MANIPULATE 4-BIT (3nnn) Variants		EXTRACT (8 nnn) Variants	
BITS 6-4	CONDITIONS	BITS 6-5	CONDITIONS
000	SET	00	X REG
001	AND	01	Y REG
010	OR	10	T REG
011	EOR	11	L REG
100	INC		
101	INC/TEST		
110	DEC		
111	DEC/TEST		
SKIP WHEN (6nnn) SKIP Test Variants		COUNT FA AND FL (06nn) Variants	
BITS 6-4	CONDITIONS	BITS 7-5	CONDITIONS
000	ANY SKIP	000	NOP
001	ALL SKIP	001	FA ↑
010	EQU SKIP	010	FL ↑
011	ALL CLR SKIP	011	FA ↑ FL ↓
100	NOT ANY SKIP	100	FA ↓ FL ↑
101	NOT ALL SKIP	101	FA ↓
110	NOT EQU SKIP	110	FL ↓
111	NOT ALL CLR SKIP	111	FA ↓ FL ↓

READ/WRITE MEMORY
(7nnn) Variants

BITS 7-6	CONDITIONS
00	X REG
01	Y REG
10	T REG
11	L REG

BITS 10-8	CONDITIONS
000	NOP
001	FA ↑
010	FL ↑
011	FA ↑ FL ↓
100	FA ↓ FL ↑
101	FA ↓
110	FL ↓
111	FA ↓ FL ↓

SWAP MEMORY
(02nn) Variants

BITS 7-6	CONDITIONS
00	X REG
01	Y REG
10	T REG
11	L REG

BIAS (003n) Variants

BITS 3-1	CONDITIONS
000	FU
001	24 OR FL
010	24 OR SFL
011	24 OR FL OR SFL
100	NOP
101	24 OR CPL OR FL
110	NOP
111	24 OR CPL OR FL OR SFL

Appendix C

A user's guide to McMIL and SMACK

SMACK is a macro-based system designed to translate statements written in a superset of MIL (the Micro Implementation Language for the Burroughs B-1700) into standard MIL for subsequent processing by the MIL assembler. The superset of MIL, hereafter referred to as McMIL, includes statements for the operating-system interface, debugging, and documentation. Utility subroutines are included with and activated by a user's McMIL program to provide the interface and debugging services. SMACK gives the casual user the impression of translating a McMIL source program into a microprogram for the B1700. The McMIL architecture is slightly different from that imposed by the B1700 operating system (MCP). This architectural change involves calculating a restart address so that logical flow of control in a microcode routine is not disturbed by calls upon the operating system. Also included are storage mapping statements to help keep track of data areas within the data region (BR-LR, base to limit register) assigned by the MCP.

The requirements of the SMACK system are few. One half scratchpad (24 bits) must be assigned to the SMACK system by giving it the name `BASE.OF.INTERPRETER`. This register is used to calculate return addresses, and holds the absolute address of the first instruction of the microcode routine. The SMACK utility subroutines must be placed ahead of any user code (except `DEFINES` and `MACROS`) for proper address calculation of the restart address. In addition, the SMACK routines use a data region for the MCP communication message that lies in the BR-LR region. This area is reserved by the use of McMIL storage allocation statements so that conflicts with other data areas can be avoided.

The SMACK processor is easily activated with two control cards, and upon terminating will automatically link to the MIL assembler with no operator intervention. SMACK handles disk maintenance of relevant files, purging old files and creating the new microcode (interpreter) file.

1 McMIL STATEMENT SYNTAX

All McMIL statements begin with an equal sign (=) in column 1 and are called E-statements. Each McMIL statement must fit on one card. One or more blanks separate the items on a McMIL statement, and commas can be freely used as an alternate for a blank. In any place where an arithmetic expression is indicated (OPEN options, sizes) no blanks are allowed within the expression.

2 McMIL STATEMENTS FOR THE OPERATION OF THE SMACK PROCESSOR

One register (a 24-bit scratchpad) must be devoted to the housekeeping chores that the SMACK subroutines perform. This register is given the name `BASE.OF.INTERPRETER`. For example, to assign scratchpad S13B for this purpose, the following line would be coded

```
DEFINE BASE.OF.INTERPRETER = S13B #
```

Alternatively the “=DF” McMIL statement could be used as follows

```
=DF BASE.OF.INTERPRETER=S13B
```

See Section 4, statement type 8, in this Appendix.

After the `BASE.OF.INTERPRETER` register is assigned, the SMACK subroutines must be included. This is done with the following McMIL statement.

1. =INITIALIZE

This statement initiates the standard section (see `=SECTION`) named “SMACK”; it also allocates areas in the BR-LR data region for system communication. These areas are used by SMACK, so user microcode should not rely on the contents of these areas. If the “=BSS” statement is used to mark off storage, the user should have no problems.

To end the McMIL expansion phase and invoke the MIL assembler, use the statement

2. =TERMINATE name

The name appearing on the `TERMINATE` statement will be the name of the assembled microcode when stored on disk after successful assembly.

3 McMIL STATEMENTS FOR DOCUMENTATION

3. =SECTION name

The appearance of a SECTION statement will separate the listing of the code following from that which appeared prior to the SECTION statement. Also, the name of the section is printed on the left side of the listing.

The source code generated by the expansion of E-statements is usually suppressed, but may be turned on for a complete section by the appearance of the following statement

4. =MLIST name1 name2 ...

The names of sections appearing on an MLIST statement will have all generated statements (from the McMIL expansion) listed along with the rest of the output. The “=MLIST” statement must appear before any section to which it refers.

5. =NOLIST name1 name2 ...

The NOLIST statement will cause suppression of the listing of any section whose name appears on that statement. The NOLIST statement, like the MLIST statement, must appear before any section named on the statement.

The listing of the SMACK subroutines is usually suppressed. However, it may be reselected by inserting the section name “SMACK” on an MLIST statement. In this case an MLIST card must appear *before* the INITIAL statement.

There can be any number of “=MLIST” or “=NOLIST” statements in a McMIL program.

4 McMIL STATEMENTS USED TO FORMAT THE LISTING

6. =STARS or =STARS *n*

This statement produces lines of asterisks on the listing, helping to visually separate lines of code from each other. The *n* indicates that $2n+1$ lines of asterisks are placed on the listing.

7. ==any-text=more-text=

This statement, containing four equal signs, will cause a comment line with the following format to appear:

```
%*      *any-text      *more-text      *
```


8. =DFsymbol=text

This line results in a MIL DEFINE with the following form:

```
DEFINE symbol      =text      #
```

giving a neater listing than DEFINE with arbitrary formats.

Similar in appearance is the following McMIL statement:

9. =DVsymbol=arithmetic-expression

All four operators (+ - * /) and parentheses can be used to evaluate an integer expression. Operator precedence is as usual. The maximum value of any operation is $2^{15}-1$. There can be no spaces (blanks) in any arithmetic expression. The effect of this statement is to issue a MIL define to set the value of symbol to the proper numerical result.

Example

```
=DV TRACE.FILE=8*5+3*3
```

produces

```
DEFINE TRACE.FILE  =49      #
```

5 McMIL STATEMENTS FOR STORAGE ALLOCATION AND ADDRESSING

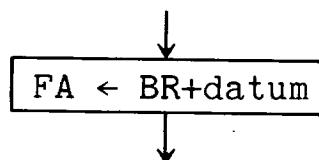
10. =data-name BSS size-in-bits

This statement defines the data-name as a displacement from BR and assigns the length in bits (evaluated as an arithmetic expression) as the length of the datum. The next BSS will assign the displacement from the next available bit position. Note that this statement does not actually reserve space but “marks off” the existing space between BR and LR. This statement is completely equivalent to the MIL statement

```
DECLARE DATA.NAME BIT(SIZE.IN.BITS);
```

11. =ADD OF datum

This statement generates code to cause FA to point to the absolute address (not base-relative) of the indicated datum. The semantics is



This statement does not change the value of any other register. The datum should have been declared with the DECLARE or “=BSS” statement.

6 McMIL STATEMENTS FOR DEBUGGING

It is possible to take a snapshot of any single register (except TAS) at any time. The output will be directed to the line printer. The state of all registers will be restored upon completion of the snap function, and execution will continue as if there were no debugging statements in the microprogram.

The debug option should *not* be used if relocation of the interpreter is possible. This could happen in a multiprogramming environment. The addresses in the stack corresponding to return addresses in the micro code could be incorrect in such a case.

The debug option can be set for any section of code (see =SECTION) with the following statement

12. =DEBUG name1 name2 . . .

There may be any number of names on the DEBUG statement. Each such section will have SNAP statements (see below) expanded; if the DEBUG statement has not selected a section, then any SNAP statements in that section are ignored.

13. =SNAP $\left\{ \begin{array}{l} \text{24-bit literal} \\ \text{register} \end{array} \right\}$ AS number $\left\{ \begin{array}{l} \text{OCTAL} \\ \text{HEX} \\ \text{EBCDIC} \end{array} \right\}$ CHARS

The literal or the value of the register will be placed in the trace buffer. Control returns to the statement immediately following the SNAP statement.

14. =PRINT SNAP

This statement causes the trace buffer to be dumped to the line printer. The buffer will automatically print if more than 110 characters are in it. The PRINT SNAP also returns control to the next statement and restores all the registers.

If the debug feature is desired, then at least one DEBUG statement must precede the =INITIALIZE statement to cause the inclusion of the proper subroutines.

6.1 Optional suppression of invoked DEBUG feature

When using the debug feature of SMACK, the leftmost console switch will suppress the printing of the trace buffer if set to the up position. In the down position, the trace buffer is printed normally.

7 McMIL STATEMENTS FOR MCP INTERFACE

ALL of the following statements restore the scratchpads, *but destroy all other registers.*

15. =DUMPFIL

The execution of this statement will cause all of the data between BR and LR to be placed on a disk file (DUMPFIL/number). Execution will continue. The dumpfile can be analyzed and printed by using the console command "PM number" (where the number is the same as was printed on the console printer at the time of the dump).

16. =STOP

Generates code such that execution of the microprogram is terminated, memory is released, all files are closed, and MCP regains control.

17. =IF NO INTERRUPTS GO TO label

When this McMIL statement is executed the hardware checks the state of all physical devices (card reader, disk, etc.) to determine if any drastic change in the state of the machine is indicated; if not, control will transfer to label. If there is a real need to return to MCP (temporarily), control will pass on to the next statement. Any housekeeping that the programmer desires to do before control is released is performed (care is necessary to preserve the system information in the "L" register). A =SERVICE INTERRUPTS (see statement 18) is then executed.

18. =SERVICE INTERRUPTS

This McMIL statement releases control of the processor to MCP. Upon return to the user, all scratchpads are restored. Control is then passed to the next sequential statement.

19. =CHECK INTERRUPTS

This statement is a combination of the above two statements; no further processing of interrupts is necessary.

Examples of interrupt handling are

```
INSTRUCTION.FETCH
=CHECK INTERRUPTS
  MOVE NEXT.INSTR.POINTER TO FA
  READ 16 BITS TO T INC FA
  MOVE FA TO NEXT.INSTR.POINTER
  : % DECODE INSTRUCTION
```

and

```
INSTRUCTION.FETCH
=IF NO INTERRUPTS GO TO +OK
  MOVE COUNT.OF.ESCAPES TO X
  MOVE 1 TO Y
  MOVE SUM TO COUNT.OF.ESCAPES
=SERVICE.INTERRUPTS
.OK MOVE NEXT.INSTR.POINTER TO FA
  READ 16 BITS TO T INC FA
  MOVE FA TO NEXT.INSTR.POINTER
  : % DECODE INSTRUCTION
```

8 McMIL STATEMENTS FOR MCP COMMUNICATION

20. =OPEN file-id WITH (option)

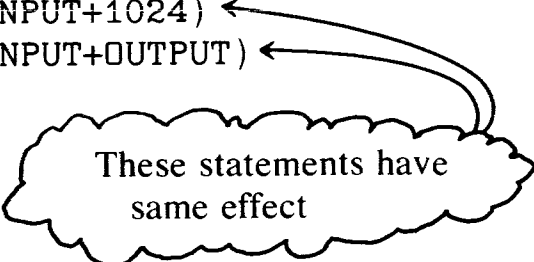
Possible open options are

NEWFILE	or	512	Force new disk file
INPUT	or	2048	Allow reading to occur
OUTPUT	or	1024	Allow writes to file

OPEN options are specified either alone or with simple addition (e.g., INPUT+OUTPUT). Parentheses are optional in this arithmetic expression, but no spaces are allowed.

Examples of OPEN and CLOSE statements are

```
=OPEN PRINT.FILE WITH (INPUT+1024)
=OPEN PRINT.FILE WITH (INPUT+OUTPUT)
```



These statements have same effect

21. =CLOSE file-id WITH (option)

Possible CLOSE options are

REEL	or	2048	Close a reel of multireel file
RELEASE	or	1024	Return buffers to memory
PURGE	or	512	Remove the file from disk directory
REMOVE	or	256	Substitute this file for another in directory
CRUNCH	or	128	Release unused disk space
NREWIND	or	64	Do not rewind
CODE	or	32	Set file type as executable
LOCK	or	16	Enter file in disk directory
conditional		8	Do not abort if already closed

An example of a CLOSE statement is

```
=CLOSE DISK.OUT.FILE WITH (REMOVE)
```

A rewind request on a file is done by a CLOSE with no options followed by an OPEN. Only one CLOSE option may be specified at a time.

The "general MCP request" statement is the most powerful statement, because of the many forms and variants permitted. Items that occur within square brackets are optional and, if included, modify the effect of the request. Items within braces indicate that one and only one is to be chosen.

$$22. = \left\{ \begin{array}{l} \text{BUFFER request USING buffer-name} \\ \text{request size } \left\{ \begin{array}{l} \text{BITS} \\ \text{BYTES} \end{array} \right\} \text{ CORE address} \end{array} \right\}$$

[FILE file-id [KEY register]]

$$\left[\left\{ \begin{array}{l} \text{OPT} \\ \text{ON} \end{array} \right\} \left\{ \begin{array}{l} \text{IN register} \\ \text{option-expression} \end{array} \right\} \left[\left\{ \begin{array}{l} \text{GO TO} \\ \text{GOTO} \end{array} \right\} \text{label} \right] \right]$$

The BUFFER option will use the size of a datum as defined with a DECLARE or as a BSS pseudo-op; otherwise the size must be explicitly specified with the BITS or BYTES option. The address specified after CORE is a base-relative displacement (i.e., absolute address-BR). If a file (i/o unit) is implied by the operation, then the FILE option must be specified, and file-id indicates (as a number) which file (logical i/o device) is to be selected (see LOADER description in Appendix D). If the file is a random-access disk file, the KEY variant must appear, and the key or record number specified in a register. (Note that random-access files start at record number 1.) Some i/o operations imply certain

options (e.g., spacing the printer) which may be specified by the appearance of the keyword OPT (or ON). If the option resides in a register, select the IN variant. GO TO will transfer to the label *only if* the proper variant has been requested (e.g., EOF).

The keyword OPT is equivalent to ON, and the keywords GO TO are equivalent to GOTO.

Some examples of this general request statement are

```
=BUFFER INPUT USING CARD.AREA FILE CARD.READER
                        % WITHOUT END FILE CHECKING
=BUFFER OUTPUT USING PRINT.LINE FILE PRINT OPT DOUBLE
=READ 180 BYTES CORE DISK.IN.AREA FILE RANDOM.DISK KEY S2B
=BUFFER ZIP USING ZIP.CONSTANT
=SEEK 0 BYTES CORE 0 FILE RANDOM.DISK KEY S2B
=OUTPUT 0 BITS CORE PRINT.AREA FILE PRINTER OPT EJECT
```

A complete list of the possible requests follows.

INPUT or READ		Transfer data from storage to file
OUTPUT or WRITE		Transfer data to file from storage
SEEK	(needs KEY)	Position random-access file
DISPLAY	(no file)	Write message to console teletypewriter
ACCEPT	(no file)	Get message from operator
ZIP	(no file)	Send control card to operating system

Possible options are

SINGLE	(printer only)	or 14	
DOUBLE	(printer only)	or 15	
EJECT	(printer only)	or 1	
EOF	(detect end file)	or 2048	Allows GO TO variant
PARITY	(detect parity)	or 1024	Allows GO TO variant
NO-ADVANCE	(overprint)	or 0	

Multiple options are specified by simple addition, for example,

EJECT+PARITY

No blanks are allowed.

Figure C.1 shows a sample deck structure and corresponding mapping of base-limit memory area. To access (the first bit of) the area named SECOND, set FA to BR+SECOND or use “=ADD OF SECOND”. The SMACK storage region will be quite large if the “=DEBUG” option is used. The question mark indicates an MCP system control card which contains a (1-2-3) overpunch in column 1.

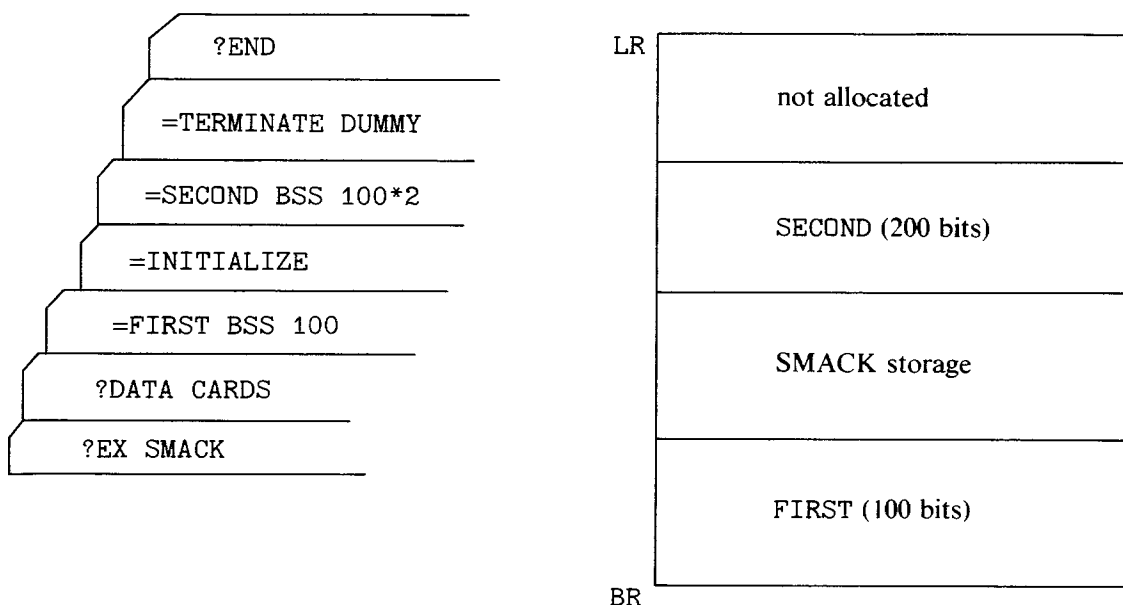


Figure C.1.

Caution Incorrect code may be generated by the general request statement 22 if any of the parameters are specified in the volatile registers X, Y, T, L or any of the outputs from the ALU (SUM, DIFF, . . .). If a situation occurs where the user really needs to specify one of the parameters as one of these registers, it is up to the user to verify that the resulting code sequence will not clobber one of the registers used as a parameter.

The most general construct will always generate code as follows.

```
=request size BYTES CORE location FILE file-id
  KEY key OPT IN option
```

generates

```
MOVE key TO TAS
MOVE location TO TAS
MOVE size TO T
SHIFT T LEFT BY 3 BITS TO TAS
MOVE request TO X
MOVE file TO Y
MOVE option TO L
MOVE 3 TO TAS ←
CALL XXFN ←
```

number of parameters on TAS

call proper subroutine

TABLE C.1 Summary of McMIL E-Statements

1	=INITIALIZE
2	=TERMINATE
3	=SECTION
4	=MLIST ^a
5	=NOLIST ^a
6	=STARS
7	==any-text=text=
8	=DF
9	=DV
10	=data.name BSS
11	=ADD OF
12	=DEBUG ^a
13	=SNAP
14	=PRINT SNAP
15	=DUMPFIL ^b
16	=STOP ^b
17	=IF NO INTERRUPTS GO TO ^b
18	=SERVICE INTERRUPTS ^b
19	=CHECK INTERRUPTS ^b
20	=OPEN ^b
21	=CLOSE ^b
22	=⟨general i/o request⟩ ^b

^a Nonexecutable. Place *before* =INITIALIZE.

^b Destroys contents of X, Y, T, L, FA, FB, CA, CB, CP, and the stack.

Appendix D

Loader primer

In most programming languages, the data and possibly input-output units are specified along with the code that operates on the data and reads and writes to the i/o devices. For example, in FORTRAN the DATA statement will preset variables, the DIMENSION statement will reserve storage, and the files (i/o units) may be declared in the program header statements. The COMPASS assembler for the CDC-6400 has many statement types (pseudo-ops) for defining data, octal constants and file tables. In one language, however, a clear distinction is made between code and data and input-output units. This language is COBOL, where data appear only in a DATA section, code only in a PROCEDURE section, and input-output assignments only in the INPUT-OUTPUT section.

In the B1700 operating environment this separation is carried to an extreme in that the data definition and code (microcode) specification are handled by completely different compilers. The code section is handled by the SMACK-MIL system, and the specification of data and input-output assignments by the LOADER. This total separation may be cumbersome and sometimes difficult to utilize in a student environment. The justification for this separation is the fact that microcode can be shared by many different users. Each user would be represented by a different data area.

All of this implies that to run a program on the B1700 one must

1. Specify the data area, the interpreter, and the input-output assignments (with the LOADER)
2. Specify the proper operations on the data area (with SMACK-MIL)
3. Request the MCP to execute the codefile-microcode package using a suitable command sequence in a job-control language (JCL).

1 THE JCL COMMAND SEQUENCES

The above operations will be performed by the following skeleton JCL stream. In the example given in Table D.1 the microcode will be named SENATOR and the data (codefile) will be named SENATE/BILL245.

TABLE D.1 Skeleton JCL Stream

JCL	COMMENTS
?CO SENATE/BILL245 WITH LOADER TO LIBRARY ^a ?DATA CARDS	Call loader to analyze data specifications contained on card deck.
:	
Interpreter, scratchpad settings, data, and file assignments	} Loader card deck.
:	
?END ^a ?EX MCMIL ^a ?DATA CARDS ^a	Call SMACK to expand, then assemble the MIL cards that follow.
:	McMIL program
=TERMINATE SENATOR	Specify the name of microcode on disk storage.
?END ^a ?EX SENATE/BILL245 INTERPRETER=SENATOR ^{a,b}	Specify the microcode for proper execution.

^a The first column of a control card marked, "?", is a 1-2-3 multipunch.

^b This command is given to the operating system after the loader and MIL assembly have been successfully completed.

In this description the term microcode has been consistently used to describe the operations on data. However, the operating system refers to the microcode as the INTERPRETER, and refers to the data as a program or codefile. Unfortunately, keywords like INTERPRETER are not truly descriptive, but they are in agreement with Burroughs conventions.

2 SYNTAX OF THE LOADER CARD DECK

```

<card deck for loader> ::=
  <program parameter specifications>;
  <scratchpad settings>;
  <file descriptions>
  DATA<data specifications>;
  FINI

```

Card boundaries are ignored, except that a decimal constant or data name (such as an input-output unit name) may not be split by a card boundary. A semicolon (;) separates the different sections (e.g., each input-output specification is terminated by a semicolon).

```

<file descriptions> ::= <empty> | <file><file descriptions>
<file> ::= FILE <input-output-attributes>;

```


will assign the bit pattern

0000	0000	0000	0100	0000	0000	to SOA	
0101	0000	0101	0000	1110	1110	to SOB	
0000	0101	0011	1001	0111	0111	to S1A	
□	□	□	□	□	□		
(0	1	2	3	4	5	6	7)

and zero to all other scratchpad registers.

2.3 File descriptions

Each file is assigned a number according to its position relative to the other FILE specifications. The first FILE is given the number 0, the next 1, and so on. The microcode refers to the input-output units by this number. The DATA segment is automatically brought into memory at the location pointed to by the base register (BR).

The FILE specification supplies information about the actual data path for program communication with input-output devices. Most of the available options under the operating system can be set with the LOADER. They are

$\langle \text{input-output attributes} \rangle ::= \langle \text{empty} \rangle \mid \langle \text{i-o-attribute} \rangle \langle \text{input-output attributes} \rangle$

$\langle \text{i-o-attribute} \rangle ::=$

PACK= $\langle \text{name} \rangle$	disk pack name for disk file
NAME= $\langle \text{name} \rangle$	name for file (internal and external)
SUBNAME= $\langle \text{name} \rangle$	second name of file
HARDWARE= $\langle \text{decimal integer} \rangle$	special hardware type
BUFFERS= $\langle \text{decimal integer} \rangle$	number of buffers
LOCK	save this file after termination of job
DEFAULT	use record length and blocking factor of file as it appears in disk directory
READER	hardware type is 80-column card reader
DISK	hardware type is any available disk
TAPE	hardware type is any magnetic tape
PRINTER	hardware type is line printer
QUEUE	queue file for interprogram communication

NO . LABEL		suppress page eject on open and close
REC = <decimal integer>		record size in bits (fixed length)
REC . P . BLK = <decimal integer>		number of records in block (fixed length)
ADVERB = <decimal integer>		options for open when implied by i/o on a closed file
RANDOM		disk-file random-access flag (set BUFFERS to 1)
AREAS = <decimal integer>		maximum number of disk areas (default is 40)
BLK . P . AREA = <decimal integer>		sets size of one area to number of blocks
REC . P . AREA = <decimal integer>		sets size of one area to number of records (record size and records per block already set)
BINARY		on 80-column card reader use binary reads/writes

Note: specify only one hardware type (don't ask for a DISK QUEUE type file, etc.). Do not ask for zero buffers.

Example

```
FILE NAME=INF DISK REC=640 DEFAULT;
```

—a disk file with record size of 80 characters.

```
FILE NAME=CARDS READER BINARY ;
```

—card reader allowing binary reads.

```
FILE NAME=PR PRINTER NO . LABEL REC=960 REC . P . BLK=1 ;
```

—printer file with 120 characters per line and no page ejects at open/close time.

Note. The microcode would refer to the disk file as file 0, the card reader as file 1 and the printer file as file 2. The operating system would refer to these files as INF, CARDS, and PR respectively.

2.4 DATA segment

Information for the DATA segment is specified in any of the following formats.

Bit string A string enclosed in “at” signs (@) is hexadecimal. If another base is desired, specify the desired bit grouping within

parentheses [e.g., (1) for binary, (2) for quartal, (3) for octal, and (4) for hexadecimal]. The base may be changed at any time within a bit string. For example the following bit pattern 1010111010111 can be specified in any of the following ways:

$$\begin{aligned} @AEB(1)1@ &= @(2)223(3)5(4)7@ = @(3)53(1)10101(2)3@ \\ &= @(1)1010111010111@ \end{aligned}$$

Bit strings may be of any length, *blanks are ignored*, and a bit string may cross a card boundary.

Character string a string of characters enclosed in double quotes (") will be treated as a sequence of EBCDIC (8-bit) characters. Two consecutive double quotes will be taken as one occurrence of a double quote. *Blanks are not ignored*, but card boundaries are.

Decimal data a decimal integer (not enclosed in double quotes or "at" signs) will be converted to its 24-bit binary representation. The suffix P (precision) followed by an integer describing the bit length will set the data item to be 0 to 24 bits long. The above string @AEB(1)1@ could be specified as 10P4 14P4 11P4 1P1.

Note. The DATA segment may be of arbitrary length and the various types of specifications may be mixed freely.

3 EXAMPLE

An input deck for the LOADER is as follows.

```
?CO FRAME/WORK WITH LOADER LIBRARY
?DATA CARDS
INTERP=EXAMPLE  STATIC=5500;  ;
FILE  NAME=PRINTER  PRINTER;
FILE  NAME=CARDS    READER;
DATA  "THE END";
FINI
?END
```

Appendix E

McMIL listing for an abridged SAMOS interpreter

This appendix presents the rudimentary or skeleton version of the SAMOS Interpreter discussed in Chapters 5 and 6, together with the LOADER program, data deck, and sample output.

	Page
Part 1 Source listing of the SAMOS interpreter input to the McMIL processor	275
Part 2 MIL assembler listing of the SAMOS interpreter	285
Part 3 Source listing of the LOADER program for generating the codefile for the SAMOS interpreter	301
Part 4 Output listing of the LOADER program	301
Part 5 Listing for a simple data deck (one SAMOS program) to be executed by the SAMOS interpreter	301
Part 6 Output produced by the SAMOS interpreter when processing the data deck given in Part 5	302
Part 7 Sequence of enhancements for the SAMOS interpreter	302

1 SOURCE LISTING OF THE SAMOS INTERPRETER INPUT TO THE McMIL PROCESSOR

Each card image is preceded by a card number followed by a colon.

```

1  :?EX McMIL
2  :?DATA CARDS
3  :      DEFINE CARD.READER          =1 #
4  :      DEFINE PRINTER              =0 #
5  :      DEFINE FIFTY.SIX            =S3A #
6  :      DEFINE MESSAGE.BASE         =S4A #
7  :      DEFINE BASE.DF.INTERPRETER  =S12A #
8  :      DEFINE SAMOS.STORE.ADDR     =S13A #
9  :      DEFINE INAREA.ADDR          =S14A #
10 :      DEFINE PRINT.AREA.ADDR      =S15A #
11 :      DEFINE EA                    =S1B #
12 :      DEFINE MESSAGE.LENGTH       =S7B #
13 :?z  DEFINE LOCATION.COUNTER       =S9B #      DEFINED IN LOAD.A.PROG
14 :      DEFINE WORK.COUNTER         =S11B #
15 :      DEFINE TERMINATION.CODE     =S12B #
16 :      DEFINE LOAD.CODE             =S13B #
17 :      DEFINE FOUND.SWITCH         =S14B #
18 :      DEFINE MASTER.SWITCH        =S15B #
19 :      DEFINE FLAG                  =FLF #
20 :      DEFINE YES                   =1 #
21 :      DEFINE NOT.YET.SET          =0 #
22 :      DEFINE SIZE                  =100 #
23 :      DEFINE WORK.LIMIT           =1500 #
24 :      DEFINE OK                    =0 #      ZVALUES FOR
25 :      DEFINE EOF                   =1 #      ZLOAD.CODE
26 :      DEFINE STAR                  =2 #

```

```

27 :      DEFINE TOO BIG          =3 #
28 :      DEFINE IX3.ADDR        =28000052 # Z -5 IN SIGNED MAG.
29 :      DEFINE IX2.ADDR        =28000042 # Z -4 IN SIGNED MAG.
30 :      DEFINE IX1.ADDR        =28000032 # Z -3 IN SIGNED MAG.
31 :      DEFINE IC.ADDR         =28000022 # Z -2 IN SIGNED MAG.
32 :      DEFINE ACC.ADDR        =28000012 # Z -1 IN SIGNED MAG.
33 :      DEFINE CHAR            =CHARACTER#
34 :      :Z
35 :      DECLARE
36 :      :Z
37 :      :Z
38 :      :Z
39 :      :Z
40 :      :Z
41 :      :Z
42 :      :Z
43 :      :Z
44 :      :Z
45 :      :Z
46 :      :Z
47 :      :Z
48 :      :Z
49 :      :Z
50 :      :Z
51 :      :Z
52 :      :Z
53 :      :Z
54 :      :Z
55 :      :Z
56 :      :Z
57 :      :Z
58 :      :Z
59 :      :Z
60 :      :Z
61 :      :Z
62 :      :Z
63 :      :Z
64 :      :Z
65 :      :Z
66 :      :Z
67 :      :Z
68 :      :Z
69 :      :Z
70 :      :Z
71 :      :Z
72 :      :Z
73 :      :Z
74 :      :Z
75 :      :Z
76 :      :Z
77 :      :Z
78 :      :Z
79 :      :Z
80 :      :Z
81 :      :Z
82 :      :Z
83 :      :Z
84 :      :Z
85 :      :Z
86 :      :Z
87 :      :Z
88 :      :Z
89 :      :Z
90 :      :Z
91 :      :Z
92 :      :Z
93 :      :Z
94 :      :Z
95 :      :Z
96 :      :Z
97 :      :Z
98 :      :Z
99 :      :Z
100 :      :Z
101 :      :Z
102 :      :Z
103 :      :Z
104 :      :Z
105 :      :Z
106 :      :Z
107 :      :Z
108 :      :Z
109 :      :Z
110 :      :Z
111 :      :Z
112 :      :Z
113 :      :Z
114 :      :Z
115 :      :Z
116 :      :Z
117 :      :Z
118 :      :Z
119 :      :Z
120 :      :Z
121 :      :Z
122 :      :Z
123 :      :Z
124 :      :Z

```

DEFINE TOO BIG
 DEFINE IX3.ADDR
 DEFINE IX2.ADDR
 DEFINE IX1.ADDR
 DEFINE IC.ADDR
 DEFINE ACC.ADDR
 DEFINE CHAR
 DECLARE
 MESSAGES
 MESS.1 CHAR(18),
 MESS.2 CHAR(15),
 MESS.3 CHAR(11),
 MESS.4 CHAR(11),
 MESS.5 CHAR(10),
 MESS.6 CHAR(19),
 MESS.7 CHAR(12),
 MESS.8 CHAR(12),
 MESS.9 CHAR(19),
 MESS.10 CHAR(15),
 MESS.11 CHAR(16),
 MESS.12 CHAR(16),
 MESS.13 CHAR(24),
 MESS.14 CHAR(25),
 INAREA CHAR(80),
 PRINT.AREA CHAR(80),
 DECLARE 01 STORAGE.FOR.SAMOS,
 02 REGISTERS,
 03 IX3 BIT(88),
 03 IX2 BIT(88),
 03 IX1 BIT(88),
 03 IC BIT(88),
 03 ACC BIT(88),
 02 SAMOS.STORE(SIZE),
 03 SIGN CHAR(1),
 03 OPCODE CHAR(3),
 03 INDEXES,
 04 INDEX1 CHAR(1),
 04 INDEX2 CHAR(1),
 04 INDEX3 CHAR(1),
 03 ADDRESSSS CHAR(4),
 MACRO DEFINITIONS(GLOBAL)
 MACRO EFF.ADDR.TO.FA =
 CALL EFFECTIVE.ADDR
 IF FLF NEQ 0 THEN GO TO EA.ERROR
 MOVE EA TO T
 CALL BINARY.TO.FA #
 END MACRO
 MACRO TEN.Y.PLUS.D(LX) =
 SHIFT T LEFT BY 1 BIT TO X
 SHIFT T LEFT BY 3 BITS TO Y
 MOVE SUM TO X
 MOVE LX TO Y
 MOVE SUM TO T #
 END MACRO
 =INITIALIZE
 =SECTION SHELL
 MOVE PRINT.AREA TO PRINT.AREA.ADDR
 MOVE INAREA TO INAREA.ADDR
 MOVE SAMOS.STORE(0) TO SAMOS.STORE.ADDR
 MOVE 56 TO FIFTY.SIX
 =OUTPUT DATA.LENGTH(MESS.14) BITS CORE MESS.14 FILE PRINTER OPT EJECT
 MOVE NULL TO MASTER.SWITCH
 MOVE NULL TO FOUND.SWITCH
 =.MAIN.SHELL.LOOP
 MOVE MASTER.SWITCH TO X
 IF X NEQ 0 THEN GO TO +STOP.STEP
 CALL FIND.A.JOB.CARD ROUTINE
 GO TO FIND.A.JOB.CARD
 =.BOX4.SHELL
 MOVE FOUND.SWITCH TO X
 MOVE 1 TO Y
 IF X NEQ Y THEN GO TO -.MAIN.SHELL.LOOP
 MOVE NULL TO FOUND.SWITCH
 CALL LOAD.A.PROGRAM ROUTINE
 GO TO LOAD.A.PROGRAM
 =.BOX6.SHELL
 MOVE LOAD.CODE TO X
 MOVE OK TO Y
 IF X EQL Y THEN GO TO +BOX7
 MOVE EOF TO Y
 IF X EQL Y THEN GO TO -.MAIN.SHELL.LOOP
 MOVE STAR TO Y
 IF X EQL Y THEN
 BEGIN
 =OUTPUT DATA.LENGTH(MESS.1) BITS CORE MESS.1 FILE PRINTER OPT DOUBLE
 =OUTPUT 80 BYTES CORE INAREA FILE PRINTER OPT SINGLE
 MOVE 1 TO FOUND.SWITCH
 END ELSE


```

125 : BEGIN
126 :
127 : ZPROGRAM TOO BIG
127 :=OUTPUT DATA.LENGTH(MESS.2) BITS CORE MESS.2 FILE PRINTER OPT DOUBLE
128 : END
129 : GO TO -MAIN.SHELL.LOOP
130 :
130 :.BOX7
131 : CALL INTERP.PROGRAM ROUTINE
132 : GO TO INTERPRET.PROGRAM
133 :.BOX8.SHELL
134 : MOVE TERMINATION.CODE TO X
135 : MOVE NOT.YET.SET TO Y
136 : IF X EQL Y THEN
137 : : Z SHOULD NOT HAPPEN
137 : : Z ERROR IN INTERPRETER
138 : BEGIN
138 : MOVE 1 TO MASTER.SWITCH
139 :=OUTPUT DATA.LENGTH(MESS.13) BITS CORE MESS.13 FILE PRINTER OPT DOUBLE
140 : END
141 : GO TO -MAIN.SHELL.LOOP
142 : Z END OF MAIN SHELL LOOP
143 :.STOP.STEP
144 :=OUTPUT DATA.LENGTH(MESS.8) BITS CORE MESS.8 FILE PRINTER OPT SINGLE
145 :.STOP
146 :=SECTION FIND.A.JOB.
147 :.FIND.A.JOB.CARD ZROUTINE BEGINS HERE. SEE FIG.5-4
148 :.FIND.LOOP
149 : MOVE FOUND.SWITCH TO X
150 : IF X NEQ 0 THEN GO TO -BOX4.SHELL
151 : MOVE MASTER.SWITCH TO X
152 : IF X NEQ 0 THEN GO TO -BOX4.SHELL
153 :=BUFFER READ USING INAREA FILE CARD.READER ON EOF GO TO +BOX4.FIND
154 : MOVE BR TO FA
155 : ADD INAREA.ADDR TO FA
156 : READ 8 BITS TO X ZCHECK FOR STAR
157 : MOVE "*" TO Y
158 : IF X EQL Y THEN
159 : : BEGIN Z FOUND A STAR
160 :=OUTPUT 80 BYTES CORE INAREA FILE PRINTER OPT DOUBLE
161 : MOVE 1 TO FOUND.SWITCH
162 : END
163 : GO TO -FIND.LOOP
164 :.BOX4.FIND
165 : MOVE 1 TO MASTER.SWITCH
166 : GO TO -FIND.LOOP
167 : Z END OF FIND.A.JOB.CARD
168 :=SECTION LOAD
169 :.LOAD.A.PROGRAM ZROUTINE BEGINS HERE. SEE FIG.5-5
170 : LOCAL.DEFINES
171 : DEFINE LOCATION.COUNTER =S9B#
172 : DEFINE UNDEFINED =4 # Z VALUE FOR LOAD.CODE
173 : Z
174 : MOVE -1 TO LOCATION.COUNTER
175 :
176 : MOVE DATA.LENGTH(STORAGE.FOR.SAMOS) TO FL ZCLEAR EMULATED SAMOS STORAGE.
177 : MOVE STORAGE.FOR.SAMOS TO S1A Z INITIALIZE FL,FA,X
178 : MOVE BR TO FA
179 : ADD S1A TO FA
180 : MOVE "000" TO X Z CHAR. ZEROES IN X
181 : MOVE "+0" TO Y
182 :.CLEAR.LOOP
183 : IF FL NEQ 0 THEN
184 : : BEGIN
185 : : WRITE 16 BITS FROM Y INC FA AND DEC FL
186 : : WRITE 24 BITS FROM X INC FA AND DEC FL
187 : : WRITE 24 BITS FROM X INC FA AND DEC FL
188 : : WRITE 24 BITS FROM X INC FA AND DEC FL
189 : : GO TO -CLEAR.LOOP
190 : : END
191 : : MOVE UNDEFINED TO LOAD.CODE ZBOX 3.1
192 :.LOAD.LOOP ZBOX 3.2
193 : : MOVE LOAD.CODE TO X
194 : : MOVE UNDEFINED TO Y
195 : : IF X NEQ Y THEN GO TO -BOX6.SHELL
196 :=BUFFER READ USING INAREA FILE CARD.READER ON EOF GO TO +BOX3.5
197 : : MOVE 78*8 TO FL Z CHECK FOR A BLANK CARD
198 : : Z INITIALIZE LOOP TO SEARCH
199 : : Z FIRST 78 CHARS OF CARD,IN
200 : : MOVE BR TO FA Z GROUPS OF THREE
201 : : ADD INAREA.ADDR TO FA
202 : : MOVE FA TO S1A
203 : : MOVE " " TO Y Z SAVE ABS ADDR OF INAREA
204 : : MOVE " " TO Y Z 3 BLANKS TO Y
205 :.CHECK.LOOP
206 : : IF FL EQL 0 THEN GO TO +LAST2.CHECK
207 : : READ 24 BITS TO X INC FA AND DEC FL
208 : : IF X EQL Y THEN GO TO -CHECK.LOOP
209 : : GO TO +BOX3.8
210 :.LAST2.CHECK
211 : : COUNT FA DOWN BY 8
212 : : READ 24 BITS TO X Z 3 CHARS OF THE CARD
213 : : IF X EQL Y THEN
214 : : : BEGIN
215 : : : MOVE OK TO LOAD.CODE Z BOX3.7 THE CARD IS BLANK
216 : : : GO TO -LOAD.LOOP
217 : : : END
218 :.BOX3.8
219 : : MOVE S1A TO FA Z RESTORE ABS ADDR OF INAREA
220 : : READ 8 BITS TO X
221 : : MOVE "*" TO Y
222 : : IF X EQL Y THEN
223 : : : BEGIN
224 : : : MOVE STAR TO LOAD.CODE
225 : : : GO TO -LOAD.LOOP
225 : : : END

```

```

226 :% BOX3.10
227 : MOVE LOCATION.COUNTER TO FL Z INCREMENT LOCATION.COUNTER
228 : COUNT FL UP BY 1
229 : MOVE FL TO LOCATION.COUNTER
230 : MOVE SIZE TO SOB Z SIZE NOW IN SFL
231 : IF FL GEQ SFL THEN
232 : BEGIN
233 : MOVE TOO BIG TO LOAD.CODE
234 : GO TO -LOAD.LOOP
235 : END
236 :
237 : MOVE FL TO T Z PUT ARGUMENT FOR NEXT CALL
238 : CALL BINARY.TO.FA Z IN T TO GET ABS ADDR OF INSTR.
239 : XCH S1 F S1 Z COPY FIRST 11 COLUMNS OF CARD
240 : MOVE 88 TO FL Z INTO SAMOS STORAGE WORD USING
241 : CALL COPY Z COPY SUBROUTINE. RECALL SOURCE
242 : Z ADDRESS WAS SAVED IN S1A
243 : Z BUFFER WRITE USING INAREA FILE PRINTER OPT SINGLE
244 : GO TO -LOAD.LOOP
245 : Z AND FINALLY,
246 : .BOX3.5
247 : MOVE EOF TO LOAD.CODE
248 : MOVE 1 TO MASTER.SWITCH
249 : GO TO -LOAD.LOOP
250 : Z END OF LOAD.A. PROGRAM ROUTINE
251 : Z SECTION INTERPRET
252 : INTERPRET.PROGRAM Z SEE FIG. 5-7
253 : Z THE IC IS ALREADY INITIALIZED
254 : Z TO "0000" BY VIRTUE OF THE
255 : Z CLEAR LOOP IN THE
256 : Z LOAD.A. PROGRAM MODULE
257 : MOVE NULL TO WORK.COUNTER
258 : MOVE NOT.YET.SET TO TERMINATION.CODE
259 : Z BOX2
260 : .INTERP.LOOP
261 : Z CHECK INTERRUPTS
262 : MOVE TERMINATION.CODE TO X
263 : MOVE NOT.YET.SET TO Y
264 : IF X NEQ Y THEN GO TO -BOX8.SHELL Z RETURN TO THE SHELL
265 : MOVE IC.ADDR TO T Z FETCH NEXT INSTRUCTION.
266 : CALL ADDRESS.TO.BINARY Z LEAVES FLAG IN Y AND BINARY
267 : IF Y EQL 0 THEN Z ADDRESS OF INSTR. IN T
268 : BEGIN
269 : MOVE YES TO TERMINATION.CODE
270 : Z OUTPUT DATA.LENGTH(MESS.7) BITS CORE MESS.7 FILE PRINTER OPT SINGLE
271 : GO TO -INTERP.LOOP
272 : END
273 : CALL BINARY.TO.FA Z POINT TO NEXT INSTRUCTION
274 : COUNT FA UP BY 8 Z AT THE OP CODE
275 : READ 24 BITS TO X INC FA Z GET OP CODE
276 : MOVE FA TO TAS Z SAVE POINTER TO REMAINDER OF
277 : MOVE X TO TAS Z INSTRUCTION AND SAVE OPCODE
278 : Z
279 : MOVE IC.ADDR TO T Z INCREMENT THE IC
280 : CALL BINARY.TO.FA
281 : MOVE FA TO S1A Z SAVE ABS ADDR OF IC
282 : CALL VALIDATE.DECIMAL Z PACKS IC INTO S5A AND S5B
283 : MOVE NULL TO S6A
284 : MOVE 1 TO S6B
285 : CALL ADD.10.COMPL
286 : MOVE S1A TO FA Z RESTORE ABS ADDR OF IC
287 : CALL UNPACK.AND.WRITE Z STORES INCREMENTED IC
288 : MOVE TAS TO X Z RESTORE OP CODE TO X
289 : MOVE TAS TO FA Z RESTORE FA PTR TO REM OF INSTR
290 : Z
291 : Z DECODE SECTION
292 : MOVE "LDA" TO Y
293 : IF X EQL Y THEN GO TO LDA.. Z
294 :
295 : MOVE "STO" TO Y
296 : IF X EQL Y THEN GO TO STO.. Z
297 :
298 : MOVE "ADD" TO Y
299 : IF X EQL Y THEN GO TO ADD.. Z
300 :
301 : MOVE "BRU" TO Y
302 : IF X EQL Y THEN GO TO BRU.. Z
303 :
304 : MOVE "BMI" TO Y
305 : IF X EQL Y THEN GO TO BMI.. Z
306 :
307 : MOVE "RWD" TO Y
308 : IF X EQL Y THEN GO TO RWD.. Z
309 :
310 : MOVE "WWD" TO Y
311 : IF X EQL Y THEN GO TO WWD.. Z
312 :
313 : MOVE "HLT" TO Y
314 : IF X EQL Y THEN GO TO HLT.. Z BAD OP CODE
315 :
316 : MOVE YES TO TERMINATION.CODE
317 : Z OUTPUT DATA.LENGTH(MESS.4) BITS CORE MESS.4 FILE PRINTER OPT SINGLE
318 : GO TO -INTERP.LOOP
319 : Z END OF DECODE SECTION
320 : Z BEGIN WORK.COUNTER UPDATE(BOXES 7-9 OF FIG. 5-7)
321 : .INC.WORK.COUNTER
322 : MOVE 24 TO CP Z INCREMENT WORK.COUNTER
323 : MOVE WORK.COUNTER TO Y

```

```

324 : MOVE 1 TO X
325 : MOVE SUM TO WORK.COUNTER
326 : MOVE WORK.LIMIT TO X Z IF BORROW,
327 : IF X LSS Y THEN Z WORK.LIMIT EXCEEDED
328 : BEGIN Z OVERWORKED
329 : MOVE YES TO TERMINATION.CODE
330 : =OUTPUT DATA.LENGTH(MESS.5) BITS CORE MESS.5 FILE PRINTER OPT SINGLE
331 : END
332 : GO TO -INTERP.LOOP
333 :Z END OF INTERPRET.PROGRAM
334 :Z SAMOS OPERATOR ROUTINES BEGIN HERE
335 :Z
336 :ADD.. Z ADD OPERATOR BEGINS HERE
337 : EFF.ADDR.TO.FA ZMACRO CALL
338 : CALL VALIDATE.DECIMAL ZLEAVES FLAG IN CB(O)
339 : IF CB(O) THEN Z 1 IF NOGOOD, 0 IF OK
340 : BEGIN ZAND (VALID) OPERAND IN S5
341 : MOVE YES TO TERMINATION.CODE ZNON-NUMERIC OPERAND
342 : =OUTPUT DATA.LENGTH(MESS.9) BITS CORE MESS.9 FILE PRINTER OPT SINGLE
343 : GO TO -INTERP.LOOP
344 : END
345 : XCH S5 F S5 Z MOVES VALIDATED OPERAND FROM
346 : XCH S6 F S6 Z S5 TO S6
347 : MOVE ACC.ADDR TO T
348 : CALL BINARY.TO.FA Z ABS ADDR OF ACC IN FA
349 : MOVE FA TO S2A Z SAVE ADDR TEMPORARILY
350 : CALL VALIDATE.DECIMAL Z LEAVES FLAG IN CB(O) AND
351 : IF CB(O) THEN Z(VALID OPERAND IN S5
352 : BEGIN Z NON-NUMERIC ACC
353 : MOVE YES TO TERMINATION.CODE
354 : =OUTPUT DATA.LENGTH(MESS.10) BITS CORE MESS.10 FILE PRINTER OPT SINGLE
355 : GO TO -INTERP.LOOP
356 : END
357 : CALL ADD.10.COMPL ZADDS TWO PACKED DECIMAL VALUES
358 : Z IN S5 AND S6 AND LEAVES RESULT
359 : Z IN S5.
360 : MOVE S2A TO FA ZNOW RESTORE ABS ADDR OF ACC
361 : CALL UNPACK.AND.WRITE Z STORES CHAR. REPRESENT. IN ACC
362 : GO TO -INC.WORK.COUNTER
363 :Z END OF ADD OPERATOR
364 :Z
365 :LDA.. Z LDA OPERATOR BEGINS HERE
366 : EFF.ADDR.TO.FA ZGET SOURCE PTR (MACRO CALL)
367 : MOVE FA TO S1A ZSAVE IT IN S1A
368 : MOVE ACC.ADDR TO T Z GET SINK POINTER
369 : CALL BINARY.TO.FA
370 : XCH S1 F S1 Z
371 : MOVE 88 TO FL ZSET LENGTH OF VALUE TO BE COPYD
372 : CALL COPY Z ARGS READY FOR COPY CALL
373 : GO TO -INC.WORK.COUNTER
374 :Z
375 :Z
376 :STO.. Z STO OPERATOR BEGINS HERE
377 : EFF.ADDR.TO.FA ZGET SINK PTR (MACRO CALL)
378 : MOVE FA TO S1A ZSAVE IT IN S1A
379 : MOVE 88 TO FL ZSET LENGTH OF VALUE TO BE COPYD
380 : MOVE ACC.ADDR TO T ZGET SOURCE PTR
381 : CALL BINARY.TO.FA Z ARGS NOW READY FOR COPY CALL
382 : CALL COPY
383 : GO TO -INC.WORK.COUNTER
384 :Z
385 :Z
386 :BRU.. Z BRU OPERATOR BEGINS HERE
387 : CALL EFFECTIVE.ADDR Z LEAVES EFFECTIVE ADDRESS AS A
388 : IF FLF NEQ 0 THEN GO TO EA.ERROR Z BINARY VALUE IN EA AND FLAG IN
389 : CALL BINARY.TO.DECIMAL ZFLF
390 : Z LEAVES PACKED DECIMAL EQUIV.
391 : Z OF EA IN S5.
392 : MOVE IC.ADDR TO T Z GET ABS ADDR OF IC
393 : CALL BINARY.TO.FA
394 : CALL UNPACK.AND.WRITE Z ASSIGN NEW VALUE TO IC
395 : GO TO -INC.WORK.COUNTER
396 :Z
397 :Z
398 :BMI.. Z BMI OPERATOR BEGINS HERE
399 : MOVE FA TO S2A ZSAVE POINTER TO REST OF INSTR
400 : MOVE ACC.ADDR TO T Z CHECK SIGN OF ACC
401 : CALL BINARY.TO.FA
402 : READ 8 BITS TO X
403 : MOVE "+*" TO Y
404 : IF X EQL Y THEN GO TO -INC.WORK.COUNTER Z NO BRANCHING (A NO OP)
405 : MOVE S2A TO FA Z RESTORE PTR TO REST OF INSTR
406 : GO TO BRU..
407 :Z
408 :Z
409 :RWD.. Z RWD OPERATOR BEGINS HERE
410 : EFF.ADDR.TO.FA Z GET ABS PTR TO SINK AND PUT IN
411 : MOVE FA TO S1A Z S1A
412 : =BUFFER READ USING INAREA FILE CARD.READER ON EOF GO TO +EOF
413 : MOVE BR TO FA Z GET ABS PTR TO
414 : ADD INAREA.ADDR TO FA Z INAREA
415 : MOVE 88 TO FL Z SET LENGTH OF INPUT
416 : CALL COPY Z COMPLETES RWD ACTION
417 : GO TO -INC.WORK.COUNTER
418 :Z
419 :Z
420 :.EOF
421 : MOVE YES TO TERMINATION.CODE ZAND (VALID) OPERAND IN S5
422 : =OUTPUT DATA.LENGTH(MESS.8) BITS CORE MESS.8 FILE PRINTER OPT SINGLE

```

```

422 : GO TO -INTERP.LOOP
423 :Z
424 :Z
425 :WWD.. Z WWD OPERATOR BEGINS HERE
426 : EFF ADDR TO FA Z SAVE ABS PTR TO SOURCE IN
427 : MOVE FA TO S1A Z S1A
428 : MOVE BR TO FA Z GET ABS PTR TO SINK,
429 : ADD PRINT.AREA ADDR TO FA Z IE, PRINT.AREA
430 : XCH S1 F S1 Z SWAP ARGS FOR CALL ON COPY
431 : MOVE 88 TO FL Z SET LENGTH OF OUTPUT
432 : CALL COPY
433 : =OUTPUT 88 BITS CORE PRINT.AREA FILE PRINTER OPT SINGLE
434 : GO TO -INC.WORK.COUNTER
435 :Z
436 :Z
437 :HLT.. Z NORMAL HALT
438 : MOVE YES TO TERMINATION.CODE
439 : =OUTPUT DATA.LENGTH(MESS.3) BITS CORE MESS.3 FILE PRINTER OPT SINGLE
440 : GO TO -INTERP.LOOP
441 :Z
442 :EA.ERROR Z EFFECTIVE ADDRESS ERR MESSAGES
443 : IF FLF(3) THEN
444 : BEGIN Z TOO MANY INDICES
445 : =OUTPUT DATA.LENGTH(MESS.12) BITS CORE MESS.12 FILE PRINTER OPT SINGLE
446 : END
447 : IF FLF(2) THEN
448 : BEGIN Z ADDR NOT DECIMAL
449 : =OUTPUT DATA.LENGTH(MESS.11) BITS CORE MESS.11 FILE PRINTER OPT SINGLE
450 : END
451 : IF FLF(1) THEN
452 : BEGIN Z BAD OPERAND ADDR
453 : =OUTPUT DATA.LENGTH(MESS.6) BITS CORE MESS.6 FILE PRINTER OPT SINGLE
454 : END
455 : MOVE YES TO TERMINATION.CODE
456 : SET FLF TO 0 Z RESET THE EA FLAG
457 : GO TO -INTERP.LOOP
458 : =SECTION SUBROUTINES
459 :Z
460 :Z UNPACK.AND.WRITE PACKED DECIMAL VALUE IN S5 IS
461 : Z UNPACKED AND WRITTEN TO G-STORE
462 : Z AT ADDRESS GIVEN BY FA.
463 : Z USES T,L,X,Y AS LOCAL STORAGE
464 : UNPACK.AND.WRITE
465 : MOVE "+0" TO T Z UNPACK AND WRITE
466 : MOVE S5A TO L Z FIRST 2 BYTES
467 : IF L(0) THEN MOVE "-0" TO T Z IN CASE OF NEGATIVE NUMBER
468 : MOVE LC TO TF
469 : WRITE 16 BITS FROM T INC FA,
470 : Z UNPACK AND WRITE NEXT 3 BYTES
471 : MOVE "000" TO T Z PUT 2FOFOFO2 IN T.
472 : MOVE LD TO TB
473 : MOVE LE TO TD
474 : MOVE LF TO TF
475 : WRITE 24 BITS FROM T INC FA
476 : Z UNPACK AND WRITE NEXT 3 BYTES
477 : MOVE S5B TO L
478 : MOVE LA TO TB
479 : MOVE LB TO TD
480 : MOVE LC TO TF
481 : WRITE 24 BITS FROM T INC FA
482 : Z UNPACK AND WRITE LAST 3 BYTES
483 : MOVE LD TO TB
484 : MOVE LE TO TD
485 : MOVE LF TO TF
486 : WRITE 24 BITS FROM T
487 : Z
488 : EXIT
489 :Z
490 :Z
491 :Z BINARY.TO.DECIMAL Z MAXIMUM VALUE TO BE CONVERTED IS 9999,
492 : Z I.E., LSS 2 TO THE 14TH POWER
493 : Z THE INPUT ARG IS IN EA (S18)
494 : Z THE OUTPUT RESULT IS LEFT IN
495 : Z S5 AS A PACKED DECIMAL VALUE
496 : BINARY.TO.DECIMAL
497 : MOVE EA TO T
498 : CLEAR X
499 : Z
500 : MOVE 2(1)001110002 TO CP Z SET UP FOR DECIMAL ARITH.
501 : MOVE 14 TO FL
502 : .LOOP
503 : IF FL NEQ 0 THEN
504 : BEGIN
505 : MOVE X TO Y Z MOVE TWICE THE VALUE OF X
506 : MOVE SUM TO Y Z TO Y
507 : EXTRACT 1 BIT FROM T(10) TO X Z GET NEXT DIGIT
508 : SHIFT T LEFT BY 1 BIT
509 : MOVE SUM TO X Z THE NEW SUM NOW IN X
510 : COUNT FL DOWN BY 1
511 : GO TO -LOOP
512 : END
513 : MOVE NULL TO S5A Z STORED IN S5
514 : MOVE X TO S5B
515 : Z
516 : EXIT
517 :Z
518 :Z
519 :Z COPY Z COPIES A SOURCE MESSAGE, WHOSE ADDRESS IS

```

```

520 :           Z IN FA, TO A SINK AREA, WHOSE ADDRESS IS
521 :           Z IN S1A.
522 :           Z LENGTH OF SOURCE IS IN FL.
523 :           Z USES CPL AND T.
524 :           Z LEAVES A COPY OF INITIAL VALUE OF FL IN
525 :           Z SFL.
526 : :COPY
527 :   MOVE FL TO SOB
528 : :.LOOP
529 :   IF FL EQL 0 THEN EXIT
530 :   BIAS BY F
531 :   READ TO T INC FA AND DEC FL      Z CPL GETS MIN(24,FL)
532 :   XCH S1 F S1                      Z READ CPL BITS
533 :   WRITE FROM T INC FA              Z WRITE CPL BITS
534 :   XCH S1 F S1
535 :   GO TO -.LOOP
536 : :Z END OF COPY
537 : :Z
538 : :Z
539 : :Z
540 : :VALIDATE.DECIMAL      ZROUTINE BEGINS HERE.  SEE FIG.6-13.
541 : :                       ZVALIDATES A SAMOS WORD POINTED TO
542 : :                       ZBY THE CONTENTS OF FA AS A DECIMAL
543 : :                       ZINTEGER AND PACKS A 4-BIT DECIMAL
544 : :                       ZREPRESENTATION IN S5. IF NOT VALID,
545 : :                       ZCB(O) IS SET TO 1, ELSE IT IS SET TO 0.
546 : :                       Z THIS ROUTINE USES X,Y,T,L, AND CP.
547 :   BEGIN
548 :   LOCAL.DEFINES
549 :   DEFINE FLAG              =CB(O) #
550 :   MACRO CHECK.F(TK) =
551 :     IF TK NEQ 2FA THEN EXIT #
552 :   CLEAR L                  ZBOX1
553 :   SET FLAG
554 :   READ 16 BITS TO T INC FA  Z TO NOGOOD
555 :                               ZGET FIRST 2 BYTES IN TC TH
556 :                               ZBOXES
557 :   EXTRACT 8 BITS FROM T(8) TO X      ZAND 6
558 :   MOVE "+" TO Y
559 :   IF X NEQ Y THEN
560 :     BEGIN                  Z TRY '-'
561 :     MOVE "-" TO Y
562 :     IF X NEQ Y THEN EXIT
563 :     MOVE 2(1)10002 TO LA
564 :   END
565 :   CHECK.F(TE)              ZBOX7
566 :   MOVE TF TO LC
567 :   READ 24 BITS TO T INC FA      ZBOX8
568 :   CHECK.F(TA)              ZCHECK AND PACK INTO L
569 :   CHECK.F(TC)
570 :   CHECK.F(TE)
571 :   MOVE TB TO LD
572 :   MOVE TD TO LE
573 :   MOVE TF TO LF
574 :   MOVE L TO SSA
575 :   READ 24 BITS TO T INC FA      ZBOX9
576 :   CHECK.F(TA)              ZCHECK AND PACK INTO L
577 :   CHECK.F(TC)
578 :   CHECK.F(TE)
579 :   MOVE TB TO LA
580 :   MOVE TD TO LB
581 :   MOVE TF TO LC
582 :   READ 24 BITS TO T
583 :   CHECK.F(TA)              ZCHECK AND PACK INTO L
584 :   CHECK.F(TC)
585 :   CHECK.F(TE)
586 :   MOVE TB TO LD
587 :   MOVE TD TO LE
588 :   MOVE TF TO LF
589 :   MOVE L TO S5B
590 :   MOVE 2(1)001110002 TO CP      ZBOX 10.1
591 :   MOVE S5A TO X              Z 10.2
592 :   CLEAR Y                    Z 10.3
593 :   MOVE SUM TO Y
594 :   IF X NEQ Y THEN EXIT
595 :   MOVE S5B TO X              ZBOX 10.4
596 :   CLEAR Y                    ZBOX 10.5
597 :   MOVE SUM TO Y
598 :   IF X NEQ Y THEN EXIT
599 :   RESET FLAG
600 :   EXIT
601 :   END Z OF VALIDATE.DECIMAL ROUTINE
602 : :BINARY.TO.FA          ZROUTINE BEGINS HERE  SEE FIG. 6-15.
603 : :                       ZINPUT VALUE,S, IS IN T REGISTER
604 : :                       ZOUTPUT RESULT IS IN FA
605 : :                       ZUSES X,Y,CB(O),AND L AS LOCAL STORAGE
606 :   BEGIN
607 :   LOCAL.DEFINES
608 :   DEFINE NEG.SIGN          =CB(O)#
609 :   DEFINE NEW.S             =L#
610 :   IF T(O) THEN            ZBOX1
611 :     BEGIN                  ZSAVE SIGN OF S
612 :     SET NEG.SIGN           ZAND REPLACE S
613 :                               ZBY ABS VALUE OF S

```

```

618 :      RESET T(0)                                ZIS THIS INSTRUCTION NECESSARY?
619 :      END ELSE
620 :      BEGIN
621 :      RESET NEG.SIGN
622 :      END
623 :      MOVE 24 TO CP                                ZSET UP FOR ARITHMETIC
624 :      SHIFT T LEFT BY 3 BITS TO X                Z8 TIMES S TO X(SIGN BIT LOST)
625 :      SHIFT T LEFT BY 4 BITS TO Y                Z16 TIMES S TO Y
626 :      MOVE SUM TO X                                Z24 TIMES S TO X
627 :      SHIFT T LEFT BY 6 BITS TO Y                Z64 TIMES S TO Y
628 :      MOVE SUM TO NEW.S                            Z88 TIMES S IN NEW.S (L)
629 :      MOVE BR TO X
630 :      MOVE SAMOS.STORE(0) TO Y                    ZBR + SAMOS.STORE IN X
631 :      MOVE SUM TO X
632 :      MOVE NEW.S TO Y                                ZBOX5
633 :
634 :      IF NEG.SIGN THEN
635 :      BEGIN
636 :      MOVE DIFF TO FA
637 :      END ELSE
638 :      BEGIN
639 :      MOVE SUM TO FA
640 :      END
641 :      EXIT
642 :      END Z BINARY.TO.FA ROUTINE
643 :
644 : Z ADDRESS.TO.BINARY ZROUTINE BEGINS HERE. SEE FIG 6-17.
645 : Z INPUT VALUE,S, IS IN T REGISTER
646 : Z OUTPUT RESULTS BINARY ADDRESS IN T
647 : Z FLAG IN Y
648 : Z 0 IF INVALID
649 : Z 1 IF VALID
650 : Z USES X,Y,L,FA AS LOCAL STORAGE
651 :
652 :      CALL BINARY.TO.FA                            ZWITH ARGUMENT IN T ZBOX1
653 :
654 :      COUNT FA UP BY 24                            ZCOUNT FA UP BY ZBOX2
655 :      COUNT FA UP BY 24                            Z6 BYTES
656 :      READ 16 BITS TO L INC FA                      ZDSUB3 NOW IN LF
657 :      MOVE LF TO T
658 :
659 :      READ 24 BITS TO L                            ZDSUB2 IN LB ZBOX3
660 :
661 :
662 :
663 :      MOVE 24 TO CP
664 :      TEN.T.PLUS.O(LB)                             Z10 TIMES T + DSUB2 TO T
665 :      TEN.T.PLUS.O(LD)                             Z10 TIMES T + DSUB1 TO T
666 :      TEN.T.PLUS.O(LF)                             Z10 TIMES T + DSUB0 TO T ZBOX5
667 :
668 :      MOVE T TO X                                    ZBOX6
669 :
670 :      MOVE SIZE TO Y                                ZSIZE IS A GLOBAL CONSTANT
671 :      IF X LSS Y THEN
672 :      BEGIN
673 :      MOVE 1 TO Y                                    ZVALID SAMOS ADDRESS
674 :      END ELSE
675 :      BEGIN
676 :      MOVE 0 TO Y                                    ZINVALID SAMOS ADDRESS
677 :      END
678 :
679 :      EXIT ZEND OF ADDRESS.TO.BINARY ROUTINE
680 :
681 : Z EFFECTIVE.ADDR ZROUTINE BEGINS HERE. SEE FIG. 6-19.
682 : Z INPUT IS A POINTER IN FA TO INDEX FIELD
683 : Z OUTPUT IS A FLAG (FLF REGISTER)
684 : Z 0 = OK
685 : Z 1 = TOO MANY INDICES
686 : Z 2 = ADDR NOT DECIMAL
687 : Z 4 = BAD OPERAND ADDR
688 : Z AND THE EFFECTIVE ADDRESS IN EA
689 : Z AS A BINARY VALUE
690 : Z ROUTINE USES X,Y,T,L,CP,FL,S0B,AND S1B
691 : Z AS LOCALS
692 :
693 :      BEGIN
694 :      LOCAL.DEFINES                                =FLF#
695 :      DEFINE FLAG                                  =FLE#
696 :      DEFINE CTR                                   =S0B#
697 :      DEFINE INDICATOR                             =S1B#
698 :      READ 24 BITS TO T INC FA
699 :      SET CTR TO 0
700 :      MOVE "0" TO Y
701 :      EXTRACT 8 BITS FROM T(0) TO X
702 :      IF X NEQ Y THEN
703 :      BEGIN
704 :      INC CTR BY 1
705 :      MOVE IX1.ADDR TO INDICATOR
706 :      END
707 :      EXTRACT 8 BITS FROM T(8) TO X
708 :      IF X NEQ Y THEN
709 :      BEGIN
710 :      INC CTR BY 1
711 :      MOVE IX2.ADDR TO INDICATOR
712 :      END
713 :      EXTRACT 8 BITS FROM T(16) TO X
714 :      IF X NEQ Y THEN
715 :      BEGIN
716 :      INC CTR BY 1
717 :      MOVE IX3.ADDR TO INDICATOR
718 :      END

```

```

719 : IF CTR(2) THEN                                % IS CTR GEQ 2
720 : BEGIN                                         % TOO MANY INDICES
721 : SET FLAG TO 1
722 : EXIT
723 : END
724 : IF CTR(3) THEN                                %BOX5
725 : BEGIN
726 : MOVE INDICATOR TO T
727 : MOVE FA TO IAS                                % SAVE ADDRESS FIELD PTR.
728 : CALL ADDRESS.TO.BINARY
729 : MOVE IAS TO FA                                % RESTORE ADDRESS FIELD PTR.
730 : MOVE T TO TEMP
731 : END ELSE
732 : BEGIN
733 : MOVE NULL TO TEMP
734 : END
735 : CLEAR L Y                                       %BOX6
736 : READ 8 BITS TO T INC FA
737 : IF TE EQL 2FA FALSE THEN GO TO +SET.FLAG.EXIT
738 : MOVE TF TO LC
739 : READ 24 BITS TO T
740 : IF TA EQL 2FA FALSE THEN GO TO +SET.FLAG.EXIT
741 : IF TC EQL 2FA FALSE THEN GO TO +SET.FLAG.EXIT
742 : IF TE EQL 2FA FALSE THEN GO TO +SET.FLAG.EXIT
743 : MOVE TB TO LD
744 : MOVE TD TO LE
745 : MOVE TF TO LF
746 :                                               % NOW L HOLDS THE PACKED DECIMAL
747 : MOVE 2(1)001110002 TO CP                       % VALUE OF THE ADDRESS FIELD
748 : MOVE L TO X                                       %SET CP FOR PACKED DECIMAL
749 : %CLEAR Y ALREADY ACCOMPLISHED                 %ADD (24 BITS)
750 : MOVE SUM TO Y
751 : IF X NEQ Y THEN
752 : BEGIN                                           %TESTS L+0=L
753 : %SET.FLAG.EXIT % ADDRESS NOT DECIMAL           %IF SO,L MUST HAVE BEEN A
754 : SET FLAG TO 2                                     %VALID,PACKED DECIMAL
755 : EXIT
756 : END
757 :
758 : MOVE 24 TO CP                                       %BOX9
759 : MOVE LC TO T                                       % SET UP FOR BINARY ARITH.
760 : TEN.T.PLUS.D(LD) %MACRO CALL
761 : TEN.T.PLUS.D(LE) %MACRO CALL
762 : TEN.T.PLUS.D(LF) %MACRO CALL
763 :
764 : MOVE T TO X                                       %BOXES 10
765 : MOVE TEMP TO Y                                       %THRU 14
766 : MOVE SUM TO X
767 : MOVE SIZE TO Y
768 : IF X LSS Y THEN
769 : BEGIN
770 : SET FLAG TO 0
771 : END ELSE
772 : BEGIN %BAD OPERAND ADDR
773 : SET FLAG TO 4
774 : END
775 : MOVE X TO EA %EFFECTIVE ADDRESS SAVED IN EA AS A BINARY VALUE
776 : EXIT
777 : END % OF EFFECTIVE.ADDR ROUTINE
778 : %
779 : %ADD.10.COMPL %ROUTINE BEGINS HERE. SEE FIG. 6-29. ARGUMENTS ARE
780 : % OP1 IN S5, OP2 IN S6, IN SIGNED MAGNITUDE FORM
781 : % AS IN FIG.6-7. THE RESULT IS LEFT IN OP1,IE,IN S5,
782 : % AND OVERFLOW FLAG IN CB(0).
783 : % THE STACK,X,Y, T, AND L ARE USED AS LOCAL STORAGE.
784 : % THE PROCEDURE COMP.T.L CONVERTS T CAT L TO 10'S
785 : % COMPLEMENT FORM
786 : BEGIN
787 : LOCAL.DEFINES
788 : DEFINE OP1A =S5A #
789 : DEFINE OP1B =S5B #
790 : DEFINE OP2A =S6A #
791 : DEFINE OP2B =S6B #
792 : DEFINE FLAG =CB(0)#
793 : %
794 : MOVE 2(1)001110002 TO CP %SETUP FOR 24-BIT
795 : %DECIMAL ARITHMETIC %BOX1
796 :
797 : MOVE OP1A TO T
798 : MOVE OP1B TO L
799 : IF T(0) THEN %IF OP1 NEGATIVE
800 : BEGIN %COMPLEMENT ABS OF OP1
801 : RESET T(0)
802 : CALL COMPL.T.L
803 : END
804 : MOVE T TO IAS %SAVE OP1 ON STACK
805 : MOVE L TO IAS
806 : MOVE OP2A TO T %BOX2
807 : MOVE OP2B TO L
808 : IF T(0) THEN %IF OP2 NEGATIVE
809 : BEGIN %COMPLEMENT ABS OF OP2
810 : RESET T(0)
811 : CALL COMPL.T.L
812 : END
813 :
814 : MOVE IAS TO X %LOW-ORDER PARTS OF OP1 AND %BOX3
815 : MOVE L TO Y %OP2 IN X AND Y,RESP.
816 : MOVE SUM TO L %LOW-ORDER PART OF OP1+OP2 IN L
817 : CARRY SUM %RECYCLE CARRY DIGIT

```

McMIL Listing for an Abridged SAMOS Interpreter

```

818 :      MOVE TAS TO X           %HIGH-ORDER PARTS OF OP1 AND
819 :      MOVE T TO Y            %OP2 IN X AND Y, RESP.
820 :      MOVE SUM TO T         %HIGH-ORDER PART OF OP1+OP2 IN T
821 :                               %BOX4
822 :      IF T(0) THEN          %IF LEADING DIGIT IS 9 OR 8
823 :      BEGIN
824 :          CALL COMPL.T.L    %THEN
825 :          SET T(0)          %COMPLEMENT IT AND
826 :                               %MARK IT MINUS
827 :      END
828 :                               %BOX5
829 :      IF TB NEQ 0 THEN      %IF 11TH DIGIT OF SUM NON-ZERO
830 :      BEGIN
831 :          SET FLAG          %THE WE HAVE OVERFLOW
832 :      END ELSE
833 :      BEGIN
834 :          RESET FLAG
835 :      END
836 :                               %EPILOGUE
837 :      MOVE T TO OP1A         %NEW RESULT LEFT IN OP1A
838 :      MOVE L TO OP1B        %AND OP1B
839 :      EXIT
840 :      END % OF ADD.10.COMPL ROUTINE
841 :
842 :      %COMPL.T.L           %PROCEDURE COMPUTES THE 10'S COMPLEMENT OF T CAT L
843 :                               %AND LEAVES THE RESULT IN T CAT L,
844 :                               %USING X AND Y AS LOCAL STORAGE.
845 :      MOVE (1)001110002 TO CP %TO BE SURE OF ARITH. SETUP
846 :      CLEAR X                %MORE SETUP
847 :      MOVE L TO Y
848 :      MOVE DIFF TO L         %LOW-ORDER PART OF COMPL IN L
849 :      CARRY DIFFERENCE      %RECYCLE THE BORROW
850 :      MOVE T TO Y
851 :      MOVE DIFF TO T        %COMPLEMENT NOW IN T CAT L
852 :      CARRY 0               %LEAVE CARRY IN 'CLEAN' STATE
853 :      EXIT                  %END OF COMPL.T.L
854 :      :=TERMINATE SAMOS/INTERP
855 :      :=?END

```


2 MIL ASSEMBLER LISTING OF THE SAMOS INTERPRETER

BURROUGHS B1700 MIL COMPILER, MARK V.0(01/24/76 18:05)

SAMOS/INTERP

FRIDAY, JUNE 04, 1976, 09:03 AM.

BLOCK NAME	CODE	MEMORY ADDRESS	SOURCE IMAGE	SEQUENCE	SEGMENT NAME	OBJ DECK ADDRESS
			DEFINE CARD.READER	=1 #		[000001] C
			DEFINE PRINTER	=0 #		[000002] C
			DEFINE FIFTY.SIX	=S3A #		[000003] C
			DEFINE MESSAGE.BASE	=S4A #		[000004] C
			DEFINE BASE.OF.INTERPRETER	=S12A #		[000005] C
			DEFINE SAMOS.STORE.ADDR	=S13A #		[000006] C
			DEFINE INAREA.ADDR	=S14A #		[000007] C
			DEFINE PRINT.AREA.ADDR	=S15A #		[000008] C
			DEFINE EA	=S18 #		[000009] C
			DEFINE MESSAGE.LENGTH	=S7B #		[000010] C
			DEFINE LOCATION.COUNTER	=S9B #	DEFINED IN LOAD.A-PROG	[000011] C
			DEFINE WORK.COUNTER	=S11B #		[000012] C
			DEFINE TERMINATION.CODE	=S12B #		[000013] C
			DEFINE LOAD.CODE	=S13B #		[000014] C
			DEFINE FOUND.SWITCH	=S14B #		[000015] C
			DEFINE MASTER.SWITCH	=S15B #		[000016] C
			DEFINE FLAG	=FLF #		[000017] C
			DEFINE YES	=1 #		[000018] C
			DEFINE NOT.YET.SET	=0 #		[000019] C
			DEFINE SIZE	=100 #		[000020] C
			DEFINE WORK.LIMIT	=1500 #		[000021] C
			DEFINE OK	=0 #	ZVALUES FOR	[000022] C
			DEFINE EOF	=1 #	ZLOAD.CODE	[000023] C
			DEFINE STAR	=2 #		[000024] C
			DEFINE TOOBIG	=3 #		[000025] C
			DEFINE IX3.ADDR	=28000053 #	Z -5 IN SIGNED MAG.	[000026] C
			DEFINE IX2.ADDR	=28000043 #	Z -4 IN SIGNED MAG.	[000027] C
			DEFINE IX1.ADDR	=28000033 #	Z -3 IN SIGNED MAG.	[000028] C
			DEFINE IC.ADDR	=28000023 #	Z -2 IN SIGNED MAG.	[000029] C
			DEFINE ACC.ADDR	=28000013 #	Z -1 IN SIGNED MAG.	[000030] C
			DEFINE CHAR	=CHARACTER#		[000031] C
			DECLARE			[000032] C
			DECLARE MESSAGES			[000033] C
[000000]			MESS.1 CHAR(18),	Z'PROGRAM INCOMPLETE'		[000034] C
[000090]			MESS.2 CHAR(15),	Z'PROGRAM TOO BIG'		[000035] C
[000108]			MESS.3 CHAR(11),	Z'NORMAL HALT'		[000036] C
[000160]			MESS.4 CHAR(11),	Z'BAD OP CODE'		[000037] C
[000188]			MESS.5 CHAR(10),	Z'OVERWORKED'		[000038] C
[000208]			MESS.6 CHAR(19),	Z'BAD OPERAND ADDRESS'		[000039] C
[0002A0]			MESS.7 CHAR(12),	Z'BAD IC VALUE'		[000040] C
[000300]			MESS.8 CHAR(12),	Z'EOF ON INPUT'		[000041] C
[000360]			MESS.9 CHAR(19),	Z'NON-NUMERIC OPERAND'		[000042] C
[0003F8]			MESS.10 CHAR(15),	Z'NON-NUMERIC ACC'		[000043] C
[000470]			MESS.11 CHAR(16),	Z'ADDR NOT DECIMAL'		[000044] C
[0004F0]			MESS.12 CHAR(16),	Z'TOO MANY INDICES'		[000045] C
[000570]			MESS.13 CHAR(24),	Z'ERROR IN THE INTERPRETER'		[000046] C
[000630]			MESS.14 CHAR(25),	Z'BEGIN BATCH RUN FOR SAMOS'		[000047] C
[0006F8]			INAREA CHAR(80),	Z'CARD IMAGE		[000048] C
[000978]			PRINT.AREA CHAR(80),	Z'LINE IMAGE		[000049] C
[000BF8]			DECLARE 01 STORAGE.FOR.SAMOS,			[000050] C
[000BF8]			02 REGISTERS,			[000051] C
[000BF8]			03 IX3 BIT(88),	ZRELATIVE ADDR = -5		[000052] C
						[000053] C

[000C50]
 [000CA8]
 [000D00]
 [000D58]
 [000DB0]
 [000DB8]
 [000DD0]
 [000DD0]
 [000DD8]
 [000DE0]
 [000DE8]

```

03 IX2      BIT(88),
03 IX1      BIT(88),
03 IC       BIT(88),
03 ACC      BIT(88),
02 SAMOS.STORE(SIZE),
03 SIGN     CHAR(1),
03 OPCODE   CHAR(3),
03 INDEXES,
04 INDEX1 CHAR(1),
04 INDEX2 CHAR(1),
04 INDEX3 CHAR(1),
03 ADDRESS CHAR(4);
Z MACRO DEFINITIONS(GLOBAL)
Z
MACRO EFF.ADDR.TO.FA =
CALL EFFECTIVE.ADDR
IF FLF NEQ 0 THEN GO TO EA.ERROR
MOVE EA TO T
CALL BINARY.TO.FA #
END MACRO

MACRO TEN.T.PLUS.D(LX) =
SHIFT T LEFT BY 1 BIT TO X
SHIFT T LEFT BY 3 BITS TO Y
MOVE SUM TO X
MOVE LX TO Y
MOVE SUM TO T #
END MACRO

Z BEGIN SHELL
SECTION SHELL

MOVE PRINT.AREA TO PRINT.AREA.ADDR

MOVE INAREA TO INAREA.ADDR

MOVE SAMOS.STORE(0) TO SAMOS.STORE.ADDR

MOVE S6 TO FIFTY.SIX

OUTPUT DATA.LENGTH(MESS.14) BITS CORE MESS.14 FILE PRINTER OPT
MOVE NULL TO MASTER.SWITCH
MOVE NULL TO FOUND.SWITCH
MAIN.SHELL.LOOP
MOVE MASTER.SWITCH TO X
IF X NEQ 0 THEN GO TO +STOP.STEP

CALL FIND.A.JOB.CARD ROUTINE TO BOX3
GO TO FIND.A.JOB.CARD Z FROM BOX3

BOX4.SHELL
MOVE FOUND.SWITCH TO X
MOVE 1 TO Y
IF X NEQ Y THEN GO TO -MAIN.SHELL.LOOP
MOVE NULL TO FOUND.SWITCH
CALL LOAD.A.PROGRAM ROUTINE TO BOX5
GO TO LOAD.A.PROGRAM
  
```

-4
 -3
 -2
 -1
 SAMOS STORE BEGINS HERE.
 IT IS SIZE WORDS LONG.
 EACH WORD HAS 11 CHARS.

[000054] C
 [000055] C
 [000056] C
 [000057] C
 [000058] C
 [000059] C
 [000060] C
 [000061] C
 [000062] C
 [000063] C
 [000064] C
 [000065] C
 [000066] C
 [000067] C
 [000068] C
 [000069] C
 [000070] C
 [000071] C
 [000072] C
 [000073] C
 [000074] C
 [000075] C
 [000076] C
 [000077] C
 [000078] C
 [000079] C
 [000080] C
 [000081] C
 [000082] C
 [000083] C
 [000084] C
 [000085] C
 [000263] C
 [000264] C
 [000265] C
 [000266] C
 [000267] C
 [000268] C
 [000269] C
 [000270] C
 [000271] C
 [000279] C
 [000280] C
 [000281] C
 [000282] C
 [000283] C
 [000284] C
 [000285] C
 [000286] C
 [000287] C
 [000288] C
 [000289] C
 [000290] C
 [000291] C
 [000292] C

[00A60]
 [00A70]
 [00A80]
 [00A90]
 [00AA0]
 [00AB0]
 [00AC0]
 [00AD0]
 [00AE0]
 [00AF0]
 [00B00]
 [00B90]
 [00BA0]
 [00BB0]
 [00BC0]
 [00BD0]
 [00BE0]
 [00BF0]
 [00C00]
 [00C10]
 [00C20]
 [00C30]

SHELL 9B00 AT [00A60]
 SHELL 0978 AT [00A70]
 SHELL 288F AT [00A80]
 SHELL 9800 AT [00A90]
 SHELL 06F8 AT [00AA0]
 SHELL 288E AT [00AB0]
 SHELL 9B00 AT [00AC0]
 SHELL 0D80 AT [00AD0]
 SHELL 288D AT [00AE0]
 SHELL 8B38 AT [00AF0]
 SHELL 2883 AT [00B00]
 SHELL 2FDF AT [00B90]
 SHELL 2FDE AT [00BA0]
 SHELL 208F AT [00BB0]
 SHELL 4D81 AT [00BC0]
 SHELL C039 AT [00BD0]
 SHELL C045 AT [00BE0]
 SHELL 208E AT [00BF0]
 SHELL 8101 AT [00C00]
 SHELL 4CD7 AT [00C10]
 SHELL 2FDE AT [00C20]
 SHELL C066 AT [00C30]

			.BOX6.SHELL		Z FROM BOX5	[000293]	C	
SHELL	208D	AT	[00C40]	MOVE LOAD.CODE TO X		[000294]	C	[00C40]
SHELL	8100	AT	[00C50]	MOVE OK TO Y		[000295]	C	[00C50]
SHELL	4CC1	AT	[00C60]	IF X EQL Y THEN GO TO +BOX7		[000296]	C	[00C60]
SHELL	C020	AT	[00C70]				I	[00C70]
SHELL	8101	AT	[00C80]	MOVE EOF TO Y	Z BOX 9.1	[000297]	C	[00C80]
SHELL	5CDF	AT	[00C90]	IF X EQL Y THEN GO TO -MAIN.SHELL.LOOP		[000298]	C	[00C90]
SHELL	8102	AT	[00CA0]	MOVE STAR TO Y	Z BOX 9.3	[000299]	C	[00CA0]
SHELL	5CC1	AT	[00CB0]	IF X EQL Y THEN		[000300]	C	[00CB0]
SHELL	C013	AT	[00CC0]			[000301]	C	[00CC0]
			BEGIN	ZPROGRAM INCOMPLETE, BOX 9.4		[000302]	C	
ZMZ			OUTPUT DATA.LENGTH(MESS.1) BITS	CORE MESS.1 FILE PRINTER OPT D		[000310]	C	
				Z ACTION FOR A STAR CARD		[000311]	C	
SHELL	8B01	AT	[00DD0]	OUTPUT 80 BYTES CORE INAREA FILE PRINTER OPT SINGLE		[000320]	C	[00DD0]
SHELL	2B9E	AT	[00DE0]	MOVE 1 TO FOUND.SWITCH			G	[00DE0]
SHELL	C007	AT	[00DF0]	END ELSE		[000321]	C	[00DF0]
			BEGIN			[000322]	C	
			ZMZ	OUTPUT DATA.LENGTH(MESS.2) BITS	ZPROGRAM TOO BIG	[000323]	C	
				CORE MESS.2 FILE PRINTER OPT D		[000324]	C	
			END			[000332]	C	
SHELL	D02D	AT	[00E70]	GO TO -MAIN.SHELL.LOOP		[000333]	C	[00E70]
			.BOX7	CALL INTERP.PROGRAM ROUTINE		[000334]	C	
SHELL	C09C	AT	[00E80]	GO TO INTERPRET.PROGRAM		[000335]	C	[00E80]
			.BOX8.SHELL	MOVE TERMINATION.CODE TO X		[000336]	C	
SHELL	208C	AT	[00E90]	MOVE NOT.YET.SET TO Y		[000337]	C	[00E90]
SHELL	8100	AT	[00EA0]	IF X EQL Y THEN	Z SHOULD NOT HAPPEN	[000338]	C	[00EA0]
SHELL	4CCA	AT	[00EB0]	BEGIN	Z ERROR IN INTERPRETER	[000340]	C	[00EB0]
				MOVE 1 TO MASTER.SWITCH		[000342]	C	[00EC0]
SHELL	8B01	AT	[00EC0]			[000343]	C	[00ED0]
SHELL	2B9F	AT	[00ED0]	ZMZ	OUTPUT DATA.LENGTH(MESS.13) BITS	[000351]	C	
				END		[000352]	C	
SHELL	D03C	AT	[00F60]	GO TO -MAIN.SHELL.LOOP		[000353]	C	[00F60]
			Z END OF MAIN SHELL LOOP			[000354]	C	
			.STOP.STEP	ZMZ	OUTPUT DATA.LENGTH(MESS.8) BITS	[000355]	C	
				STOP		[000363]	C	
			END			[000369]	C	
			PAGE			[000370]	C	
			BEGIN FIND.A.JOB.			[000371]	C	
ZMZ			SECTION FIND.A.JOB.			[000372]	C	
FIND.A.JOB	208F	AT	[01040]	FIND.A.JOB.CARD	ZROUTINE BEGINS HERE. SEE FIG.5-4	[000373]	C	
FIND.A.JOB	4D81	AT	[01050]	.FIND.LOOP		[000374]	C	
FIND.A.JOB	D048	AT	[01060]	MOVE FOUND.SWITCH TO X		[000375]	C	[01040]
FIND.A.JOB	208F	AT	[01070]	IF X NEQ 0 THEN GO TO -BOX4.SHELL		[000376]	C	[01050]
FIND.A.JOB	D081	AT	[01080]				I	[01060]
FIND.A.JOB	4048	AT	[01090]	MOVE MASTER.SWITCH TO X		[000377]	C	[01070]
FIND.A.JOB	D048	AT	[01090]	IF X NEQ 0 THEN GO TO -BOX4.SHELL		[000378]	C	[01080]
			ZMZ	BUFFER READ USING INAREA FILE CARD.READER ON EOF GO TO +BOX4.FI		[000379]	C	[01090]
FIND.A.JOB	16A8	AT	[01160]	MOVE BR TO FA		[000389]	C	[01160]
FIND.A.JOB	080E	AT	[01170]	ADD INAREA.ADDR TO FA		[000390]	C	[01170]
FIND.A.JOB	7008	AT	[01180]	READ 8 BITS TO X	ZCHECK FOR STAR	[000391]	C	[01180]
FIND.A.JOB	815C	AT	[01190]	MOVE "*" TO Y		[000392]	C	[01190]
FIND.A.JOB	4CCB	AT	[011A0]	IF X EQL Y THEN		[000393]	C	[011A0]
			BEGIN	Z FOUND A STAR		[000394]	C	
ZMZ			OUTPUT 80 BYTES CORE	INAREA FILE PRINTER OPT DOUBLE		[000395]	C	
FIND.A.JOB	8B01	AT	[01240]	MOVE 1 TO FOUND.SWITCH		[000404]	C	[01240]
FIND.A.JOB	2B9E	AT	[01250]				G	[01250]
			END			[000405]	C	

FIND.A.JOB	D023	AT	[01260]	GO TO -FIND.LOOP	[000406]	C	[01260]
FIND.A.JOB	6801	AT	[01270]	MOVE 1 TO MASTER.SWITCH	[000407]	C	[01270]
FIND.A.JOB	289F	AT	[01280]	GO TO -FIND.LOOP	[000409]	C	[01280]
FIND.A.JOB	D026	AT	[01290]	Z END OF FIND.A.JOB.CARD	[000410]	C	[01290]
				END	[000411]	C	
				PAGE	[000412]	C	
				BEGIN LOAD	[000413]	C	
				SECTION LOAD	[000414]	C	
ZMX				ZROUTINE BEGINS HERE. SEE FIG.5-5	[000415]	C	
LOAD.A.PROGRAM				LOCAL DEFINES	[000416]	C	
				DEFINE LOCATION.COUNTER =S9B#	[000417]	C	
				DEFINE UNDEFINED =4 # % VALUE FOR LOAD.CODE	[000418]	C	
Z				MOVE -1 TO LOCATION.COUNTER	[000419]	C	[012A0]
					[000420]	C	[012B0]
LOAD	98FF	AT	[012A0]	ZCLEAR EMULATED SAMOS STORAGE.	[000421]	C	[012C0]
LOAD	FFFF	AT	[012B0]	MOVE DATA.LENGTH(STORAGE.FOR.SAMOS) TO FL % INITIALIZE FL,FA,X	[000422]	C	[012D0]
LOAD	2B99	AT	[012C0]	MOVE STORAGE.FOR.SAMOS TO S1A	[000423]	C	[012E0]
LOAD	9A00	AT	[012D0]			C	[012F0]
LOAD	2418	AT	[012E0]			C	[01300]
LOAD	9800	AT	[012F0]			C	[01310]
LOAD	0BF8	AT	[01300]			C	[01320]
LOAD	2B81	AT	[01310]	MOVE BR TO FA	[000424]	C	[01330]
LOAD	16A8	AT	[01320]	ADD S1A TO FA	[000425]	C	[01340]
LOAD	0801	AT	[01330]	MOVE "000" TO X % CHAR. ZEROES IN X	[000426]	C	[01350]
LOAD	90F0	AT	[01340]			C	[01360]
LOAD	F0F0	AT	[01350]	MOVE "+0" TO Y	[000427]	C	[01370]
LOAD	9100	AT	[01360]			C	[01380]
LOAD	4EF0	AT	[01370]			C	[01390]
				END	[000428]	C	[013A0]
LOAD	4785	AT	[01380]	MOVE UNDEFINED TO LOAD.CODE %BOX 3.1	[000429]	C	[013B0]
					[000430]	C	[013C0]
LOAD	7850	AT	[01390]	WRITE 16 BITS FROM Y INC FA AND DEC FL	[000431]	C	[013D0]
LOAD	7818	AT	[013A0]	WRITE 24 BITS FROM X INC FA AND DEC FL	[000432]	C	[013E0]
LOAD	7818	AT	[01380]	WRITE 24 BITS FROM X INC FA AND DEC FL	[000433]	C	[013F0]
LOAD	7818	AT	[01380]	WRITE 24 BITS FROM X INC FA AND DEC FL	[000434]	C	[01400]
LOAD	7818	AT	[01380]	WRITE 24 BITS FROM X INC FA AND DEC FL	[000435]	C	[01410]
LOAD	D006	AT	[013D0]	GO TO -CLEAR.LOOP	[000436]	C	[01420]
LOAD	8804	AT	[013E0]		[000437]	C	[01430]
LOAD	2B9D	AT	[013F0]			C	[01500]
				END	[000438]	C	[01510]
				MOVE UNDEFINED TO LOAD.CODE %BOX 3.2	[000439]	C	[01520]
LOAD	208D	AT	[01400]		[000440]	C	[01530]
LOAD	8104	AT	[01410]	MOVE LOAD.CODE TO X	[000441]	C	[01540]
LOAD	5CC1	AT	[01420]	MOVE UNDEFINED TO Y	[000442]	C	[01550]
LOAD	D080	AT	[01430]	IF X NEQ Y THEN GO TO -BOX6.SHELL	[000443]	C	[01560]
					[000444]	C	[01570]
ZMX				BUFFER READ USING INAREA FILE CARD.READER ON EOF GO TO +BOX3.5	[000445]	C	[01580]
				Z CHECK FOR A BLANK CARD	[000446]	C	[01590]
				Z INITIALIZE LOOP TO SEARCH	[000447]	C	[015A0]
LOAD	9A00	AT	[01500]	MOVE 78*8 TO FL	[000448]	C	[015B0]
LOAD	0270	AT	[01510]		[000449]	C	[015C0]
				Z FIRST 78 CHARS OF CARD, IN	[000450]	C	[015D0]
				Z GROUPS OF THREE	[000451]	C	[015E0]
LOAD	16A8	AT	[01520]		[000452]	C	[015F0]
LOAD	080E	AT	[01530]	MOVE BR TO FA	[000453]	C	[01600]
LOAD	2881	AT	[01540]	ADD INAREA.ADDR TO FA	[000454]	C	[01610]
LOAD	9140	AT	[01550]	MOVE FA TO S1A	[000455]	C	[01620]
LOAD	4040	AT	[01560]	MOVE " " TO Y % SAVE ABS ADDR OF INAREA	[000456]	C	[01630]
				MOVE " " TO Y % 3 BLANKS TO Y	[000457]	C	[01640]
					[000458]	C	[01650]
LOAD	4783	AT	[01570]		[000459]	C	[01660]
LOAD	7318	AT	[01580]	IF FL EQL 0 THEN GO TO +LAST2.CHECK	[000460]	C	[01670]
LOAD	5CD3	AT	[01590]	READ 24 BITS TO X INC FA AND DEC FL	[000461]	C	[01680]
LOAD	C006	AT	[015A0]	IF X EQL Y THEN GO TO -CHECK.LOOP	[000462]	C	[01690]
				GO TO +BOX3.8	[000463]	C	[01700]
					[000464]	C	[01710]
				LAST2.CHECK		C	

LOAD	06A8 AT [01580]	COUNT FA DOWN BY 8	[000465] C	[01580]
LOAD	7018 AT [015C0]	READ 24 BITS TO X	[000466] C	[015C0]
LOAD	4CC3 AT [015D0]	IF X EQL Y THEN	[000467] C	[015D0]
		BEGIN	[000468] C	
LOAD	8B00 AT [015E0]	MOVE OK TO LOAD.CODE	[000469] C	[015E0]
LOAD	2B9D AT [015F0]			[015F0]
LOAD	0021 AT [01600]	GO TO -LOAD.LOOP	[000470] C	[01600]
		END	[000471] C	
		.BOX3.8	[000472] C	
LOAD	28A1 AT [01610]	MOVE S1A TO FA	[000473] C	[01610]
LOAD	7008 AT [01620]	READ 8 BITS TO X	[000474] C	[01620]
LOAD	815C AT [01630]	MOVE "*" TO Y	[000475] C	[01630]
LOAD	4CC3 AT [01640]	IF X EQL Y THEN	[000476] C	[01640]
		BEGIN	[000477] C	
LOAD	8B02 AT [01650]	MOVE STAR TO LOAD.CODE	[000478] C	[01650]
LOAD	2B9D AT [01660]			[01660]
LOAD	0028 AT [01670]	GO TO -LOAD.LOOP	[000479] C	[01670]
		END	[000480] C	
		Z BOX3.10	[000481] C	
LOAD	2AB9 AT [01680]	MOVE LOCATION.COUNTER TO FL	[000482] C	[01680]
LOAD	0641 AT [01690]	COUNT FL UP BY 1	[000483] C	[01690]
LOAD	2A99 AT [016A0]	MOVE FL TO LOCATION.COUNTER	[000484] C	[016A0]
LOAD	8B64 AT [016B0]	MOVE SIZE TO SOB	[000485] C	[016B0]
LOAD	2B90 AT [016C0]			[016C0]
LOAD	57A3 AT [016D0]	IF FL GEQ SFL THEN	[000486] C	[016D0]
		BEGIN	[000487] C	
LOAD	8B03 AT [016E0]	MOVE TOOBIG TO LOAD.CODE	[000488] C	[016E0]
LOAD	2B9D AT [016F0]			[016F0]
LOAD	0031 AT [01700]	GO TO -LOAD.LOOP	[000489] C	[01700]
		END	[000490] C	
		Z BOX 3.13	[000491] C	
LOAD	1AA2 AT [01710]	MOVE FL TO T	[000492] C	[01710]
LOAD	E1A4 AT [01720]	CALL BINARY.TO.FA	[000493] C	[01720]
LOAD	0711 AT [01730]	XCH S1 F S1	[000494] C	[01730]
LOAD	8A58 AT [01740]	MOVE 88 TO FL	[000495] C	[01740]
LOAD	E15E AT [01750]	CALL COPY	[000496] C	[01750]
		Z BOX 3.14	[000497] C	
LOAD	0040 AT [017F0]	ZMZ BUFFER WRITE USING INAREA FILE	[000498] C	[017F0]
		GO TO -LOAD.LOOP	[000506] C	
		Z AND FINALLY,	[000507] C	
		.BOX3.5	[000508] C	
LOAD	8B01 AT [01800]	MOVE EOF TO LOAD.CODE	[000509] C	[01800]
LOAD	2B9D AT [01810]			[01810]
LOAD	8B01 AT [01820]	MOVE 1 TO MASTER.SWITCH	[000510] C	[01820]
LOAD	2B9F AT [01830]			[01830]
LOAD	0045 AT [01840]	GO TO -LOAD.LOOP	[000511] C	[01840]
		Z END OF LOAD.A.PROGRAM ROUTINE	[000512] C	
		END	[000513] C	
		PAGE	[000514] C	
		BEGIN INTERPRET	[000515] C	
		ZMZ SECTION INTERPRET	[000516] C	
		INTERPRET.PROGRAM	[000517] C	
		Z SEE FIG. 5-7	[000518] C	
		Z THE IC IS ALREADY INITIALIZED	[000519] C	
		Z TO "0000" BY VIRTUE OF THE	[000520] C	
		Z CLEAR LOOP IN THE	[000521] C	
		Z LOAD.A.PROGRAM MODULE	[000522] C	
INTERPRET	2FDB AT [01850]	MOVE NULL TO WORK.COUNTER	[000523] C	[01850]
INTERPRET	8B00 AT [01860]	MOVE NOT.YET.SET TO TERMINATION.CODE		[01860]
INTERPRET	2B9C AT [01870]			[01870]
		Z BOX2	[000524] C	
		.INTERP.LOOP	[000525] C	

			ZMZ	GO TO +XXOK IF ALL OK	[000526]	C	
			ZMZ	RELINQUISH CONTROL TO MCP	[000534]	C	
			.XXOK		[000542]	C	
INTERPRET	208C	AT [01960]		MOVE TERMINATION.CODE TO X	[000543]	C	[01960]
INTERPRET	8100	AT [01970]		MOVE NOT.YET.SET TO Y	[000544]	C	[01970]
INTERPRET	5CC1	AT [01980]		IF X NEQ Y THEN GO TO -BOX8.SHELL	[000545]	C	[01980]
INTERPRET	00B1	AT [01990]				I	[01990]
INTERPRET	9280	AT [019A0]		MOVE IC.ADDR TO T	[000546]	C	[019A0]
INTERPRET	0002	AT [019B0]				G	[019B0]
INTERPRET	E18F	AT [019C0]		CALL ADDRESS.TO.BINARY	[000547]	C	[019C0]
INTERPRET	50AB	AT [019D0]		IF Y EQL 0 THEN	[000548]	C	[019D0]
				BEGIN	[000549]	C	
				MOVE YES TO TERMINATION.CODE	[000550]	C	[019E0]
INTERPRET	8B01	AT [019E0]				G	[019E0]
INTERPRET	2B9C	AT [019F0]				C	[019F0]
			ZMZ	OUTPUT DATA.LENGTH(MESS.7) BITS CORE MESS.7 FILE PRINTER OPT SI	[000551]	C	
INTERPRET	D021	AT [01A80]		GO TO -INTERP.LOOP	[000559]	C	[01A80]
				END	[000560]	C	
INTERPRET	E16D	AT [01A90]		CALL BINARY.TO.FA	[000561]	C	[01A90]
INTERPRET	0628	AT [01AA0]		COUNT FA UP BY 8	[000562]	C	[01AA0]
INTERPRET	7118	AT [01AB0]		READ 24 BITS TO X INC FA	[000563]	C	[01AB0]
INTERPRET	18A8	AT [01AC0]		MOVE FA TO TAS	[000564]	C	[01AC0]
INTERPRET	10AB	AT [01AD0]		MOVE X TO TAS	[000565]	C	[01AD0]
					[000566]	C	
				MOVE IC.ADDR TO T	[000567]	C	[01AE0]
INTERPRET	9280	AT [01AE0]				G	[01AE0]
INTERPRET	0002	AT [01AF0]				C	[01AF0]
INTERPRET	E166	AT [01B00]		CALL BINARY.TO.FA	[000568]	C	[01B00]
INTERPRET	2881	AT [01B10]		MOVE FA TO S1A	[000569]	C	[01B10]
INTERPRET	E12A	AT [01B20]		CALL VALIDATE.DECIMAL	[000570]	C	[01B20]
INTERPRET	2FC6	AT [01B30]		MOVE NULL TO S6A	[000571]	C	[01B30]
INTERPRET	8B01	AT [01B40]		MOVE I TO S6B	[000572]	C	[01B40]
INTERPRET	2B96	AT [01B50]				G	[01B50]
INTERPRET	E1E2	AT [01B60]		CALL ADD.10.COMPL	[000573]	C	[01B60]
INTERPRET	28A1	AT [01B70]		MOVE S1A TO FA	[000574]	C	[01B70]
INTERPRET	EOF4	AT [01B80]		CALL UNPACK.AND.WRITE	[000575]	C	[01B80]
INTERPRET	1BA0	AT [01B90]		MOVE TAS TO X	[000576]	C	[01B90]
INTERPRET	1BA8	AT [01BA0]		MOVE TAS TO FA	[000577]	C	[01BA0]
					[000578]	C	
				Z	[000579]	C	
				Z	[000580]	C	
INTERPRET	91D3	AT [01BB0]		MOVE "LDA" TO Y		C	[01BB0]
INTERPRET	C4C1	AT [01BC0]				G	[01BC0]
INTERPRET	4CC1	AT [01BD0]		IF X EQL Y THEN GO TO LDA..	[000581]	C	[01BD0]
INTERPRET	C062	AT [01BE0]				I	[01BE0]
					[000582]	C	
INTERPRET	91E2	AT [01BF0]		MOVE "STO" TO Y	[000583]	C	[01BF0]
INTERPRET	E3D6	AT [01C00]				G	[01C00]
INTERPRET	4CC1	AT [01C10]		IF X EQL Y THEN GO TO STO..	[000584]	C	[01C10]
INTERPRET	C06B	AT [01C20]				I	[01C20]
					[000585]	C	
INTERPRET	91C1	AT [01C30]		MOVE "ADD" TO Y	[000586]	C	[01C30]
						G	
INTERPRET	C4C4	AT [01C40]			[000587]	C	[01C40]
INTERPRET	4CC1	AT [01C50]		IF X EQL Y THEN GO TO ADD..		C	[01C50]
INTERPRET	C031	AT [01C60]				I	[01C60]
					[000588]	C	
INTERPRET	91C2	AT [01C70]		MOVE "BRU" TO Y	[000589]	C	[01C70]
INTERPRET	D9E4	AT [01C80]				G	[01C80]
INTERPRET	4CC1	AT [01C90]		IF X EQL Y THEN GO TO BRU..	[000590]	C	[01C90]
INTERPRET	C06F	AT [01CA0]				I	[01CA0]
					[000591]	C	
INTERPRET	91C2	AT [01CB0]		MOVE "BMI" TO Y	[000592]	C	[01CB0]
INTERPRET	D4C9	AT [01CC0]				G	[01CC0]
INTERPRET	4CC1	AT [01CD0]		IF X EQL Y THEN GO TO BMI..	[000593]	C	[01CD0]
INTERPRET	C074	AT [01CE0]				I	[01CE0]
					[000594]	C	

INTERPRET	91D9	AT	[01CF0]	MOVE "RWD" TO Y		[000595]	C	[01CF0]
INTERPRET	E6C4	AT	[01D00]				G	[01D00]
INTERPRET	4CC1	AT	[01D10]	IF X EQL Y THEN GO TO RWD..		[000596]	C	[01D10]
INTERPRET	C07A	AT	[01D20]				I	[01D20]
					Z	[000597]	C	
INTERPRET	91E6	AT	[01D30]	MOVE "WWD" TO Y		[000598]	C	[01D30]
INTERPRET	E6C4	AT	[01D40]				G	[01D40]
INTERPRET	4CC1	AT	[01D50]	IF X EQL Y THEN GO TO WWD..		[000599]	C	[01D50]
INTERPRET	C098	AT	[01D60]				I	[01D60]
					Z	[000600]	C	
INTERPRET	91C8	AT	[01D70]	MOVE "HLT" TO Y		[000601]	C	[01D70]
INTERPRET	D3E3	AT	[01D80]				G	[01D80]
INTERPRET	4CC1	AT	[01D90]	IF X EQL Y THEN GO TO HLT..		[000602]	C	[01D90]
INTERPRET	C0A8	AT	[01DA0]				I	[01DA0]
					Z BAD OP CODE	[000603]	C	
INTERPRET	8B01	AT	[01DB0]	MOVE YES TO TERMINATION.CODE		[000604]	C	[01DB0]
INTERPRET	2B9C	AT	[01DC0]				G	[01DC0]
INTERPRET	D05E	AT	[01E50]	ZMZ OUTPUT DATA.LENGTH(MESS.4) BITS CORE MESS.4 FILE PRINTER OPT SI		[000605]	C	[01E50]
				GO TO -INTERP.LOOP		[000613]	C	
				Z END OF DECODE SECTION		[000614]	C	
				Z BEGIN WORK.COUNTER UPDATE (BOXES 7-9 OF FIG. 5-7)		[000615]	C	
				.INC.WORK.COUNTER		[000616]	C	
				MOVE 24 TO CP	Z INCREMENT WORK.COUNTER	[000617]	C	[01E60]
INTERPRET	8C18	AT	[01E60]				G	[01E60]
INTERPRET	218B	AT	[01E70]	MOVE WORK.COUNTER TO Y		[000618]	C	[01E70]
INTERPRET	8001	AT	[01E80]	MOVE 1 TO X		[000619]	C	[01E80]
INTERPRET	20DB	AT	[01E90]	MOVE SUM TO WORK.COUNTER		[000620]	C	[01E90]
INTERPRET	9000	AT	[01EA0]	MOVE WORK.LIMIT TO X	Z IF BORROW,	[000621]	C	[01EA0]
INTERPRET	050C	AT	[01EB0]				G	[01EB0]
INTERPRET	4CAA	AT	[01EC0]	IF X LSS Y THEN	Z WORK.LIMIT EXCEEDED	[000622]	C	[01EC0]
				BEGIN	Z OVERWORKED	[000623]	C	
INTERPRET	8B01	AT	[01ED0]	MOVE YES TO TERMINATION.CODE		[000624]	C	[01ED0]
INTERPRET	2B9C	AT	[01EE0]				G	[01EE0]
				ZMZ OUTPUT DATA.LENGTH(MESS.5) BITS CORE MESS.5 FILE PRINTER OPT SI		[000625]	C	
				END		[000633]	C	
INTERPRET	D070	AT	[01F70]	GO TO -INTERP.LOOP		[000634]	C	[01F70]
				Z END OF INTERPRET.PROGRAM		[000635]	C	
				SAMOS OPERATOR ROUTINES BEGIN HERE		[000636]	C	
				Z		[000637]	C	
				Z ADD..	Z ADD OPERATOR BEGINS HERE	[000638]	C	
				# EFF.ADDR.TO.FA	ZMACRO CALL	[000639]	C	
INTERPRET	E0DF	AT	[01FD0]	CALL VALIDATE.DECIMAL	Z LEAVES FLAG IN CB(0)	[000640]	C	[01FD0]
					Z 1 IF NOGOOD, 0 IF OK	[000641]	C	
INTERPRET	476B	AT	[01FE0]	IF CB(0) THEN	Z AND (VALID) OPERAND IN S5	[000642]	C	[01FE0]
				BEGIN	Z NON-NUMERIC OPERAND	[000643]	C	
INTERPRET	8B01	AT	[01FF0]	MOVE YES TO TERMINATION.CODE		[000644]	C	[01FF0]
INTERPRET	2B9C	AT	[02000]				G	[02000]
				ZMZ OUTPUT DATA.LENGTH(MESS.9) BITS CORE MESS.9 FILE PRINTER OPT SI		[000645]	C	
INTERPRET	D082	AT	[02090]	GO TO -INTERP.LOOP		[000653]	C	[02090]
				END		[000654]	C	
INTERPRET	0755	AT	[020A0]	XCH S5 F S5	Z MOVES VALIDATED OPERAND FROM	[000655]	C	[020A0]
INTERPRET	0766	AT	[020B0]	XCH S6 F S6	Z S5 TO S6	[000656]	C	[020B0]
INTERPRET	9280	AT	[020C0]	MOVE ACC.ADDR TO T		[000657]	C	[020C0]
INTERPRET	0001	AT	[020D0]				G	[020D0]
INTERPRET	E108	AT	[020E0]	CALL BINARY.TO.FA	Z ABS ADDR OF ACC IN FA	[000658]	C	[020E0]
INTERPRET	2882	AT	[020F0]	MOVE FA TO S2A	Z SAVE ADDR TEMPORARILY	[000659]	C	[020F0]
INTERPRET	E0CC	AT	[02100]	CALL VALIDATE.DECIMAL	Z LEAVES FLAG IN CB(0) AND	[000660]	C	[02100]
INTERPRET	476B	AT	[02110]	IF CB(0) THEN	Z (VALID OPERAND IN S5	[000661]	C	[02110]
				BEGIN	Z NON-NUMERIC ACC	[000662]	C	
INTERPRET	8B01	AT	[02120]	MOVE YES TO TERMINATION.CODE		[000663]	C	[02120]
INTERPRET	2B9C	AT	[02130]				G	[02130]
				ZMZ OUTPUT DATA.LENGTH(MESS.10) BITS CORE MESS.10 FILE PRINTER OPT		[000664]	C	
INTERPRET	D095	AT	[021C0]	GO TO -INTERP.LOOP		[000672]	C	[021C0]

INTERPRET	E17B	AT	[021D0]	END	ZADDS TWO PACKED DECIMAL VALUES	[000673]	C	[021D0]
				CALL ADD.10.COMPL	Z IN S5 AND S6 AND LEAVES RESULT	[000674]	C	
					Z IN S5.	[000675]	C	
INTERPRET	28A2	AT	[021E0]	MOVE S2A TO FA	Z NOW RESTORE ABS ADDR OF ACC	[000676]	C	[021E0]
INTERPRET	E08D	AT	[021F0]	CALL UNPACK.AND.WRITE	Z STORES CHAR. REPRES. IN ACC	[000677]	C	[021F0]
INTERPRET	D03B	AT	[02200]	GO TO -INC.WORK.COUNTER		[000678]	C	[02200]
				Z END OF ADD OPERATOR		[000679]	C	
				Z LDA..	Z LDA OPERATOR BEGINS HERE	[000680]	C	
				# LDA..	Z GET SOURCE PTR (MACRO CALL)	[000681]	C	
INTERPRET	2881	AT	[02260]	EFF.ADDR.TO.FA	ZSAVE IT IN S1A	[000682]	C	[02260]
INTERPRET	9280	AT	[02270]	MOVE FA TO S1A	Z GET SINK POINTER	[000683]	C	[02270]
INTERPRET	0001	AT	[02280]	MOVE ACC.ADDR TO T		[000684]	C	[02280]
INTERPRET	E0ED	AT	[02290]			[000685]	C	[02290]
INTERPRET	0711	AT	[022A0]	CALL BINARY.TO.FA	Z SET LENGTH OF VALUE TO BE COPYD	[000686]	C	[022A0]
INTERPRET	8A58	AT	[022B0]	KCH S1 F S1	Z ARGS READY FOR COPY CALL	[000687]	C	[022B0]
INTERPRET	E0A7	AT	[022C0]	MOVE 88 TO FL		[000688]	C	[022C0]
INTERPRET	D048	AT	[022D0]	CALL COPY		[000689]	C	[022D0]
				GO TO -INC.WORK.COUNTER		[000690]	C	
				Z STO..	Z STO OPERATOR BEGINS HERE	[000691]	C	
				# STO..	Z GET SINK PTR (MACRO CALL)	[000692]	C	
INTERPRET	2881	AT	[02330]	EFF.ADDR.TO.FA	ZSAVE IT IN S1A	[000693]	C	[02330]
INTERPRET	8A58	AT	[02340]	MOVE FA TO S1A	Z SET LENGTH OF VALUE TO BE COPYD	[000694]	C	[02340]
INTERPRET	9280	AT	[02350]	MOVE 88 TO FL	Z GET SOURCE PTR	[000695]	C	[02350]
INTERPRET	0001	AT	[02360]	MOVE ACC.ADDR TO T		[000696]	C	[02360]
INTERPRET	E0DF	AT	[02370]			[000697]	C	[02370]
INTERPRET	E09B	AT	[02380]	CALL BINARY.TO.FA	Z ARGS NOW READY FOR COPY CALL	[000698]	C	[02380]
INTERPRET	D054	AT	[02390]	CALL COPY		[000699]	C	[02390]
				GO TO -INC.WORK.COUNTER		[000700]	C	
				Z BRU..	Z BRU OPERATOR BEGINS HERE	[000701]	C	
INTERPRET	E10E	AT	[023A0]	CALL EFFECTIVE.ADDR	Z LEAVES EFFECTIVE ADDRESS AS A	[000702]	C	[023A0]
					Z BINARY VALUE IN EA AND FLAG IN	[000703]	C	
					Z FLF	[000704]	C	
INTERPRET	65A0	AT	[023B0]	IF FLF NEQ 0 THEN GO TO EA-ERROR		[000705]	C	[023B0]
INTERPRET	C051	AT	[023C0]			[000706]	C	[023C0]
INTERPRET	E087	AT	[023D0]	CALL BINARY.TO.DECIMAL	Z LEAVES PACKED DECIMAL EQUIV.	[000707]	C	[023D0]
					Z OF EA IN S5.	[000708]	C	
INTERPRET	9280	AT	[023E0]	MOVE IC.ADDR TO T	Z GET ABS ADDR OF IC	[000709]	C	[023E0]
INTERPRET	0002	AT	[023F0]			[000710]	C	[023F0]
INTERPRET	E0D6	AT	[02400]	CALL BINARY.TO.FA	Z ASSIGN NEW VALUE TO IC	[000711]	C	[02400]
INTERPRET	E06B	AT	[02410]	CALL UNPACK.AND.WRITE		[000712]	C	[02410]
INTERPRET	D05D	AT	[02420]	GO TO -INC.WORK.COUNTER		[000713]	C	[02420]
				Z		[000714]	C	
				Z			C	
				Z BMI..	Z BMI OPERATOR BEGINS HERE	[000715]	C	
INTERPRET	2882	AT	[02430]	MOVE FA TO S2A	ZSAVE POINTER TO REST OF INSTR	[000716]	C	[02430]
INTERPRET	9280	AT	[02440]	MOVE ACC.ADDR TO T	Z CHECK SIGN OF ACC	[000717]	C	[02440]
INTERPRET	0001	AT	[02450]			[000718]	C	[02450]
INTERPRET	E0D0	AT	[02460]	CALL BINARY.TO.FA		[000719]	C	[02460]
INTERPRET	7078	AT	[02470]	READ 8 BITS TO X		[000720]	C	[02470]
INTERPRET	814E	AT	[02480]	MOVE "*" TO Y		[000721]	C	[02480]
INTERPRET	4CC1	AT	[02490]	IF X EQL Y THEN GO TO -INC.WORK.COUNTER	Z NO BRANCHING (A NO OP)	[000722]	C	[02490]
INTERPRET	D065	AT	[024A0]			[000723]	C	[024A0]
INTERPRET	28A2	AT	[024B0]	MOVE S2A TO FA	Z RESTORE PTR TO REST OF INSTR	[000724]	C	[024B0]
INTERPRET	D013	AT	[024C0]	GO TO BRU..		[000725]	C	[024C0]
				Z RWD..	Z RWD OPERATOR BEGINS HERE	[000726]	C	
				# RWD..	Z GET ABS PTR TO SINK AND PUT IN	[000727]	C	
INTERPRET	2881	AT	[02520]	EFF.ADDR.TO.FA	Z S1A	[000728]	C	[02520]
				MOVE FA TO S1A			C	

INTERPRET	16A8	AT	[025F0]	ZMZ	BUFFER READ USING INAREA FILE CARD.READER ON EOF GO TO +EOF	[000729]	C	
INTERPRET	080E	AT	[02600]		MOVE BR TO FA	[000739]	C	[025F0]
INTERPRET	8A58	AT	[02610]		ADD INAREA.ADDR TO FA	[000740]	C	[02600]
INTERPRET	E071	AT	[02620]		MOVE 88 TO FL	[000741]	C	[02610]
INTERPRET	D07E	AT	[02630]		CALL COPY	[000742]	C	[02620]
					GO TO -INC.WORK.COUNTER	[000743]	C	[02630]
						[000744]	C	
INTERPRET	8801	AT	[02640]	-.EOF	MOVE YES TO TERMINATION.CODE	[000745]	C	[02640]
INTERPRET	2B9C	AT	[02650]		ZAND (VALID) OPERAND IN S5	[000745]	C	[02650]
						[000746]	C	
INTERPRET	D0E7	AT	[026E0]	ZMZ	OUTPUT DATA.LENGTH(MESS.8) BITS CORE MESS.8 FILE PRINTER OPT SI	[000746]	C	[026E0]
					GO TO -INTERP.LOOP	[000754]	C	
						[000755]	C	
						[000756]	C	
						[000757]	C	
						[000758]	C	
INTERPRET	2881	AT	[02740]	#	EFF.ADDR.TO.FA	[000759]	C	[02740]
INTERPRET	16A8	AT	[02750]		MOVE FA TO S1A	[000759]	C	[02750]
INTERPRET	080F	AT	[02760]		MOVE BR TO FA	[000760]	C	[02760]
INTERPRET	0711	AT	[02770]		ADD PRINT.AREA.ADDR TO FA	[000761]	C	[02770]
INTERPRET	8A58	AT	[02780]		XCH S1 F S1	[000762]	C	[02780]
INTERPRET	E05A	AT	[02790]		MOVE 88 TO FL	[000763]	C	[02790]
					CALL COPY	[000764]	C	
INTERPRET	D09D	AT	[02820]	ZMZ	OUTPUT 88 BITS CORE PRINT.AREA FILE PRINTER OPT SINGLE	[000765]	C	
					GO TO -INC.WORK.COUNTER	[000773]	C	[02820]
						[000774]	C	
						[000775]	C	
						[000776]	C	
INTERPRET	8B01	AT	[02830]	Z		[000777]	C	[02830]
INTERPRET	2B9C	AT	[02840]	Z		[000777]	C	[02840]
						[000778]	C	
INTERPRET	D106	AT	[028D0]	ZMZ	OUTPUT DATA.LENGTH(MESS.3) BITS CORE MESS.3 FILE PRINTER OPT SI	[000778]	C	[028D0]
					GO TO -INTERP.LOOP	[000786]	C	
						[000787]	C	
INTERPRET	4588	AT	[028E0]	Z	EA.ERROR	[000788]	C	[028E0]
					IF FLF(3) THEN	[000789]	C	
					BEGIN	[000790]	C	
INTERPRET	45A8	AT	[02970]	ZMZ	OUTPUT DATA.LENGTH(MESS.12) BITS CORE MESS.12 FILE PRINTER OPT	[000791]	C	[02970]
					END	[000799]	C	
					IF FLF(2) THEN	[000800]	C	
					BEGIN	[000801]	C	
INTERPRET	45C8	AT	[02A00]	ZMZ	OUTPUT DATA.LENGTH(MESS.11) BITS CORE MESS.11 FILE PRINTER OPT	[000802]	C	[02A00]
					END	[000810]	C	
					IF FLF(1) THEN	[000811]	C	
					BEGIN	[000812]	C	
INTERPRET	8B01	AT	[02A90]	ZMZ	OUTPUT DATA.LENGTH(MESS.6) BITS CORE MESS.6 FILE PRINTER OPT	[000813]	C	[02A90]
					END	[000821]	C	
					MOVE YES TO TERMINATION.CODE	[000822]	C	
						[000823]	C	
INTERPRET	2B9C	AT	[02AA0]		SET FLF TO 0	[000823]	G	[02AA0]
INTERPRET	3580	AT	[02AB0]		GO TO -INTERP.LOOP	[000824]	C	[02AB0]
INTERPRET	D125	AT	[02AC0]		END	[000825]	C	[02AC0]
					PAGE	[000826]	C	
						[000827]	C	
						[000828]	C	
						[000829]	C	
						[000830]	C	
						[000831]	C	
						[000832]	C	
						[000833]	C	
						[000834]	C	
						[000834]	C	

SUBROUTINE 9200 AT [02AD0]	MOVE "+0" TO T	% UNPACK AND WRITE	[000835]	C	[02AD0]
SUBROUTINE 4EFO AT [02AE0]				G	[02AE0]
SUBROUTINE 23A5 AT [02AF0]	MOVE S5A TO L	% FIRST 2 BYTES	[000836]	C	[02AF0]
SUBROUTINE 4862 AT [02B00]	IF L(0) THEN MOVE "-0" TO T	% IN CASE OF NEGATIVE NUMBER	[000837]	C	[02B00]
SUBROUTINE 9200 AT [02B10]				G	[02B10]
SUBROUTINE 60F0 AT [02B20]				G	[02B20]
SUBROUTINE 1A05 AT [02B30]	MOVE LC TO TF		[000838]	C	[02B30]
SUBROUTINE 7990 AT [02B40]	WRITE 16 BITS FROM T INC FA		[000839]	C	[02B40]
	MOVE "000" TO T	% UNPACK AND WRITE NEXT 3 BYTES	[000840]	C	
		% PUT 2F0F0F03 IN T.	[000841]	C	[02B50]
SUBROUTINE 92F0 AT [02B50]				G	[02B60]
SUBROUTINE F0F0 AT [02B60]			[000842]	C	[02B70]
SUBROUTINE 1801 AT [02B70]	MOVE LD TO TB		[000843]	C	[02B80]
SUBROUTINE 1C03 AT [02B80]	MOVE LE TO TD		[000844]	C	[02B90]
SUBROUTINE 1D05 AT [02B90]	MOVE LF TO TF		[000845]	C	[02BA0]
SUBROUTINE 7998 AT [02BA0]	WRITE 24 BITS FROM T INC FA	% UNPACK AND WRITE NEXT 3 BYTES	[000846]	C	
			[000847]	C	[02BB0]
SUBROUTINE 2385 AT [02BB0]	MOVE S5B TO L		[000848]	C	[02BC0]
SUBROUTINE 1801 AT [02BC0]	MOVE LA TO TB		[000849]	C	[02BD0]
SUBROUTINE 1903 AT [02BD0]	MOVE LB TO TD		[000850]	C	[02BE0]
SUBROUTINE 1A05 AT [02BE0]	MOVE LC TO TF		[000851]	C	[02BF0]
SUBROUTINE 7998 AT [02BF0]	WRITE 24 BITS FROM T INC FA	% UNPACK AND WRITE LAST 3 BYTES	[000852]	C	
			[000853]	C	[02C00]
SUBROUTINE 1801 AT [02C00]	MOVE LD TO TB		[000854]	C	[02C10]
SUBROUTINE 1C03 AT [02C10]	MOVE LE TO TD		[000855]	C	[02C20]
SUBROUTINE 1D05 AT [02C20]	MOVE LF TO TF		[000856]	C	[02C30]
SUBROUTINE 7898 AT [02C30]	WRITE 24 BITS FROM T	%	[000857]	C	
			[000858]	C	[02C40]
SUBROUTINE 18A4 AT [02C40]	EXIT		[000859]	C	
			[000860]	C	
	% BINARY.TO-DECIMAL	% MAXIMUM VALUE TO BE CONVERTED IS 9999,	[000861]	C	
		% I.E., LSS 2 TO THE 14TH POWER	[000862]	C	
		% THE INPUT ARG IS IN EA (S18)	[000863]	C	
		% THE OUTPUT RESULT IS LEFT IN	[000864]	C	
		% S5 AS A PACKED DECIMAL VALUE	[000865]	C	
			[000866]	C	
	BINARY.TO-DECIMAL		[000867]	C	[02C50]
SUBROUTINE 22B1 AT [02C50]	MOVE EA TO T		[000868]	C	[02C60]
SUBROUTINE 0310 AT [02C60]	CLEAR X		[000869]	C	
		% SET UP FOR DECIMAL ARITH.	[000870]	C	[02C70]
SUBROUTINE 8C38 AT [02C70]	MOVE 3(1)00110003 TO CP		[000871]	C	[02C80]
SUBROUTINE 8A0E AT [02C80]	MOVE 14 TO FL		[000872]	C	
	-LOOP		[000873]	C	[02C90]
SUBROUTINE 4787 AT [02C90]	IF FL NEQ 0 THEN		[000874]	C	
	BEGIN	% MOVE TWICE THE VALUE OF X	[000875]	C	[02CA0]
SUBROUTINE 10A1 AT [02CA0]	MOVE X TO Y	% TO Y	[000876]	C	[02CB0]
SUBROUTINE 10E1 AT [02CB0]	MOVE SUM TO Y	% GET NEXT DIGIT	[000877]	C	[02CC0]
SUBROUTINE B581 AT [02CC0]	EXTRACT 1 BIT FROM T(10) TO X		[000878]	C	[02CD0]
SUBROUTINE A281 AT [02CD0]	SHIFT T LEFT BY 1 BIT	% THE NEW SUM NOW IN X	[000879]	C	[02CE0]
SUBROUTINE 10E0 AT [02CE0]	MOVE SUM TO X		[000880]	C	[02CF0]
SUBROUTINE 06C1 AT [02CF0]	COUNT FL DOWN BY 1		[000881]	C	[02D00]
SUBROUTINE 0008 AT [02D00]	GO TO -LOOP		[000882]	C	
	END	% STORED IN S5	[000883]	C	[02D10]
SUBROUTINE 2FC5 AT [02D10]	MOVE NULL TO S5A				
			[000884]	C	[02D20]
SUBROUTINE 2095 AT [02D20]	MOVE X TO S5B		[000885]	C	
			[000886]	C	[02D30]
SUBROUTINE 18A4 AT [02D30]	EXIT		[000887]	C	
			[000888]	C	
	% COPY	% COPIES A SOURCE MESSAGE,WHOSE ADDRESS IS	[000889]	C	
		% IN FA, TO A SINK AREA,WHOSE ADDRESS IS	[000890]	C	
		% IN S1A.	[000891]	C	
		% LENGTH OF SOURCE IS IN FL.	[000892]	C	
		% USES CPL AND T.	[000893]	C	
		% LEAVES A COPY OF INITIAL VALUE OF FL IN	[000894]	C	
		% SFL.	[000895]	C	

SUBROUTINE 2A90	AT [02D40]	COPY		[000896]	C	
		MOVE FL TO SOB		[000897]	C	[02D40]
		.LOOP		[000898]	C	
SUBROUTINE 5781	AT [02D50]	IF FL EQL 0 THEN EXIT		[000899]	C	[02D50]
SUBROUTINE 1BA4	AT [02D60]				G	[02D60]
SUBROUTINE 0032	AT [02D70]	BIAS BY F	% CPL GETS MIN(24,FL)	[000900]	C	[02D70]
SUBROUTINE 7380	AT [02D80]	READ TO T INC FA AND DEC FL	% READ CPL BITS	[000901]	C	[02D80]
SUBROUTINE 0711	AT [02D90]	XCH S1 F S1		[000902]	C	[02D90]
SUBROUTINE 7980	AT [02DA0]	WRITE FROM T INC FA	% WRITE CPL BITS	[000903]	C	[02DA0]
SUBROUTINE 0711	AT [02DB0]	XCH S1 F S1		[000904]	C	[02DB0]
SUBROUTINE D008	AT [02DC0]	GO TO -LOOP		[000905]	C	[02DC0]
		% END OF COPY		[000906]	C	
		%%		[000907]	C	
		%%		[000908]	C	
		%%		[000909]	C	
		VALIDATE.DECIMAL	%ROUTINE BEGINS HERE. SEE FIG.6-13.	[000910]	C	
			%VALIDATES A SAMOS WORD POINTED TO	[000911]	C	
			%BY THE CONTENTS OF FA AS A DECIMAL	[000912]	C	
			%INTEGER AND PACKS A 4-BIT DECIMAL	[000913]	C	
			%REPRESENTATION IN SS. IF NOT VALID,	[000914]	C	
			%CB(0) IS SET TO 1, ELSE IT IS SET TO 0.	[000915]	C	
			% THIS ROUTINE USES X,Y,T,L, AND CP.	[000916]	C	
		BEGIN		[000917]	C	
		LOCAL.DEFINES		[000918]	C	
		DEFINE FLAG	=CB(0) #	[000919]	C	
		MACRO CHECK.F(TK) =		[000920]	C	
		IF TK NEQ 0 THEN EXIT #		[000921]	C	
				[000922]	C	
SUBROUTINE 0380	AT [02DD0]	CLEAR L		[000923]	C	[02DD0]
SUBROUTINE 3728	AT [02DE0]	SET FLAG	% TO NOGOOD	[000924]	C	[02DE0]
SUBROUTINE 7190	AT [02DF0]	READ 16 BITS TO T INC FA	%GET FIRST 2 BYTES IN TC THRU TF	[000925]	C	[02DF0]
			%BOXES 4,5,	[000926]	C	
			%AND 6	[000927]	C	
SUBROUTINE B808	AT [02E00]	EXTRACT 8 BITS FROM T(8) TO X		[000928]	C	[02E00]
SUBROUTINE 814E	AT [02E10]	MOVE "+" TO Y		[000929]	C	[02E10]
SUBROUTINE 5CC4	AT [02E20]	IF X NEQ Y THEN		[000930]	C	[02E20]
		BEGIN	% TRY "--"	[000931]	C	
SUBROUTINE 8160	AT [02E30]	MOVE "--" TO Y		[000932]	C	[02E30]
SUBROUTINE 5CC1	AT [02E40]	IF X NEQ Y THEN EXIT		[000933]	C	[02E40]
SUBROUTINE 1BA4	AT [02E50]			[000934]	C	[02E50]
SUBROUTINE 3808	AT [02E60]	MOVE 2(1)10002 TO LA		[000935]	C	[02E60]
		END		[000936]	C	
				[000937]	C	
SUBROUTINE 150A	AT [02E90]	# CHECK.F(TE)		[000938]	C	[02E90]
		MOVE TF TO LC		[000939]	C	
SUBROUTINE 7198	AT [02EA0]	READ 24 BITS TO T INC FA		[000940]	C	[02EA0]
		CHECK.F(TA)	%CHECK AND PACK INTO L	[000941]	C	
		CHECK.F(TC)		[000942]	C	
		CHECK.F(TE)		[000943]	C	
SUBROUTINE 1108	AT [02F10]	MOVE TB TO LD		[000944]	C	[02F10]
SUBROUTINE 130C	AT [02F20]	MOVE TD TO LE		[000945]	C	[02F20]
SUBROUTINE 150D	AT [02F30]	MOVE TF TO LF		[000946]	C	[02F30]
SUBROUTINE 2385	AT [02F40]	MOVE L TO SSA		[000947]	C	[02F40]
SUBROUTINE 7198	AT [02F50]	READ 24 BITS TO T INC FA		[000948]	C	[02F50]
		CHECK.F(TA)	%CHECK AND PACK INTO L	[000949]	C	
		CHECK.F(TC)		[000950]	C	
		CHECK.F(TE)		[000951]	C	
SUBROUTINE 1108	AT [02FC0]	MOVE TB TO LA		[000952]	C	[02FC0]
SUBROUTINE 1309	AT [02FD0]	MOVE TD TO LB		[000953]	C	[02FD0]
SUBROUTINE 150A	AT [02FE0]	MOVE TF TO LC		[000954]	C	[02FE0]
SUBROUTINE 7098	AT [02FF0]	READ 24 BITS TO T		[000955]	C	[02FF0]

```

# CHECK.F(IA) ZCHECK AND PACK INIU L [000956] C
# CHECK.F(IC) [000957] C
# CHECK.F(TE) [000958] C
SUBROUTINE 110B AT [03060] MOVE TB TO LD [000959] C [03060]
SUBROUTINE 130C AT [03070] MOVE TD TO LE [000960] C [03070]
SUBROUTINE 150D AT [03080] MOVE TF TO LF [000961] C [03080]
SUBROUTINE 2395 AT [03090] MOVE L TO S5B [000962] C [03090]
SUBROUTINE 8C38 AT [030A0] MOVE 2(1)001110002 TO CP ZBOX 10.1 ZBOX10 [000963] C [030A0]
SUBROUTINE 20A5 AT [030B0] MOVE S5A TO X Z 10.2 [000964] C [030B0]
SUBROUTINE 0320 AT [030C0] CLEAR Y Z 10.3 [000965] C [030C0]
SUBROUTINE 10E1 AT [030D0] MOVE SUM TO Y [000966] C [030D0]
SUBROUTINE 5CC1 AT [030E0] IF X NEQ Y THEN EXIT [000967] C [030E0]
SUBROUTINE 18A4 AT [030F0] [000968] C [030F0]
SUBROUTINE 20B5 AT [03100] MOVE S5B TO X ZBOX 10.4 [000969] C [03100]
SUBROUTINE 0320 AT [03110] CLEAR Y ZBOX 10.5 [000970] C [03110]
SUBROUTINE 10E1 AT [03120] MOVE SUM TO Y [000971] C [03120]
SUBROUTINE 5CC1 AT [03130] IF X NEQ Y THEN EXIT [000972] C [03130]
SUBROUTINE 18A4 AT [03140] [000973] C [03140]
SUBROUTINE 3717 AT [03150] RESET FLAG ZBOX11 [000974] C [03150]
SUBROUTINE 18A4 AT [03160] EXIT [000975] C [03160]
END Z OF VALIDATE.DECIMAL ROUTINE
BINARY.TO.FA ZROUTINE BEGINS HERE SEE FIG. 6-15.
ZINPUT VALUE,S, IS IN T REGISTER
ZOUTPUT RESULT IS IN FA
ZUSES X,Y,CB(O),AND L AS LOCAL STORAGE
BEGIN
LOCAL.DEFINES
DEFINE NEG.SIGN =CB(O)#
DEFINE NEW.S =L#
SUBROUTINE 4063 AT [03170] IF T(O) THEN ZSAVE SIGN OF S ZBOX1 [000984] C [03170]
SUBROUTINE 3728 AT [03180] BEGIN ZAND REPLACE S [000985] C [03180]
SUBROUTINE 3017 AT [03190] SET NEG.SIGN ZBY ABS VALUE OF S [000986] C [03190]
SUBROUTINE C001 AT [031A0] RESET T(O) ZIS THIS INSTRUCTION NECESSARY? [000987] C [031A0]
SUBROUTINE 3717 AT [031B0] END ELSE [000988] C [031B0]
SUBROUTINE 8C18 AT [031C0] BEGIN RESET NEG.SIGN [000989] C [031C0]
SUBROUTINE A083 AT [031D0] END [000990] C [031D0]
SUBROUTINE A184 AT [031E0] MOVE 24 TO CP ZSET UP FOR ARITHMETIC [000991] C [031E0]
SUBROUTINE 10E0 AT [031F0] SHIFT T LEFT BY 3 BITS TO X Z8 TIMES S TO X(SIGN BIT LOST) [000992] C [031F0]
SUBROUTINE A186 AT [03200] SHIFT T LEFT BY 4 BITS TO Y Z16 TIMES S TO Y [000993] C [03200]
SUBROUTINE 10E3 AT [03210] MOVE SUM TO X Z24 TIMES S TO X [000994] C [03210]
SUBROUTINE 16A0 AT [03220] SHIFT T LEFT BY 6 BITS TO Y Z64 TIMES S TO Y [000995] C [03220]
SUBROUTINE 9100 AT [03230] MOVE SUM TO NEW.S Z88 TIMES S IN NEW.S (L) [000996] C [03230]
SUBROUTINE 0DB0 AT [03240] MOVE BR TO X [000997] C [03240]
SUBROUTINE 10E0 AT [03250] MOVE SAMOS.STORE(O) TO Y [000998] C [03250]
SUBROUTINE 13A1 AT [03260] [000999] C [03260]
SUBROUTINE 4762 AT [03270] ZBR + SAMOS.STORE IN X [001000] C [03270]
SUBROUTINE 18E8 AT [03280] IF NEG.SIGN THEN ZBOX5 [001001] C [03280]
SUBROUTINE C001 AT [03290] BEGIN MOVE DIFF TO FA [001002] C [03290]
SUBROUTINE 10E8 AT [032A0] END ELSE [001003] C [032A0]
SUBROUTINE 18A4 AT [032B0] BEGIN MOVE SUM TO FA [001004] C [032B0]
EXIT [001005] C
END Z BINARY.TO.FA ROUTINE [001006] C
Z [001007] C
[001008] C
[001009] C
[001010] C
[001011] C
[001012] C
[001013] C

```

	ADDRESS.TO.BINARY	ZROUTINE BEGINS HERE. SEE FIG 6-17.	[001014]	C	
		ZINPUT VALUE,S, IS IN T REGISTER	[001015]	C	
		ZOUTPUT RESULTS BINARY ADDRESS IN T	[001016]	C	
		Z FLAG IN Y	[001017]	C	
		Z 0 IF INVALID	[001018]	C	
		Z 1 IF VALID	[001019]	C	
		ZUSES X,Y,L,FA AS LOCAL STORAGE	[001020]	C	
			[001021]	C	
			[001022]	C	
			[001023]	C	
			[001024]	C	
			[001025]	C	
			[001026]	C	
			[001027]	C	
			[001028]	C	
			[001029]	C	
			[001030]	C	
			[001031]	C	
			[001032]	C	
			[001033]	C	
			[001034]	C	
			[001035]	C	
			[001036]	C	
			[001037]	C	
			[001038]	C	
			[001039]	C	
			[001040]	C	
			[001041]	C	
			[001042]	C	
			[001043]	C	
			[001044]	C	
			[001045]	C	
			[001046]	C	
			[001047]	C	
			[001048]	C	
			[001049]	C	
			[001050]	C	
			[001051]	C	
			[001052]	C	
			[001053]	C	
			[001054]	C	
			[001055]	C	
			[001056]	C	
			[001057]	C	
			[001058]	C	
			[001059]	C	
			[001060]	C	
			[001061]	C	
			[001062]	C	
			[001063]	C	
			[001064]	C	
			[001065]	C	
			[001066]	C	
			[001067]	C	
			[001068]	C	
			[001069]	C	
			[001070]	C	
			[001071]	C	
			[001072]	C	
			[001073]	C	
			[001074]	C	
			[001075]	C	
			[001062]	C	
			[001063]	C	
			[001064]	C	
			[001065]	C	
			[001066]	C	
			[001067]	C	
			[001068]	C	
			[001069]	C	
			[001070]	C	
			[001071]	C	
			[001072]	C	
			[001073]	C	
			[001074]	C	
			[001075]	C	
			[001062]	C	
			[001063]	C	
			[001064]	C	
			[001065]	C	
			[001066]	C	
			[001067]	C	
			[001068]	C	
			[001069]	C	
			[001070]	C	
			[001071]	C	
			[001072]	C	
			[001073]	C	
			[001074]	C	
			[001075]	C	
			[001062]	C	
			[001063]	C	
			[001064]	C	
			[001065]	C	
			[001066]	C	
			[001067]	C	
			[001068]	C	
			[001069]	C	
			[001070]	C	
			[001071]	C	
			[001072]	C	
			[001073]	C	
			[001074]	C	
			[001075]	C	
			[001062]	C	
			[001063]	C	
			[001064]	C	
			[001065]	C	
			[001066]	C	
			[001067]	C	
			[001068]	C	
			[001069]	C	
			[001070]	C	
			[001071]	C	
			[001072]	C	
			[001073]	C	
			[001074]	C	
			[001075]	C	
			[001062]	C	
			[001063]	C	
			[001064]	C	
			[001065]	C	
			[001066]	C	
			[001067]	C	
			[001068]	C	
			[001069]	C	
			[001070]	C	
			[001071]	C	
			[001072]	C	
			[001073]	C	
			[001074]	C	
			[001075]	C	
			[001062]	C	
			[001063]	C	
			[001064]	C	
			[001065]	C	
			[001066]	C	
			[001067]	C	
			[001068]	C	
			[001069]	C	
			[001070]	C	
			[001071]	C	
			[001072]	C	
			[001073]	C	
			[001074]	C	
			[001075]	C	
			[001062]	C	
			[001063]	C	
			[001064]	C	
			[001065]	C	
			[001066]	C	
			[001067]	C	
			[001068]	C	
			[001069]	C	
			[001070]	C	
			[001071]	C	
			[001072]	C	
			[001073]	C	
			[001074]	C	
			[001075]	C	
			[001062]	C	
			[001063]	C	
			[001064]	C	
			[001065]	C	
			[001066]	C	
			[001067]	C	
			[001068]	C	
			[001069]	C	
			[001070]	C	
			[001071]	C	
			[001072]	C	
			[001073]	C	
			[001074]	C	
			[001075]	C	
			[001062]	C	
			[001063]	C	
			[001064]	C	
			[001065]	C	
			[001066]	C	
			[001067]	C	
			[001068]	C	
			[001069]	C	
			[001070]	C	
			[001071]	C	
			[001072]	C	
			[001073]	C	
			[001074]	C	
			[001075]	C	
			[001062]	C	
			[001063]	C	
			[001064]	C	
			[001065]	C	
			[001066]	C	
			[001067]	C	
			[001068]	C	
			[001069]	C	
			[001070]	C	
			[001071]	C	
			[001072]	C	
			[001073]	C	
			[001074]	C	
			[001075]	C	
			[001062]	C	
			[001063]	C	
			[001064]	C	
			[001065]	C	
			[001066]	C	
			[001067]	C	
			[001068]	C	
			[001069]	C	
			[001070]	C	
			[001071]	C	
			[001072]	C	
			[001073]	C	
			[001074]	C	
			[001075]	C	
			[001062]	C	
			[001063]	C	
			[001064]	C	
			[001065]	C	
			[001066]	C	
			[001067]	C	
			[001068]	C	
			[001069]	C	
			[001070]	C	
			[001071]	C	
			[001072]	C	
			[001073]	C	
			[001074]	C	
			[001075]	C	
			[001062]	C	
			[001063]	C	
			[001064]	C	
			[001065]	C	
			[001066]	C	
			[001067]	C	
			[001068]	C	
			[001069]	C	
			[001070]	C	
			[001071]	C	
			[001072]	C	
			[001073]	C	
			[001074]	C	
			[001075]	C	
			[001062]	C	
			[001063]	C	
			[001064]	C	
			[001065]	C	
			[001066]	C	
			[001067]	C	
			[001068]	C	
			[001069]	C	
			[001070]	C	
			[001071]	C	
			[001072]	C	
			[001073]	C	
			[001074]	C	
			[001075]	C	
			[001062]	C	
			[001063]	C	
			[001064]	C	
			[001065]	C	
			[001066]	C	
			[001067]	C	
			[001068]	C	
			[001069]	C	
			[001070]	C	
			[001071]	C	
			[001072]	C	
			[001073]	C	
			[001074]	C	
			[001075]	C	

	#	TEN.T.PLUS.D(LO)	ZMACRO CALL	[001130]	C	
	#	TEN.T.PLUS.D(LE)	ZMACRO CALL	[001131]	C	
	#	TEN.T.PLUS.D(LF)	ZMACRO CALL	[001132]	C	
				ZBOXES 10		
				ZTHRU 14		
SUBROUTINE	12A0	AT	[038F0]	[001133]	C	
SUBROUTINE	21B1	AT	[03900]	[001134]	C	[038F0]
SUBROUTINE	10E0	AT	[03910]	[001135]	C	[03900]
SUBROUTINE	8164	AT	[03920]	[001136]	C	[03910]
SUBROUTINE	4CA2	AT	[03930]	[001137]	C	[03920]
		MOVE T TO X		[001138]	C	[03930]
		MOVE TEMP TO Y		[001139]	C	
		MOVE SUM TO X		[001140]	C	
		MOVE SIZE TO Y		[001141]	C	
		IF X LSS Y THEN		[001142]	C	
		BEGIN		[001143]	C	
		SET FLAG TO 0		[001144]	C	[03940]
SUBROUTINE	3580	AT	[03940]	[001145]	C	[03940]
SUBROUTINE	C001	AT	[03950]	[001146]	C	[03950]
		END ELSE		[001147]	C	
		BEGIN	ZBAD OPERAND ADDR	[001148]	C	
		SET FLAG TO 4		[001149]	C	[03960]
SUBROUTINE	3584	AT	[03960]	[001150]	C	[03960]
		END		[001151]	C	
		MOVE X TO EA	ZEFFECTIVE ADDRESS SAVED IN EA AS A BINARY VALUE	[001152]	C	[03970]
SUBROUTINE	2091	AT	[03970]	[001153]	C	[03970]
SUBROUTINE	1BA4	AT	[03980]	[001154]	C	[03980]
		EXIT		[001155]	C	
		END Z	OF EFFECTIVE.ADDR ROUTINE	[001156]	C	
	Z	ADD.10.COMPL	ZROUTINE BEGINS HERE. SEE FIG. 6-29. ARGUMENTS ARE	[001157]	C	
			Z OP1 IN S5, OP2 IN S6. IN SIGNED MAGNITUDE FORM	[001158]	C	
			Z AS IN FIG.6-7. THE RESULT IS LEFT IN OP1,IE,IN S5,	[001159]	C	
			Z AND OVERFLOW FLAG IN CB(0).	[001160]	C	
			Z THE STACK,X,Y, T, AND L ARE USED AS LOCAL STORAGE.	[001161]	C	
			Z THE PROCEDURE COMP.T.L CONVERTS T CAT L TO 10'S	[001162]	C	
			Z COMPLEMENT FORM	[001163]	C	
		BEGIN		[001164]	C	
		LOCAL.DEFINES		[001165]	C	
		DEFINE OP1A	=S5A #	[001166]	C	
		DEFINE OP1B	=S5B #	[001167]	C	
		DEFINE OP2A	=S6A #	[001168]	C	
		DEFINE OP2B	=S6B #	[001169]	C	
		DEFINE FLAG	=CB(0)#	[001170]	C	
	Z	MOVE a(1)00111000a TO CP	ZSETUP FOR 24-BIT	[001171]	C	[03990]
SUBROUTINE	8C38	AT	[03990]	[001172]	C	[03990]
			ZDECIMAL ARITHMETIC	[001173]	C	
			ZBOX1	[001174]	C	
SUBROUTINE	22A5	AT	[039A0]	[001175]	C	[039A0]
SUBROUTINE	23B5	AT	[039B0]	[001176]	C	[039B0]
SUBROUTINE	4062	AT	[039C0]	[001177]	C	[039C0]
		MOVE OP1A TO T		[001178]	C	
		MOVE OP1B TO L		[001179]	C	
		IF T(0) THEN	ZIF OP1 NEGATIVE	[001180]	C	
		BEGIN	ZCOMPLEMENT ABS OF OP1	[001181]	C	
		RESET T(0)		[001182]	C	[039D0]
SUBROUTINE	3017	AT	[039D0]	[001183]	C	[039D0]
SUBROUTINE	E019	AT	[039E0]	[001184]	C	[039E0]
		CALL COMPL.T.L		[001185]	C	
		END		[001186]	C	
		MOVE T TO TAS	ZSAVE OP1 ON STACK	[001187]	C	[039F0]
SUBROUTINE	12AB	AT	[039F0]	[001188]	C	[039F0]
SUBROUTINE	13AB	AT	[03A00]	[001189]	C	[03A00]
SUBROUTINE	22A6	AT	[03A10]	[001190]	C	[03A10]
SUBROUTINE	23B6	AT	[03A20]	[001191]	C	[03A20]
SUBROUTINE	4062	AT	[03A30]	[001192]	C	[03A30]
		MOVE OP2A TO T		[001193]	C	
		MOVE OP2B TO L		[001194]	C	
		IF T(0) THEN	ZIF OP2 NEGATIVE	[001195]	C	
		BEGIN	ZCOMPLEMENT ABS OF OP2	[001196]	C	
		RESET T(0)		[001197]	C	[03A40]
SUBROUTINE	3017	AT	[03A40]	[001198]	C	[03A40]
SUBROUTINE	E012	AT	[03A50]	[001199]	C	[03A50]
		CALL COMPL.T.L		[001200]	C	
		END		[001201]	C	
			ZBOX3	[001202]	C	
		MOVE TAS TO X	ZLOW-ORDER PARTS OF OP1 AND	[001203]	C	[03A60]
SUBROUTINE	1BA0	AT	[03A60]	[001204]	C	[03A60]
SUBROUTINE	13A1	AT	[03A70]	[001205]	C	[03A70]
SUBROUTINE	10E3	AT	[03A80]	[001206]	C	[03A80]
SUBROUTINE	0064	AT	[03A90]	[001207]	C	[03A90]
SUBROUTINE	1BA0	AT	[03AA0]	[001208]	C	[03AA0]
SUBROUTINE	12A1	AT	[03AB0]	[001209]	C	[03AB0]
SUBROUTINE	10E2	AT	[03AC0]	[001210]	C	[03AC0]
		MOVE T TO Y	ZOP2 IN X AND Y, RESP.	[001211]	C	
		MOVE SUM TO T	ZHIGH-ORDER PART OF OP1+OP2 IN T	[001212]	C	
			ZBOX4	[001213]	C	

SUBROUTINE 4062 AT [03AD0]	IF T(0) THEN	ZIF LEADING DIGIT IS 9 OR 8	[001192] C	[03AD0]
SUBROUTINE E009 AT [03AE0]	BEGIN	ZTHEN	[001193] C	
SUBROUTINE 3028 AT [03AF0]	CALL COMPL.T.L	Z COMPLEMENT IT AND	[001194] C	[03AE0]
	SET T(0)	Z MARK IT MINUS	[001195] C	[03AF0]
	END		[001196] C	
SUBROUTINE 6160 AT [03B00]	IF TB NEQ 0 THEN	ZIF 11TH DIGIT OF SUM NON-ZERO	[001197] C	[03B00]
SUBROUTINE C002 AT [03B10]	BEGIN	ZTHE WE HAVE OVERFLOW	[001198] C	[03B10]
SUBROUTINE 3728 AT [03B20]	SET FLAG		[001199] C	
SUBROUTINE C001 AT [03B30]	END ELSE		[001200] C	[03B20]
SUBROUTINE 3717 AT [03B40]	BEGIN		[001201] C	[03B30]
	RESET FLAG		[001202] C	
	END		[001203] C	[03B40]
SUBROUTINE 2285 AT [03B50]	MOVE T TO OP1A	ZNEW RESULT LEFT IN OP1A	[001204] C	
SUBROUTINE 2395 AT [03B60]	MOVE L TO OP1B	Z AND OP1B	[001205] C	[03B50]
SUBROUTINE 18A4 AT [03B70]	EXIT		[001206] C	[03B60]
	END Z OF ADD.10-COMPL ROUTINE		[001207] C	[03B70]
			[001208] C	
			[001209] C	
			[001210] C	
			[001211] C	
	Z	ZPROCEDURE COMPUTES THE 10'S COMPLEMENT OF T CAT L	[001212] C	
	Z	Z AND LEAVES THE RESULT IN T CAT L	[001213] C	
		Z USING X AND Y AS LOCAL STORAGE	[001214] C	
SUBROUTINE 8C38 AT [03B80]	MOVE Z(1)00111000 TO CP	ZTO BE SURE OF ARITH. SETUP	[001215] C	[03B80]
SUBROUTINE 0310 AT [03B90]	CLEAR X	ZMORE SETUP	[001216] C	[03B90]
SUBROUTINE 13A1 AT [03BA0]	MOVE L TO Y		[001217] C	[03BA0]
SUBROUTINE 18E3 AT [03BB0]	MOVE DIFF TO L	ZLOW-ORDER PART OF COMPL. IN L	[001218] C	[03BB0]
SUBROUTINE 0068 AT [03BC0]	CARRY DIFFERENCE	ZRECYCLE THE BORROW	[001219] C	[03BC0]
SUBROUTINE 12A1 AT [03BD0]	MOVE T TO Y		[001220] C	[03BD0]
SUBROUTINE 18E2 AT [03BE0]	MOVE DIFF TO T	ZCOMPLEMENT NOW IN T CAT L	[001221] C	[03BE0]
SUBROUTINE 0061 AT [03BF0]	CARRY 0	ZLEAVE CARRY IN 'CLEAN' STATE	[001222] C	[03BF0]
SUBROUTINE 18A4 AT [03C00]	EXIT		[001223] C	[03C00]
	ZEND OF COMPL.T.L		[001224] C	
	END		[001225] C	
	FINI			

NUMBER OF ERRORS DETECTED = 000
NUMBER OF WARNING MESSAGES = 000
MICRO INSTRUCTION COUNT = 00965

CAUTION: \$ SUBSET WAS NOT SPECIFIED; THEREFORE, THIS PROGRAM SHOULD NOT BE USED ON A B1712/B1714.

6 OUTPUT PRODUCED BY THE SAMOS INTERPRETER WHEN PROCESSING THE DATA DECK GIVEN IN PART 5

```

INTRP          SAMOS      INTERP
BEGIN BATCH RUN FOR SAMOS

* FIBONNACI A LA CS 105
+LDA0000017   THIS IS FIBONNACI
+STO0000050   NEXT
+LDA0000018
+STO0000051   LATEST
+LDA0000051   LATEST
+ADD0000050   NEXT
+STO0000052   SUM
+LDA0000052   SUM
+ADD0000019   (NEGATIVE OF LIMIT PLUS ONE)
+BMI0000012   TO BOX 4A
+HWD0000052   PRINT SUM
+HLT0000000
+LDA0000051   THIS IS BOX4A
+STO0000050
+LDA0000052
+STO0000051
+BRU00000004   TO BOX2
+0000000000   (0)
+0000000001   (1)
-0000000009   (-9)
+0000000013
NORMAL HALT
EOF ON INPUT

```

7 SEQUENCE OF ENHANCEMENTS FOR THE SKELETAL VERSION OF THE SAMOS INTERPRETER

The interpreter displayed in this appendix may be enhanced or modified by a series of interesting exercises. The following are exercises that were assigned to students at the University of Utah in an undergraduate software laboratory course, Spring quarter 1976. Readers having access to a B1726 may wish to consider working similar problems.

1. The ADD instruction in the base interpreter does not take advantage of the overflow-sensing ability of the ADD.10.COMPL routine. Modify the ADD instruction coding to abort program execution upon sensing accumulator overflow.
2. Extend the instruction set of the base interpreter by coding the SAMOS subtract instruction. (No more than about six new microinstructions are needed.)
3. Add a dump routine to the interpreter which is called whenever a fatal error is encountered in executing a SAMOS program (e.g., INVALID OPERAND). The dump routine should produce a pretty-print display of all the SAMOS registers and all the storage cells. The name of each register should be displayed above or beside its content. In dumping the SAMOS storage, display the contents of at least 5 SAMOS storage cells per line. Display at the extreme left of each such line the location of the leftmost cell shown on that line. As a further enhancement of the dump, display the

number of SAMOS instructions that have been executed, the number of data cards that were read, and the number of lines printed.

4. Enhance the appearance of the output produced by the LOAD.A.PROGRAM module of the interpreter so the *location* of each loaded instruction appears to the left of each displayed instruction. Skip a line after the last loaded instruction word is displayed and/or display a line like

SAMOS PROGRAM HAS BEEN LOADED.

5. Extend the instruction set supported by the interpreter to include MPY, DIV, the shift instructions SHL and SHR, and the indexing instructions, LIX, SIX, and TIX (load index, store index, and test index instructions). See Appendix F for definitions of these instructions.
6. The base interpreter wastes space in representing storage words. It uses 88 bits per word [11 EBCDIC characters], but a SAMOS word as originally defined uses 61 bits per word [a sign bit and 10 BCD (6-bit) characters]. Modify the utility routines and all other code required to represent SAMOS storage words in 61 bits (and also modify the representation of the SAMOS registers in a similar way). After completing these modifications, discuss the trade-offs (space versus time) between the “88-bit” and the “61-bit” interpreter.
7. The interpreter as presented in the base case suffers in speed because SAMOS registers are represented in G-store. Some or all of the SAMOS registers might instead be represented in scratchpads with increased speed of interpretation—provided, of course, there is sufficient scratchpad space available. Note that there is a better chance to find the space needed in the scratchpads if the index registers are represented faithfully as four 6-bit characters rather than as 88-bit or even 61-bit words (see preceding exercise).

Make all the necessary modifications so as to represent as many of the SAMOS registers as possible in the scratchpads. Design and run speed tests to determine the increase in speed achieved by these modifications for a set of representative SAMOS programs. How much bigger or smaller (number of microinstructions) is the modified interpreter?

Appendix F





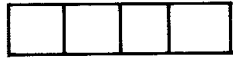
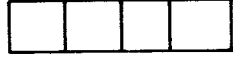
The SAMOS computer

SAMOS is a hypothetical computer, introduced for pedagogical purposes in 1965 in the introductory text *Algorithms, Computation and Mathematics*, produced for high-school students by the School Math Study Group at Stanford University. SAMOS appeared again in some college-level texts, also for pedagogical purposes.¹

It has been common to simulate SAMOS at institutions where students are asked to write a few practice programs in SAMOS machine language or in an equivalent symbolic assembly language. SAMOS simulators often take the form of programs coded in FORTRAN, BASIC or similar high-level languages.

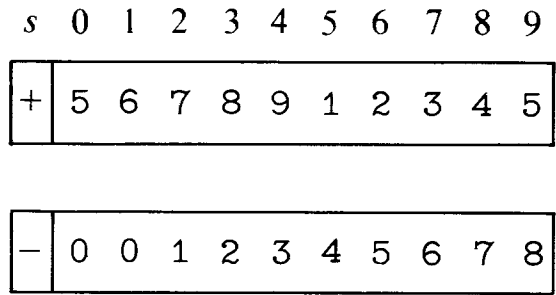
SAMOS is a one-address von Neumann-style computer. Its storage-address space ranges from 0000 to 9999. Each storage word is ten characters in length. There is also an eleventh (sign) position in each storage word for use in representing signed, ten-decimal-digit integers. Figure F.1 illustrates the use of SAMOS storage words for representing integers, ten-character strings, and SAMOS instructions.

The control unit of SAMOS contains the following six registers.

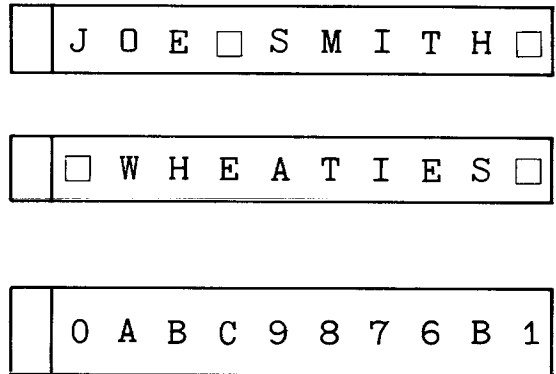
1. Instruction counter	IC		4 decimal digits
2. Operation register	OR		3 characters
3. Address register	AR		4 decimal digits
4. Index registers (3)	R1		} 4 decimal digits each
	R2		
	R3		

¹ The most recent full description is in A. I. Forsythe et al., *Computer Science: A First Course* (2nd Ed., Wiley, 1975).

Decimal integer numbers



Character strings



SAMOS instructions

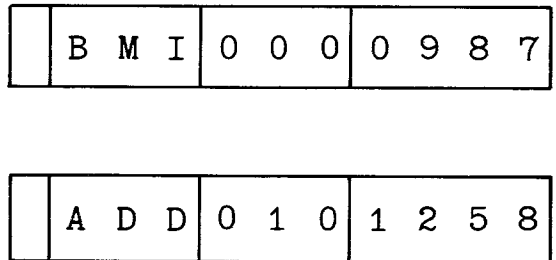
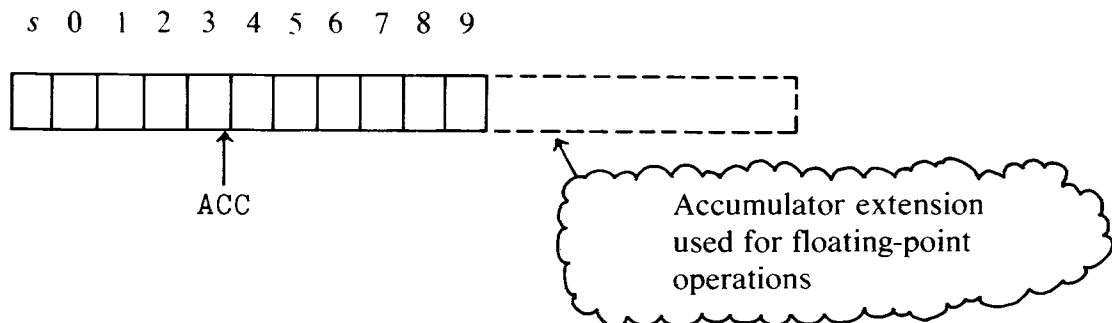


Figure F.1.

The processing unit of SAMOS contains only a single register, the accumulator, or ACC, as shown.



The sign position, s , participates only in arithmetic operations whose operands must be signed decimal integers (ADD, SUB, MPY, and DIV; see Table F.1 below). For example, the instruction

s	0	1	2	3	4	5	6	7	8	9											
<table style="border-collapse: collapse; width: 100%; border: none;"> <tr> <td style="border: 1px solid black; width: 10%;"></td> <td style="border: 1px solid black; width: 10%; text-align: center;">A</td> <td style="border: 1px solid black; width: 10%; text-align: center;">D</td> <td style="border: 1px solid black; width: 10%; text-align: center;">D</td> <td style="border: 1px solid black; width: 10%; text-align: center;">0</td> <td style="border: 1px solid black; width: 10%; text-align: center;">1</td> <td style="border: 1px solid black; width: 10%; text-align: center;">0</td> <td style="border: 1px solid black; width: 10%; text-align: center;">1</td> <td style="border: 1px solid black; width: 10%; text-align: center;">2</td> <td style="border: 1px solid black; width: 10%; text-align: center;">5</td> <td style="border: 1px solid black; width: 10%; text-align: center;">8</td> </tr> </table>												A	D	D	0	1	0	1	2	5	8
	A	D	D	0	1	0	1	2	5	8											

means: add to the contents of the ACC the value copied from the storage word whose (effective) address is 1258 plus the contents of index register R2, mod 10^4 . The sign position of the ADD instruction is ignored, but the operands referred to by the ADD instruction, namely the ACC register and the value found at the effective address, must be properly signed decimal integer. A nonzero digit in position 3, 4, or 5 indicates that index registers R1, R2, or R3, respectively, “participates” in the computation of the effective address. Thus, if the contents of R2 in the above ADD instruction were 8889, the effective address would be $(1258 + 8889) \bmod 10^4$ or 0147.² In most simulated versions of SAMOS more than one nonzero digit in positions 3, 4, and 5 of an instruction is regarded as an error, implying that only one index register may participate in the interpretation of a SAMOS instruction. (But note that some SAMOS simulators permit all three index registers to participate, in which case they participate in an *additive* fashion.)

1 THE SAMOS EXECUTION CYCLE

Steps in the execution cycle of the SAMOS machine may be described as follows

1. Fetch a copy of the instruction whose storage location is given by the IC (instruction counter) register.
2. The operation code of the fetched instruction (positions 0, 1, 2) is assigned to the OR (operation) register; the address field of the fetched instruction (positions 6, 7, 8, 9) is assigned to the AR (address) register. If there is one nonzero digit in position 3, 4, or 5, the index register R1, R2, or R3 respectively is enabled.
3. The IC register is incremented by 1.
4. The instruction, whose op-code is defined by the contents of the OR register, and whose effective address is defined by the contents of the AR register and the enabled index register, if any, is executed according to the semantics given in Table F.1.

² If a SAMOS computer is simulated with a smaller storage size (for example, one with only 200 storage words), it is appropriate to simulate an error condition, signaling an out-of-bounds address, whenever the effective address exceeds 200.

2 INSTRUCTION REPERTOIRE

Table F.1 uses the following notation to express the semantics of SAMOS instructions.

Parentheses surrounding a register name or storage location signify the *contents* of that register or storage location.

Thus, (ACC) means the contents of ACC; (0151) means the contents of storage location 0151. Also, (ACC) ← (0151) means that the contents of the ACC become a copy of the contents of storage location 0151. Moreover,

$$(ACC) \leftarrow (0153 + (R2))$$

means that the contents of the ACC become a copy of the contents of storage location $0153 + (R2)$, i.e., of storage location 0153 plus the contents index register R2. Note that $(0153 + (R2))$ is actually a shorthand in this case for $\{(0153 + (R2))\} \bmod 10^4$. It is to be understood that the effective address is always computed mod 10^4 .

3 ERROR CONDITIONS

Error conditions are sensed as follows:

1. Two or more nonzero digits or characters are present in positions 3, 4, 5 of a fetched instruction. (*Exception:* Such an instruction is acceptable when the opcode is SHL or SHR.)
2. Nondigits in positions 6, 7, 8, 9 of the fetched instruction. (*Exception:* such an instruction is acceptable when the opcode is HLT.)
3. Either the ACC or the operand fetched from the effective address of an instruction about to be executed is not a signed decimal integer when the opcode is ADD, SUB, MPY, or DIV.

4 THE SAMOS LOADER

The LOAD button on the console of the SAMOS computer is used to load a program and to transfer control to that program. Pressing the LOAD button amounts to invoking a built-in (primitive) load instruction whose semantics are as follows.

The computer reads successive cards one after another until a card is read that is blank in columns 1 through 11 inclusive. For each nonblank

TABLE F.1 The SAMOS Instruction Repertoire

OP-CODE	EXAMPLE INSTRUCTION	MEANING
LDA	LDA 000 0151	$(ACC) \leftarrow (0151)$
STO	STO 001 0041	$(0041) + (R3) \leftarrow (ACC)$
ADD ^a	ADD 200 1001	$(ACC) \leftarrow (ACC) + (1001 + (R1))$
SUB ^a	SUB 000 1011	$(ACC) \leftarrow (ACC) - (1011)$
MPY ^b	MPY 010 0049	$(ACC) \leftarrow (ACC) \times (0049 + (R2))$
DIV ^c	DIV 000 1079	$(ACC) \leftarrow (ACC) / (1079)$
HLT	HLT 000 2011	The machine stops. If the START button on the operator's console is then pressed, the next instruction will be taken from location 2011; i.e., resumption after the halt is specified by $(IC) \leftarrow 2011$.
BRU	BRU 000 1108	Branch unconditionally to 1108, i.e., $(IC) \leftarrow 1108$.
BMI	BMI 001 1045	Branch on minus sign in the ACC, i.e., if sign of ACC is "-" then $(IC) \leftarrow 1045 + (R3)$.
RWD	RWD 000 0056	Read the next card and assign the data value in the first 11 columns of the card (sign and 10 characters) to location 0056; i.e., $(0056) \leftarrow$ (first 11 columns of the next data card).
WWD	WWD 010 0059	Write on a new output line a copy of the value found at location $0059 + (R2)$.
SHL ^d	SHL 100 0006 ^e	Shift the accumulator value <i>left</i> 6 positions, with zero right fill. The sign position of the ACC is left unchanged. (Positions 3, 4, 5 of the instructions are ignored.)
SHR ^d	SHR 000 0004 ^e	Shift the accumulator value <i>right</i> 4 positions, with zero left fill. The sign position of the ACC is left unchanged.
LI1 LI2 LI3	LI2 000 1741	Load index register R2 with a copy of the address part (character positions 6, 7, 8, 9) of the value at storage location 1741; $(R2) \leftarrow (1741)_{6-9}$
SI1 SI2 SI3	SI3 000 2229	Store a copy of the value in index register R3 in the address part (character positions 6, 7, 8, 9) of storage location 2229; i.e., $(2229)_{6-9} \leftarrow (R3)$. The remainder of the storage word at location 2229 is left unchanged.
TI1 TI2 TI3	TI1 000 7711	Index register R1 is decremented by 1. Then, if the resulting value is zero, the IC is assigned 7711.

^a Overflow digits are lost, but if overflow occurs, the machine halts.

^b The lowest-order 10 digits of the product go to the ACC. Overflow digits are lost, but if overflow occurs, the machine halts.

^c The integer quotient goes to the ACC. Digits of the remainder, if any, are lost. Attempt to divide with a zero divisor leaves the ACC unchanged, and the machine halts.

^d If the number of shift positions exceeds 9, the ACC will contain 0 (10 zero digits) upon completion of a SHL or a SHR instruction.

card read, the contents of columns 1 through 11 (corresponding to the positions $s, 0, 1, \dots, 8, 9$ of a storage word) are assigned to successive storage locations beginning with storage location 0000. When the blank card is sensed, the value of the IC register is set to 0000 and the execution cycle is initiated. If there are no cards in the input hopper at the time the LOAD button is pressed, SAMOS simply hangs until the cards are placed in the hopper.

Index

- 24-BIT FUNCTION BOX, 31, 104ff
- 4-BIT MANIPULATE
 - microinstruction, 232
- A register, 36, 215
- A stack, 12, 41
- ADD
 - MIL statement, 173
 - microinstruction, 39
 - routine, 98, 137
- ADD OF McMIL statement, 261
- ADDRESS MIL expression, 210
- ADDRESS . TO . BINARY routine, 124
- ADJUST MIL statement, 205
- AND MIL statement, 173
- ANY . INTERRUPT condition, 218
- ASCII, 89

- BARTON, ROBERT S., 3
- BASE . OF . INTERPRETER, 64
- base register (*see* BR), 14
- BELGARD, R.A., 59, 160
- BIAS
 - MIL statement, 174
 - microinstruction, 19, 23, 28, 220
- BICN register, 217
- binary arithmetic, 104
- BINARY . TO . FA routine, 121
- BIND microinstruction, 221
- BIOPSI, 59
- BIT TEST microinstruction, 222, 223
- BR register, 16, 35
- BRANCH microinstruction, 224
- BSS McMIL statement, 261
- BUFFER McMIL statement, 60, 265
- B1700
 - as an interpreting machine, 10
 - family, 4
 - instruction cycle, 217
 - model of storage, 7
- B1710, 5
- B1720, 5, 6
- B1726, 7, 10, 11
- B1800, 4, 150

- CA register, 218
- cache memory, 5, 150
- CALL
 - MIL statement, 176
 - microinstruction, 41, 225, 226
- CARRY MIL statement, 106, 176
- CB register, 218
- CC register, 218
- CD register, 218
- CHECK INTERRUPTS McMIL statement, 77, 263
- CLEAR
 - MIL statement, 177
 - microinstruction, 226
- CLOSE McMIL statement, 64, 264
- CMPX register, 31
- CMFY register, 31
- COBOL, 80
- code segments, 43
- codefile, 45
- COMPLEMENT MIL statement, 177
- control, 37
 - stack (*see* A stack), 12
 - store, 6, 36, 150
 - transfer between interpreters, 164
- copy loop, 25
- COUNT
 - MIL statement, 178
 - microinstruction, 227
- CP register, 32, 213

- CPL register, 19, 23, 31, 104, 213
- CPU register, 32, 104, 213
- CYD condition, 31, 104, 218
- CYF register, 31, 32, 104, 213, 217
- CYL condition, 31, 104, 218
- data
 - fetch and addressing, 16
 - examination and manipulation, 28
- DATA.LENGTH MIL expression, 55, 210
- DEBUG McMIL statement, 262
- DEC MIL statement, 179
- decimal arithmetic, 32, 104
- DECLARE MIL statement, 50, 206
- definable field, 4
- DEFINE MIL statement, 50, 205
- descriptors, 24
- DF McMIL statement, 261
- DIFF register, 31, 104
- DUMPFIL McMIL statement, 263
- DV McMIL statement, 261
- DYNAMIC region, 43
- EBCDIC, 89
- EFFECTIVE.ADDRESS routine, 127
- emulator, 1
- EOR MIL statement, 181
- EXCHANGE microinstruction, 253
- EXIT MIL statement, 41, 181
- EXTRACT
 - MIL statement, 181
 - microinstruction, 29, 229
- FA register, 14, 35, 213
- fault detection, 74
- FB register, 214, 218
- field
 - address register, 14
 - definable, 4
 - length register, 14
- FILE declarations, 69, 272
- FL register, 14, 214
- FLC register, 214
- FLCN register, 218
- FLD register, 214
- FLE register, 214
- FLF register, 214
- FORTRAN, 80
- FT register, 26, 214
- function box, 31, 104
- FU register, 26, 28, 214
- G-machine, 2
- G-store (*see also* S-memory), 3, 150
- GO TO MIL statement (*see also* JUMP), 39, 183
- guest, 2
- H-machine, 2
- H-store (*see also* M-memory), 3, 36, 150
- HALT microinstruction, 41, 230
- host, 2
 - universal, 4
- host environment, 80
- IF
 - MIL statement, 183
 - conditions, 217
- IF NO INTERRUPTS McMIL statement, 263
- INC MIL statement, 185
- INITIALIZE McMIL statement, 65, 259
- INPUT/OUTPUT, 59
- instruction cycle, 37, 217
 - decoding, 33
 - fetch and increment logic, 155
- interpreter, 1
 - displacement, 159
 - faults sensed, 76
 - shell, 80
 - structure, 75, 87
- interpreters, switching of, 164
- interrupts, 74, 77
- io request McMIL statement, 265
- JCL (MCP control statements), 64, 269

- JUMP
 - MIL statement, 39, 186
 - microinstruction (*see* BIT TEST), 40
 - jump table, 39

- L register, 28, 213, 218
- LA register, 29, 213
- labels, 39
- LB register, 29, 213
- LC register, 29, 213
- LD register, 29, 213
- LE register, 29, 213
- LF register, 29, 213
- limit register (*see* LR), 14
- literals, 57
- LOAD F
 - MIL statement, 187
 - microinstruction, 231
- LOADER, 46, 68, 269
- local labels, 39
- LOCAL .DEFINES MIL statement, 51, 53
- LR register, 16
- LSUX condition, 218
- LSUY condition, 217

- M register, 36, 215
 - ORing feature, 37
- M .MEMORY .BOUNDARY
 - MAXIMUM, 157
 - MINIMUM, 161
- M-memory (*see* H-store), 7, 150
- MACRO MIL statement, 50, 56, 208
- MANIPULATE 4-BIT
 - microinstruction, 232
- Master Control Program (MCP), 8
- MBR register, 154
- McMIL, 59
- McMIL statement summary, 268
- MCP, 8, 42
- microcode, 10
 - transfer between G- and H-store, 161
- microinstruction
 - summary, 254
 - variants, 256
- microprocessor, 16
- MIL, 19
- MIL statements
 - alphabetical list, 170
 - executable, 173
 - nonexecutable, 205
- MINUS routine, 142
- MLIST McMIL statement, 260
- MONITOR
 - MIL statement, 187
 - microinstruction, 233
- MOVE
 - MIL statement, 12, 187
 - REGISTER microinstruction, 241
 - 24-BIT LITERAL microinstruction, 235
 - 8-BIT LITERAL microinstruction, 234
- MSBX condition, 218
- MSKX register, 31
- MSKY register, 31

- NO OPERATION microinstruction, 41, 236
- NOLIST McMIL statement, 260
- NOP microinstruction, 41, 236
- NORMALIZE
 - MIL statement, 190
 - microinstruction, 236

- OPEN McMIL statement, 64, 264
- OR MIL statement, 191
- OVERLAY
 - MIL statement, 191
 - microinstruction, 36, 161, 237

- packed decimal, 104
- point labels, 39
- printer spacing, 60
- PRINT SNAP McMIL statement, 262

- READ
 - MIL statement, 192
 - H-STORE microinstruction, 240
 - microinstruction, 14, 239
 - MSML MIL statement, 193
- read loop, 21
- recursive organization, 2
- REGISTER MOVE microinstruction, 241
- RESET MIL statement, 193
- REVERSE option, 23, 49
- ROTATE T
 - MIL statement, 194
 - microinstruction, 246
- ROTATE X
 - MIL statement, 196
 - microinstruction, 248
- ROTATE XY
 - MIL statement, 196
 - microinstruction, 249
- ROTATE Y
 - MIL statement, 196
 - microinstruction, 248
- run-structure nucleus, 42
- running a program, 269

- S-memory (*see* G-store), 7, 16, 150
- SAMOS description, 303
- SAMOS interpreter, 275, 285
 - LOADER program, 300
 - sample output, 301
 - shell, 81
 - storage representation, 88
 - test data, 301
 - utility routines, 92
- scratchpad, 12, 24, 214
 - settings, 69
- SCRATCHPAD
 - MOVE microinstruction, 243
 - RELATE microinstruction, 244
- SDL, 154
- SECTION McMIL statement, 65, 260
- SERVICE INTERRUPTS McMIL statement, 263
- SET
 - MIL statement, 196
 - CYF microinstruction, 245
- SFL register, 26
- SFU register, 26
- shell
 - interpreter, 80
 - SAMOS, 81
- SHIFT T
 - MIL statement, 194
 - microinstruction, 246
- SHIFT X
 - MIL statement, 196
 - microinstruction, 248
- SHIFT XY
 - MIL statement, 196
 - microinstruction, 249
- SHIFT Y
 - MIL statement, 196
 - microinstruction, 248
- simulator, 1
- SKIP
 - MIL statement, 197
 - microinstruction, 40, 250
- SMACK system, 59
- SNAP McMIL statement, 262
- STACK (*see* A stack), 214
- STARS McMIL statement, 260
- STATIC region, 43
- STOP McMIL statement, 65, 263
- STORE F
 - MIL statement, 199
 - microinstruction, 251
- SUBTRACT MIL statement, 200
- SUM register, 31, 104
- SWANSON, M., 160
- SWAP
 - MIL statement, 200
 - microinstruction, 252
- Systems Development Language (SDL), 154
- S0 scratchpad, 26

- T register, 13, 14, 28, 212, 218
- TA register, 29, 212
- TABLE MIL statement, 209
- tabular data, 151, 157
- TAS register (*see* A stack), 41, 214

- TB register, 29, 212
- TC register, 29, 212
- TD register, 29, 212
- TE register, 29, 212
- TERMINATE McMIL statement, 65, 259
- TEST microinstruction, 40
- testable conditions, 217
- TF register, 29, 212
- TOPM register, 154
- TRANSFER.CONTROL MIL statement, 164, 202

- universal host, 4
- UNPACK.AND.WRITE routine, 147
- UPL, 52, 53, 80

- VALIDATE.DECIMAL routine, 106

- WRITE
 - MIL statement, 202
 - H-STORE microinstruction, 240
 - microinstruction, 15, 239
 - MSML MIL statement, 204

- X register, 18, 28, 31, 212
- XANY register, 31
- XCH MIL statement, 204
- XCH microinstruction, 24, 25, 253
- XEDY register, 31
- XORY register, 31
- XYCN register, 218
- XYST register, 218

- Y register, 28, 31, 212

