OPERATING AND PROGRAMMING SYSTEMS SERIES

Peter J. Denning, *Editor*

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

Elliott I. Organick
James A. Hinds

THE COMPUTER SCIENCE LIBRARY

PERATING 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

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

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

Contents

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 inacessible 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

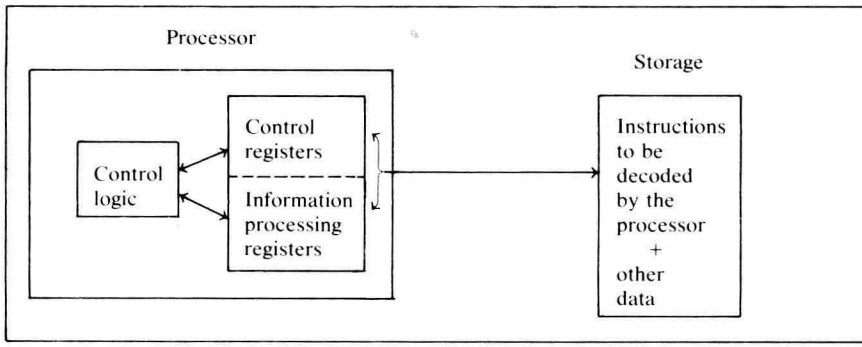

Figure 1.1. A view of a typical computer architecture.

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 thc 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.)

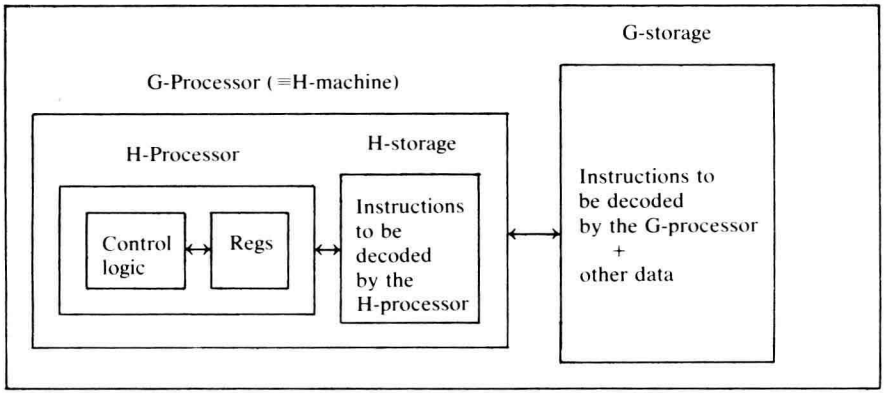

Figure 1.2. Structure of a two-level host/guest system.

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

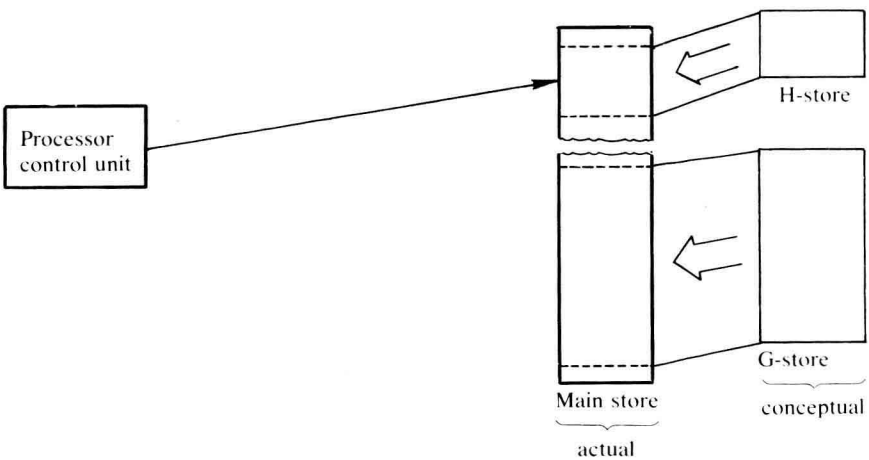

Figure 1.3. B1710 Processor access to H-store code.

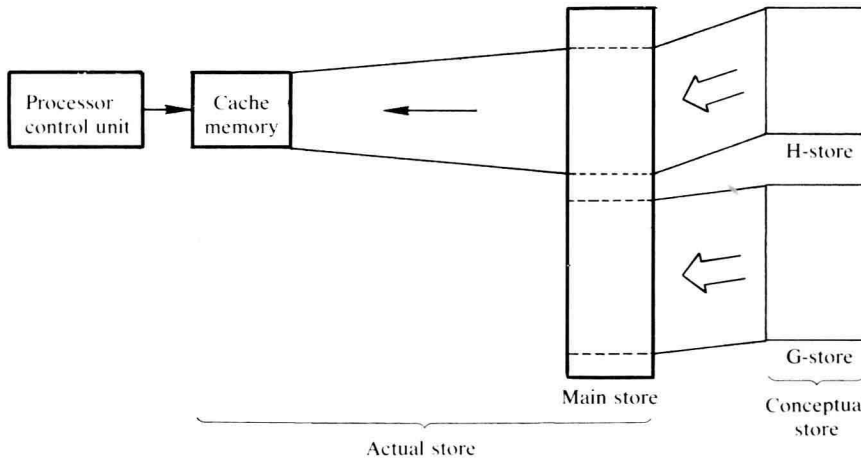

Figure 1.4. B1800 Processor access to H-store code.

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

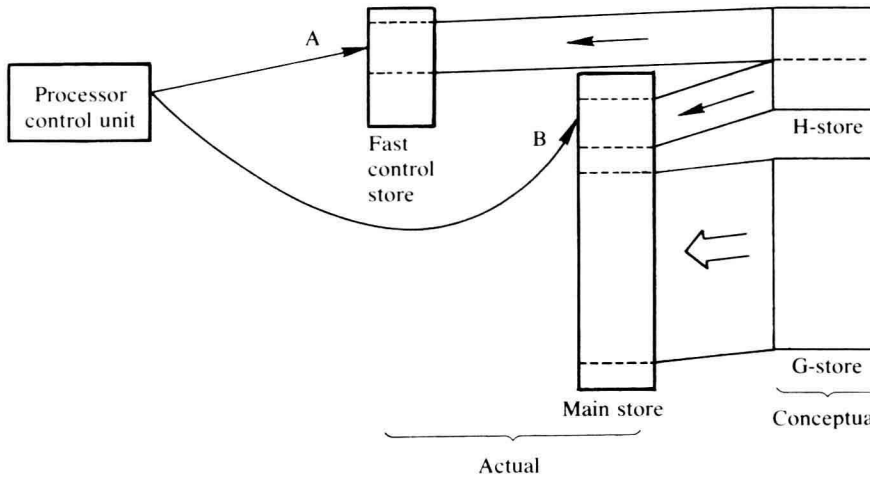

Figure 1.5. B1720 Processor access to H-store code.

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" (*M*aster *C*ontrol *P*rogram) 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.