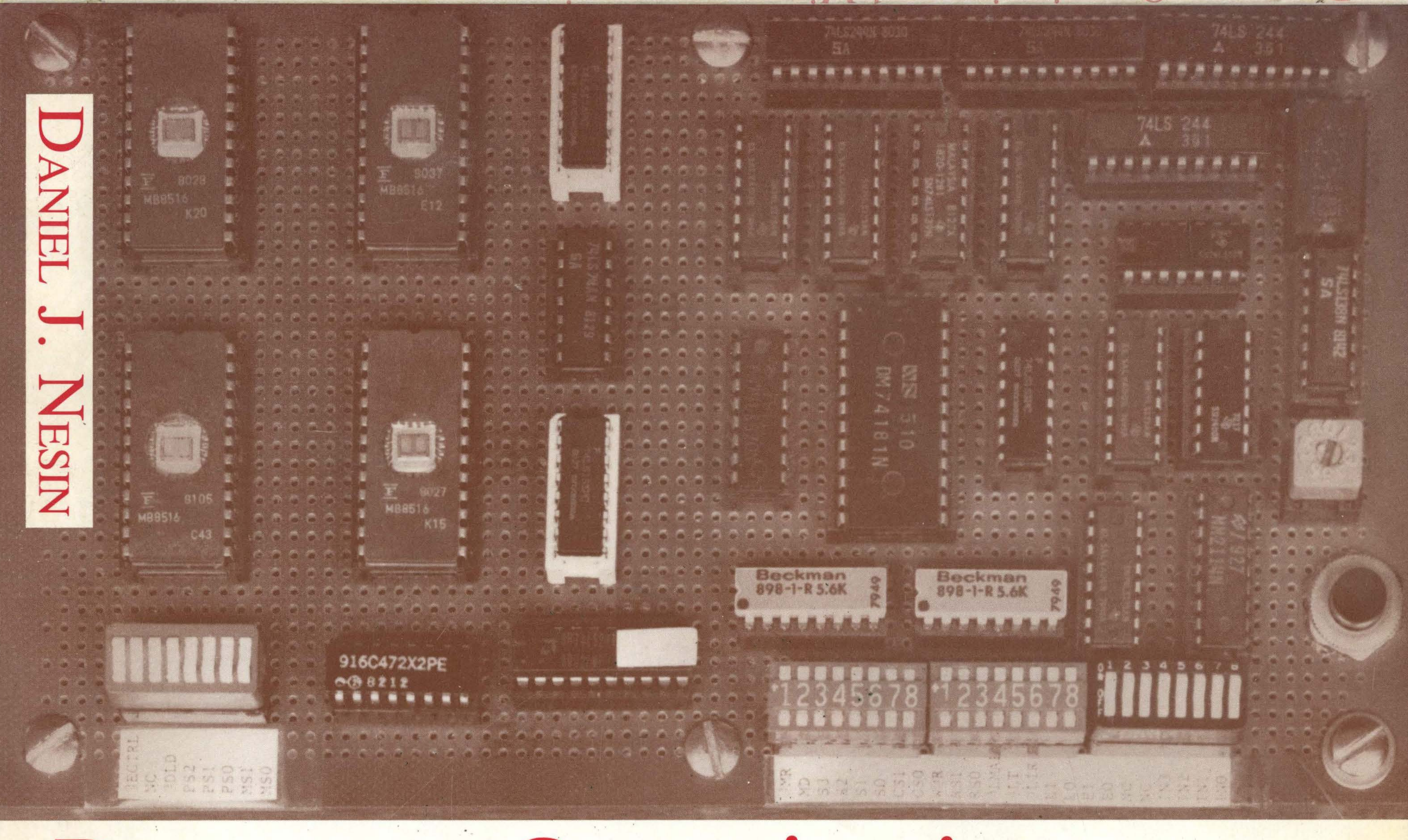
Processor Organization and Microprogramming • A Project Case Study — NESIN

SRA



DANIEL J. NESIN

Processor Organization and Microprogramming

A Project Case Study

PROCESSOR ORGANIZATION AND MICROPROGRAMMING

Processor Organization and Microprogramming

A Project Case Study

Daniel J. Nesin

California State Polytechnic University, Pomona

SRA®

SCIENCE RESEARCH ASSOCIATES, INC.
Chicago, Henley-on-Thames, Sydney, Toronto
A Subsidiary of IBM

Acquisition Editor	Al Lowe
Project Editor	Geof Garvey
Copy Editor	Dan Kirklin
Production Director	Arthur Kuntz
Text Design, Cover Design, and Composition	Baskerville Book

Library of Congress Cataloging in Publication Data

Nesin, Daniel J., 1924–
Processor organization and microprogramming.

1. Computer architecture. 2. Microprocessors. 3. Microprogramming. I. Title.
QA76.9.A73N46 1985 001.64'2 84-23560
ISBN 0-574-21770-3

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

CONTENTS

5 STUPIDD V–A Microprocessor Analysis and Construction Project 155

6 The Control System 241

Ready Reference of Key Topics 307

PREFACE

The material of this text arose out of a series of student projects in digital design initiated over a decade ago with electrical engineering students. The projects answered two observed needs of students to enhance their comprehension of digital processor systems: (1) to experience a complete processor—as opposed to partial amorphous architectural representations—in order to gain a comprehension of the nature of processing systems; (2) to work with real (rather than hypothetical) architectural constructs—thus demonstrating their mastery of an actual archetypical processor, which proved a particular satisfaction to them.

The associated Central Processor Unit construction project was designed to be built and operated at home, to encourage an appreciation of the development of the hardware and microprogramming.

In 1981 the author began instructing software-oriented computer science students who had very little, if any, prior exposure to hardware. It was felt that removing the mystery and hardware fright of these students should significantly improve their future careers, as the distinction between hardware and software knowledge becomes increasingly obsolete. These computer science students' comprehension of the material formalized into this text appeared to be no different from that of electrical engineers: 25 percent culture shock, 50 percent varying degrees of comprehension, and 25 percent enthusiasm accompanied by innovative suggestions and activities. A significant number of these software-emphasis students actually found that the material helped open up new career opportunities, such as software development for microprocessor-controlled instrumentation systems.

Chapter 1 presents a historical perspective on computers and introduces the major fundamental systems architectures. It is shown that a computer system is composed of a number of processors, which possess unified underlying systems principles coordinated by the Central Processing Unit (CPU) and the systems software developed for it. The notion of the cycle of computation is introduced, with its two major states: Instruction Fetch and Instruction Execute. The role of the Program Counter in von Neumann architectures is explained, as is its effect on memory-usage maps. The increasing need for systems insight into the hardware/software relationship is stressed, with its current trends and impacts on people.

Chapters 2, 3, and 4 are included in this text in response to the many questions posed by the software-oriented student with little hardware exposure. It is presented as a broad survey or review of device and architecture fundamentals that an electrical engineering student may or may not already have been exposed to. The goal of these chapters is to create a working understanding and intuitive mental image of fundamental logic devices and how they are incorporated into architectural structures. The manner in which many of these devices of a digital system function can often be readily understood in terms of logically equivalent analogs. For example, the transistor's behavior, in the systems context, can be perceived as that of a simple switch. The functioning of logic and tri-state gates, buses, memory, Arithmetic/Logic Unit (ALU) devices, and system-clock coordination are presented.

These structures are then incorporated into the larger architectural block-diagram structures of a processor system. The ALU is emphasized as the device that both transforms information and originates the system signals that control the conditional flow of software and state paths of the system. Chapter 4 reviews sequential machine theory and implementation, stressing the State Table form of representation and its relationship to microprogramming. The time spent on these chapters will depend on the students' prior exposure to hardware. It does not exceed four to five weeks with the author's students.

Chapters 5 and 6 present the actual processor construction and microprogramming projects. The CPU is presented as the archetypical microprogrammable processor. Construction guidelines and a debugging overview are presented. It should be emphasized that the Central Processor Unit project has been constructed at home by computer science students and therefore requires no school laboratory facilities. A very simple wire-wrap tool and an inexpensive volt/ohmeter are all the student requires. The power-supply can be a 6-volt camping lantern battery and some silicon diodes. Several drilled holes and no more than five soldered connections are needed.

The wire-wrap form of construction is recommended as best for students who have little experience with hardware. Student feedback has made it clear that the hours spent handling each integrated circuit and consulting the data catalogs provide an extra dimension of "hands-on" experience. This exposure is especially helpful when the students are later required to work with the data catalogs of the programmable peripheral microcomputer devices, such as communications, disk, and Direct Memory Access (DMA) controllers. This good basic training has proved to be worth the time and expense of the project.

The author's computer science students have built the CPU portion of the project and demonstrated the switch-controlled operation of microcode, in a one-quarter course. The control system can be—and actually has been—constructed on an elective follow-up basis. It can, however, also be incorporated into a full-semester course. Very little extra construction is required for the control system. The advanced microprogramming activities of this phase of the projects require of the student great creativity, total systems understanding, and the drive to be innovative. These advanced activities require access to a PROM programmer. The last section of Chapter 6 covers the use of an IBM PC, with an aftermarket PROM programmer, for automating the development and demonstration of microprograms. (*Aftermarket* refers to components supplied by a secondary vendor.) Most flexible personal computer systems these days support the programming of PROM's (Programmable Read Only Memories). Advanced microcomputer hobbyists often already have such equipment.

A NOTE OF APPRECIATION

The drive to dominate the machine, as opposed to the reverse, grew out of actual hardware-innovation experiences with students, which the author would like to acknowledge here. In 1972 Xerox Corporation donated a 930 computer system on an "as-is" basis. This was a gift of considerable value. The author and dedicated students revived the system. They created a real-time operating system for it, with memory mapping and instructional traps, implemented interactive graphics, computer-aided drafting, etc. Their innovative hands-on hardware experiences had significant effects on industry. Companies were formed, jobs were created, and products most readers would be familiar with were marketed. It is impossible to name all these movers and shakers. They know who they are, and the author certainly remembers them. The author does, however, wish to acknowledge the many contributions from the students of the old Extracurricular Student Computer Lab–we all grew from our experiences with hardware and software basket cases. The author is appreciative because, after seventeen years in industry, another seventeen years in teaching could only offer the satisfaction of positively affecting the world we live in. These students provided that satisfaction–and then some.

CHAPTER 1
OVERVIEW OF PROCESSOR ARCHITECTURE

THE EVOLUTION OF COMPUTERS

The rapid pace of recent events, particularly in microprocessors, appears to make computing a completely new phenomenon to some. But computing is not new, nor has it always been electronic and digital in nature. For a better perspective, let us briefly review the historical background of computing.

There can be no doubt that the ability to predict seasonal variations was important to early man. Using data to predict is one form of computation. Cro-Magnon bone carvings, 300 centuries old, record the lunar cycles and seasonal changes noted by members of a hunting society, in relation to the flora and fauna of interest to them. While anthropologists may dispute some interpretations of these records, they definitely constitute an early form of data base—most likely used to predict seasons, migrations, and so on. Later agricultural societies were very dependent on the computation of the seasons and resulting harvest yields. Accurate astronomical observations, the development of some numbering systems, and their use in predicting seasons and accounting for harvests are, again, indications of data gathering for computational use. The likelihood that a few unscientific spirits were invoked to ensure the success of these processes only serves to indicate that ancient programmers may have shared something with a few modern ones. Thus we see our own natural propensities for computation applied to data collection, prediction, simulation, control, and the execution of events. These are the basis for the mechanization of applied computation. Efficient computation requires the application of advanced technologies.

EARLY CALCULATORS

The early development of modern computation was extremely slow by today's standards. Much of the mathematics we use evolved before the existence of calculating machines, if we ignore our fingers and toes. The abacus was an early calculating instrument first used about the fourth century B.C. While useful for rapid addition of numbers, its chief advantage lay in its ability to "remember" partial results, thus enabling nimble fingers and unsure minds to perform tedious calculations. During the 1600s Schickhardt (1624), Pascal (1642), and Leibniz (1673) first developed the mechanical calculators that were widely applied in the 1800s, when the manufacturing technology was available. As technology evolved, these calculators were replaced by ones using solid-state devices. The industrial revolution is a reflection of our evolving ability to develop mechanisms for computational uses. The problem with these early calculators was that none of them had the ability to store a program. Each step had to be dictated by the human operator—thus delineating the difference between these early calculators and the computers that followed them.

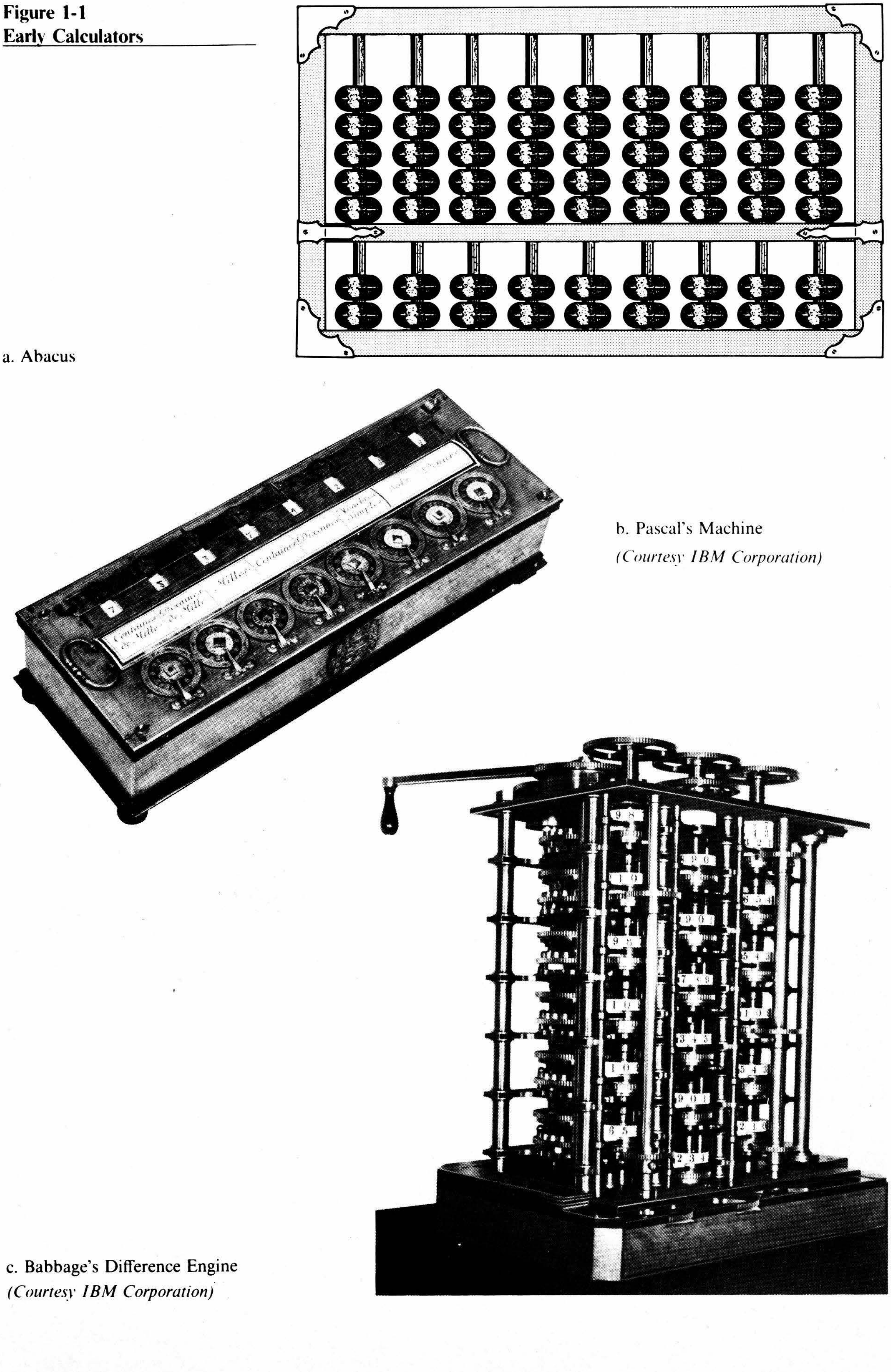
EARLY COMPUTER DEVELOPMENTS

Babbage's difference and analytical engines show that mechanical technology alone could not produce an effective computer. In the 1820s, Charles Babbage, an Englishman and a founder of the Royal Astronomical Society, became interested in developing a mechanical computer for the solution of mathematical equations. He was assisted in these efforts by Ada Augusta, Countess of Lovelace, after whom the programming language *Ada* is named. She is often referred to as the first programmer, but she was far more than that, for she understood the hardware, the software, and theoretical basis of the computer. They applied a good deal of thought and ingenuity to the task of designing a computer. The many original concepts developed foreshadowed the development of the modern computer. It is sad that, in the end, their valid concepts could not be incorporated into a useful machine because of the inability to hold to the required mechanical tolerances in its manufacture. These manufacturing obstacles were eventually surmounted by Scheutz, but the resulting machine's accuracy, speed, and cost would be unacceptable today. This is not to take away credit from these pioneers. New technologies and new concepts were necessary for further progress. Figure 1-1 displays some of these early calculators.

Some of the new concepts were provided by George Boole, who in 1854 developed an algebra of logic. This algebra was primarily of interest to mathematicians and philosophers until Claude Shannon published "A Symbolic Analysis of Relay and Switching Circuits" in 1938. Shannon dealt with relay logic, the technology of that day, as a result of his research in developing telephone switching circuits. Shannon made the first application of Boolean algebra to switching circuits, paving the way for it to become the important tool that it is today. Meanwhile, the need for an effective form of a stored-program digital computer was growing. We have to go back a bit in time to see how it evolved.

Babbage began work on his analytical engine in 1833. On a previous visit to France, he had seen the Jacquard looms in action, weaving fabrics. Jacquard developed this system of controlling looms by pasteboard cards with holes "punched" in them, in 1805. (See Figure 1-2.) Babbage was inspired by the Jacquard loom to formulate the design of his new engine, which was to use two sets of these punched cards. One set was the *operation* set of cards; the other, the *variable* set. Thus, the concept for the first computer was born. We now call the operation cards the *program* and the variable cards the *data.* Babbage was frustrated by the lack of an adequate implementation technology, but the machine he conceived became the basis for the Harvard-IBM Mark I computer, produced a century later. Jacquard's pierced pasteboards also provided the inspiration for Herman Hollerith's development of punched-card tabulation, used for the United States census of 1890. Hollerith, seeking a way to tabulate census data mechanically rather than manually, was advised to look at the Jacquard looms. In 1906 he founded the company that evolved into the International Business Machines Corporation. Tabulating systems had thus arrived, as well as their associated printer-listers, and they became common office

Figure 1-1
Early Calculators



a. Abacus

b. Pascal's Machine
(Courtesy IBM Corporation)

c. Babbage's Difference Engine
(Courtesy IBM Corporation)

equipment. This business office equipment was in turn used in the implementation of Babbage's concepts in the development of the Harvard-IBM Mark I computer (1939–44).

HARVARD MACHINES

The Harvard Mark I used relay logic; as an electromechanical rather than just mechanical device, it represents the beginning of the era of modern computers. The Mark I could multiply 23-digit numbers in about six seconds–not impressive by today's standards, but remarkable a relatively short time ago. One of its important tasks was the calculation and analysis of ballistics equations, a vital contribution to the nation's efforts in World War II. The success of the Harvard machines, the Mark I, II, III, and IV, spurred researchers to find even better technological answers to the problems of implementing computation.

The Harvard machines had two storage areas: one for the program and the other for data and results. The two storage areas were isolated from each other and were typically of different word size. These machines were termed *automatic-sequence calculators* and did not have the capability of modifying their own programs. They were dedicated special-purpose machines, whose principles are still used in many small microprocessors today. They are most advantageous in applications where the program is not expected to be modified during operation, such as in vending machines or toys. They also offer a hypothetical two-to-one speed advantage over the von Neumann-type machine, discussed next. This important advantage is based on the fact that the separate memories can be accessed simultaneously. Most 4-bit and some early 8-bit microprocessors are Harvard-type machines, huge numbers of which are still in use. Special-purpose high-speed dedicated processors, often called controllers, can use the Harvard (really, Babbage) approach to good advantage in their design. Today, the trend is to use the Harvard architecture in custom-designed integrated-circuit controllers (processors), where an inherent speed advantage can be an important consideration.

VON NEUMANN MACHINE

The shortcomings of the Harvard architecture were perceived in the mid-1940s by von Neumann, who recommended that a computer have only a single storage area for *both* programs and data. This gave rise to the prevalant form of digital computer extant today in business and scientific applications. Von Neumann's recommendations appear in a summary article (Burks, Goldstine, and von Neumann 1946) and are worthy of the reader's time. The essence of his idea is that, if a computer can operate on data, it can also operate on its own program, thus obtaining the ability to alter its course conditionally, *without operator intervention.* This was the beginning of the stored-program digital-computer concept, as distinct from the Harvard concept, discussed earlier.

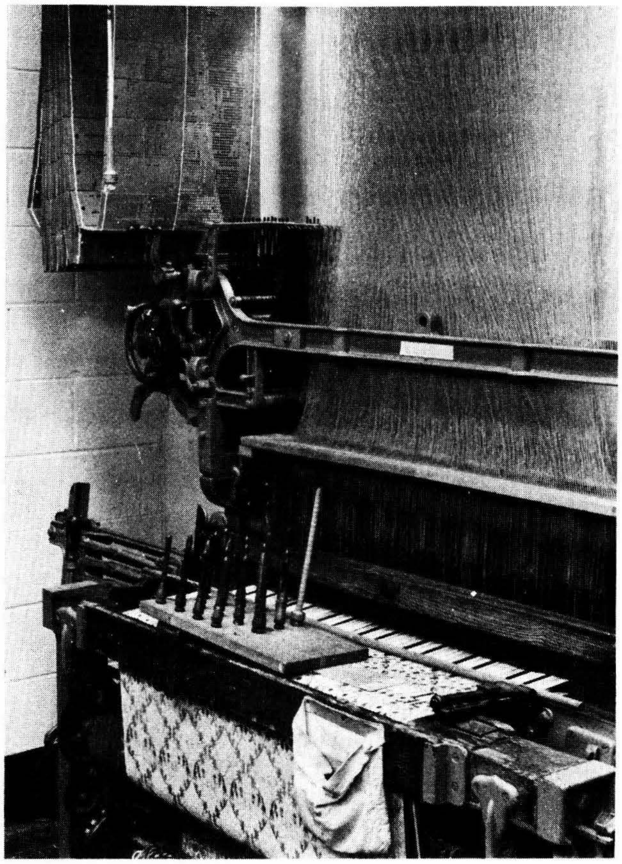


Figure 1-2
The Jacquard Loom
Photography by Jan Forman
Philadelphia College of
Textiles and Science

The very idea of letting a program operate on itself is repugnant to modern structured programmers, but it was a milestone in computer development and remains a powerful concept.

Microprogramming Concept

The late forties and the fifties produced many ideas on how to improve, design, and build computers. It was a decade of realization of what was practical in computation. Three-state logic devices (+, 0, −) were considered, for example, but these lost out because of the greater feasability of the two-state (on-off) transistor devices used today. Among the ferment of ideas produced in this era was Wilkes's (1951) concept of microprogramming. This important modification of the von Neumann concept was too powerful to lose out permanently in the feasibility contest. Rather, once a suitable modern implementation technology had evolved, Wilkes's concept became the rational basis for designing a computer's control system. Today, it is the internal organizational foundation around which we structure most processors, computers, and—what amounts to the same thing—microprocessors that the reader is likely to encounter. *An applied understanding of microprogramming is a major goal of this text.*

Computer Implementation Technologies

Implementation technologies were rapidly improving. ENIAC, a Harvard-architecture program calculator, became operational in 1945. It was the first all-electronic computer, using vacuum-tube technology. EDSAC and EDVAC, von Neumann-type stored-program computers, became operational in 1949 and 1950. Subsequent development was extremely rapid; although interesting, the full details are beyond the scope of this book. By now, the major ways of organizing an architecture were well established. Let us then refocus on the development of implementation technologies. Digital computers were becoming better because we were finding better ways to implement them with emergent semiconductor technologies. The basic conceptual groundwork for the organization of the types of processor we are most likely to encounter had already been developed.

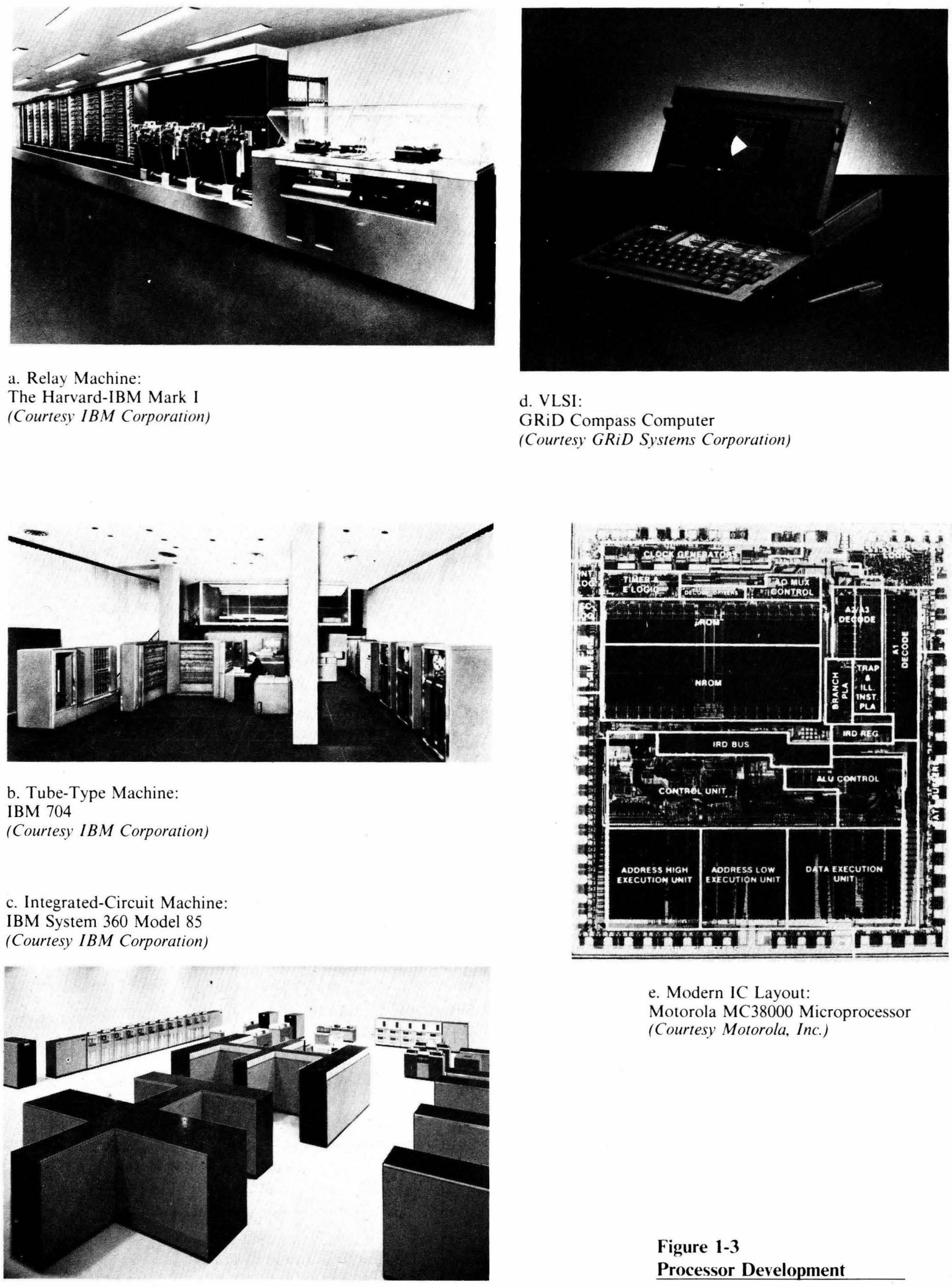
Tube-type computers are referred to as the "first generation." One could argue the point, if we consider the Harvard relay machines, but in fact they formed the first generation of stored-program digital computers, from 1950 to the early 1960s. In the early 1960s "second-generation" computers–based on transistor technology–came on the market. The "third generation"–based around small- and medium-scale integrated circuits (MSI)–began to appear in the late 1960s. The emphasis in that decade was primarily on how to build computers, for a lot was known about architecture, but there was little economical implementation technology available. Some very significant variations on von Neumann-type architecture were being developed, though. Noteworthy were Barton's concepts for stack architecture (1961), actual implementations of microprogrammed architectures, virtual memory, and interactive real-time computing. Each new "generation" of computers involved a reduction in size and power consumption by an order of magnitude. The need for air conditioning, for example, was once a major consideration. Figure 1-3 illustrates the relative sizes of these "generations" of computers. We shall not discuss here the evolution of programming and operating systems, except to note that they developed along with the increasingly available hardware.

BUS ORGANIZATION

By 1970 part of the new hardware expertise developed concerned bus organization and *tri-state* logic. A *bus* is simply a data path for computer signals, consisting of one or more physical conductors of information. Bus organization includes the study of methods for time-sharing the use of a single bus, thus reducing the total number of buses required in a given computer. *Tri-stating* is a means of interfacing separate entities to a time-shared bus. We shall study this important implementation technology subsequently. The combination of the ideas for bus organization and tri-state logic was widely applied in the 1970s. They became an important step in making microprocessors feasible. Bus organization is still a vital topic, involving federal standards, networking, multiprocessing, and many other aspects of computing.

SEMICONDUCTOR TECHNOLOGY

Integrated Circuit (IC) manufacturing technologies continue to develop at an unabated pace. MSI combines the equivalent of several hundred transistor logic gates in a single design. Large Scale Integration (LSI) can put on the order of 10,000 transistor equivalents into a



a. Relay Machine:
The Harvard-IBM Mark I
(Courtesy IBM Corporation)

d. VLSI:
GRiD Compass Computer
(Courtesy GRiD Systems Corporation)

b. Tube-Type Machine:
IBM 704
(Courtesy IBM Corporation)

c. Integrated-Circuit Machine:
IBM System 360 Model 85
(Courtesy IBM Corporation)

e. Modern IC Layout:
Motorola MC38000 Microprocessor
(Courtesy Motorola, Inc.)

Figure 1-3
Processor Development

single monolithic structure. In the mid-1960s attempts to apply MSI and LSI technologies to commercial desk calculators were not fruitful, but these efforts led to the monolithic microprocessor in the early 1970s. As in the past, if we can calculate, why can we not compute? Thus, in 1971, Intel Corporation marketed the first microprocessor—the Intel 4004. It was a Harvard machine with a 4-bit data bus, and it contained the equivalent of 2,450 transistors on a "real estate" silicon chip 0.117 × 0.159 inches in size. This led to the flood of microprocessors that continues to this day, as Very Large Scale Integrated (VLSI) circuits, approaching 500,000 transistor equivalents on a single monolithic IC, are produced.

PROCESSOR SYSTEMS

LSI and VLSI technology spawned another breed of processors besides the computer itself. These belong to the very important class of processor-support peripheral IC's. We have not only the computer in an IC today, but all the memory, communication channels, floating-point arithmetic processors, disk and display controllers, and so on, as well. Three important facts should be made apparent:

> First, a computer is now a *system,* consisting of a collection of IC's ruled by the software and "firmware" created for it. The firmware is a program permanently recorded in a Read Only Memory (ROM).
>
> Second, many of these IC's in the system are processors.
>
> Third, by studying the organization and function of an archetypical microprogrammable processor, we can obtain an appreciation of all the others.

All these processors, combined into the system we call a computer, can be and in fact now are organized around a common set of microprogrammed design principles. An understanding of these principles is another of our main objectives in this text. Therefore, there is a common point of view through which we can gain an understanding of the design, function, and operation of all the processor-type IC's used in the modern computer. This insight is as essential to the programmer as it is to the hardware designer. It is not easy to write efficient code for the software that drives a synchronous communications IC, a disk controller, etc., without a secure grasp of the intrinsic nature of these devices.

Thus, with our technological advances in computing, we find that the distinction between hardware and software types is rapidly breaking down. This artificial distinction never existed for the creators, Charles Babbage and Ada Lovelace, who truly understood the nature of what they wrote programs for. They would have had no difficulty in recognizing that the modern computer is a *system* composed of a microprocessor and a collection of peripheral support IC's, all having many functional features in common. At this writing, one major producer is shipping 100,000 personal computers a month—each one having as much computing power as early mainframes and many minicomputers still in use. The microprocessor is not a "different" thing or a break with the past. It is a result of modern industry's ability to capitalize on past technological breakthroughs, now occurring at a bewildering pace. The development of the microprocessor has been so impressive that it leads to speculation about potential future applications. The microprocessor is becoming the mainframe computer of today. True 32-bit microprocessors are on the market now, with their full potential yet to be realized.

Personal Impacts

And that is probably why you are reading this text. The utility of microprocessors (really, computers or just processors) is having a profound effect on our lives. Can any of us, whether software or hardware types, afford *not* to understand philosophically the tools by which we earn our livelihood? We all need insight into the intrinsic nature of these devices if we are to stay current in our respective fields and understand the world we live in. Perhaps our individual emphasis may not be on the computations of seasonal migrations, crops, ballistics, or astronomical calculations. These topics are still of interest to many. Our interests, on the other hand, might be on the business, scientific, educational, sociological, or even amusement aspects of applied computation. We are still interested in planning, controlling, simulating, and executing–except that highly sophisticated technology is now available to us for these purposes, on a mass basis, that would simply amaze the early pioneers–let alone Cro-Magnon man.

(Those of us with a strange philosophical bent may ponder what might have happened if Cro-Magnon man had turned his caves into video arcades, if he had had the microprocessor. This might have deprived us of some of our most moving works of art. The antitechnologist might presume that they would have planned anti-Neanderthal games with the computer.)

We have always been interested in computation, for better or worse. We can see the continuing evolution of the technologies used for applied computation all around us and, it seems, the evolution of our fascination with it. Table 1-1 summarizes this brief introduction into the history of computing. We hope that it also conveys a feeling for the rapidly accelerating pace of innovation that we are all caught up in. Some essential concepts, however, change only slowly. It is the innovative use of these concepts that avoids individual obsolescence. Our goal will be to gain an understanding of the fundamental concepts of microprogrammable processor organization. We will deal with the topic not hypothetically but rather in the very real terms of analyzing, implementing, and microprogramming an instructional 4-bit processor. In doing this, we can easily penetrate the mystique that still surrounds the computer CPU and other processors–to eliminate the "hardware fright" that accompanies a lack of understanding of these systems.

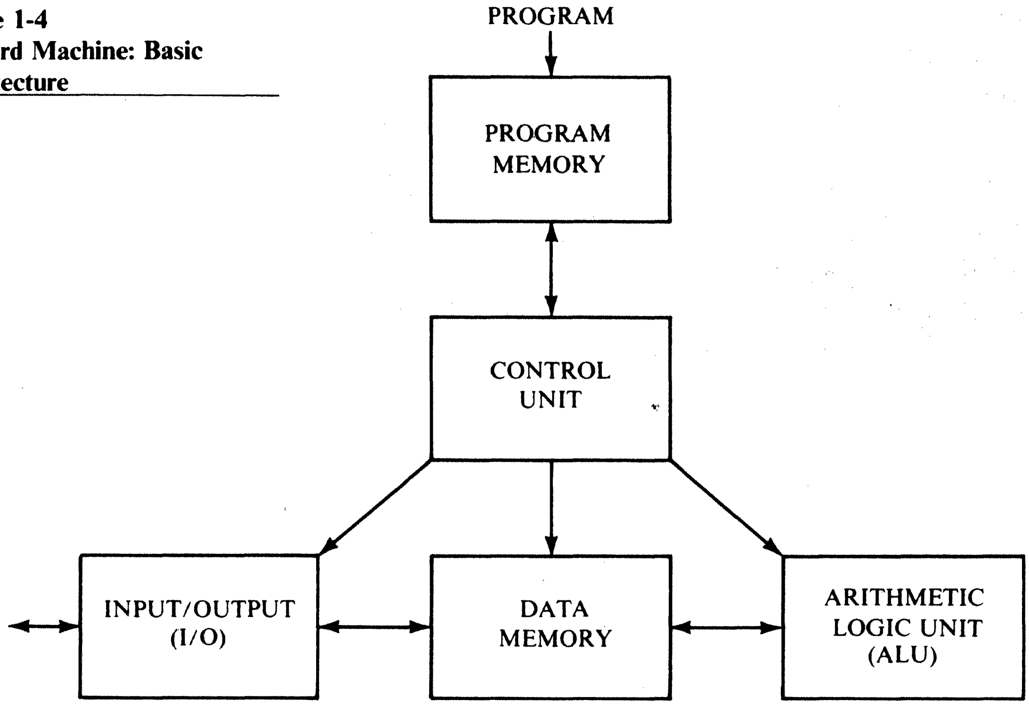
Table 1-1
Historical Computing Perspective: Selected Key Benchmarks

Year	*Event in the Evolution of Computers*
30,000 B.C.	Bone carvings show evidence of calculation of lunar and seasonal cycles by Cro-Magnon man.
400 B.C.	Abacus first used.
1600–1700 A.D.	Schickhardt, Pascal, and Liebniz develop mechanical calculators.
1805	Jacquard loom uses "punched" pasteboards.
1820	Babbage conceives idea of difference engine.
1833	Babbage conceives idea of analytical engine. Device never built due to mechanical complexities. First true computer concept, using punched pasteboards for program storage.
1854	Boole developes an algebra of logic.
1890	Hollerith uses punched cards for 1890 census tabulation; later founds IBM Corporation.
1938	Shannon publishes "A Symbolic Analysis of Relay Switching Circuits." Applied logic design flowers as a result.
1939–1944	Harvard-IBM Mark I–IV computers–relay logic implementations of Babbage's concepts.
1945	ENIAC–Harvard-type tube computer.
1946	Von Neumann concept of stored-program computer architecture published.
1949	EDSAC–first von Neumann-type computer becomes operational, using vacuum-tube technology.
1952	Wilkes proposes microprogramming as rational approach to computer control-system design.
1950s	Tube computers–first generation.
1960s	Transistor computers–second generation.
Late 1960s	MSI technology employed: 1,000 transistor equivalents on a monolithic IC.
1960s–1970s	LSI technology employed. 10,000 transistor equivalents on a monolithic IC.
1968	Calculator IC's produced.
1971	First microprocessor produced (Intel 4004). Harvard-type 4-bit machine.
1970s	Personal computers. Processor-design methods applied to peripheral-support IC's.
1980s	VLSI in production. 500,000 transistor equivalents on an IC. Microprocessors perform mainframe functions. 16-bit and 32-bit machines produced on a single IC. *Wall Street Journal* reports on personal computers almost daily.

ARCHITECTURAL TYPES AND THE CYCLE OF COMPUTATION

In sketching the evolution of computers, we have referred to three major types: the Harvard architecture, the von Neumann approach, and its modification–the Wilkes's concept of a microprogrammed architecture. As noted, all three methods of organizing a processor's architecture are currently used. This categorization by major type

Figure 1-4
Harvard Machine: Basic Architecture

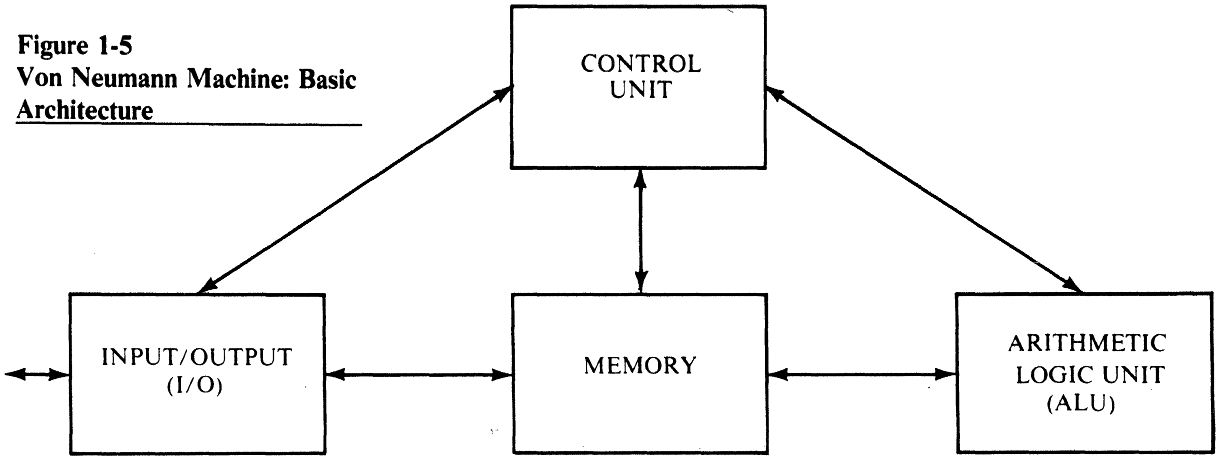


(Harvard vs. von Neumann) is apparent to a user who observes operational behavior. The recent VLSI processors are internally constructed on the basis of microprogramming principles, due to their complexity and this design method's simplicity. A review of the basic architectural features of these types, with comments on behaviorial characteristics, is in order. In the end, we shall study the microprogrammed type in depth.

These architectural types may be reduced to their fundamental architectures. Figure 1-4 presents the essential block diagram for the Harvard architecture. Note that the control section communicates with all other blocks. It issues the command signals that dictate performance. The arrows of the diagram are important because, in this case, they indicate that the completely separate *program* memory communicates *only* with the control unit and not with any of the other structural blocks. There is another, also separate, memory for storage of data. These two typically possess different word sizes. They are not expected to communicate with each other, in normal operation. Of particular significance, in this Harvard approach, is the fact that the ALU does not interact with the program memory at all: no path exists for the ALU to operate on program information. The Input/Output block (I/O) transfers information between the data memory and the "outside world." The ALU performs all transformations on data. That is, all arithmetic and logical manipulations take place in this block. It contains only combinational logic and no storage. It receives data from and returns it to the data memory. These are the basic blocks of the Harvard machine.

Already, the features of the Harvard machine that could affect our selection of a processor begin to emerge. First, its program is fixed in the program memory. The program is not dynamically alterable during operation, because only the ALU contains the power to alter information, and no path exists between it and the program store. We can expect this architecture to lend itself best to the less

Figure 1-5
Von Neumann Machine: Basic Architecture



complex, unchanging types of applications—as noted, toys, terminals, microwave ovens, etc. Somewhat less obvious is the fact that both memories may be active simultaneously. That is, the data memory may be used to fetch the current instruction's data at the same instant that the next instruction in the program store is also being fetched. While this inherent two-to-one speed advantage is not fully realized in practice, the fact remains that—for the same implementation technology and clock rate—Harvard machines can be faster than von Neumann machines.

Figure 1-5 presents the von Neumann concept's fundamental block architecture, which contains the broad applicability features of the stored-program digital computer. In this case, we have only one memory system. Both data and instructions reside in this single memory. In their formats, both use the same-sized unit of addressing to communicate with memory. The first important ramification of this is that, if we look at a random location in memory, we cannot be sure whether the bit configuration is an instruction or merely data that happens to look like an instruction. This implies that we need a tool to separate instructions from data. This is provided by reserving one of the memory locations for use *only* as a Program Counter (PC), the function of which is to keep track of where the next program step resides in the rest of the memory. The second ramification is that, since the instructions reside in the same memory as the data, they can also be transformed by the ALU. Now the machine can dynamically alter its own program. The other essential blocks perform the same functions as in the Harvard architecture.

A disadvantage of the early von Neumann machines was that they were constructed on a *hard-wired* basis. That is, the instruction set was a fixed, wired entity. Obtaining a large and modifiable set of instructions requires resorting to microprogrammable-design methodology. Even though Wilkes's concepts on microprogramming, as a rational approach to the design of a computer's control system, were well known by 1952, they did not achieve widespread use until semiconductor Read Only Memory (ROM) technology became very economical. Actual implementation of computer concepts has often depended on the availability of suitable fabrication technologies. The microprogrammed architecture is the last one we shall consider at this point, but we shall work with it in the remainder of the text. This is not a separate architecture from the von Neumann approach, only a better way to implement it.

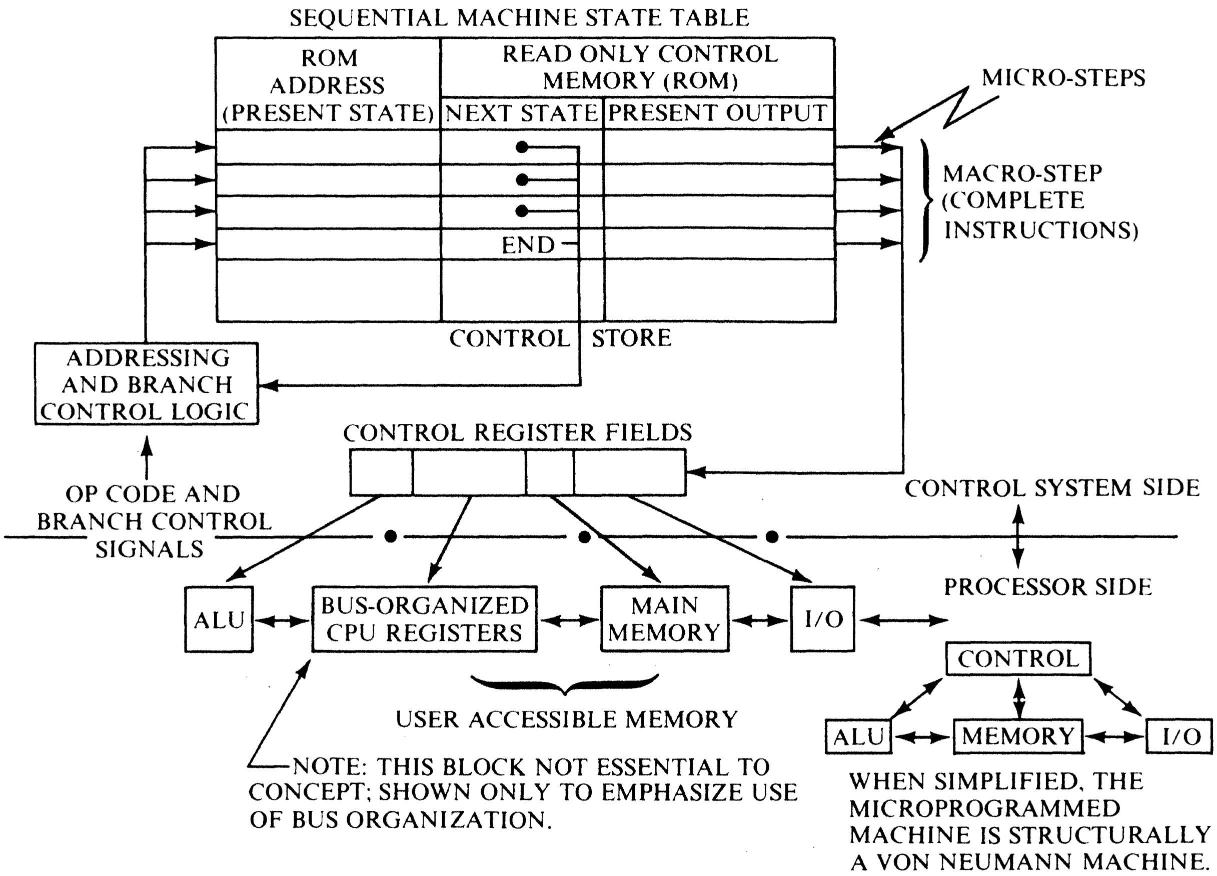
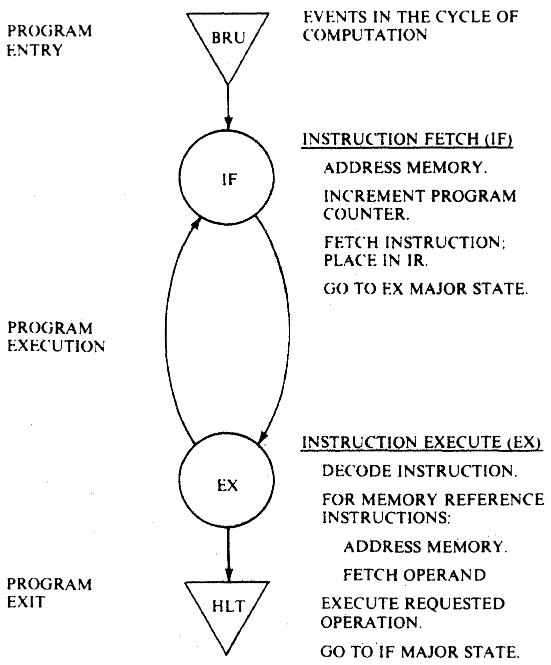

SEQUENTIAL MACHINE STATE TABLE
ROM ADDRESS (PRESENT STATE)
READ ONLY CONTROL MEMORY (ROM)
NEXT STATE
PRESENT OUTPUT
END
MICRO-STEPS
MACRO-STEP (COMPLETE INSTRUCTIONS)
CONTROL STORE
ADDRESSING AND BRANCH CONTROL LOGIC
CONTROL REGISTER FIELDS
OP CODE AND BRANCH CONTROL SIGNALS
CONTROL SYSTEM SIDE
PROCESSOR SIDE
ALU
BUS-ORGANIZED CPU REGISTERS
MAIN MEMORY
I/O
USER ACCESSIBLE MEMORY
NOTE: THIS BLOCK NOT ESSENTIAL TO CONCEPT; SHOWN ONLY TO EMPHASIZE USE OF BUS ORGANIZATION.
CONTROL
ALU
MEMORY
I/O
WHEN SIMPLIFIED, THE MICROPROGRAMMED MACHINE IS STRUCTURALLY A VON NEUMANN MACHINE.

Figure 1-6
Microprogrammable Machine: Basic Architecture

Figure 1-6 portrays the organization of the Wilkes's microprogrammed architecture, stressing its State Table organizational features. Many simple things appear complex because they are sophisticated. This figure contains more detail than is comprehensible right away. Since it is the system we really aim to explore, let us introduce some of this detail in an overview. A ROM is the key feature of its control system. Again, we have a processor with two separate memory systems. The main memory contains both the data and the program's instructions that are to be executed during operation. The ROM memory is referred to as the Control Store. It, too, contains instructions, *but of an entirely different class.* These instructions are the sequence of marching orders that control the step-by-step operation of the system. Each discrete control order is properly termed a *microstep.* A collection of these microsteps, sequentially issued, forms a *macro* (or algorithm) that the machine performs. These macros contain the processor's sequence of orders for the execution of an instruction that has been fetched from main memory.

The collection of macros that reside in the control store makes up the stored algorithms for the step-by-step execution of each instruction of a program. This collection of macros makes up the *instruction set* of the system. It is of utmost importance to realize, at the very outset, that the contents of the control store ROM must have the form of a state table for a sequential machine. After all, a processor is a sequential machine that can be described (and controlled) by its state table. Each instruction step in your program is sequentially presented to the control store. The control store, figuratively, says "Aha! so that's what you want me to do. O.K., I will look up the steps for performing

Figure 1-7
The Basic Cycle of Computation



your program's instruction in my State Table, and I will sequentially issue them to the hardware of the Central Processing Unit (CPU)." Therefore, your instructions in the program lead to the selection of the sequence of steps, found in the state table of the control store, that actually get executed. This addressing of control store's ROM and the subsequent receipt of a word (a command) are stressed in the figure.

If we analyze Figure 1-6 closely, we find that we can reduce it to the essential blocks of the von Neumann architecture, as shown. It only looks complex, now, because we have prematurely enlarged upon the details associated with the control system, to introduce its fundamental block structure and state-table organization.

CYCLE OF COMPUTATION

Because architectural types are capable of affecting the computer's sequence and timing of the events in a cycle of computation, we must define this cycle. We shall do this for the architecture of interest in this text, the von Neumann architecture. A processor is a sequential machine. That is, it is a machine that can be in a finite, albeit large, number of states or conditions. Each "next state" of the system must be predictable from a knowledge of the "present state" and the "external inputs" applied to it in this state. For every input, the system must possess a defined state path. This information is contained in its control store. While we discuss these points in more detail later, the overall behavior of the computer can be reduced in complexity by considering its *two major states*—Instruction Fetch (IF) and EXecute (EX). All other states are minor states within these two. This is instructive because we can simply display the principal occurrences in a single cycle of computation in relation to these two major states.

The cycle of computation is portrayed in Figure 1-7. How is the cycle initiated? Let us assume a simple case, in which the operator has loaded the program to be run into the computer's main memory and has given the PC an initial value, which is the program's entry point.

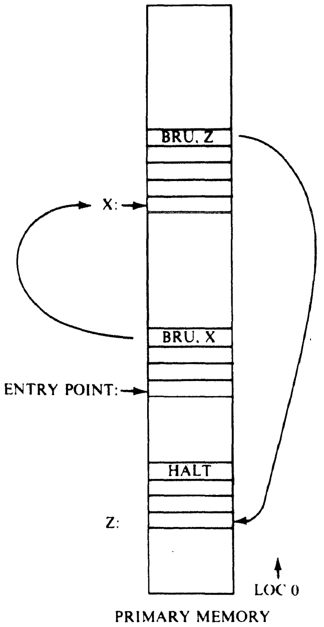
The front-panel controls are assumed to have these capabilities, as in the old-time computers that required an operator. By way of initializing the PC, the operator could set the console switches to order the system to perform a BRanch Unconditional (BRU) to some starting address. At this point, the operator hits the Run switch and stands back. The cycle of computation's sequences commences with the operator's BRU command. The BRU execution consists of placing the branch address of this initial instruction, forced by the operator, into the PC. Thereafter the machine behaves as shown in the figure until a HALT instruction, say, is encountered.

After the execution of the forced BRU, the machine automatically enters the IF major state. The system PC is now pointing to the memory location of the next instruction to be fetched. The essential events of the IF sequence are shown in Figure 1-7. The first event is to address memory by transferring the contents of the PC to the Memory Address Register (MAR). MAR is another portion of the system's memory reserved for a special purpose. This register *only* selects a location to be communicated with in main memory. It does not handle data. Since this address came from the PC, the *contents* of this memory location had better be a valid instruction. Let us assume, as in early computers, that each word in main memory contained the complete instruction format. The second event performed in the IF state is to readjust the PC so that it will point to the *next* instruction in main memory. This is done by incrementing its contents. Now, the PC is looking ahead again. *As we use the PC, we also increment it.* MAR still contains the original contents of the PC or the address of the instruction we wish to fetch. In another event, we read the contents of this location and transfer it into the Instruction Register (IR). The IR, like the PC, is yet another example of a dedicated specialized use of the system's generalized memory. In the final event, the system ends the IF phase by issuing the command to proceed on to the Execute (EX) phase of operation. We stress the point that a small portion of the system's total generalized memory is dedicated to these special usages.

These four events of the IF phase of operation occur on every von Neumann-type machine. Some of them may be performed in parallel, so it is not always true that a particular machine requires four separate clock pulses to complete the sequence. The control system records the fact that the EX state has been entered. In the EX state, it looks at the *contents* of IR and figuratively says "You want me to do *that*!" and so on. From decoding the contents of IR, it now can find the macro in its control-store state table for the sequential execution of "that." The OPeration portion (or field) of IR is, in reality, the encoded address of a macro's starting location in the control store. The last step of every macro contains an END order, to return the system to the IF phase. The system continuously cycles through this cycle of computation—IF to EX, and back again. Each time the machine returns to the IF state, the PC must already be conveniently pointing to the address of the next instruction. The microprogrammer that creates an instruction (macro) is responsible for properly advancing the PC; we shall practice this later.

In the EX major state, the processor may have to use memory again. This is important to recognize, since about 85 percent of all instructions are memory-reference instructions. That is, the instruction word itself contains a field that provides the address of an operand. In this case, the address field of the instruction is transferred to MAR

Figure 1-8
Memory Map of a Program's Blocks



during the EX phase. The object, a piece of data, is brought in from memory to be operated on in the course of executing the instruction. Memory, then, is typically accessed *twice* within the full cycle of computation. The first time, during IF, was for the purpose of fetching the instruction (which usually contains an operand address). The second time memory is accessed, if the instruction mandates this during the EX state, the data to be operated on is fetched. Therefore, two memory accesses in a single cycle of operation are usual. This is natural, since *both* data and instructions reside in main memory.

The novice should carefully study the events in the cycle of computation, until they are fully understood. This cycle is the basic description of the operation of the von Neumann-type computer, the type of architecture used in most personal or business computers. It also illustrates when and why memory is accessed–usually twice during the cycle of computation. An understanding of these events is essential to the microprogrammer's proper control of the PC. It also leads to an appreciation of how a given system actually executes a particular instruction.

The von Neumann architecture's cycle of computation also tells us something about the way memory space is utilized by this type of machine. We can now see how a program is mapped into total main-memory address space, shown in Figure 1-8. Since the PC carries the information of where the instructions are in memory–and since the PC is incremented only during the cycle of computation–one instruction *must* follow another in memory, except for branches. That is, instructions reside in contiguous locations in main memory. These contiguous locations of instructions may be formed into blocks of memory. For a given program, these blocks are linked by the address fields of the branch (or jump) type of instruction. Recall, the execution of a branch instruction simply amounts to a reloading of the PC. The rest of memory that is not occupied with blocks of instructions may now contain the data these instructions access or the results they store.

The cycle of computation simplifies the computer into a machine with two major states. This state diagram may be slightly expanded to

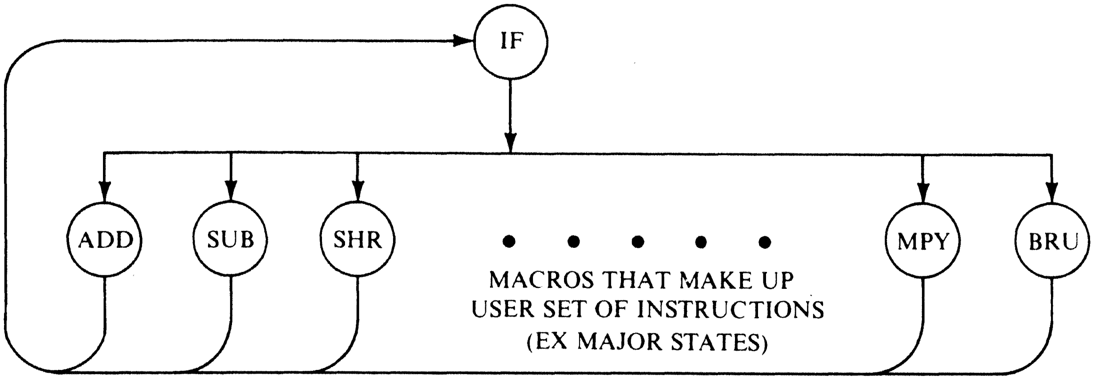


Figure 1-9
Expanded IF-EX Major State Diagram

better represent the actual case, as is shown in Figure 1-9. In this illustration, we have only one IF major state. After all, IF is IF, and we need only one copy of it in the control store's state table. However, the other macros in the user's instruction set are not used in every cycle of computation. Therefore, we require a separate state-table segment, in control store, for each macro of the EX major state. The actual cycle proceeds, in IF, to the fetch and subsequent decoding of the instruction word. This decoding process, during the EX major state, selects the particular macro specified in the instruction word. That is, the system executes only one user macro at a time and then returns to IF to fetch another. IF therefore is not categorized as a user macro. It is transparent to the user. Nevertheless, it *is* a macro–automatically invoked and performed by the system in the course of executing our programs. In a microprogrammable system, anything that the machine is capable of performing must be a macro residing in the state table of control store. We shall refer to the above cycle of computation as a basic frame of reference many, many times.

Forms and Uniformity among Processors

Having reviewed the major basic architectural types among computers which, as we have said, are one form of processor, we may now consider the question–what is a processor? We propose to answer this question in detail in this text by examining the internal organization, microprogramming, and functioning of an instructional archetypical Central Processor Unit. The essential structure of this CPU's architecture will be shown to be closely related to that of the Intel 8080 microprocessor, the DEC PDP-11 minicomputer, the AMD 2901 bit-slice IC, etc. Therefore, we are stressing the universality of the principles behind the organization of a processor. Here, we must be careful to distinguish between the organization of a computer's CPU and that of the total computer system. Briefly put, the organization of a computer system largely consists of the CPU *and a number of other processors;* a means of intercommunication, the so-called busing structure; and the system software that is executed by the CPU to coordinate the whole. Our emphasis is on the concept of the generalized processor and its essential characteristics.

To be more explicit, let us examine the block diagram of a computer system, Figure 1-10. This figure displays channelized computer-system organization, in the form most commonly seen in microcomputer systems. Variations of this structure are applied to large, so-

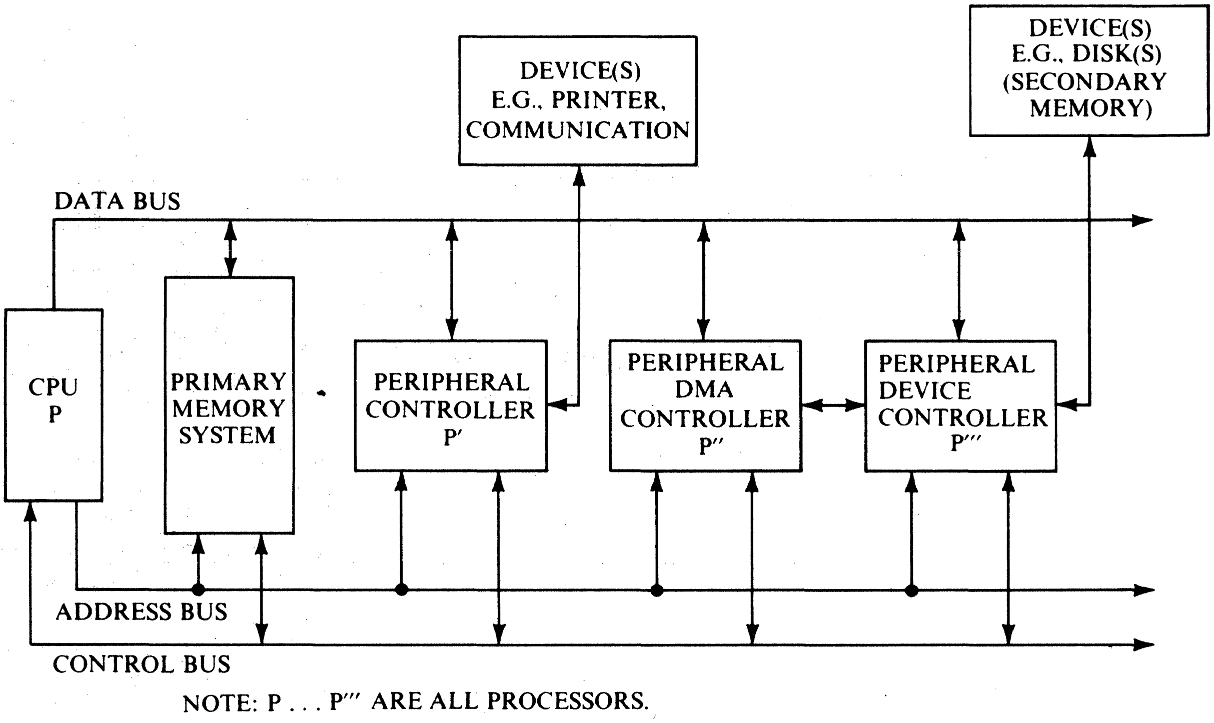


Figure 1-10
Channelized Computer Systems Organization

called mainframe computer systems, as well as to small systems. It is a universally used form of organization and could be the starting point for a discussion on operating systems and their environment. What we wish to highlight now, however, is the large number of processors present within the system. The term CPU itself implies that a computer system consists of a central processor and a number of other ones. These other processors are the peripheral controllers whose coordination is orchestrated by the CPU. As shown, these processors peripheral to the CPU are of two types—Direct Memory Access (DMA) and those with indirect access to memory. By *access* we mean access to the system's primary memory—as opposed to access to mass-storage devices or to secondary memory, such as disks.

The indirect-access processors exchange data with primary memory via the CPU. The direct-access processors can assume direct control of primary memory for a transfer of information between it and another entity that does not involve the CPU.

Our point is, then, that a computer system consists of a number of processors, coordinated by the system's software. The peripheral processors are the Arithmetic Processor Units, the DMA controllers, communications controllers, disk (secondary-memory) controllers, graphics controllers, and others. These days, they are most often sophisticated programmable devices housed in a single IC. Any one of them can be as complex as the CPU. An 80-bit floating-point arithmetic processor peripheral is as complex as most CPU's. So far, we have said that these processors are all sequential machines. We will study the organization of the microprogrammable form of the CPU to learn what a processor really is—and we do mean what it *really* is. If we understand this one, we will have gained an intuitive feel for all the other forms of processors, for they all reduce essentially to sequential automata. In developing microprograms for an instructional CPU, we have to display an intimate knowledge of sequential machine behavior. If we gain an appreciation of the essential common features of processor organization, then we should be better prepared to understand, work with, and program the other processors of the total computer system. This information is also a background for the synthesis and implementation of processors.

Thus we consciously used the term *processor*—as opposed to *computer, microprocessor, dedicated processor, arithmetic processor, DMA channel,* etc.—to stress the unity of their fundamental design and function. They are all *sequential machines.* Chapter 4 reviews sequential-machine principles, because the fundamentals of sequential automata are the foundation for our understanding of processors. Microprogramming is a generalized approach to their design that, with today's emphasis, can have a strong software component. We wish eventually to explicitly illustrate how a microprogrammable type of architecture is implemented. Its basic nature is no different from all the other types of processors, regarded as sequential machines. An internal or even a user-accessible microprogrammed memory, used as the core of the sequential machine's control system, is an integral part of most present large-scale systems organizations. Via the techniques of microprogramming, we can view the rationalization of control-system design for processors as one of the technological advances that has made VLSI fabrication feasible.

Let us briefly look at the external physical details of a processor, as represented by some microprocessors. They are usually classified as members of one of the three previously mentioned architectural types. That is not to say that other important architectures do not exist. The array processor, for example, has a distinctly different systems organization, yet its individual functional blocks likely behave as a member of one of our major categories. Do microprocessors differ from what we mean by *computers*? Not really—the differences are only of degree, not kind. They still are computer-system processor units, implemented today through VLSI technology. The differences are variations in physical size, word size, instruction formats, program throughput, and the complexity of the instruction set. A microprocessor's instruction set now may be as sophisticated as any of the older mainframe computers. Even its bus and word size have entered the 32-bit region, formerly the domain of the mainframe. The speed and corresponding productivity of the microprocessor can be much better, due to the physically small, monolithic nature of its IC. The fact is that microprocessors are beginning to replace mainframes, in many applications. The mainframe will still be around in the future, but it will be a *mainframe.*

Some physical statistics for a few IC processors are of interest. They indicate advances in fabrication techniques—rather than the essential nature—of processors. The first production microprocessor, the Intel 4004, came in a 16-pin Dual In-line Package (DIP), whose external measurements were 0.30 × 0.78 inches. The silicon surface area (the amount of silicon "real estate") used for the fabrication of the actual IC measured 0.117 × 0.159 inches. The number of active devices contained within this area, in several layers, is referred to as the number of *transistor equivalents.* The 4004 contained on the order of 2,500 transistor equivalents. Its word size was four bits: that is, the data bus handled and it internally operated on four bits of information at a time. It was followed by the 4040 microprocessor, packaged in a 24-pin DIP, externally measuring 0.6 × 1.25 inches. The microprocessor surface area size measured only 0.118 × 0.163 inches. The Motorola 6800 8-bit microprocessor was packaged in a 40-pin DIP measuring 0.6 × 2.0 inches, as was the Intel 8080 and most other 8-bit microprocessors. The Intel 8086, a 16-bit machine, was also packaged in the same size 40-pin DIP. Many of the peripheral support processors are also packaged in the 40-pin DIP. For a time, 40 pins was the practical limit to package size. Current packages are larger, with approximately 60 interface pin connections. As manufacturing technologies have improved, more and more logic has been designed into the monolithic IC. The 8086 contains on the order of 80,000 transistor equivalents. The Intel 80286 16-bit processor contains approximately 130,000 transistor equivalents. The current frontier is about 500,000 transistor equivalents in a single monolithic IC.

Because of the growing sophistication of microprocessors and the limitations on package size, extensive time sharing of the interface pins was resorted to. Time sharing of an interface pin, also called *Time Division Multiplexing* (TDM), is frequently employed to reduce the number of required interface connections. Generally, manufacturing and reliability problems increase with the number of interface connections. TDM typically allows a particular interface pin to function, say, as an address-bus line in one clock period—yet serve as a data-bus line in another. TDM is common practice, often confusing to the beginner. It may help to imagine that an interface pin can be switch-connected to an internal address bus in one time frame, then to an internal data bus line in another. Thus, a single interface connection performs entirely different functions at different times.

All the preceding indicates two things. First, today's microprocessor is becoming the previous decade's smaller mainframe, because of the amount of logic a monolithic IC may now contain and because of improving design methodologies. Second, the typical processor of today is housed in a monolithic IC—from peripheral controllers to the CPU's of microprocessors and mainframes. What we hope will become apparent in the course of this text is the role of the techniques of microprogramming in the creative shaping of the functions these processors can perform; equally important is the fact that we can be in control of the specification of these functions.

We now also raise the rhetorical question: How does one learn about these production processors? The answer is that, after the introductory orientation of a text, we read manufacturers' data manuals. Much of the material of many good texts is obviously derived or even taken straight from the data catalogs of these IC's, unadulterated. We must stress the importance of learning how to go—independently—straight to the source, thus gaining self sufficiency.

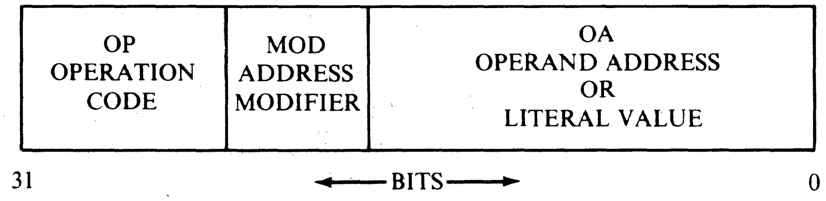
Mention should be made of a special group of processor IC's, the bit-sliced–based dedicated controllers, CPU's, and computer emulators. They have played an essential role in the technology of computation, with less publicity than the mainframes and microprocessors. These processors have been organized around the use of *bit-sliced* architectures. The bit-slice is a basic bus-organized central processor unit, containing only an Arithmetic/Logic Unit, a register array, and a general-purpose load and shift register, with associated bus paths and combinational logic. It does not have a control system built into it. The user supplies this, via microprogrammed design. In creating a computer or dedicated processor with these bit slices, we simply employ as many of the slices (in parallel) as we need to achieve the desired machine size. Typically, we now add to this a microprogrammable control store, to form the processor's control system. The resulting machines have been used to create dedicated controllers, such as high-speed disk controllers. They have also been used to create fast emulators of existing older mainframes and to form specialized computers, such as those employed in avionics applications.

A popular bit slice has been the Advanced Micro Devices Am2901 and Am2903 4-bit slices, housed in the same 40-pin DIP as many of the other microprocessors and peripheral controllers. There is a family of other related IC's that support the creation of the total bit-sliced processing system. This means that the designer forms a processing system, using several different types of IC's. In this do-it-yourself approach, the processor is not a single IC but consists of several coordinated IC's from the family. The control system that we shall study and implement here reflects the function and nature of these other IC's in the family. The 4-bit processor we shall study, construct, and microprogram is a simplified model of a bit-sliced IC. As we shall see, our own archetypical processor project will be a bus-organized CPU containing a Register Array, an ALU, and a few dedicated registers–much like the bit slice. To this we will add a microprogrammable control store for sequential behavior control and an external memory system. These major structural features form the basis for the type of processing system we call the computer. It is a hands-on, nonhypothetical approach to the subject matter.

At the present writing, a new trend has emerged–due to the availability of advanced VLSI IC-development technologies. It is now possible for any of us to specify and design custom processor IC's. This, too, is a do-it-yourself approach to the design and implementation of custom IC's, which relies on the very same system structural-block approach we shall consider here. Many applications–communications, disk and instrument controllers, for example–have sufficiently high production volumes to justify the development of custom processors in the form of the monolithic IC. We can apply the same organizational principles of microprogrammable systems to this task, too. This important phenomenon is one indication of the spread of processor design and implementation knowledgeability to the mass production, low-cost stage that each of us can participate or lead in. It is now realistic for us to anticipate our own involvement in any of the design, microprogramming, and software-development phases of processor-controlled products–provided that we understand the systems organization of the archetypical processor.

From the discussion so far, we see that processors may be anything from a simple dedicated controller to a microprocessor to a computer mainframe. To understand this myriad of devices, we need a common center of reference. This we provide by examining the sequential machine organization and microprogramming of a 4-bit CPU. The applied microprogramming-systems principles that we shall study are exactly what a processor is organized around–this is the unifying factor, which encompasses both hardware and software. These processors may be used for many different purposes, but their fundamental sequential-machine organization is the same. Our view, then, is that processors consist of the systems application of a set of principles for the sequential-machine organization of computation and control. Differences are largely due to scale and end-function adaptations, rather than to their essential features.

Figure 1-11
Mainframe Instruction Word: Typical Fields (Single Address)



INSTRUCTION-WORD FORMATS

Let us now proceed on to an important examination of the *apparent* difference of form between the instruction word formats among those processors we refer to as computers. A computer is a processor that contains a large, flexible, instruction set suitable for multipurpose adaptation, via user programming, to a variety of applications. These range from the 4-bit microprocessor to the 64-bit or larger mainframe. Common word sizes in microprocessors are 4, 8, 16, and now 32 bits. Minicomputers typically have 12- to 18-bit word sizes; mainframes are considered to have 24-bit and larger word sizes. An early number-crunching machine even had 128-bit words.

It is totally incorrect to say that microprocessors are slower than mainframes. The frequency of the driving clock for a microcomputer is often greater than that of a mainframe system. Execution times, say, for addition, are often comparable—even better. The real differences in performance, for a given clock rate, depend on the number of bits operated on at a single time in the process of executing a command. An 8-bit micro adds two 8-bit quantities in one clock period, while a mainframe may handle two 48-bit values in one period. By using repeated addition (with carries), the microprocessor can add two large 48-bit numbers just as well as a mainframe. Since this also requires a good deal of thrashing about between I/O, memory, register exchanges, etc., many extra operations are performed by the small-word machine to obtain the same resulting precision. If the small machine is not superfast, then its *throughput* suffers in comparison with the large machine.

Throughput is a key criterion of application performance. In measuring throughput, we refer to the techniques for evaluating how long it takes to process a set of problems relative to some proposed application. Two machines of the same word size may perform in significantly different ways in a given situation, often due to differences in their instruction sets. This important aspect of processor evaluation is called *benchmark* testing. A number of firms specialize in providing benchmark tests for competing processors for given applications. In practicing microprogramming, we shall *implement* an instruction set. This will create an acute awareness of the effects of architectural design and of the choice of instruction sets on the speed of operation, which affects a system's throughput.

The monolithic-IC computer often utilizes a clock frequency significantly higher than that of distributed mainframes. Associated with it is a smaller word size. It has to hustle to obtain the same throughput. Therefore, the distinctions between word sizes, instruction formats, instruction words, and the addressable unit of memory deserve to be discussed here. We will constantly be dealing with these concepts later.

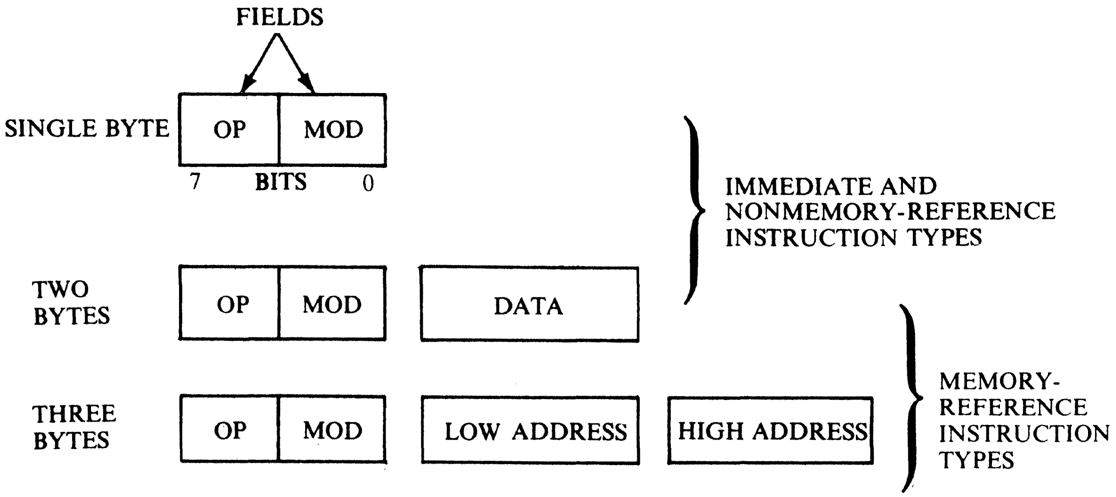

FIELDS
SINGLE BYTE
OP
MOD
7
BITS
0
TWO BYTES
OP
MOD
DATA
THREE BYTES
OP
MOD
LOW ADDRESS
HIGH ADDRESS
IMMEDIATE AND NONMEMORY-REFERENCE INSTRUCTION TYPES
MEMORY-REFERENCE INSTRUCTION TYPES

Figure 1-12
Eight-Bit Microprocessor Instruction Formats: Typical Fields

Further, these concepts begin to make it clear that *apparent* differences are not fundamental ones.

In the early computers, instruction formats were straightforward. The complete instruction word was stored in a single word in memory and could be held in a single register within the CPU. The format of this instruction word, however, is invariant for most machines of interest to us. That is, they all are divided into the same sets of fields. This is presented in Figure 1-11. The typical complete single-address instruction word consists of three subfields, as shown. These fundamental fields are the OP code, MODifier, and Operand Address (OA) fields. Two- or three-address machines simply make repeated use of one of these field types—the OA field. The OP field contains the binary code naming the operation to be executed. The OA field contains either the address of an operand or some value specified in the program. Whether this field contains an address or a value depends on the nature of the instruction specified in the OP field, that is, on whether the instruction is ADD or SHIFT. If the OA field is in fact an operand address, then the`MOD field is used to specify how the final Effective Address (EA) is to be calculated. The MOD field specifies the addressing modes we can select for our programs.

The modes of addressing supported by a computer are important factors in how effective its throughput is. The usual ones, which we shall later microprogram, are the register, immediate, direct, indirect, indexed, and the autoincrement/decrement modes.

The early single-address machines stored all three fields in one memory word. The sizes of the instruction word, memory word (addressable unit of memory), and the register were the same. Minicomputers and microprocessors have word sizes in the 4–18-bit range. This requires that we distinguish between the complete instruction word and the size of the addressable unit in memory (the memory word). Currently, memory is often addressed using the *byte* (eight bits) as the unit for numbering its addressable locations. Certainly, this is true for the 8-bit machine. How does this affect the typical single-address instruction format? The answer is—not at all. What it does affect is its *distribution* in memory in the addressable units of memory required to contain it.

A comparison of the instruction word format of a typical 8-bit microprocessor with the early mainframe type above should clarify this. This format for the 8-bit machine is presented in Figure 1-12. The 8-bit machine customarily uses the byte as the addressable unit of memory, as do many larger processors today. It is also customary for the first memory word addressed in the process of fetching the complete instruction word to contain the OP and MOD fields of the entire instruction. If the instruction can be completely specified in this one byte, e.g., a shift operation, then there is no need to use any more addressable units of memory to form the complete instruction. Therefore we can find single-byte complete instruction words on these small register-size machines.

There is another type of instruction, which we shall later implement, that moves the byte following the OP and MOD fields of the first byte of the complete instruction word into a register. This instruction word is completed in two bytes (words) of memory, as is also illustrated in Figure 1-12. Thus we have one-byte and two-byte instruction formats, consistent with the formats of the early mainframes. Memory space not needed is not simply wasted. Finally, another type of instruction—let us use addition as an example—specifies the operand's address in primary memory in the complete instruction word. Again, the customary practice in 8-bit machines is to provide an address-bus size of two bytes' (sixteen bits') worth of memory-address space. An instruction that specifies a complete address in memory requires three bytes to form the complete instruction word. Its format is, again, the same as the one for the mainframe—with all fields of the format required and present. Note, then, that complete instruction words on small machines can contain 24 or more bits—the same as mainframes.

Thus we see that machines with small word sizes have the same instruction formats as the very large-sized ones. Insofar as instruction formats are concerned, we are looking only at apparent differences—not real ones. Because they do not waste the space for fields not used in a specific instruction, the complete instruction word may consist of one or more physical words (addressable units) in memory. The small-register-size machines simply use a variable amount of memory space to contain a complete instruction, for reasons of economy—a privilege not available to the early mainframe. From this we can see that there need be no fundamental differences among microprocessors, minicomputers, and mainframes, regarding the function and formatting of their instruction words.

Practice varies. The Signetics 2650 8-bit microprocessor extended the OP and MOD fields to nine bits. The ninth bit, the most significant bit of the second byte of the instruction word, was used to specify indirect addressing. This reduced the memory-address space to fifteen bits. It truly had a minicomputerlike instruction set, which was extremely satisfying to use, at the expense of the amount of memory it could directly support. The PDP-11 16-bit minicomputer uses variable-word-length instruction formats, too. In addition, many of its instructions support two-operand addresses. The first word (two bytes) of an instruction in memory defines the OP, MOD, and OA fields, in a manner analogous to those previously discussed. The PDP-11 can define two MOD and OA fields—one for the source, the other for the destination—for its two operands. Yet, the fundamental fields employed in its sophisticated instruction word formats are the same as those named in Figure 1-12.

In summary, then, a processor is any of a myriad of sequential machine devices, which may have many apparent differences of form. They are all, however, programmable sequential machines. They may be comprehended, in their essence, by examining the broad fundamental organizational principles of a simple microprogrammable system. If we take advantage of the hands-on approach presented later in the text, by constructing and microprogramming the instructional system out of readily available IC's, we shall face the real and creative challenges of conquering the system. This is a far better alternative than remaining its victims.

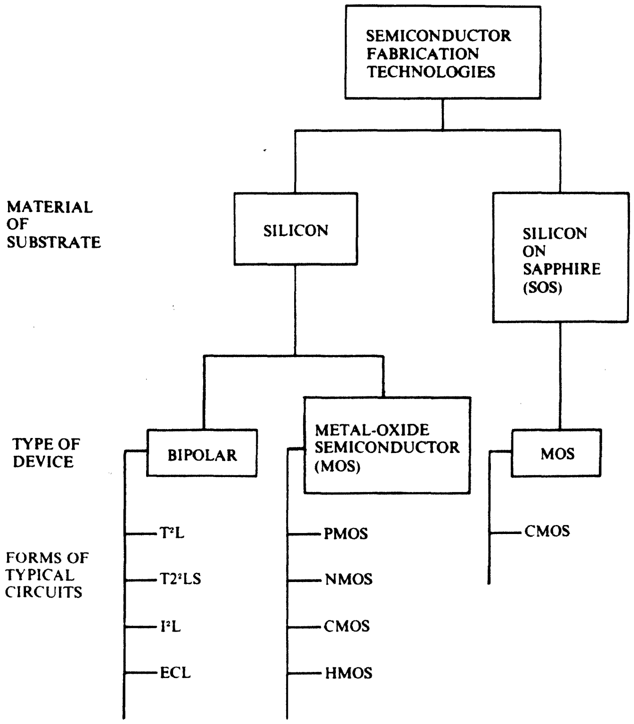
Processor Fabrication Technologies and Personal Impacts

Included in this introductory view of processors are the programmable microprocessor-support IC's. This is precisely why the software-oriented person must comprehend processor organization to do systems software development with confidence. The Direct Memory Access (DMA) controller, the floating-point Numeric Data Processor unit, the graphics or CRT controllers, the Floppy Disk Controller, and the Communications interface IC's are all examples of programmable devices–just the sort of devices we need to understand if we are to dominate the computer system, instead of the reverse. They, too, have instruction sets. Their instruction sets are the set of *activation commands* issued by the so-called central processor *under its programmer's software control.* To program these devices via the central processor, it helps to understand their intrinsic nature, which is almost universally uniform. The manufacturers of microprocessor-controlled systems require ever more software support in such projects as electronics instruments, hospital-patient monitors, and the production of applications software for personal computers; it will help if we demystify what this very large range of processors is about. That is our major goal in constructing, microprogramming, and operating a processor.

Previously we introduced the concept of the existence of generations of processors. These generations were roughly based on the fabrication technologies employed in their construction. Broadly, the mainstream of computer development proceeded from mechanical devices to relays to tubes to semiconductors. It has been asserted that the widely used concepts on how to organize these processors, that is, their architectures, have been few but persistently employed. Therefore it is primarily in fabrication development that the dynamics of processor construction resides. As noted, we are most likely in practice to encounter the Harvard, the von Neumann, and its variant–the Wilkes microprogrammable concept–system architectures. There are important organizational exceptions, such as array processors, but these are special-purpose machines. Generally, they are design responses to specialized, important, but limited types of computational needs.

The interesting fact is that a computer system can contain many processors. The peripheral LSI and VLSI processors mentioned outnumber the CPU's. Their organization as sequential machines, describable in terms of microprogrammed-design methodology, implies their fundamental unity. There are many of these mass-produced processors in current computer systems. In fact, they are just as much the basis for the current widespread use of computers as is the CPU itself. They are fabricated as monolithic IC's, too. Thus we see that advances in the

Figure 1-13
Summary of Some Current Microprocessor Fabrication Technologies


SEMICONDUCTOR FABRICATION TECHNOLOGIES
MATERIAL OF SUBSTRATE
SILICON
SILICON ON SAPPHIRE (SOS)
TYPE OF DEVICE
BIPOLAR
METAL-OXIDE SEMICONDUCTOR (MOS)
MOS
FORMS OF TYPICAL CIRCUITS
T²L
T2²LS
I²L
ECL
PMOS
NMOS
CMOS
HMOS
CMOS

art of building the processor IC's of a computer have helped make IC's such a common feature of our lives and that there exists a common basis for understanding them. The personal impacts of advancements in fabrication technologies of processors are widespread. A short discussion of recent trends in the evolution of computers shows why this is occurring and how it affects all of us.

Semiconductor-fabrication technology commenced with the germanium bipolar transistor and diode as its basis. This was soon replaced with the silicon bipolar discrete-device technology. Then came the metal oxide semiconductors, which were progressively incorporated into the monolithic structures of the small-, medium-, and large-scale monolithic logic IC's. The emphasis on developing better fabrication techniques has been such that an entire tree-structured spectrum of these technologies exists, with important application-dependant choices to be made. Figure 1-13 introduces this perspective. This figure presents just some of the more widely used semiconductor-fabrication technologies. Experimentation is constantly proceeding, for example, with gallium arsenide semiconductors, in the attempt to improve speed, reduce power, and achieve economic viability. The technology at the forefront of development efforts changes almost yearly. In the late 1970s, IIL transistor arrays attracted much attention. Currently, CMOS processor and memory IC implementations are emphasized.

Speed and power are important parameters in the selection of a fabrication technology that addresses the application needs of the end user. The early electronic machines were slow and consumed tremendous amounts of power to remove the heat they produced. Roughly the same amount of air-conditioning power is required as the machine employs in its operation. Office automation with processors simply is not feasible if the cooling system cannot handle the load. The inherent speed capability of a particular semiconductor technology affects the throughput that it can achieve for a given architecture. Early microprocessors used PMOS technology and the Harvard architecture. This choice of architecture also masked the slower speed of PMOS, as compared to NMOS or HMOS fabrication. The Harvard architecture continues to be considered for specific applications, to further enhance the inherent speed/power properties of a given technology.

Table 1-2 presents relative factors of merit for some of the microprocessor-fabrication technologies. As noted, architectural choices can use inherent speed and power properties as a platform for further improvement of performance. All this is subject to the economics of the costs and yields obtainable with a particular technology, but the development process continues unabated. The result is that several successful implementation technologies are readily available for the rapid implementation of new processing systems. These new IC's meet the speed, power, and production criteria required for mass marketing. A good example of this is the portable personal computer.

Table 1-2
Relative Factors of Merit in Microprocessor Fabrication Technologies

Technologies		*MOS*		*SOS*		*Bipolar*	
Factors	*PMOS*	*NMOS*	*CMOS*	*CMOS*	*T²LS*	*I²L*	*ECL*
Speed 1 = Slowest	1	2	4	5	6	3	7
Power Req. 1 = Most	4	5	7	7	2	6	1
Packaging Density 1 = Least	5	6	3	5	2	5	1
Process Complexity 1 = Most	7	6	4	1	3	3	2
Experience Factor 1 = Least	7	7	4	1	7	2	3

As semiconductor-fabrication technologies improved, modular IC building-block components were introduced. The availability of these components and their subsequent inclusion in large-scale fabrication designs have had a great impact. These formed the basis for regularizing organizational approaches to the implementation of sequential machines. The first of these popular devices to achieve wide usage was the Programmable Read Only Memory (PROM). This is a form of user-programmable permanent memory, which we shall discuss again, that retains its contents even when the power is turned off. Anyone who has had to bootstrap load an early machine with paper tape, on powering it up, heaves a sigh of relief when turning power on to a personal computer, then relaxing while the boot program in PROM brings in the disk operating system. The availability of the PROM solved the problem of how to have a fixed program permanently available on power-up. It also solved the problem of having an immediately available state table in control store for the control of a processor. The PROM, with added registers, is employed in the construction of microprogrammable-control systems.

In the 1970s, the Programmable Logic Array (PLA) came on the market. It was followed by a variant, the PAL, in the early 1980s.

Here, user programmable gate logic permitted incorporation of large amounts of combinational logic into a single IC. The PLA, in conjunction with registers, is also used extensively in control systems. This construct is called a Programmable Logic Sequencer (PLS). A pattern is beginning to emerge here. As modular components gained widespread use, they were also incorporated into the design and construction of the processor IC. This continues to this day. The discrete IC's that we shall work with in our analysis, construction, and microprogramming all have their functional equivalents incorporated into the design of a monolithic processor IC.

Gate arrays are semiconductor IC's containing a large number of gates. The end user specifies how they are to be connected. From the basic gate, we may construct an entire processor. The logic gate may be perceived as a fundamental building block. Most of these modular do-it-yourself structures are regular in nature, and they are therefore easily incorporated into a custom IC. The craft of VLSI design has evolved to the point where it is no longer necessary to work with fixed arrays. Individual gates can be specified in Computer Aided Design procedures to tailor their speed and power properties to the needs of the application. Further, libraries of previously developed modules are available for immediate incorporation into proposed processing systems.

The pattern that emerges here is that processors are constructed by using fundamental constructs that were originally available in separate IC form. These constructs are currently available in library form, which the designer merely names (with associated parameters), making it instantly available for incorporation into a custom IC's system. New, programmable processor IC's are reaching mass markets at a rapidly accelerating rate. Therefore the personal impact here is on our level of understanding of processing systems, so that we may knowledgeably work with them. This applies equally to hardware and software types. The computer we work with or own contains not one but several programmable processors. In the end, it is the system software that rules the roost. To design, use, and program these processors effectively, we require insight into them. An understanding of both hardware and software is essential to current computer-systems development. Operating-systems development, graphics, communications, video games, and business software all interact with a wide variety of programmable processors.

To illustrate some effects of these last statements, the author sat with a former student who demonstrated the use of a Computer Aided Design system to specify a PAL, by naming it and providing the system with the number of inputs and outputs. This person was less than four years out of college. In a few seconds, the design features were displayed on the monitor screen for future integration into a larger and complete custom processor IC. Also demonstrated was the design of a gate with specifically enhanced performance properties. The final processor design that used these building blocks was developed in five months, contains over 9,000 transistor equivalents in a single IC—and it worked from the very start. This feat was performed by a capable individual who does not specialize in IC design, in response to a corporate need for a processor IC in a product. The age of rapid response to both hardware and software computational needs is here—and it affects us all.

The construction project we shall study in this text follows the pattern expressed above. We shall incorporate separate building-block IC's into a total processor-system design, to gain an understanding of processor-system principles. These principles, using functional modules, are exactly what is employed in the system-concept development of a custom IC. This, then, is the frontier fabrication technologies has brought us to. System developers may now rapidly construct their own processors, using VLSI custom IC-design technology. Users, designers, and programmers must respond to this imperative if they wish to participate in the implemetation aspects of computer systems applications—as opposed to being restricted to being an end user of a system. That is all the more reason for us to understand these *universal* processor-system principles, to enhance our abilities to either program or design these devices.

BIBLIOGRAPHY

Bell, C.G., and Newell, A. *Computer Structures, Readings and Examples.* New York: McGraw-Hill, 1971.

Burks, A.W., Goldstine, H.H., and von Neumann, J. "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument." 28 June 1946. Reprinted in *Datamation,* September and October 1962.

Goldstine, H.H. *The Computer.* Princeton, New Jersey: Princeton University Press, 1972.

Marshack, A. "Exploring the Mind of Ice Age Man." *National Geographic,* pp. 65–89, January 1975.

Mead, C., and Conway, L. *An Introduction to VLSI Systems.* Reading, Massachusetts: Addison-Wesley, 1980.

Shannon, C.E. "A Symbolic Analysis of Relay and Switching Circuits." *AIEE* 57, 713–23 (1928).

Spencer, D.D., *An Introduction to Computers.* Westerville, Ohio: Charles E. Merrill, 1983.

Wilkes, M.V. "The Best Way to Design an Automatic Calculating Machine." Paper read at Manchester University Computer Inaugural Conference, July 1951.

Wilkes, M.V., and Stringer, J.B. "Microprogramming and the Design of the Control Circuits in an Electronic Digital Computer." *Proc. Cambridge Phil. Soc.,* Pt. 2, Vol. 49, pp. 230–38, April 1953.

PROBLEMS

1. Count the number of microprocessor controlled products in your home or in your car.
2. What are the essential differences between the Harvard and von Neumann architectures, from the standpoint of a computer's organization?

3. Draw a simple timing diagram that illustrates and explains why the Harvard architecture has an inherent speed advantage over the von Neumann approach.
4. Define the meaning and original intent behind the term *microprogramming.*
5. What form of tabular representation is microprogrammed architecture organized around?
6. Define the terms *micro* and *macro,* with respect to microprogramming.
7. For a microprogrammed computer system, what is the relationship between a user's instruction in the instruction set and a macro in the system's control store?
8. Describe the cycle of computation and the principal events that occur in each of its phases.
9. What is the role of the Program Counter in the von Neumann architecture?
10. How does the von Neumann architecture affect the manner of space allocation in the primary memory of a computer system? Draw a memory map that illustrates your comments.
11. How many programmable processors are there in the following popular personal computer systems? Investigate at least one, and list your findings. Do you understand the organizational basis for these processors?

    a. IBM PC or XT
    b. Apple IIe
    c. Apple Macintosh
    d. Hewlett-Packard 150
    e. DEC Rainbow

12. Describe the channelized computer system pattern of organization and its relationships to programmable processor IC's.
13. What are the typical fields of a complete instruction word, and what are the functions that each serves?
14. What is the difference between a complete instruction word and addressable units of memory?
15. If you plan to participate in systems programming and development of computers and processors, check off all the following areas for which you feel adequately prepared, at the entry level:

    a. The logical nature (not semiconductor physics) of semiconductors and digital IC's
    b. Fundamental bus organizational and operational concepts
    c. The logical nature and elementary physical principles behind memory devices
    d. Fundamental clock and timing relationships of a synchronous digital system
    e. The actual functioning of an ALU and how data is altered in a CPU

f. How the ALU influences conditional instructions and the flow of software

g. Basic sequential machine organization, implementation, and operating principles

h. Data flow and transformation within the context of a bus-oriented system, including the block diagram of a computer's architecture

i. How logic IC's–and systems that incorporate their features–actually function

j. Freedom from "hardware fright"

k. Participation in microprogramming activities

l. How an instruction in the instruction set of a processor is actually executed

m. Implementing a digital processor system

n. How the various addressing modes of a computer function and are executed

o. The fundamental organization behind the typical programmable peripheral processor, such as CPU's, disk controllers, graphics and CRT controllers, communications IC's, numeric data processors, etc. An elementary background for reading their data manuals and for using and programming these devices

If you checked none of the items above, there are two choices: either change your major, or carry on. Remember: This is only the beginning.

CHAPTER 2
FEATURES AND ARCHITECTURE OF PROCESSORS

Large-scale computers, minicomputers, microprocessors, and dedicated programmable processors are all derived from the same common heritage of basic features. The differences are more likely to be in the scale on which each machine is constructed or in how the same features are used, rather than in the varieties of features employed in their architectures. For this reason, we shall refer to all of the foregoing as *processors* in the future, to indicate that certain unified concepts regarding structural features apply to all.

There are many levels at which one can study the nature of processors. These range from the component level (resistors, transistors, etc.) at one extreme to the use of abstract computer-description languages that detail the register and state transactions of a machine. At the entry level, however, it is beneficial to obtain some insight into the major basic working features of processors–and to understand their mechanisms. As we review the basic logical devices employed in a processor's design and examine the fundamental concepts associated with their use, observe that a limited number of devices and concepts are used repeatedly. These form the "blocks" of logic employed in describing and constructing an architecture; these are herein referred to as "features" of an architecture. As we proceed, the fundamental features that are now somewhat briefly introduced, in an overview of their properties, will eventually be interconnected to form the meaningful groups collectively referred to as a computer's architecture.

There is a subtle relationship between architecture and software. The instruction-set capability is embedded in the architectural features–a fact that places a limit on software performance. This is important nowadays, when software and application objectives are considered *before* prescribing the requisite architectural arrangement of the features. The hardware design of the VAX family of computers, for example, was designed with certain software-performance objectives in mind. Since we have to start somewhere, we shall introduce the applicable basic terminology, devices, and features to arrive at the architectural structures common to computers, microprocessors, and dedicated processors and controllers. From this base, one should gain some understanding of the analysis and implementation of a given processor's architecture discussed in the subsequent chapters.

It should be noted that the greatest cost associated with computing is that of software development and maintenance–the pain never goes away. Thus, a "good" architecture supplies the vehicle for the development of useful and enduring software. Plentiful software is the reason some older systems (poor ones included) refuse to fade away. Software and operating-systems development is expensive. Microprocessors have affected this situation in a strange way. The manufacturer frequently markets devices, not software systems. The end user is tempted to use the microprocessor chip for which the most software is available–whatever the source. Thus, we see that, in gaining an understanding of the architectural features of processors, we become better prepared to cope with the interrelated hardware and software problems. This chapter, Chapter 3, and Chapter 4 are primarily for the benefit of the more software-oriented student. The final goal is to dispel the mystique that unfortunately still surrounds the internal structure and functioning of processors. By this means, we can bring both hardware and software interests closer.

Finally, some comments on our approach in this chapter are in order. While some exposure to and knowledge of logic devices is presumed, it is not always assumed in this text that one actually possesses hardware experience. This is not intended as a first course in logic design. The basic nature of fundamental logic elements is described here in a manner intended to help one *visualize* their function and, in so doing, to become comfortable with them. The reader should possess a TTL 7400 series logic family data catalog from the very outset. *This is essential.* As noted, the basic devices and building blocks described in the TTL data catalogs will be combined to produce the more sophisticated logic structures of which an architecture consists. The emphasis is on the *logical* behavior of these computing structures, as opposed to the electronics-design details of a particular device or system. It is necessary, though, to explain logical computing structures in terms of the familiar and commercially available small- and medium-scale TTL and MOS logic devices available to you at any electronics parts store.

One soon learns that the features associated with the basic commercial logic devices described in the manufacturer's data catalogs, are—logically speaking—the same features utilized in a microprocessor's design. It can be demonstrated that the microprocessor evolved out of the ability to incorporate more and more of these basic features into a monolithic circuit. Thus, from a study of these readily available devices, one can gain an intuitive understanding of how Very Large Scale Integrated (VLSI) processor systems are organized and how they function. This is also good basic preparation for gaining an understanding of the data sheets on microprocessors and programmable support logic, such as communications, disk, printer, and input/output controllers. They too are processors. Thus there is a big picture out there, consisting of an understanding of the shared features of programmable processors and the reader's willingness to consult their data sheets.

An important point is that the task of gaining an understanding of hardware is not hard. However, you must be willing to study the information in data catalogs on your own. From a study of these basic logic devices, you gain the ability to understand both function and performance in processor systems, such as microprocessors, communications devices, etc. By taking the time to really understand the functioning of about *ten readily available IC's* and the way they are used in a systems context at the end of this text, you can gain a strong hardware understanding of what computer architecture is about. The 4-bit microprocessor we shall build and work with is capable of demonstrating the operation of most instruction types, as well as several addressing modes, if you are willing to microprogram it. First, we must learn to understand it.

BUSES, DATA PATHS, GATES, AND BUFFERS

We shall start by forming a visual image of a total busing structure, followed by a overview of the operating principles of typical devices

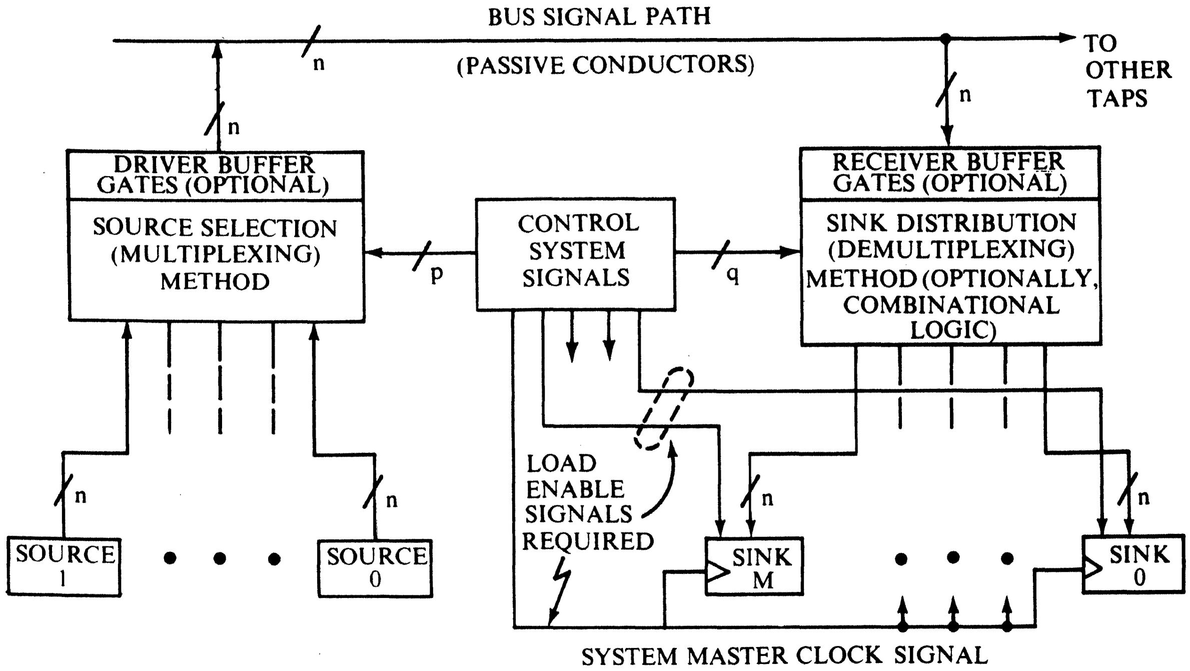

BUS SIGNAL PATH
(PASSIVE CONDUCTORS)
TO OTHER TAPS
n
n
n
DRIVER BUFFER GATES (OPTIONAL)
SOURCE SELECTION (MULTIPLEXING) METHOD
p
CONTROL SYSTEM SIGNALS
q
RECEIVER BUFFER GATES (OPTIONAL)
SINK DISTRIBUTION (DEMULTIPLEXING) METHOD (OPTIONALLY, COMBINATIONAL LOGIC)
n
n
SOURCE 1
SOURCE 0
LOAD ENABLE SIGNALS REQUIRED
n
n
SINK M
SINK 0
SYSTEM MASTER CLOCK SIGNAL

Figure 2-1
Bus System Block Diagram

used at bus interfaces. A *bus* may be defined as a data path containing one or more transmission lines over which information is transmitted in a coordinated manner. Usually, a bus is *time-shared* by several *sources* of information and will have one or more recipients, or *sinks,* of the information it handles. A general block diagram of a bus's structure is presented in Figure 2-1. More than one data-source selection block, like the one shown in the figure, may interface with the bus at its taps. This characterizes only the path from any one source to any of the chosen sinks. Where a single tap contains both sources and sinks, we say that it has *bidirectional* properties.

Bus Control

Associated with the bus are control signals emanating from the processor's control system. These control signals coordinate the activity on the bus. We are discussing clock-driven (synchronous) systems. The control-system signals decide which single potential source to the bus becomes the *master* of the bus during the current clock period. When the selected source or its data selector (also called a *multiplexer*) does not have the necessary power to drive the bus lines, a power-amplifying driver called a *buffer* may be used to speed up signal-level transitions. In some systems, receiver-buffer gates are optionally used to detect the driven signal. The use of combinational-logic data distributers (called demultiplexers) to sort out which sink will receive the bus information is seldom employed in large-scale current practice. What is commonly used are a set of signals, as noted in the figure. One of these is the control system master *clock* signal, which is distributed over a common line to the clock input of each sink register. The other elements of this set of signals are the separate control system load-enable signals, one for each sink. The control system may enable none or several sinks to receive information in any one time frame. This separate enable and common clocking is typical practice. If not enabled, a particular sink ignores the clock. If enabled, it stores the current period's bus information.

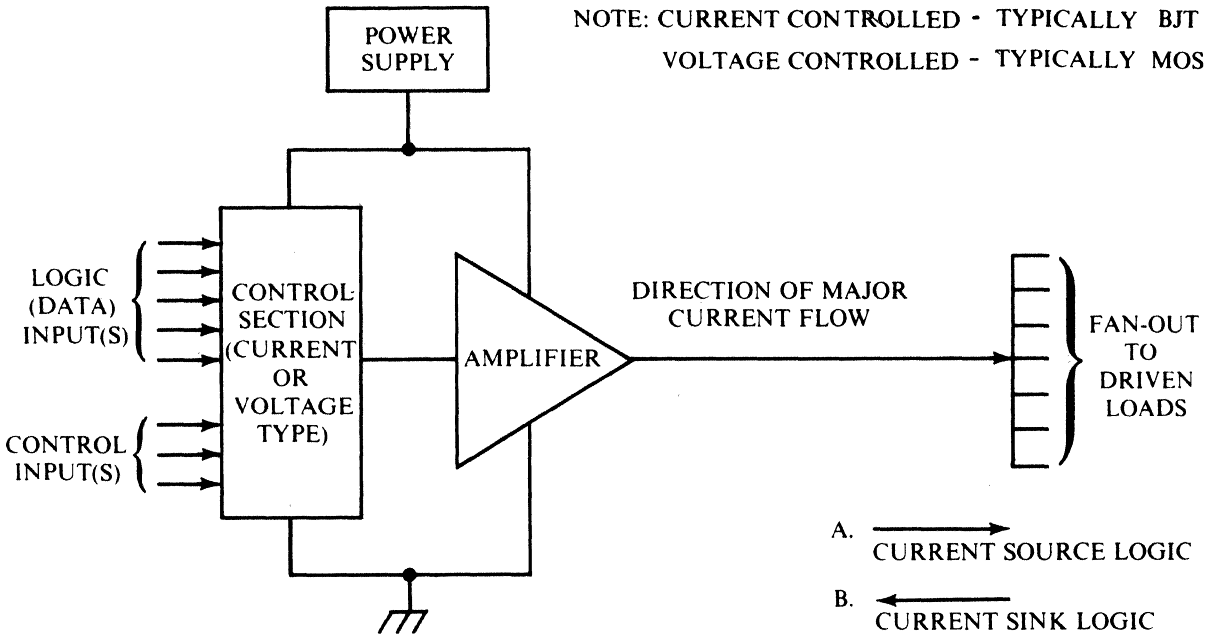
A *bus system,* then, is simply an associated passive bundle of wires, traces on a printed circuit board, or deposited sets of conductors within an integrated circuit. Associated with it are control-system-driven source- and sink-selection signals. When driven by a selected source, each line of the bus presents its information to all the sinks simultaneously. Note that there can be only one source acting as master of the bus at a particular moment; in principle, there are no limitations to the number of sinks that may receive information from the bus. The load-enable signals from the control system dictate which sink(s) shall respond to the clock in the current period and thus store the information at their inputs. For clarity, the slash (/) and an associated number are used in the figures when the bus contains more than a single line. Thus $-\!/\!-_8$ indicates an eight-line bus.

Above we have characterized a bus system. This is a simple but effective picture of what a bus is. Henceforth, when we speak of a bus, this image should be recalled. Two basic but important tools (*characterization* and *visualization*) for understanding digital systems architecture have thus been introduced in this bus-organization example. By *characterization,* we mean the understanding of the basic functional and behavioral nature of the things we are talking about, perhaps in an analogy. It is not at all necessary to understand semiconductor-design details completely in order to understand computers. However, we *must* be able to adequately characterize and visualize the system's components in our own minds. Putting these characterized components together visually as a meaningful system is the essence of computer-systems comprehension—and of what follows here. Some students may be new to this way of thinking, so remember that characterization and visualization are two major keys to understanding processor systems behavior. Computers should always be approached from a systems point of view.

A brief aside: Microprogramming was originally defined by M.V. Wilkes as "a rational approach to the design of a computer's control system." We shall discuss microprogrammed system-design principles later in detail. It enters the scene here, since the bus-control signals mentioned above often come from a microprogrammed controller. Sometimes the microprogrammer is responsible for enforcing the single-source rule. The microprogrammer is always responsible for the selection of the sinks that receive information at the end of the current period. Bus control is an important aspect of microprogramming, and the microprogrammer must always have an image of the system in mind. The preceding presented one way to visualize a bus. Let us now look at the principles, characterizations, and terminology behind the design details of the physical devices within a busing structure.

LOGIC GATE CHARACTERIZATIONS

A fundamental appreciation of the nature of these devices substantially reduces hardware fright. The logic gate is the basic building block of interest. The major structural features of a logic gate are shown in Figure 2-2. These consist of a control section (receiving both external data and—in some cases—control inputs), followed by an



POWER SUPPLY
NOTE: CURRENT CONTROLLED - TYPICALLY BJT
VOLTAGE CONTROLLED - TYPICALLY MOS
LOGIC (DATA) INPUT(S)
CONTROL INPUT(S)
CONTROL SECTION (CURRENT OR VOLTAGE TYPE)
AMPLIFIER
DIRECTION OF MAJOR CURRENT FLOW
FAN-OUT TO DRIVEN LOADS
A. CURRENT SOURCE LOGIC
B. CURRENT SINK LOGIC

Figure 2-2
Logic Gate–Typical Organization

amplifier. This amplifier may be either an inverting or noninverting one–the choice is design-dependent. The control section is either current- or voltage-controlled. Current-controlling inputs are typical of the Bipolar Junction Transistor (BJT) logic families, such as Transistor-Transistor Logic (TTL) and Emitter Coupled Logic (ECL). Voltage-controlling inputs are typical of the Metal Oxide Semiconductor (MOS) transistor families, such as PMOS, NMOS, and CMOS. Whatever the type of input control, the output must have the necessary power to source or sink current on the driven line, to obtain the desired speed of response at the output. All logic modules can both *source* (feed onto a line) and *sink* (drain from a line) current. As noted, though, current-source logic *primarily* drives current down the line; current-sink logic primarily drains it from the line.

Briefly, look ahead at the components in the basic 7400 TTL NAND gate, Figure 2-4. It is composed of resistors, transistors, and diodes. The diodes are disguised as an intrinsic part of the multiple-emitter input transistor. This structure simply means that the input transistor is being used as a diode-logic gate. We need to *characterize* the nature of these components to understand their computer system's behaviorial characteristics. We start with the resistor, symbolically shown in Figure 2-3a. By analogy, the resistor restricts the flow of electricity–as a restriction in a pipe limits the flow of water. The resistor is often used to limit the flow of current to avoid damage from improper short circuits. The resistor is thus characterized, for our purposes, as a restrictor or current limiter. It has other uses, but these need not concern us now.

How does one characterize a diode? Its electrical and analog symbols are shown in Figure 2-3b. In semiconductor terms, it is a junction of *p* and *n*-type silicon materials. More important to our understanding of its nature, the diode's behavior is analogous to that of a check or one-way valve in a pipe. Water (electricity) flows in one direction if the pressure (electrical potential or voltage) is high enough to open the valve. High pressure on the opposite side only causes the

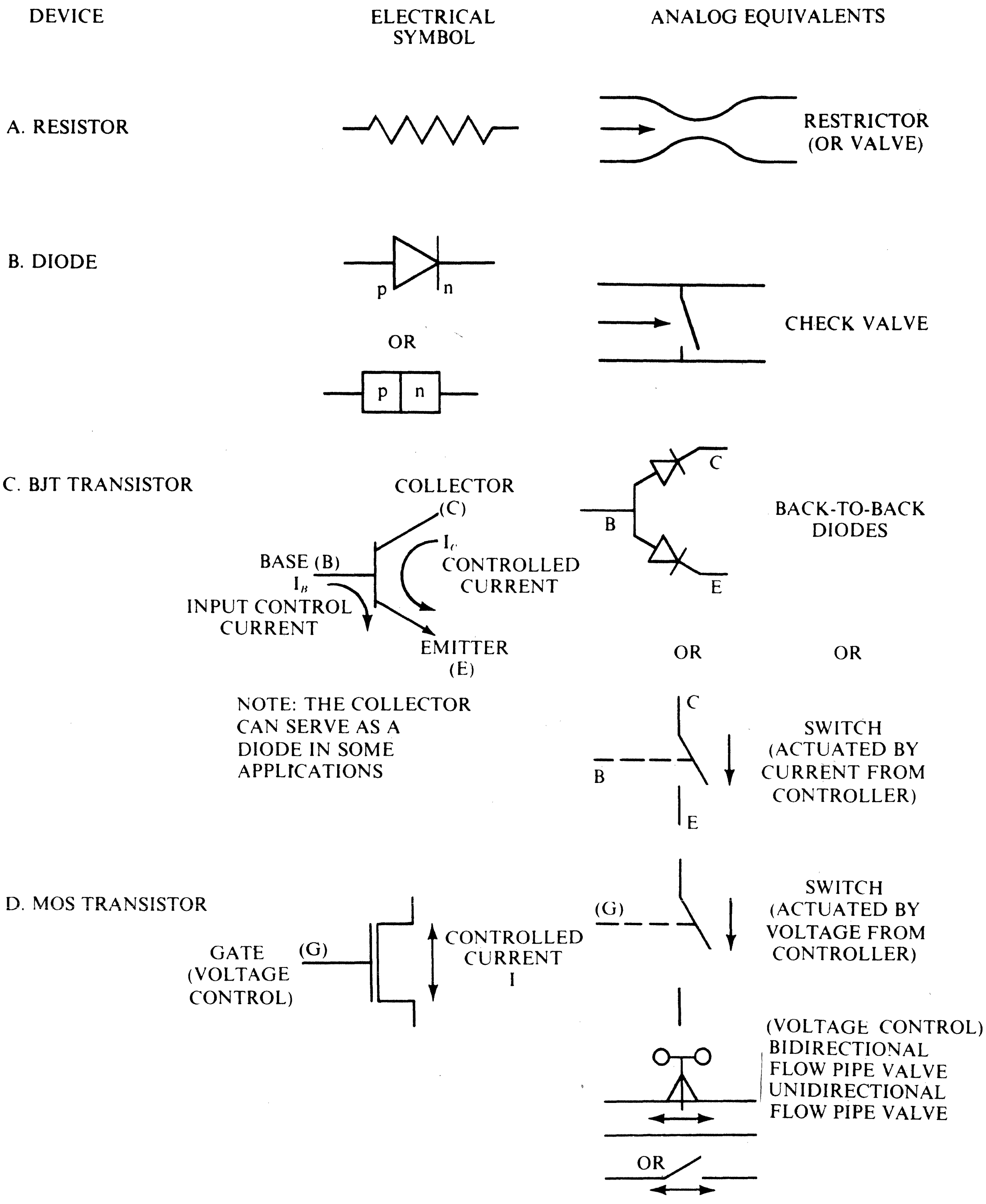


Figure 2-3
Characterizations of Basic Electrical Devices

valve to seat, thus sealing off all fluid flow. The diode, for our purposes, is characterized as an electrical one-way check valve. The flow of conventional current takes place in the direction of the arrow in the symbol when the voltage at the base of the arrow is great enough to "open the valve." (This is usually about 0.7 volts for silicon diodes.) When this is done, we say that the diode is *forward-biased.* A high voltage difference of the opposite polarity does not cause current to flow—

within reason: let's not destroy the device! Under these conditions, we refer to the diode as being *back-biased.*

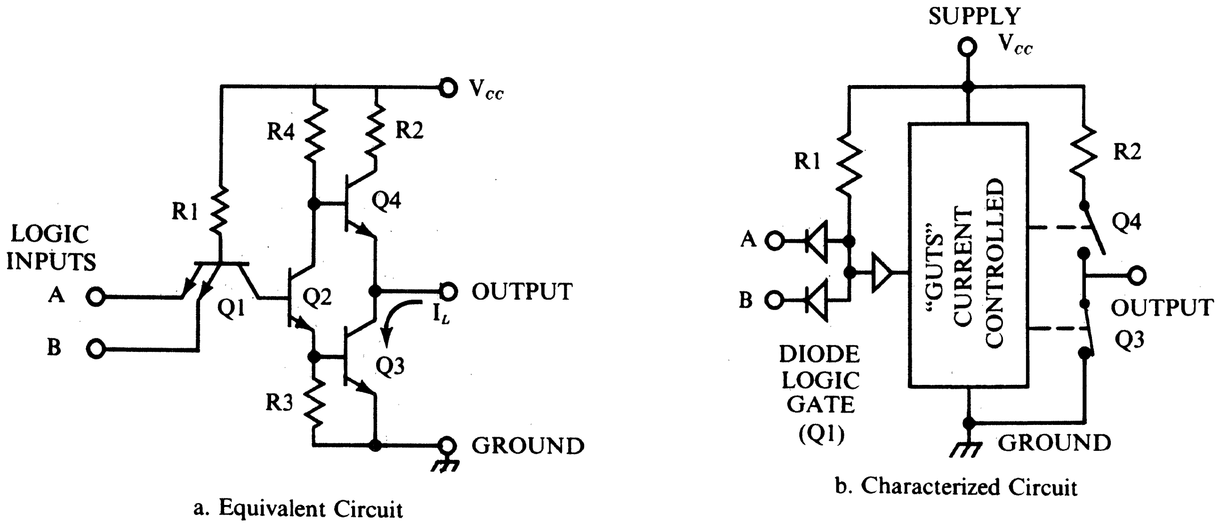
The BJT is often characterized as two diodes, back to back. As shown in Figure 2-3c, the usual symbol for this device uses the arrow to indicate the diode we refer to as the *emitter.* In fact, the *collector* is simply another diode that shares a common junction with the emitter—which is the so-called *base* of the transistor. If the base is at a relatively high voltage, in the type illustrated, current flows through whichever diode has the lower potential—be it the emitter or collector. This is not what is referred to as transistor action, but it does illustrate the fact that transistors are often used as diodes in an IC's design. *As such, they often serve as diode-based logic gates at a device's inputs.*

When transistor action prevails in the BJT, the structure behaves as a current-controlled current amplifier. The amount of input (base) current determines which of the three operating regions the transistor is in. In the active region, the transistor acts as a linear amplifier, which is not relevant to a description of the output's interface behavior in a logic gate. In digital circuits, the output transistors of the amplifier are driven either into the full "on" (saturated) region or into the full "off" (cutoff) region. This brings us to yet another way in which we may characterize the transistor. That is as a current-controlled *switch* in the BJT family of transistors and as a voltage-controlled *switch* in the MOS family. *This means that the physics and electronics of transistors are often reducible to the analog of the current- or voltage-controlled switch when we are examining the resultant behavior of the output circuits of digital systems.*

The Metal Oxide Semiconductor (MOS) transistor is characterized in Figure 2-3d. (It will soon be discussed in detail.) For now, let us state that we can visualize it as a voltage-controlled switch, for the unidirectional control of current flow, as the BJT is. For our switching purposes, the valve is either fully open or fully closed. Thus its action is the same as that of a switch that supports bidirectional current flow. A second use of MOS transistors is as voltage-controlled bidirectional gate valves that manage current flow. In this case, the direction of flow is determined solely by which side of the valve is at the higher "pressure" (voltage), as illustrated.

To illustrate the preceding, let us first look at the equivalent circuit of a basic TTL NAND gate, Figure 2-4. Transistor Q1 of Figure 2-4a is used not as a transistor but as multiple-emitter diodes placed back to back with a single collector diode. This structure forms a diode AND gate (if the high voltage level is interpreted as the "true" condition). Only if all the logic inputs are at the relatively high voltage level, H, does current flow through the collector diode of Q1 to drive the base of Q2, switching it on. The voltage at the top of resistor R3 is driven up by the flow of current, causing some current to be fed to the base of Q3, switching it on, in turn. Note that, in addition, Q2 is also diverting current away from the base of Q4. Thus, Q3 is now in saturation while Q4 is cut off. Q3 is *sinking* current from the gate's output to ground and may be thought of as a low impedance path to ground. This path handles the current supplied to the output by the equivalent of resistor R1 contained in *other* devices driven by this one, and any charge stored in the parasitic capacitances of the driven line.

Conversely, if any *one* of the input diodes is held at the relatively low voltage, L, current is diverted away from the base of Q2. The result is that Q3 is no:v cut off, and Q4 is driven on. The analysis of this situ-



a. Equivalent Circuit

b. Characterized Circuit

Figure 2-4
Basic TTL Logic Gate: NAND

ation requires circuit expertise beyond our current scope. Resistor R2 is chosen as a compromise between the conflicting desires to protect against short circuits and yet provide as low an impedance path as possible between the supply voltage, Vcc, and the driven line. The current sourced to the line from a single input is much lower than the total sink-current capacity of Q3, due to the high impedance resistor, R1, in the driven multiple-emitter input circuit. Nothing, however, protects Q3 of the output except the common sense of the user in limiting the total sink current arriving from all external sources. In summary, we note that this NAND gate really consists of a diode AND gate followed by an inverting amplifier. Thus, the NAND gate is treated as an AND-NOT gate, but only when the high voltage level is designated as the "true" level.

Observe that the output consists of one transistor atop another, both acting as switches. This important structure, referred to as a *totem pole,* is found very frequently in digital-logic devices. When serving as a logic gate, the two transistors of the totem pole are always *mutually exclusive.* That is, if one is on, then the other is off, and vice versa. The characterized model of Figure 2-4b summarizes all this. In characterizing these computer components, we are particularly concerned with the nature of the input and output interfaces. Everything that lies between these often can be referred to simply as the *"guts"* of the device in question. That is, we must understand a device's overall function; but what electronics lies between the input and output requires only an intuitive understanding for systems-application purposes. It is important to visualize the output as a tap that sits between the two switches of a totem pole, which also contains a current-limiting resistor at the top.

ACTIVITY LEVELS VERSUS ELECTRICAL LEVELS

Either of the two voltage levels, H or L, can be designated as the *true* logic level. This choice is made by the user. The important connotations of this fact are often confusing to beginners. Let us try to clear this up by adopting the following conventions: In this text, the *true* level is *always* referred to as a logic 1, and it may be either the high or the low voltage. Once the true level is selected, the other voltage

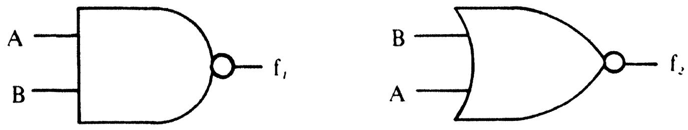
DEVICE'S INVARIANT
ELECTRICAL CHARACTERISTICS

INPUTS		OUTPUT
A	B	f
L	L	H
L	H	H
H	L	H
H	H	L

a. ELECTRICAL LEVEL
CHARACTERISTICS TABLE,
TTL NAND GATE
(SEE FIG. 2-4)

H = HIGH LOGIC VOLTAGE LEVEL
L = LOW LOGIC VOLTAGE LEVEL

RESULTING LOGICAL BEHAVIOR
VS.
ADOPTED LOGICAL ACTIVITY LEVEL



b. GATE SYMBOL FOR
POSITIVE LOGIC
APPLICATION

(ACTIVE HIGH)

c. GATE SYMBOL FOR
NEGATIVE LOGIC
APPLICATION

(ACTIVE LOW)

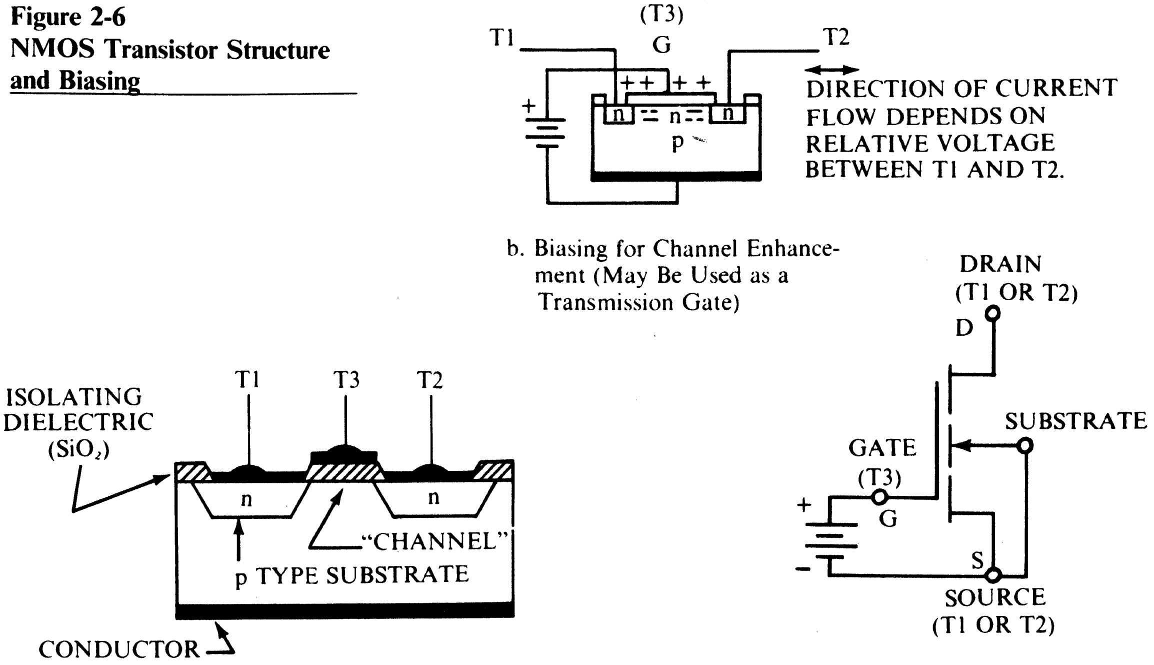
Figure 2-5
Dual Logical Behavior of TTL NAND Gate

level becomes the false, or logic-0 level. Much confusion can be overcome by relating 1 and 0 *only* to true and false, respectively. When the H level is selected as 1, we have what is termed a *positive,* or *active-high,* logic system. If the L level is designated as 1, we have a *negative,* or *active-low,* logic system. There are far more negative logic signals at a processor's interface than the beginner suspects. We shall work with active-low logic signals later. There is no avoiding them in practice.

The consequences of the choice of the activity level to be applied in a logic device can be substantial. It may come as a surprise to the reader to learn that there is, in reality, no such a thing as a NAND gate. There is only a *NAND function,* performed by a physical device for one chosen activity level. If we think about the device of Figure 2-4 again, we see that it can be truly and consistently characterized by its electrical behavior only as a device whose output must be high when any of its inputs are low. Figure 2-5 presents a characteristics table for this device's *electrical* levels of L and H. In the context of a positive-logic system, this produces the NAND function. In a negative-logic system, it yields the NOR function. This electrical-level form of table is often used in data manuals, as it is the only consistent way to describe a device's physical behavior. Logical behavior depends on whether the active-high or the active-low logic convention has been chosen.

Conversion of the H's and L's to the 1s and 0s of a selected activity-level convention of this and other logic gates is an important exercise in the problem set, to let the reader discover what functions the 1s and 0s produce for each chosen logical activity level. First, though, one must determine the electrical-level characteristics of a device in terms of H's and L's—since that is its *real* physical characteristic. This is done in the laboratory by applying all possible input combinations to the device, while recording the associated output response in terms of H's and L's. The logic function performed is a consequence of the logical activity level we later choose when we employ the device.

Figure 2-6
NMOS Transistor Structure and Biasing



b. Biasing for Channel Enhancement (May Be Used as a Transmission Gate)

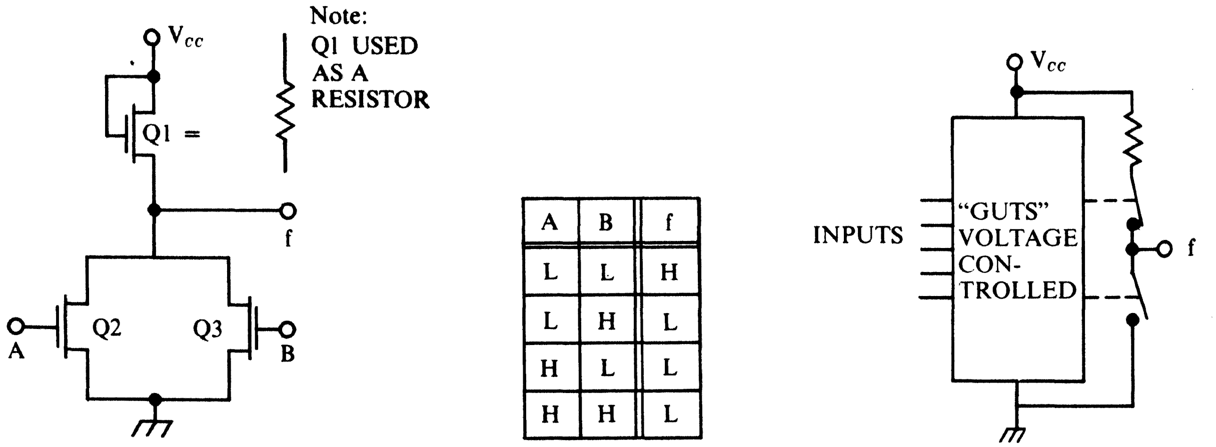
a. Simplified Device Structure Cross-Section

c. NMOS Transistor Gate Symbol with Enhancement Mode Biasing

Further, by consistently treating logic 1 and 0 as true and false, respectively, we can correctly perform arithmetic computations in either positive or negative logic systems. In the arithmetic problems in this text, the binary digits 0 and 1 are *always* associated with logic-0 and logic-1 truth levels, respectively. In summary, the H and L tables consistently characterize device *physical* behavior, regardless of the selected logical-activity level used. On the other hand, 1s and 0s characterize *only* truth-value behavior for a selected and known logical-activity level. We consistently maintain these distinctions in this text.

MOS LOGIC CHARACTERIZATIONS

The MOS family of transistors generally consists of low-power devices, often requiring auxiliary buffers. Most processor-type IC's employ MOS technology, including MOS bus interfaces. An understanding of their fundamental means of operation is useful. They may be implemented in a variety of ways. One basic manner widely used in logic devices is illustrated in Figure 2-6. The voltage-controlled device in this figure is an *n-channel enhancement-mode Metal Oxide Semiconductor Field Effect Transistor (MOSFET).* It was created by starting with a piece of p-type silicon, into which two n-type silicon pockets were diffused. The *n* and the *p* refer to the type of majority charge carriers present in this region, negative electrons or positive "holes," respectively. An insulating metal-oxide layer is then deposited over the top surface, except for the conducting pads T1 and T2. A third conducting pad, T3, is applied over the insulating dielectric to form the gate of the device. The *gate* and the *substrate,* separated by the insulator, form a "capacitor."



A	B	f
L	L	H
L	H	L
H	L	L
H	H	L

a. Simple MOS Logic Gate

b. Electrical Level Characteristics Table

c. Characterized MOS Logic Gate

Figure 2-7
MOS Logic Gate: Simplified Enhancement Mode Operation

Whatever polarity of voltage is applied between T1 and T2 (assuming T3 is open), no current flows because of the back-to-back diode behavior of the n-p-n path between T1 and T2. One of these diodes is always back-biased. The gate is capable of affecting the channel region between the two n-type regions, making the device conductive. If a positive voltage is applied to the gate, electrons (n-type minority charge carriers always exist in a p-type region) are attracted to the channel area, which is the other plate of the capacitor. This is called the *enhancement,* or formation, of the channel. In enhancing the channel, we have converted the former n-p-n structure to an n-n-n structure. Thus, the diode behavior no longer applies, and there is a current-conducting path between T1 and T2, as shown in Figure 2-6b.

An important result is that the direction of current flow depends only on the polarity of the voltage between T1 and T2. We have thus created the *transmission* gate, which supports bidirectional current flow. When this same device is refined in manufacture to handle primarily one direction of current flow, one end (T1 or T2) is optimized to become the source of charge carriers; this end is referred to as the *source.* The carrier-receiving end is called the *drain.* This modified form of construction is common when the device is to be used as an MOS transistor. Let us now connect both the substrate terminal (Figure 2-6c) and the source to the negative pole of a battery. The battery's positive pole drives the gate. Since the gate dielectric has extremely high resistance–on the order of 10^{18} ohms–no appreciable current flows between the gate and the channel. Still, the channel is enhanced and therefore is capable of conduction. Thus, an input gate voltage controls the flow of channel current. This basic structure serves in two ways: first, as a bilateral-current-flow *transmission gate* and, second, as a unilateral-current-flow *MOS transistor.* The difference lies solely in the manner of use. For digital circuits, both may be characterized as behaving like a *switch.* Where current flow is regulated in both directions, as in analog gates, the structure can be compared to a voltage-controlled *valve.*

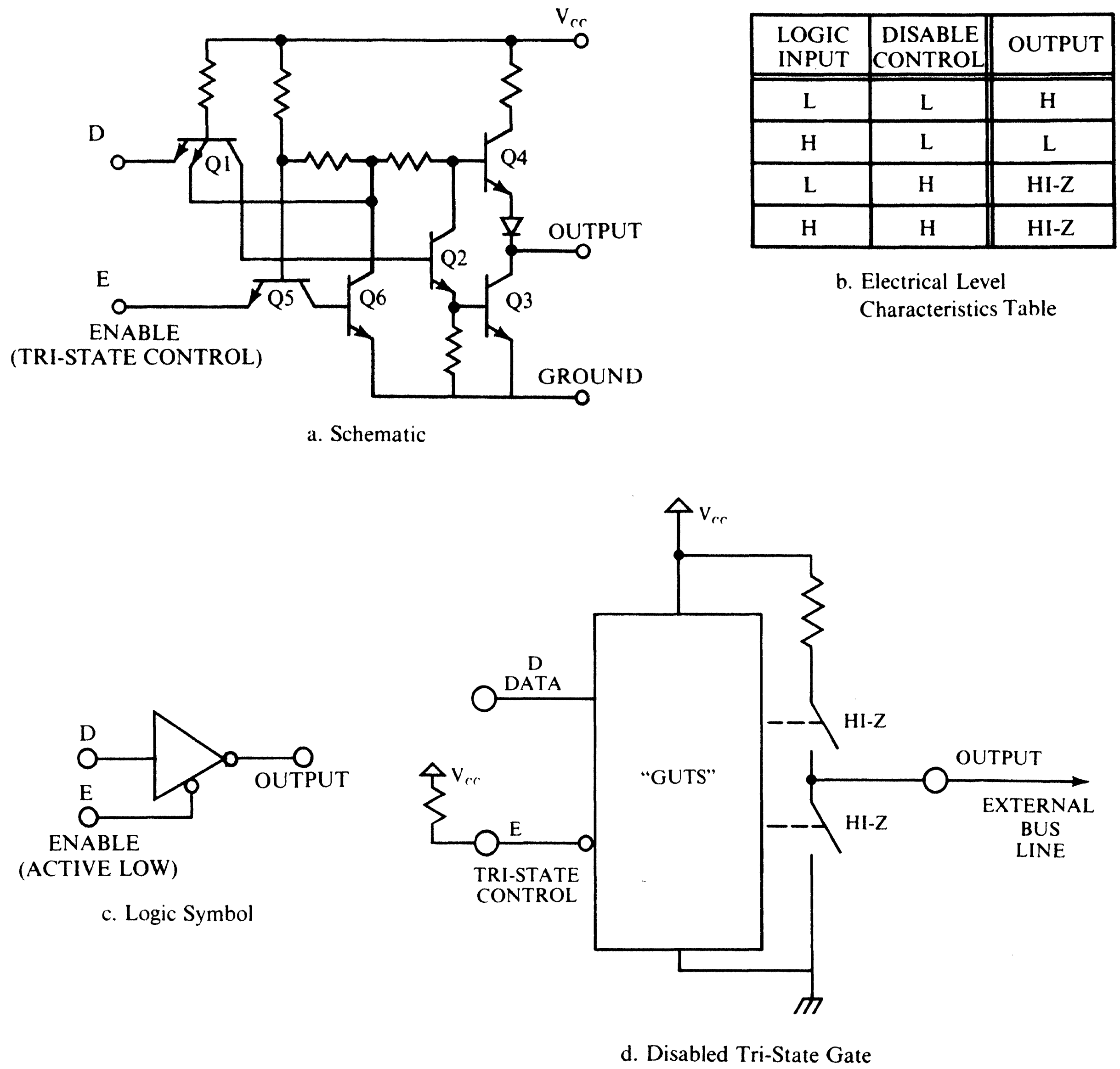
Several of these devices can be used to form a logic gate, Figure 2-7a. Simplified MOS-transistor symbols are used in the illustration. Transistor Q1 is the MOS-technology equivalent of a current-limiting resistor. Since the gate is always connected to the source and the power supply, the channel of this transistor is always enhanced. The current flow is limited by the intrinsic resistance of the channel, which is designed to limit current flow sufficiently to avoid harm. Current flow at the output of this gate is always a possibility. If either input A or B is pulled high, the current of Q1 is diverted from the output, f, and shunted to ground. The result is a low output voltage, used to control other gates or devices. When this gate is used within a positive-logic system, it produces the NOR function. The electrical-level-characteristics table of Figure 2-7b characterizes the device's physical behavior in actual practice. The logic function actually performed depends on the *logical-activity-level convention* adopted. Again, we see that the characterization of logic devices starts with their *observed* electrical behavior, expressed in terms of *electrical levels.* Boolean logic functions, where relevant, are *derived* from this.

This has been a functional explanation of just one of the many FET technology families, which are similar in nature: all share the advantages of simplicity, low cost, high packaging density, low power consumption, and ease of fabrication. Coupled with reasonable speed of response, these factors have contributed to the wide use of MOSFET's in the manufacture of LSI and VLSI processors and memory systems. We have seen the MOS transistor used as a transmission gate, as a logic gate, and as a resistor. All these forms are frequently used. In looking at the structure of MOS interfaces to the outside world, we shall also find the ubiquitous totem pole formed with MOS devices. It is apparent from Figure 2-7c that the patterns we are learning to visualize contain a good deal of repetition. Here, a MOS logic gate (shown with totem-pole outputs, as most commercial IC gates are) has almost the same characterization as that of the BJT logic gate. The "guts" now manage voltage-controlled switches, and the current-limiting resistor of the output is, in reality, a disguised MOS transistor whose source and gate are connected.

As the use of bus structures evolved, it became necessary to develop new techniques of interfacing logic gates to them. One important technique was the development of methods for time-sharing a bus among several sources. Let us look at the modifications to the inverting gate that make it useful as a time-sharing structure. It is referred to as the tri-state (or 3-state) gate. The third state is not a logical level but, as we shall see, a state of electrical isolation.

TRI-STATE CHARACTERIZATIONS

If the inputs of the previously discussed NAND gate are tied together, the result is the simple inverter. Looking at the output structure of the gate, we notice that it consists of the totem-pole structure. Let us focus on the BJT NAND gate, to demonstrate its transformation into the tri-state interface. As presented so far, transistors Q3 and Q4 of Figure 2-4 occupy mutually exclusive states. That is, if one is on, the other is off. If *both* could be turned *off* simultaneously, then the gate's output pin would be *electrically isolated* from both power and ground by the very high impedance of both *off* transistors. As far as the driven line is concerned, it is as if the gate is nonexistent, because of the high impedance of its output. Figure 2-8 illustrates the conversion of a TTL inverting gate into a tri-state buffer. An extra level of inversion would produce a noninverting buffer.



a. Schematic

LOGIC INPUT	DISABLE CONTROL	OUTPUT
L	L	H
H	L	L
L	H	HI-Z
H	H	HI-Z

b. Electrical Level Characteristics Table

c. Logic Symbol

d. Disabled Tri-State Gate

Figure 2-8
Elementary Tri-State Logic Gate (TTL)

This gate was created by adding Q5 and Q6 to the circuitry of Figure 2-4 and feeding one of the former inputs back to the collector of Q6. What we have now gained is a buffer gate with one data input and one tri-state control input. This last input is often referred to as an *enable* input. This term relates to the enabling (or disabling) of the gate's capacity to pass data through to the output. When the enable line of Figure 2-8a is low, Q6 is deprived of its base current and in consequence is cut off. In this state, one emitter of Q1 is internally held high, while the other is driven by input data. As the truth table shows, the output is the complement of the input logic state for this inverter. Making this line high saturates Q6, thereby pulling both the internally connected emitter and the base of Q4 low at the same time. This action deprives both Q3 and Q4 of their base-current drives. The net result is that both of these transistors are *cut off,* and the output is *electrically isolated* from *both* power and ground. The data-input line can no longer influence the output. The gate is now in the *Hi-Z,* or high impedance state. This is the state of electrical isolation, as shown in the table of Figure 2-8b. The logic symbol used to represent the tri-state buffer gate is presented in Figure 2-8c.

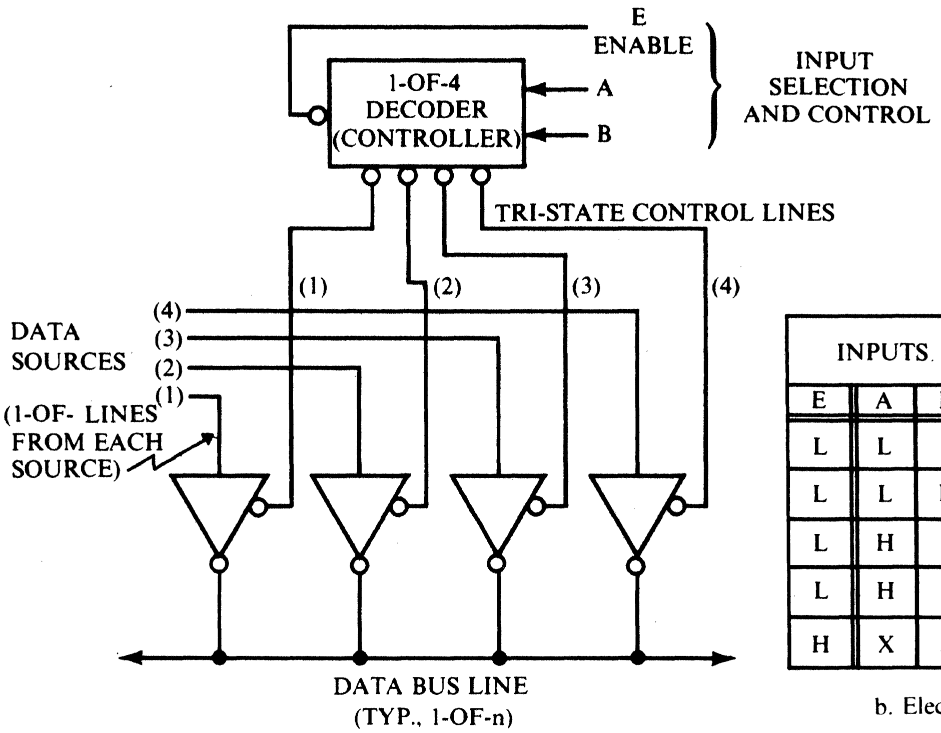
Figure 2-8d shows our simple switch-analog picture for the Hi-Z state of the totem pole. Note that *both* switches are open, thus electrically isolating the gate from its driven line. This is an important visual image, since bus interfacing and multiplexing often employs this construct. Time-sharing of a bus line is accomplished by enabling only one tri-state gate per line at any one time. The same tri-stating principles are applied to all the many semiconductor fabrication technologies that support controlled-switch behavior.

A comment on drawing notation may help prevent confusion. The small circle at the control input, E, tells us that it is an active-low input. Its absence would be indicative of an active-high input. The real purpose of the small circles on logic diagrams is to indicate active-low inputs and outputs. Previously, we referred to the NAND gate as an AND-NOT gate. The small circle at the output of the AND symbol merely indicates that the preceding function (the AND) was actuated whenever there is a low output on that line. This has the same effect as inversion. The small circle means *only* that the logic function associated with it is actuated when the logic line connected to it is low. This criterion applies to both inputs and outputs. Unfortunately, to confuse the situation, somewhat larger circles are often shown on data-sheet logic diagrams. They merely represent actual output pins and have no logical significance.

These are the principles underlying tri-state operation. They work equally well with BJT, MOS, and other technologies. The technique is ubiquitous at the interfaces of LSI and VLSI integrated circuits, particularly where bus interfaces are concerned. When devices are not capable of delivering power, tri-state buffers (usually BJT devices) are used both to amplify power and to permit selective enabling of the outputs of the source devices onto a bus.

Tri-stating is widely used to time-share access to a bus. This technique is referred to as *Time Division Multiplexing* (TDM). The basic mechanism is illustrated in Figure 2-9. Here, two selection input lines and an output enable line drive a one-of-four decoder (also called a *demultiplexer*) with active-low outputs. An example of just such a device is the 74LS139 IC. When enabled, only one of the selector's output lines is low at any time—corresponding to the present state of the device's addressing inputs. Each of the output lines in turn controls the enabling inputs of one tri-state gate. At a given instant, therefore, only one of these gates can be active-low enabled, to pass its data on through to the single bus line illustrated. With this system, we are assured that only one gate can be master of the bus at any time. If the decoder is not enabled (E is high in this case), no source is placed on the bus at all. That is, the bus can be totally isolated from all sources, when desired.

Naturally, a bus with more lines would require *sets* of tri-state gates, each arranged as in Figure 2-9. There will be one set of gates per driven bus line, the number of tri-state gates being equal to the number of sources to the line. The corresponding source enable-control lines of each gate would be tied to the related common output from a decoder. In our example, the data inputs to the tri-state gates come from four different sources. The control system of a processor can now determine which one of the four sources shall present its



a. Logic Diagram

INPUTS			TRI-STATE CONTROL LINES			
E	A	B	1	2	3	4
L	L	L	L	H	H	H
L	L	H	H	L	H	H
L	H	L	H	H	L	H
L	H	H	H	H	H	L
H	X	X	H	H	H	H

b. Electrical Characteristics

NOTE: WHEN E IS HIGH, NO BUS SOURCE IS SELECTED.

Figure 2-9
Tri-State Application Example

data to the bus line, by both enabling the selector and forcing its two addressing inputs to one of their four possible states. Thus three of the tri-state control lines are always high, isolating their corresponding outputs from the bus line. The control input of one tri-state set of buffers is low, thus permitting it to present its selected input data to the bus line(s). This approach to time-sharing a data bus with many sources is very flexible. Since TTL logic modules and data are likely to be most readily available to the reader, it is used here most often, to enable the reader to experiment with implementation examples. Where a processor-system design is fabricated with a number of MOS, CMOS, TTL, and other components, some details change—but few principles do.

The ability to select no source at all is inherent in tri-state operation. Where no source is selected, it is usually best to be sure that the driven bus assumes a known quiescent state. Noise-immunity and power-conservation considerations lead to the selection of the electrically high level for the bus in these cases. Figure 2-10 presents one method of implementing this type of bus structure. Notice that the individual bus lines are forced high when all connected tri-state drivers are in the Hi-Z state, by the typical "pull-up" resistors used on each line. Similar methodologies apply to the internal workings of IC's, but this figure illustrates a typical problem to be resolved in interfacing an IC to other IC's. Eight separate sources are shown; the A's and B's indicate the possibility that they belong to different classes of devices. Each distinct source, however, has an associated *set* of tri-state gates, through which it gains access to the bus, subject to some control scheme. The size of each set of tri-state gates for a source must relate to the number of lines of the bus that are accessed. The design of the system must guarantee that it does not attempt to source more lines than the bus can handle at a given instant.

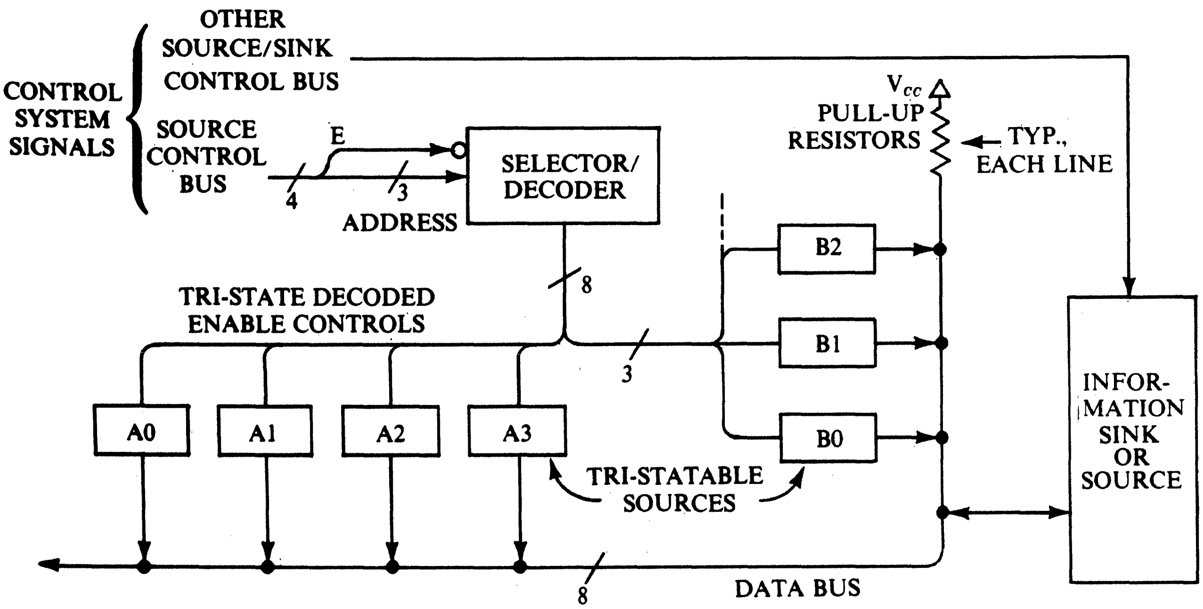


Figure 2-10
Tri-State Bus-Sourcing Control Block Diagram

Thus far, this chapter has been a general review of the characterization and visualization of some of the functional blocks and components of a *computer busing structure.* We have also examined how to visualize the application of these blocks and components in forming a bus system. In a following section, we shall look at the specifications for a few typical devices used in bus interfaces. These are available at your local electronics-parts shop and may be used for experimentation. The following section briefly discusses the bus calculations required for proper application of IC devices. These calculations begin to disclose the considerations involved in interfacing, say, MOS microprocessors with external buses.

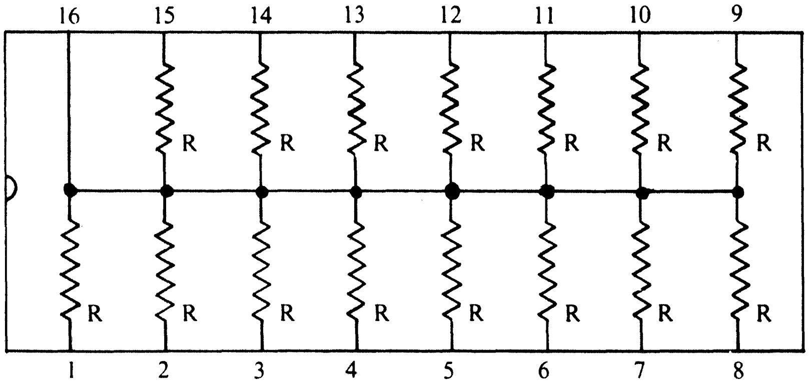
BUS HARDWARE, CALCULATIONS, AND DATA CATALOGS

We shall now examine some typical examples of commercial hardware available to us for use in a data-path construct. The electrically high state of the quiescent bus is maintained by "pull-up" resistors to the power supply. These resistors are supplied in easy-to-use packages, such as the Dual In-line Packages (DIP) used for other IC's, as shown in the schematic of Figure 2-11. Typically, there can be fifteen resistors in a 16-pin DIP, where the last pin is a common connecter to the power supply. In other modules, each resistor is treated as a separate unit, with pin connections at both ends.

The following IC data sheets contain a great deal of useful and practical information; the sheets should be studied in detail as we proceed. Figure 2-12 presents the manufacturer's data on a typical decoder/selector logic module that could be used to enable the tri-state line drivers of a bus. The 74LS138 is a low-power Schottky TTL decoder. The properties that enhance its usefulness are lower than standard TTL power dissipation, which improves reliability by reducing heat failures, and low input-drive requirements. The low input drive is especially interesting. Having a large number of devices connected to a bus can severely tax the driving element's ability to handle the current load. Many sources, such as MOS IC's, have very limited

Figure 2-11
Fifteen-Resistor Array DIP Schematic: 16-Pin DIP Package

Note: Pin 16 is common to all resistors.



drive (current sink when driving TTL) capability and can handle an adequate number of devices only when they have low input-drive requirements. An example of a tri-state driver that has low input/high output-drive characteristics is the 8T97 buffer shown in Figure 2-13.

The manner in which the preceding modules may be employed is further detailed in Figure 2-14, a schematic of a single selected source being fed onto a data bus. Pins 1, 2, and 3 (the address inputs) of the 74138 decoder select which of its outputs is low, provided that it is enabled. Pins 4, 5, and 6 are the enabling inputs. Should pin 6 be low, all outputs are high, and no source is selected. Pins 4 and 5 are additional enabling inputs, but they are active only when low. This large number of enabling inputs is useful in decoding selection addresses aimed at the chip, as well as in expanding the number of outputs by adding more chips. To select a device with multiple enable inputs, *all* enables must be active, whether they are active high or low. Output pin 7, as illustrated, is used as a typical tri-state buffer-control line for a single set of gates. It drives the gate enables on pins 1 and 15 of the 8T97 modules, which control access to an eight-line bus.

There is an important difference in the enabling of a tri-statable interface, such as the 8T97, and that of a module such as the 138. The 74138 does not have a tri-state front-end interface. It possesses the conventional totem-pole output structure. In this case, when it is disabled, *all* outputs are forced to the electrically high state, not the Hi-Z tri-stated state. This way, all driven tri-state active-low control lines are actively held high when the 138 is disabled. The *driven* tri-state gates of this example are in the Hi-Z state under these conditions. Therefore, we should be aware that some disabled IC's can be actively driving their outputs, as opposed to disabled tri-statable ones.

FAN-OUT

The current-handling capacity of a logic module has been referred to a number of times, already. It is important to be aware of and check this parameter in the use of logic modules, particularly since common practice often intermixes logic families. For example, it is common to see MOS and one of the TTL families interfaced to each other. In technical terms, the important parameters are referred to as *fan-out* (FO) and *fan-in* (FI). *Fan-in* is defined as the number of inputs a gate can support and still operate properly. A two-input NAND gate has an FI of two. Thus we see that this parameter has already been accounted for by the circuit designer, and we need not concern ourselves with it,

TTL
MSI

TYPES SN54LS138, SN54LS139, SN54S138, SN54S139, SN74LS138, SN74LS139, SN74S138, SN74S139 DECODERS/DEMULTIPLEXERS

BULLETIN NO. DL-S 7611804, DECEMBER 1972–REVISED OCTOBER 1976

- **Designed Specifically for High-Speed:**
  Memory Decoders
  Data Transmission Systems
- **'S138 and 'LS138 3-to-8-Line Decoders Incorporate 3 Enable Inputs to Simplify Cascading and/or Data Reception**
- **'S139 and 'LS139 Contain Two Fully Independent 2-to-4-Line Decoders/ Demultiplexers**
- **Schottky Clamped for High Performance**

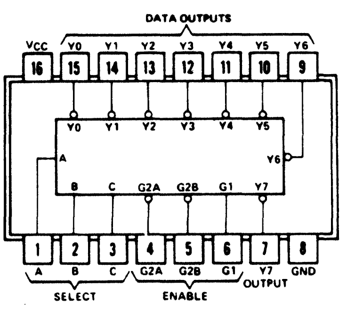
TYPE	TYPICAL PROPAGATION DELAY (3 LEVELS OF LOGIC)	TYPICAL POWER DISSIPATION
'LS138	22 ns	32 mW
'S138	8 ns	245 mW
'LS139	22 ns	34 mW
'S139	7.5 ns	300 mW

description

These Schottky-clamped TTL MSI circuits are designed to be used in high-performance memory-decoding or data-routing applications requiring very short propagation delay times. In high-performance memory systems these decoders can be used to minimize the effects of system decoding. When employed with high-speed memories utilizing a fast-enable circuit the delay times of these decoders and the enable time of the memory are usually less than the typical access time of the memory. This means that the effective system delay introduced by the Schottky-clamped system decoder is negligible.

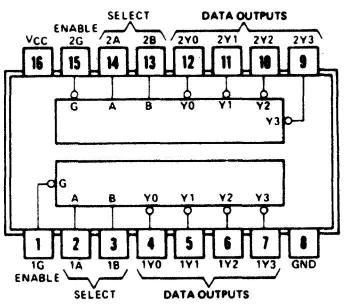
The 'LS138 and 'S138 decode one-of-eight lines dependent on the conditions at the three binary select inputs and the three enable inputs. Two active-low and one active-high enable inputs reduce the need for external gates or inverters when expanding. A 24-line decoder can be implemented without external inverters and a 32-line decoder requires only one inverter. An enable input can be used as a data input for demultiplexing applications.

SN54LS138, SN54S138 . . . J OR W PACKAGE
SN74LS138, SN74S138 . . . J OR N PACKAGE
(TOP VIEW)



positive logic: see function table

SN54LS139, SN54S139 . . . J OR W PACKAGE
SN74LS139, SN74S139 . . . J OR N PACKAGE
(TOP VIEW)



positive logic: see function table

The 'LS139 and 'S139 comprise two individual two-line-to-four-line decoders in a single package. The active-low enable input can be used as a data line in demultiplexing applications.

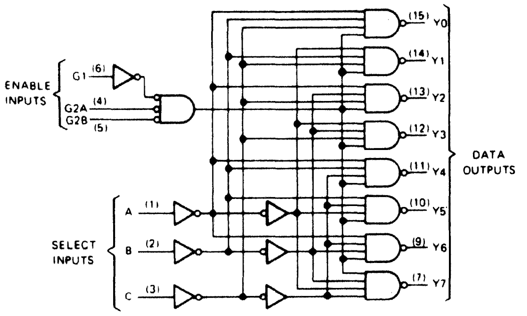
All of these decoders/demultiplexers feature fully buffered inputs each of which represents only one normalized Series 54LS/74LS load ('LS138, 'LS139) or one normalized Series 54S/74S load ('S138, 'S139) to its driving circuit. All inputs are clamped with high-performance Schottky diodes to suppress line-ringing and simplify system design. Series 54LS and 54S devices are characterized for operation over the full military temperature range of −55°C to 125°C; Series 74LS and 74S devices are characterized for 0°C to 70°C industrial systems.

Figure 2-12
74LS138 and 74LS139 Data Sheets
For educational purposes only. Data may be old and obsolete. Courtesy of Texas Instruments, Inc. © 1984, Texas Instruments, Inc.

TYPES SN54LS138, SN54S138, SN54LS139, SN54S139 SN74LS138, SN74S138, SN74LS139, SN74S139 DECODERS/DEMULTIPLEXERS

functional block diagrams and logic

'LS138, 'S138

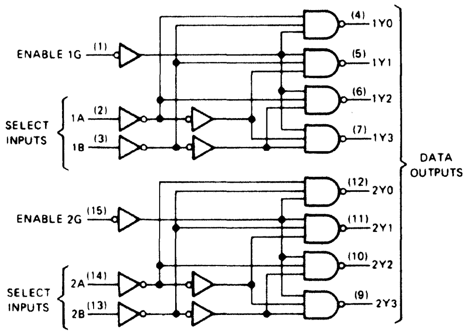


'LS138, 'S138
FUNCTION TABLE

INPUTS					OUTPUTS							
ENABLE		SELECT										
G1	G2*	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	H	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	H	L	H	H	L	H	H	H	H	H
H	L	L	H	H	H	H	H	L	H	H	H	H
H	L	H	L	L	H	H	H	H	L	H	H	H
H	L	H	L	H	H	H	H	H	H	L	H	H
H	L	H	H	L	H	H	H	H	H	H	L	H
H	L	H	H	H	H	H	H	H	H	H	H	L

*G2 = G2A + G2B
H = high level, L = low level, X = irrelevant

'LS139, 'S139

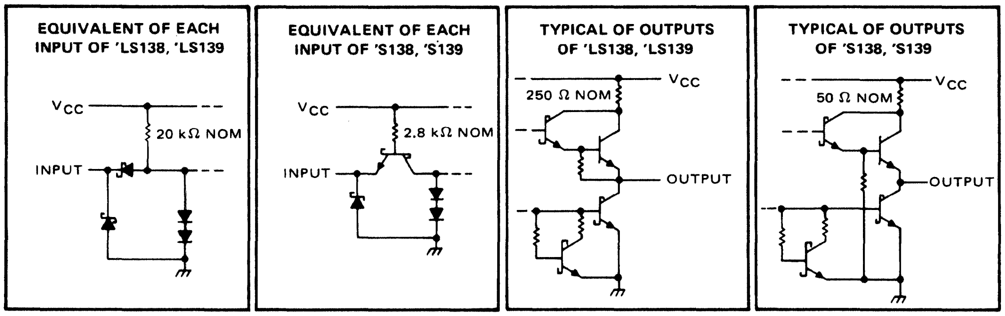


'LS139, 'S139
(EACH DECODER/DEMULTIPLEXER)
FUNCTION TABLE

INPUTS			OUTPUTS			
ENABLE	SELECT					
G	B	A	Y0	Y1	Y2	Y3
H	X	X	H	H	H	H
L	L	L	L	H	H	H
L	L	H	H	L	H	H
L	H	L	H	H	L	H
L	H	H	H	H	H	L

H = high level, L = low level, X = irrelevant

schematics of inputs and outputs


EQUIVALENT OF EACH INPUT OF 'LS138, 'LS139

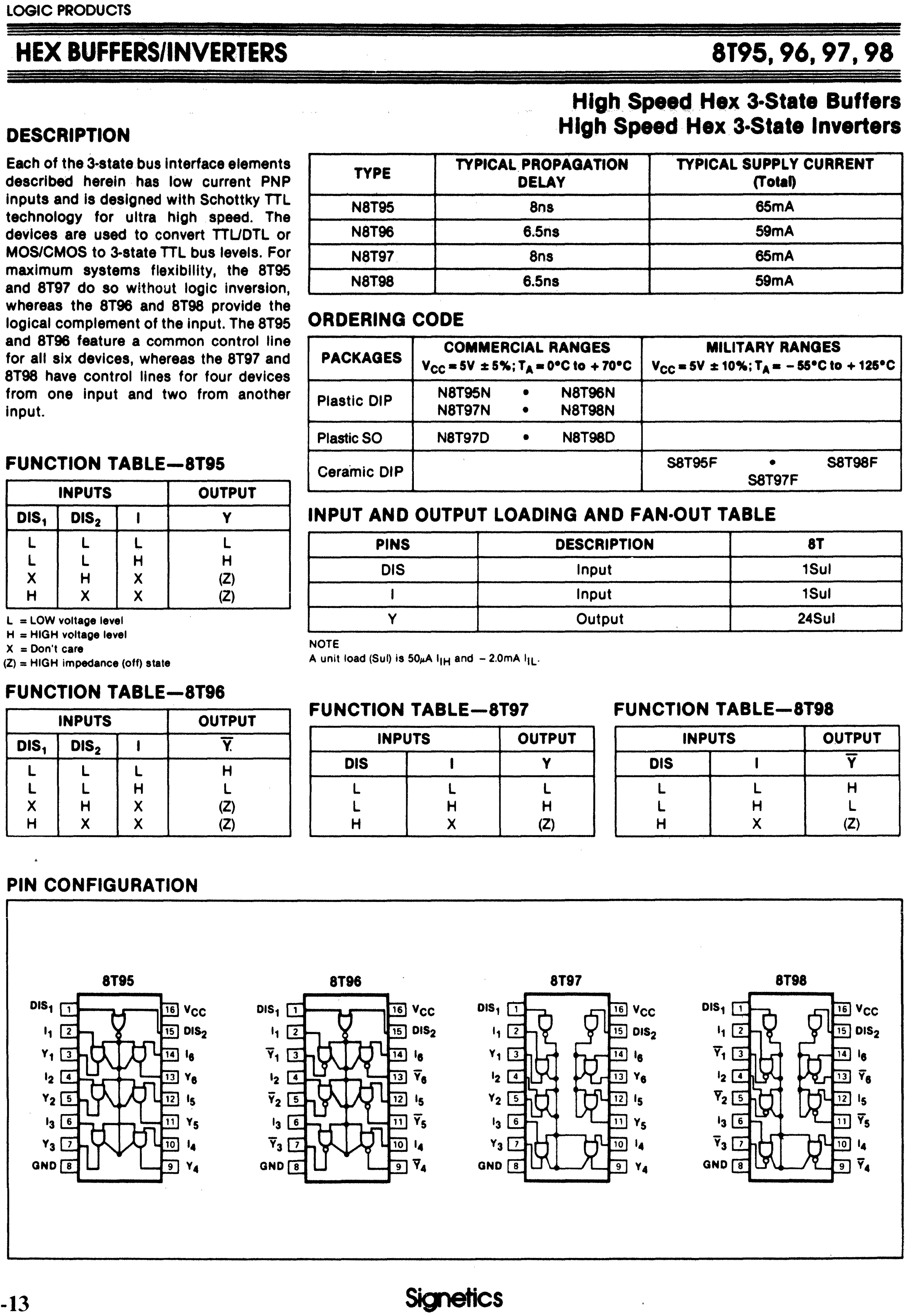
EQUIVALENT OF EACH INPUT OF 'S138, 'S139

TYPICAL OF OUTPUTS OF 'LS138, 'LS139

TYPICAL OF OUTPUTS OF 'S138, 'S139

TEXAS INSTRUMENTS
INCORPORATED
POST OFFICE BOX 5012 • DALLAS, TEXAS 75222

Figure 2-12
74LS138 and 74LS139 Data Sheets
For educational purposes only. Data may be old and obsolete. Courtesy of Texas Instruments, Inc. © 1984, Texas Instruments, Inc.



LOGIC PRODUCTS

HEX BUFFERS/INVERTERS 8T95, 96, 97, 98

High Speed Hex 3-State Buffers
High Speed Hex 3-State Inverters

DESCRIPTION

Each of the 3-state bus interface elements described herein has low current PNP inputs and is designed with Schottky TTL technology for ultra high speed. The devices are used to convert TTL/DTL or MOS/CMOS to 3-state TTL bus levels. For maximum systems flexibility, the 8T95 and 8T97 do so without logic inversion, whereas the 8T96 and 8T98 provide the logical complement of the input. The 8T95 and 8T96 feature a common control line for all six devices, whereas the 8T97 and 8T98 have control lines for four devices from one input and two from another input.

TYPE	TYPICAL PROPAGATION DELAY	TYPICAL SUPPLY CURRENT (Total)
N8T95	8ns	65mA
N8T96	6.5ns	59mA
N8T97	8ns	65mA
N8T98	6.5ns	59mA

ORDERING CODE

PACKAGES	COMMERCIAL RANGES $V_{CC} = 5V \pm 5\%; T_A = 0°C$ to $+70°C$	MILITARY RANGES $V_{CC} = 5V \pm 10\%; T_A = -55°C$ to $+125°C$
Plastic DIP	N8T95N • N8T96N N8T97N • N8T98N	
Plastic SO	N8T97D • N8T98D	
Ceramic DIP		S8T95F • S8T98F S8T97F

FUNCTION TABLE—8T95

INPUTS			OUTPUT
DIS_1	DIS_2	I	Y
L	L	L	L
L	L	H	H
X	H	X	(Z)
H	X	X	(Z)

L = LOW voltage level
H = HIGH voltage level
X = Don't care
(Z) = HIGH impedance (off) state

INPUT AND OUTPUT LOADING AND FAN-OUT TABLE

PINS	DESCRIPTION	8T
DIS	Input	1Sul
I	Input	1Sul
Y	Output	24Sul

NOTE
A unit load (Sul) is 50μA I_{IH} and − 2.0mA I_{IL}.

FUNCTION TABLE—8T96

INPUTS			OUTPUT
DIS_1	DIS_2	I	$\overline{Y}$
L	L	L	H
L	L	H	L
X	H	X	(Z)
H	X	X	(Z)

FUNCTION TABLE—8T97

INPUTS		OUTPUT
DIS	I	Y
L	L	L
L	H	H
H	X	(Z)

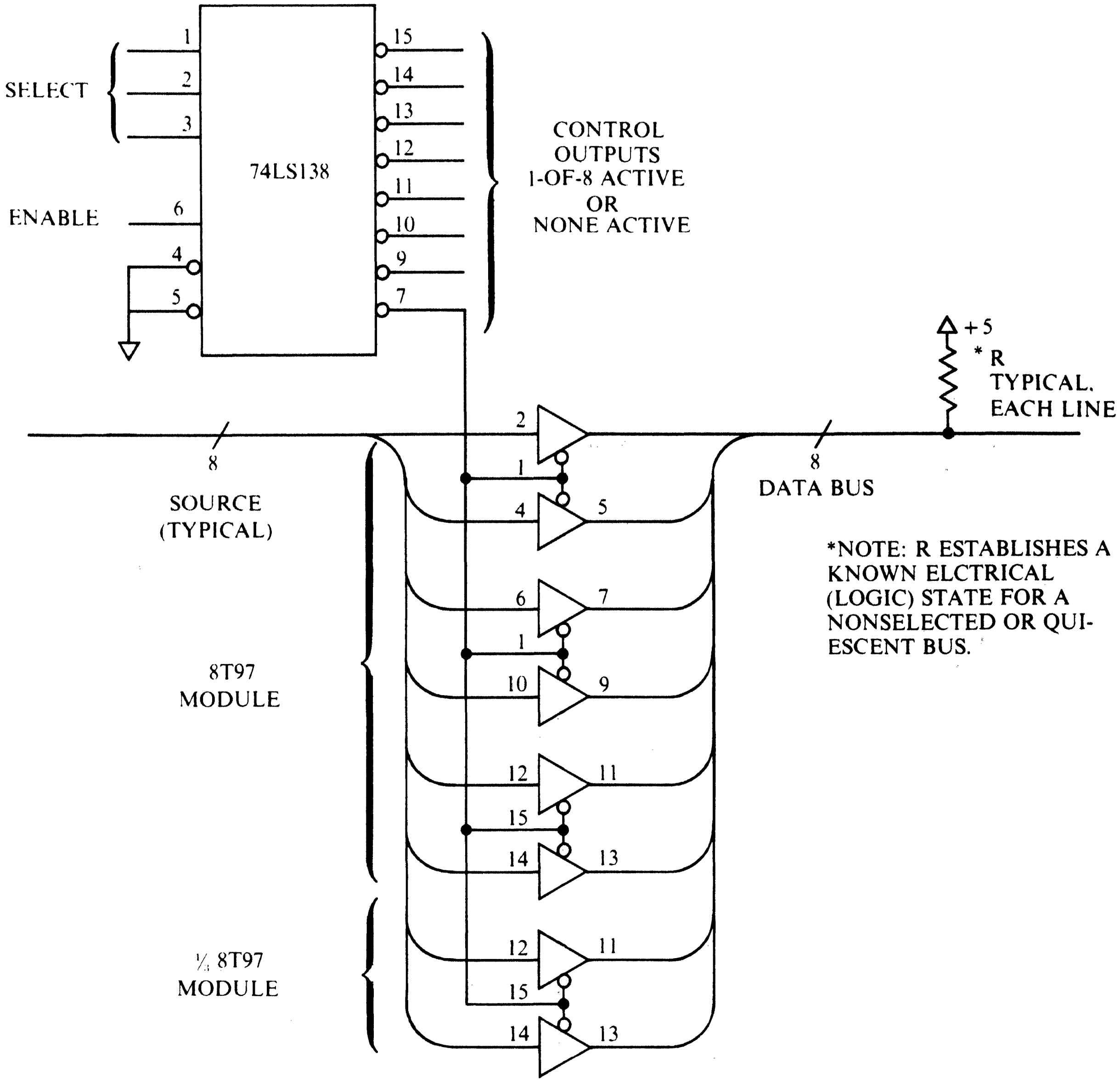
FUNCTION TABLE—8T98

INPUTS		OUTPUT
DIS	I	$\overline{Y}$
L	L	H
L	H	L
H	X	(Z)

PIN CONFIGURATION

Signetics

Figure 2-13
Tri-State Line Drivers Data Sheet
(Courtesy of Signetics Corporation © 1984 Signetics Corporation.)



*NOTE: R ESTABLISHES A KNOWN ELCTRICAL (LOGIC) STATE FOR A NONSELECTED OR QUIESCENT BUS.

Figure 2-14
Typical Source Feed onto a Data Bus

beyond an understanding of the terminology. Fan-out is another matter altogether. We must learn to check this parameter carefully, a process that also gives us an appreciation of why such a large number of buffers and other interface IC's are required in many systems. It should be noted that VLSI relieves the user of many of these problems by resolving them for us within the IC. Another modern trend, that of having custom IC's produced, eliminates many of the user's interfacing problems as well. In spite of these trends, we still have to handle interfacing considerations–be it on our home computers or in evaluating what we see in some commercial computer product.

Let us look at the calculations involved in checking drive capacity at an interface. *FO* refers to the situation that exists while a module is driving the inputs of *other* modules. Any logic module can both source and sink current, but a module usually is designed to have good rrent capacity in only one of these modes. Standard TTL, in fact, sinks 16 milliamps and is rated to source only 0.8 milliamps. These IC parameters, found in a data catalog, tell us how many other modules the module can drive. Fan-out restrictions are often easy to violate in use, and the results would be comic if they were not so sad. Quantitatively, FO is defined as the ratio:

$$\text{FO} = \frac{\text{Current Capacity of } \textit{Driver} \text{ Output}}{\text{Total Current Requirements of } \textit{Driven} \text{ Inputs}}$$

This factor should be checked whenever logic modules are interfaced. Above, we refer to driver (output) and driven (input) pins, which are not necessarily members of the same family of logic devices. In practice, it comes down to ascertaining whether the driver can handle the total current requirements of the several types of driven elements it may be interfaced with, e.g., MOS, TTL, LSTTL, among others.

Table 2-1 summarizes these parameters for a number of devices of different families, as an indication of the spread of values that are encountered in practice. Several logic families are included for comparison of their relative drive capacities. Values from this table are used to illustrate the following examples. Since these values vary somewhat by manufacturer, by temperature, and between different devices of the same family, one should consult a data catalog or a manufacturer's representative for precise data. These are only guidelines.

Table 2-1
FO Current Drive Parameters

FO Calculation Parameter	*MOS*		*TTL*			*Typical Interface IC's*		*Units*
	CMOS Gate	*NMOS μ Processor Interface*	*74LS*	*74*	*74S*	*74LS240*	*8T97*	
I_{IH}	1*	± 10*	20	40	50	20	40	μ A
I_{IL}	1*	± 10*	−0.36	−1.6	−2.0	−0.2	−0.4	mA
I_{OH}	2.8	−0.4	−0.4	−0.4	−1.0	−15	−5.2	mA
I_{OL}	2.8	2.0	8	16	20	24	40	mA

*Notes: 1. *Leakage*

2. Caution! These parameters vary with temperature, manufacturer, and device. The values given here are only relative guides.

Two types of fan-out must be calculated: that for the electrically high and that for the electrically low state. In industrial terminology, the high-state parameter is *FO1* and the low state parameter is *FO0*. Since we already know that *0* and *1* are truth not electrical values, we can use the terms FOH and FOL instead in what follows. Below are the calculations for the 74LS, driving 74 TTL modules:

$$\text{FOH} = \frac{I_{OH}}{I_{IH}} = \frac{400\ \mu\text{A}}{40\ \mu\text{A}} = 10$$

$$\text{FOL} = \frac{I_{OL}}{I_{IL}} = \frac{8\ \text{mA}}{1.6\ \text{mA}} = 5$$

What are the terms I_{OH}, I_{OL}, I_{IH}, and I_{IL}? *I refers to current, $_O$ to output, and $_I$ to input. The $_L$ and $_H$ are of course the electrical-state levels. Almost any good logic-family data catalog contains a glossary of terms such as these.*

The need for the student to have a data catalog for each family of interest cannot be emphasized enough. A good deal of software is written for specific devices. One must start early by learning the basic data-catalog terminology, to comprehend the data sheets that describe the functioning of floppy-disk controller or arithmetic processor IC's and how to program them. As a matter of fact, much of the information in these texts is taken verbatim from the data catalog. Why not go straight to the source? The advanced microprocessor and peripheral-support IC data sheets start with this type of information and proceed to the higher levels of systems application and programming. Therefore, we are being exposed to basic training on how to read a data catalog as we proceed.

We can see that the 74LS family will adequately sink current from five 74-series standard TTL inputs. This is the lower of the two values calculated and must be used as the limit to the number of modules that can be interfaced safely. In comparison, each 74LS138 output must handle eight 8T97 inputs in the interface scheme presented in Figure 2-14. Since the 8T-series module features low current sourcing inputs, this may be possible. The calculations that follow decide this.

$$\text{FOH} = \frac{I_{OH}}{I_{IH}} = \frac{400\ \mu\text{A}}{40\ \mu\text{A}} = 10$$

$$\text{FOL} = \frac{I_{OL}}{I_{IL}} = \frac{8\ \text{mA}}{0.4\ \text{mA}} = 20$$

In this case, the 74LS138's fan-out of ten in the FOH state is the limiting factor, assuring us that we can drive the eight inputs of the schematic and satisfying the fan-out criterion.

Notice, however, that the last calculation showed FOH to be the limiting factor, not FOL, as in the first example. All this illustrates that interfacing considerations are a large part of the work associated with the implementation of processors and bus systems. Again, our best allies are the data manuals of each family being used. Here, we are examining the nature of the bus and its associated logic modules. The terminology learned now—and the interpretive skills gained in the process—will carry forward into the reading of microprocessor and peripheral-support IC data catalogs.

Bi-Directional Bus Drivers

An important example of another bus-module interface construct, either internal or external to an IC, is the bidirectional bus driver. We will illustrate the features involved with the application of an 8T26 bidirectional bus transceiver module. There are many related buffers, such as the more modern 74LS240 through 74LS245 IC modules. The 8T26 is very flexible and instructive, because it can be externally connected in several different ways, where some of the possible connections of most other modules are internally committed. It offers a high current capacity to the driven bus tap and a moderate drive capability to the side designated as the receiver, since this side does not usually interface to the "outside world." Its manufacturer's specifications are presented in Figure 2-15.

The essential characteristics of how this bidirectional driver may be utilized are presented in the schematic and applications of Figure 2-16, a bus relationship between a CPU and its external memory. The control lines are used to avoid confusion as to which is

LOGIC PRODUCTS

BUS TRANSCEIVERS 8T26A, 28

3-State Quad Bus Transceiver

- **High speed Schottky quad transceivers**
- **48mA LOW-state drive**
- **200μA bus loading**
- **Ideal for:**
  Half-duplex data transmission
  Memory interface buffers
  Data routing in bus oriented systems
  High current drivers
  MOS/CMOS-to-TTL interface

TYPE	TYPICAL PROPAGATION DELAY	TYPICAL SUPPLY CURRENT (Total)
N8T26A	7ns	48mA
N8T28	10ns	67mA

ORDERING CODE

PACKAGES	COMMERCIAL RANGES V_{CC} = 5V ±5%; T_A = 0°C to +70°C	MILITARY RANGES V_{CC} = 5V ±10%; T_A = −55°C to +125°C
Plastic DIP	N8T26AN • N8T28N	
Ceramic DIP		S8T26AF • S8T28F

INPUT AND OUTPUT LOADING AND FAN-OUT TABLE

PINS	DESCRIPTION	N8T	S8T
I_N	Input	0.5Sul	0.5Sul
D/E, R/E	Inputs	0.5Sul	0.5Sul
D_{OUT}	Output	24Sul	16Sul
R_{OUT}	Output	10Sul	6Sul

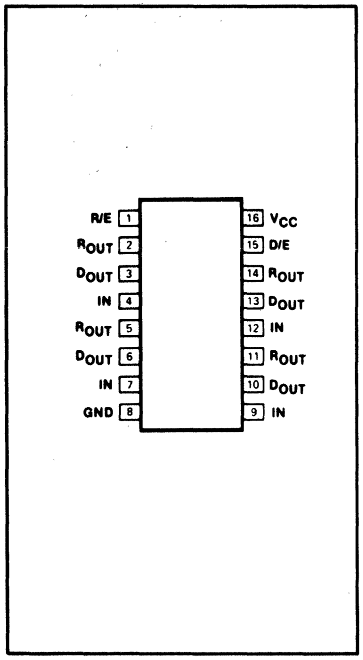
NOTE
A unit load (Sul) is 50μA I_{IH} and -2.0mA I_{IL}.

DESCRIPTION

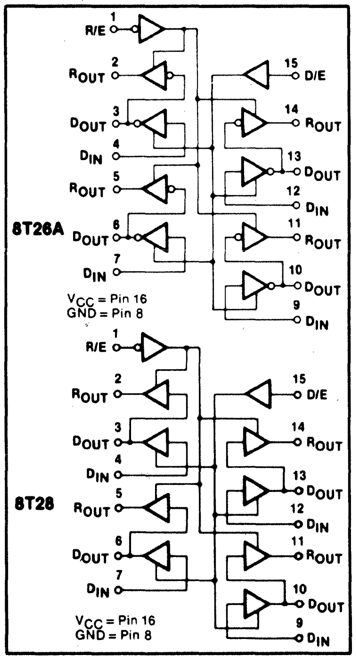
The 8T26A/28 consists of four pairs of 3-state logic elements configured as quad bus drivers/receivers, along with separate buffered receiver enable and driver enable lines. This single IC quad transceiver design distinguishes the 8T26A/28 from conventional multi-IC implementations. In addition, the 8T26/28's ultra high speed while driving heavy bus capacitance (300pF) makes these devices particularly suitable for memory systems and bidirectional data buses.

Both the driver and receiver gates have 3-State outputs and low-current PNP inputs. 3-State outputs provide the high switching speeds of totem-pole TTL circuits while offering the bus capability of open collector gates. PNP inputs reduce input loading to 200μA maximum.

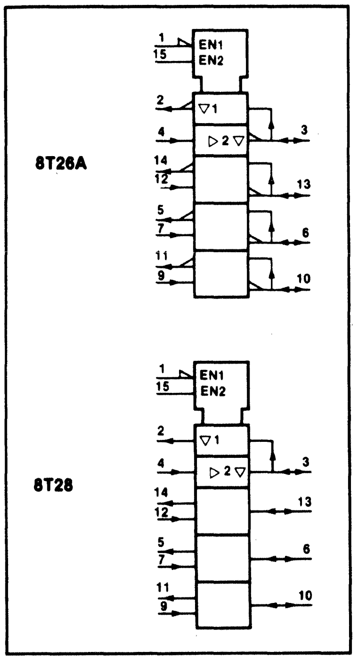
PIN CONFIGURATION



LOGIC SYMBOL



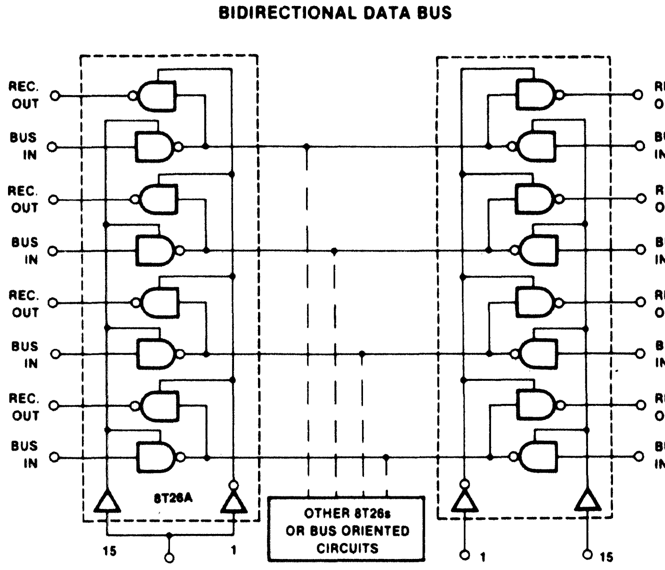
LOGIC SYMBOL (IEEE/IEC)



Signetics

Figure 2-15a
Tri-State Quad Bus Transceiver Data Sheet
(Courtesy of Signetics Corporation © 1984 Signetics Corporation.)

TYPICAL APPLICATION



Control lines may be tied together, such that logical "1" transmit, logical "0" receive.

Logical "0" = active Logical "11" = Hi-z

Logical "1" = active Logical "0" = Hi-z

Signetics

Figure 2-15b
Tri-State Quad Bus Transceiver Data Sheet
(Courtesy of Signetics Corporation © 1984 Signetics Corporation.)

master of the bus. They must be properly coordinated by the microprocessor's CPU control signals in the application. The CPU possesses the ability to tell the external bus driver whether it wants to read or write memory, either directly or through decoding logic. These signals manage the tri-state bus driver's bidirectional behavior. In this example, each end of the bus is connected in a different manner. Note that the CPU's bus interface is connected *both* to the receiver output and to the driver input and that proper tri-stating prevents electrical confusion. This is termed single-port, or *common,* I/O. Some memory systems have dual-port I/O. That is, some modules have separate inputs and outputs, as opposed to the common I/O of the CPU. In this case, the 8T26 has the flexibility to be used in both cases, as illustrated at the memory interface. Follow the direction-control logic action to observe how conflicts are avoided on a typical line of a bidirectional bus system.

We will not go into the details of noise immunity, voltage margins, and other design criteria in this text, important as they are, because this book is aimed at a general readership, not only designers, for the purpose of providing an understanding of *how* a processor functions internally for microprogramming purposes. The goal is to enable one to recognize that hardware organization and software operation are really one unified subject. Fan-out is discussed because that is one design criterion that even the home hobbyist must handle, and as an insight into design specifics in hardware organization. If the fan-out criteria are observed, other considerations are generally not violated.

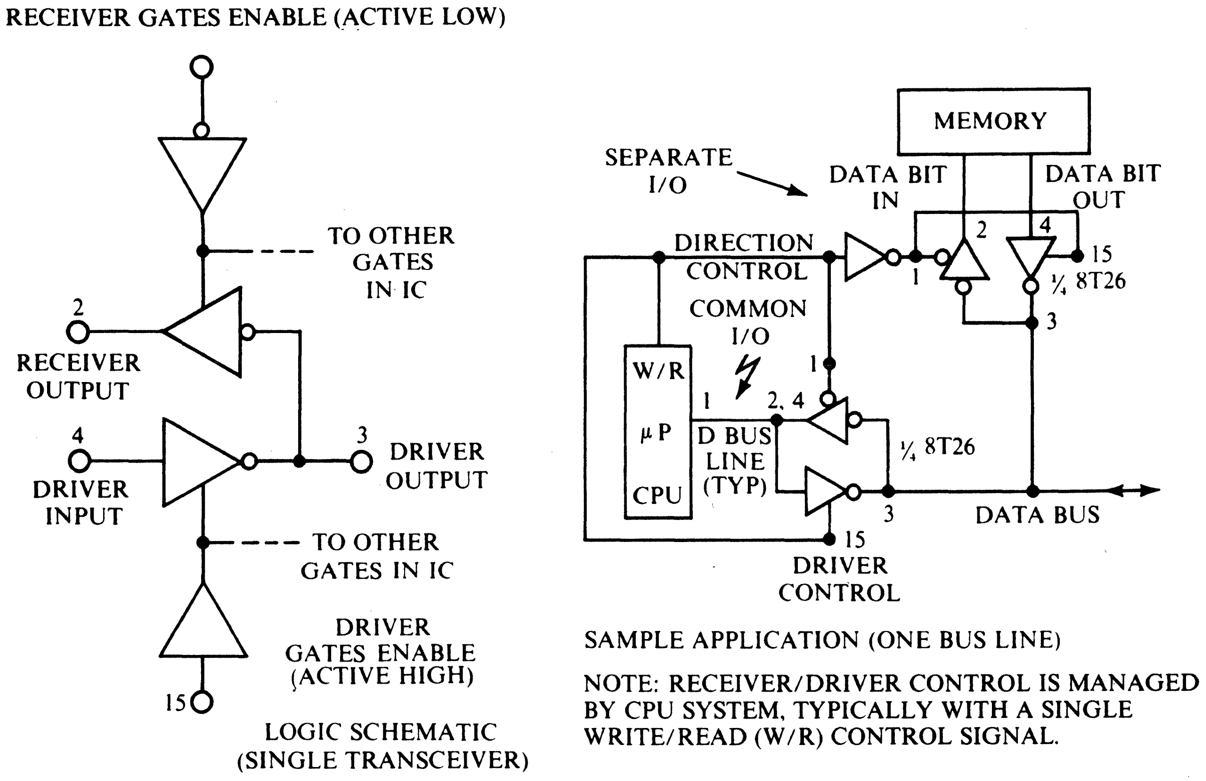

RECEIVER GATES ENABLE (ACTIVE LOW)
TO OTHER GATES IN IC
2
RECEIVER OUTPUT
4
DRIVER INPUT
3
DRIVER OUTPUT
TO OTHER GATES IN IC
15
DRIVER GATES ENABLE (ACTIVE HIGH)
LOGIC SCHEMATIC (SINGLE TRANSCEIVER)
MEMORY
SEPARATE I/O
DATA BIT IN
DATA BIT OUT
DIRECTION CONTROL
COMMON I/O
W/R
μP
CPU
D BUS LINE (TYP)
1
2, 4
2
4
15
¼ 8T26
3
1
¼ 8T26
3
DATA BUS
15
DRIVER CONTROL
SAMPLE APPLICATION (ONE BUS LINE)
NOTE: RECEIVER/DRIVER CONTROL IS MANAGED BY CPU SYSTEM, TYPICALLY WITH A SINGLE WRITE/READ (W/R) CONTROL SIGNAL.

Figure 2-16
Bus Transceiver: Schematic and Application Example

The advanced hardware considerations have been kept to the minimum that still enables us to practice computer "architecture" (vs. design) as we proceed. The reading of data manuals has been emphasized as essential to practicing modern computer architecture, microprogramming, and systems organization. It is also essential in the development of systems software. Knowledge obtained from manufacturer's data manuals, which often include software examples for the programmable devices, gradually creates an awareness of the many other factors that cannot be adequately treated here, lest we bury our main points in extraneous detail. The TTL data catalog is basic training for these future purposes.

In summation, we have seen that the organization of sources and sinks around a data path that employs gates, tri-state drivers, selection modules, load enable signals, and a system clock are basically what a bus system is composed of. Some additional elements are introduced as we proceed, but fundamentally, we have already seen a busing system. The visual image of a bus system as a communication network is important. The bus has been referred to as the "skeleton" of a computer system, around which the system itself is constructed. Certainly, it is the communication structure through which the computer system operates, by relating to its elements in the structured manner managed by the control system. The concept of the skeletal functioning of buses in systems structure even extends to distributed processing systems via their networks. The topic is of such importance that the federal government issues standards on busing systems organization. The Anderson and Jensen article listed in the References, on the taxonomy and characteristics of computer interconnection structures (buses), offers an excellent perspective on systems-interconnection methodology.

Multiplexing and Demultiplexing

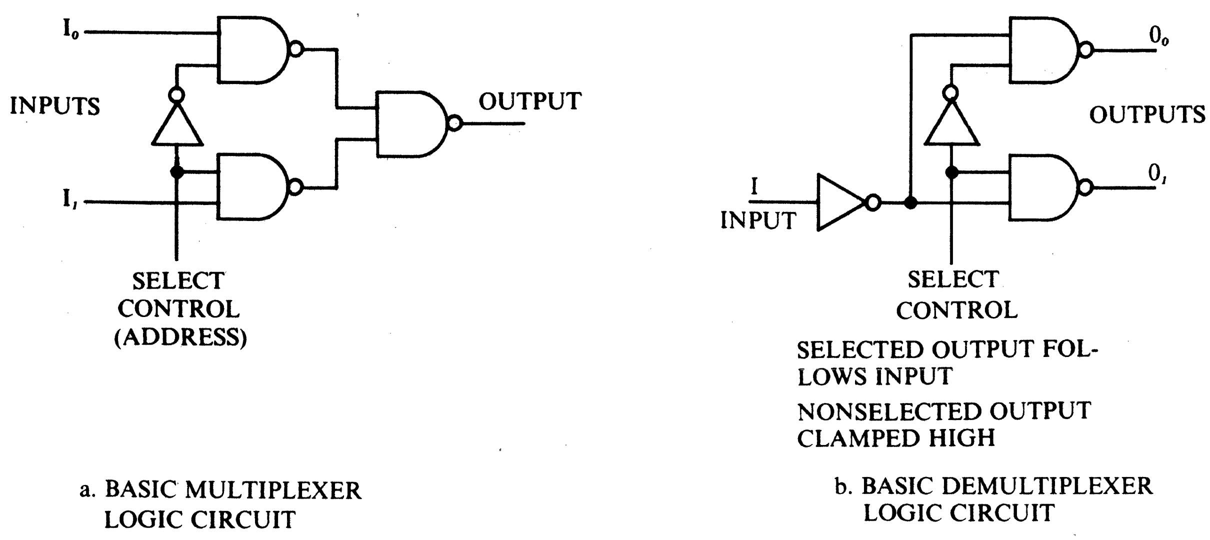
We have already referred to the use of multiplexing (MPX) as a method for time-sharing a communications channel. The main example given so far was the tri-stating of buses, as detailed in the previous section. Multiplexing can be implemented in several ways. In communications, *Frequency Division Multiplexing* (FDM) is often performed. Here, separate sources share a communications medium simultaneously, each channel separated from the others only by the bandwidths assigned to each of the sources. This method finds frequent use with digital systems in the simultaneous transmission of several encoded data packets. *Time Division Multiplexing* (TDM), previously mentioned, occurs when a single channel, such as a bus, is shared by several sources, each using the path in a separate time slot. The time slots are often derived from the system's coordinated control and clock signals. Our interest at present is to pursue TDM methodologies further.

We will also look further into the opposite of multiplexing, called–as you might guess–*demultiplexing* (DMPX). Demultiplexing, when employed as the counterpart of TDM, is the technique of placing information from a single time-shared channel onto the appropriate separate channels, to reach the desired sink. It is a means of distributing data. These methods of data selection and distribution may be based on the use of combinational logic, tri-stating, open collector, and other techniques. While tri-stating has some superior properties for larger external interfaces, it is not the only important technique. Combinational-logic function and open-collector multiplexers are frequently used, too. The major drawback of the combinational-logic approach is the large number of gates or IC's required, as compared to the others, where there are many bus lines to interface to.

Multiplexing, then, means selecting one of several sources of information and placing it on a single data path. Demultiplexing, on the other hand, takes information from a single input source and selectively distributes it to one of several outputs. Let us examine some data sheets and the principles of logic related to these techniques.

Many IC's are available to accomplish demultiplexing. Actually, the three-line-to-eight-line 74LS138 decoder of Figure 2-12, which we have already seen in the previous section, can also serve as a demultiplexer. For example, with reference to this figure, suppose that, for a given state of the three select (address) inputs A, B, and C, a corresponding sink is to accept information from the output of the 138 when this IC is enabled. One way to achieve this is to bring G_{2A} and G_{2B} low. This only partially enables the 138. We must yet account for the role of G_1 in this scheme.

If the last enable input, G_1, of the 138 demultiplexer is controlled by the incoming data, the complement of the data appears at its selected output. Unselected outputs are always high, as are *all* outputs of the device when it is not enabled by G_1. All separate sink inputs are connected to their corresponding outputs of the 138. The appropriate sink, as managed by the control system, can clock in the complemented data arriving at its input from the *addressed* output of the 138 via G_1. When G_1 is high, the IC is enabled and the selected output is low. When G_1 is low, the IC is disabled and all outputs are low. The chip-enable lines have thus been partitioned into two parts–the actual enabling of the IC and a data input. How does this differ from tri-stat-



a. BASIC MULTIPLEXER LOGIC CIRCUIT

b. BASIC DEMULTIPLEXER LOGIC CIRCUIT

Figure 2-17
MPX and DMPX with Combinational Logic

ing? How would you alter these arrangements if you did not want to handle complemented data?

MPX AND DMPX LOGIC

The fundamental circuit concepts behind combinational-logic multiplexers and demultiplexers are shown in Figure 2-17. In Figure 2-17a, which illustrates the multiplexer, the control-input line and its inversion force one of the two input NAND gates to have its output high. Recall that the NAND function for active-high conventions is performed by a device whose output is high when *any* single input goes low. If one of the two input-stage gates receives a low control signal, then the other input of this gate, the data input, has no effect on this gate's output, since it is clamped high. Data levels will not propagate further. For the other, the control-enabled gate, the complement of the data appears at its output, and the output changes with the input data. The final stage, also a NAND gate, always has one input held high and the other active. Again, a NAND function was produced by a device whose output is low only when *both* inputs are high. This results in the recomplementation of the selected data at the final output. These principles are simply extended to select more than the two inputs illustrated, in commercial IC's. The functional explanation of the basic combinational-logic circuitry behind demultiplexing is left as an exercise.

An example of an IC that performs multiplexing through combinational logic is the 74LS251 IC module in Figure 2-18. Its three select inputs, $S_2 \ldots S_0$, determine which selected input appears at the output when the chip is enabled. In addition, this IC offers tri-stated true and complement outputs of the selected input. This tri-state capability is interesting, in that the output can now also be TDMed onto a bus line, along with several other outputs from other 251s. Figure 2-19 presents a generalized block diagram of several multiplexers that time-share an n-bit-wide bus between p different sources, also assumed to be n lines wide. Each separate multiplexer is a p:1 data selector. As shown, each individual MPX IC is responsible for picking up the *same* bit from each separate source and placing it onto the corresponding bit line of the driven bus that its output is connected to. That is, the ith bit of each source is multiplexed into *i*th multiplexer and placed on the *i*th line of the driven bus, when selected. Thus, if a source address is presented to all of the multiplexers in common, the parallel data of the source is reconstructed on the data bus. The source selection control bus consists of *m* lines, such that $2^m \geq p$, these *m* lines being connected to all MPX's in common, as address inputs. The tri-state control line is generally connected to all modules in common, too.

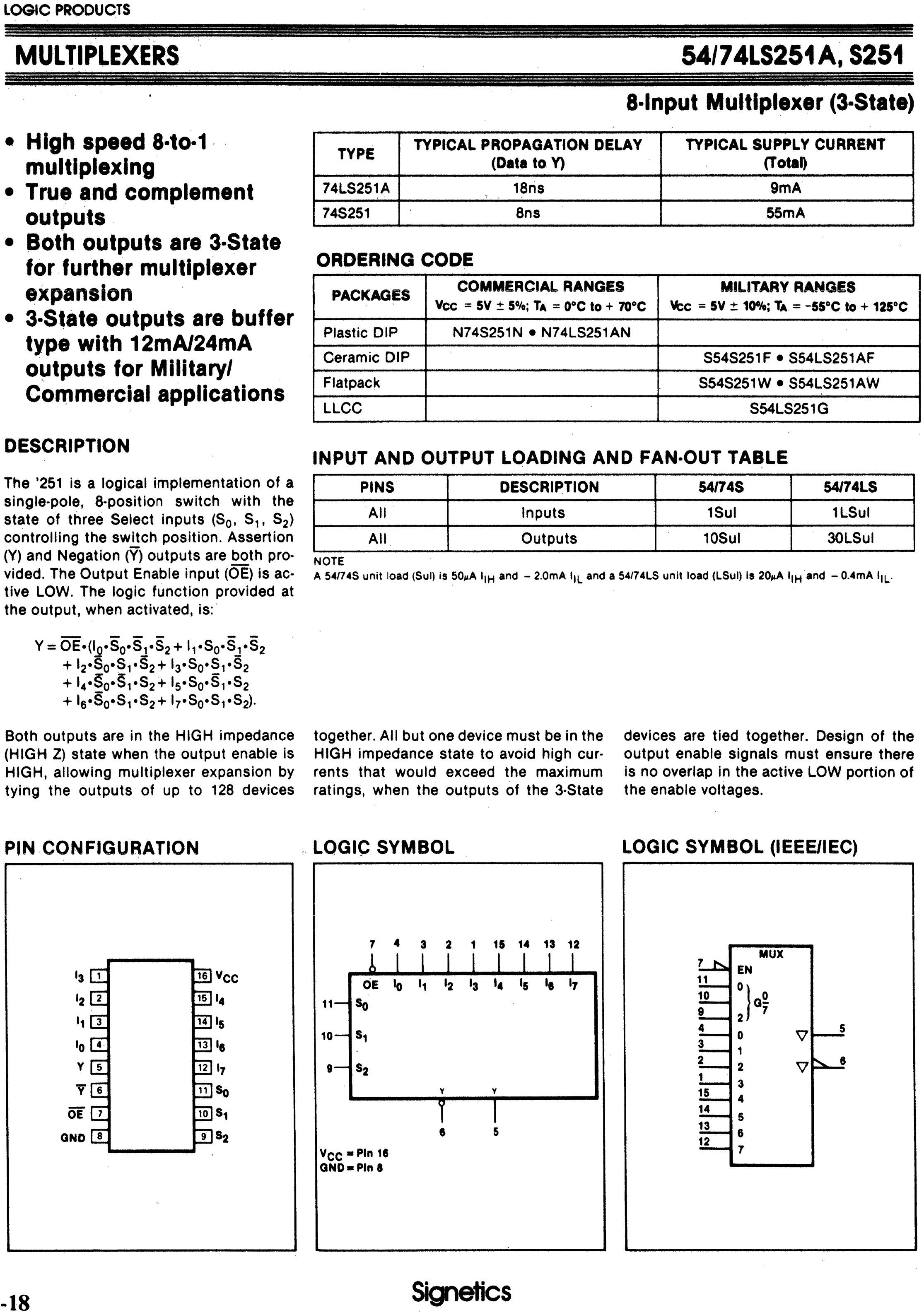
Open-Collector Interfaces

Some of these types of devices have *open-collector* outputs, which use external pull-up resistors on the driven bus line. An open collector driver contains only one transistor, instead of a totem pole, at its output interface. This structure is shown in Figure 2-20. If this device is disabled, the output(s) behave as an open switch. In this state, the line is pulled high by the external pull-up resistor. If the device is enabled and its data input(s) close the output switch, that line is pulled low. This type of connection scheme is often referred to as wire-ANDing or wire-ORing, because the drivers actually perform a logic function in driving a common bus line. Tri-stating is not the principle used here. True, disabling opens the output switch, but the driven line is always pulled to the high (quiescent) level, when disabled, by a pull-up resistor. No Hi-Z state is involved in these actions. Before tri-stating was developed, open-collector operation was very popular for interfacing to a bus. It has the advantage of requiring fewer gates than other methods, particularly for MPX applications. Since open-collector methods make sparing use of gates, they are used in the internal design of IC's.

Open-collector operation is often used with IC's intended for modular use, where a logical form of voting is required. The 74181 ALU, which we will soon study, has an open-collector interface at its comparison output pin. If several of these ALUs are operated in parallel, to form a larger ALU, their open-collector outputs are wired together along with a single common pull-up resistor. The comparison output can be high only when each individual ALU submodule in the chain agrees (i.e., has voted) that equality is detected. (The vote consists of all open-collector interfaces asserting a high—the wired AND.)

Figure 2-21, an illustration of TDM applied to a bus, shows that the use of tri-stating can be very simple. Many sources (registers, etc.) now come equipped with built-in tri-statable outputs and associated control-enable input(s). A good example is the 74LS173 4-bit register, which we shall make extensive use of later. Therefore, *all* the separate source-output interfaces can be directly connected to the appropriate driven bus lines. In these cases, one only has to enable the output of one source at a time with tri-state control lines, as shown. The source selection is often performed through the use of a 1-of-*p* decoder. Tri-stating is widely used, but, as noted, combinational-logic multiplexers may be more feasible where small amounts of data are handled.

Usually, in multiplexing, each source feeding an *n*-line bus contains the same number of bits as the data bus it is fed onto. In some important cases, two different sources of, say, *x* and *y* bits may be concatenated and then multiplexed onto a bus in parallel. In these instances, $x + y \leq n$. An example is the formation of 16-bit results



LOGIC PRODUCTS

MULTIPLEXERS 54/74LS251A, S251

8-Input Multiplexer (3-State)

- **High speed 8-to-1 multiplexing**
- **True and complement outputs**
- **Both outputs are 3-State for further multiplexer expansion**
- **3-State outputs are buffer type with 12mA/24mA outputs for Military/Commercial applications**

TYPE	TYPICAL PROPAGATION DELAY (Data to Y)	TYPICAL SUPPLY CURRENT (Total)
74LS251A	18ns	9mA
74S251	8ns	55mA

ORDERING CODE

PACKAGES	COMMERCIAL RANGES Vcc = 5V ± 5%; T_A = 0°C to + 70°C	MILITARY RANGES Vcc = 5V ± 10%; T_A = -55°C to + 125°C
Plastic DIP	N74S251N • N74LS251AN	
Ceramic DIP		S54S251F • S54LS251AF
Flatpack		S54S251W • S54LS251AW
LLCC		S54LS251G

DESCRIPTION

The '251 is a logical implementation of a single-pole, 8-position switch with the state of three Select inputs (S_0, S_1, S_2) controlling the switch position. Assertion (Y) and Negation ($\overline{Y}$) outputs are both provided. The Output Enable input ($\overline{OE}$) is active LOW. The logic function provided at the output, when activated, is:

$$\begin{aligned}Y = \overline{OE}\bullet(&I_0\bullet\overline{S}_0\bullet\overline{S}_1\bullet\overline{S}_2 + I_1\bullet S_0\bullet\overline{S}_1\bullet\overline{S}_2\\ &+ I_2\bullet\overline{S}_0\bullet S_1\bullet\overline{S}_2 + I_3\bullet S_0\bullet S_1\bullet\overline{S}_2\\ &+ I_4\bullet\overline{S}_0\bullet\overline{S}_1\bullet S_2 + I_5\bullet S_0\bullet\overline{S}_1\bullet S_2\\ &+ I_6\bullet\overline{S}_0\bullet S_1\bullet S_2 + I_7\bullet S_0\bullet S_1\bullet S_2).\end{aligned}$$

INPUT AND OUTPUT LOADING AND FAN-OUT TABLE

PINS	DESCRIPTION	54/74S	54/74LS
All	Inputs	1Sul	1LSul
All	Outputs	10Sul	30LSul

NOTE
A 54/74S unit load (Sul) is 50μA I_{IH} and − 2.0mA I_{IL} and a 54/74LS unit load (LSul) is 20μA I_{IH} and − 0.4mA I_{IL}.

Both outputs are in the HIGH impedance (HIGH Z) state when the output enable is HIGH, allowing multiplexer expansion by tying the outputs of up to 128 devices together. All but one device must be in the HIGH impedance state to avoid high currents that would exceed the maximum ratings, when the outputs of the 3-State devices are tied together. Design of the output enable signals must ensure there is no overlap in the active LOW portion of the enable voltages.

PIN CONFIGURATION

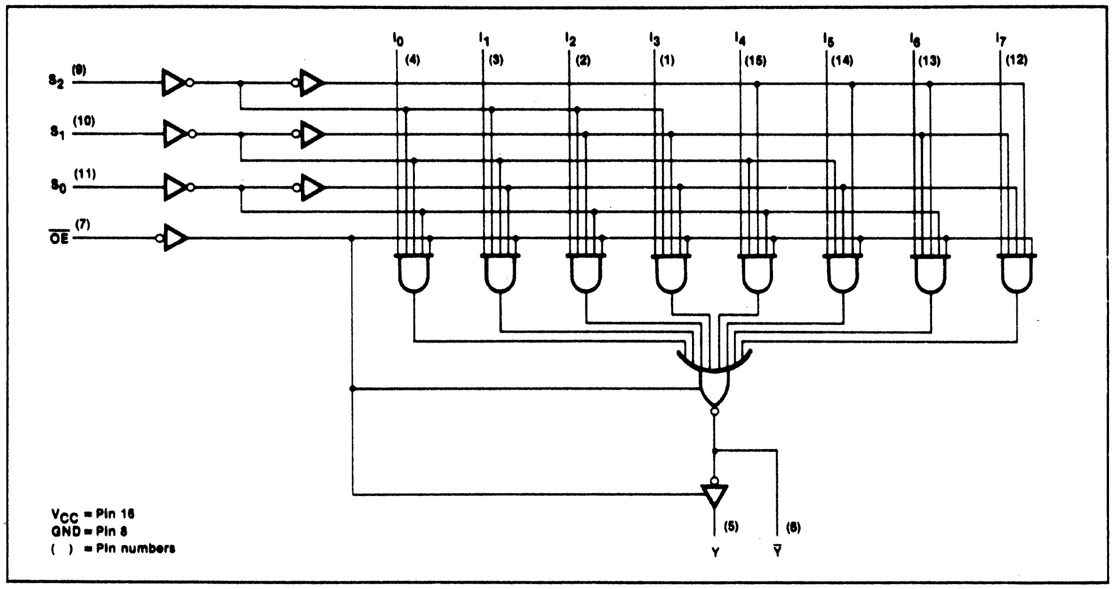
LOGIC SYMBOL

LOGIC SYMBOL (IEEE/IEC)

Signetics

Figure 2-18
Tri-State Line Drivers Data Sheet
(Courtesy of Signetics Corporation © 1984 Signetics Corporation.)

LOGIC DIAGRAM



FUNCTION TABLE

INPUTS												OUTPUTS	
$\overline{OE}$	S_2	S_1	S_0	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	$\overline{Y}$	Y
H	X	X	X	X	X	X	X	X	X	X	X	(Z)	(Z)
L	L	L	L	L	X	X	X	X	X	X	X	H	L
L	L	L	L	H	X	X	X	X	X	X	X	L	H
L	L	L	H	X	L	X	X	X	X	X	X	H	L
L	L	L	H	X	H	X	X	X	X	X	X	L	H
L	L	H	L	X	X	L	X	X	X	X	X	H	L
L	L	H	L	X	X	H	X	X	X	X	X	L	H
L	L	H	H	X	X	X	L	X	X	X	X	H	L
L	L	H	H	X	X	X	H	X	X	X	X	L	H
L	H	L	L	X	X	X	X	L	X	X	X	H	L
L	H	L	L	X	X	X	X	H	X	X	X	L	H
L	H	L	H	X	X	X	X	X	L	X	X	H	L
L	H	L	H	X	X	X	X	X	H	X	X	L	H
L	H	H	L	X	X	X	X	X	X	L	X	H	L
L	H	H	L	X	X	X	X	X	X	H	X	L	H
L	H	H	H	X	X	X	X	X	X	X	L	H	L
L	H	H	H	X	X	X	X	X	X	X	H	L	H

H = HIGH voltage level
L = LOW voltage level
X = Don't care
(Z) = HIGH impedance (off) state

Figure 2-18
Tri-State Line Drivers Data Sheet
(Courtesy of Signetics Corporation © 1984 Signetics Corporation.)

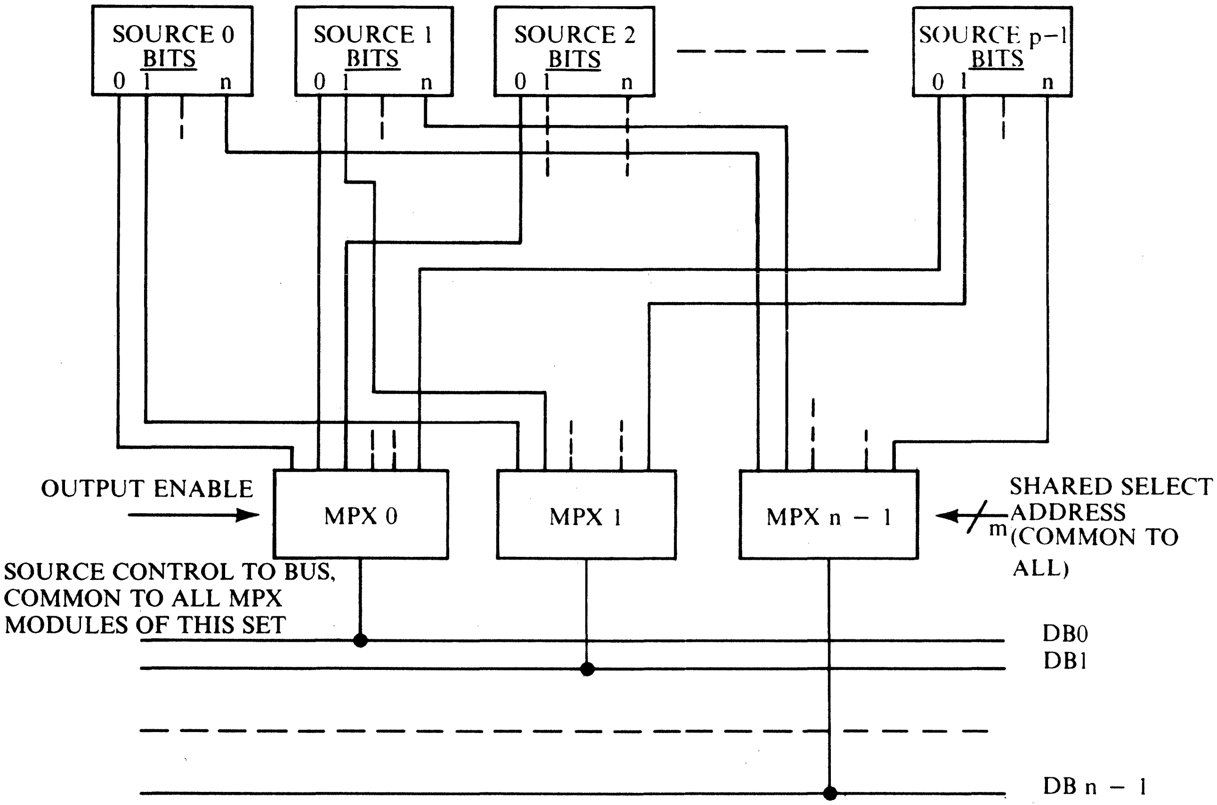


Figure 2-19
Multiplexing *p* Sources onto *n* Bus Lines

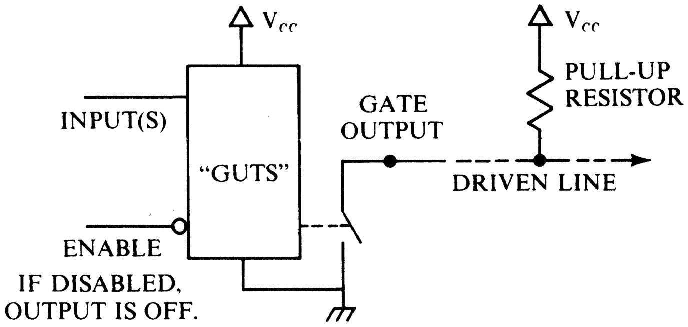


Figure 2-20
Open-Collector Operating Principle

NOTE: IF PULL-UP RESISTOR NOT USED, DRIVEN LINE CONDITION IS NOT DEFINED AND DRIVEN INPUTS WILL BE SUBJECTED TO NOISE PULSES.

from two 8-bit concatenated registers, with their subsequent multiplexing onto a bus. This method is often used to concatenate separate internal subregisters, within a processor, thus forming a larger entity for either internal or external reference. A convenient method, then, that we may use to control the reconfiguration of separate sources being fed onto a bus is through the addressing and enabling of the tri-

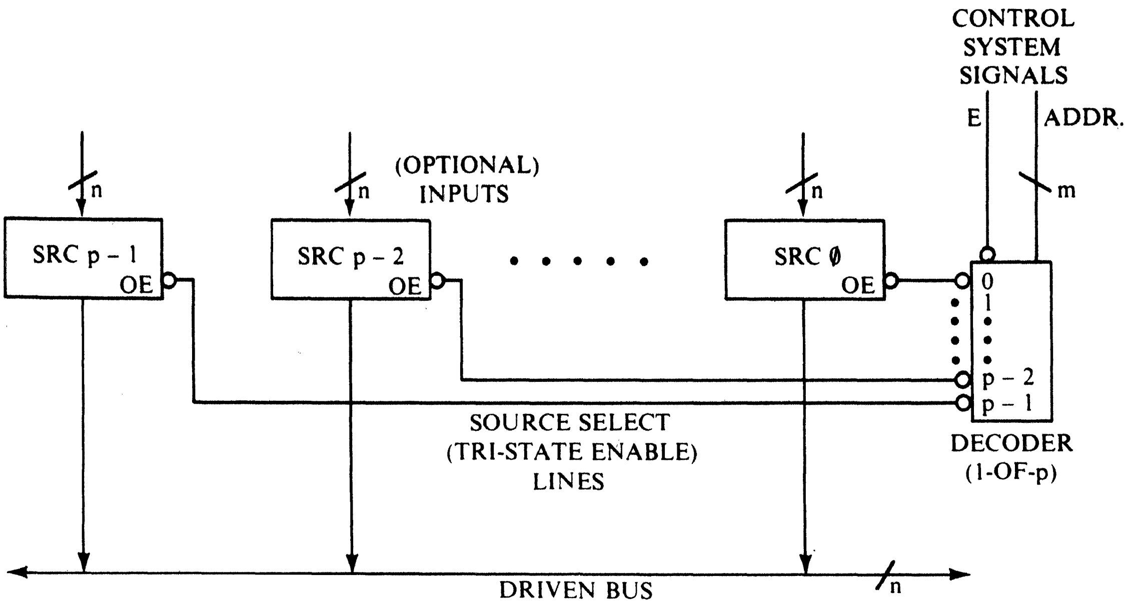


Figure 2-21
MPX via Tri-Stated Sources

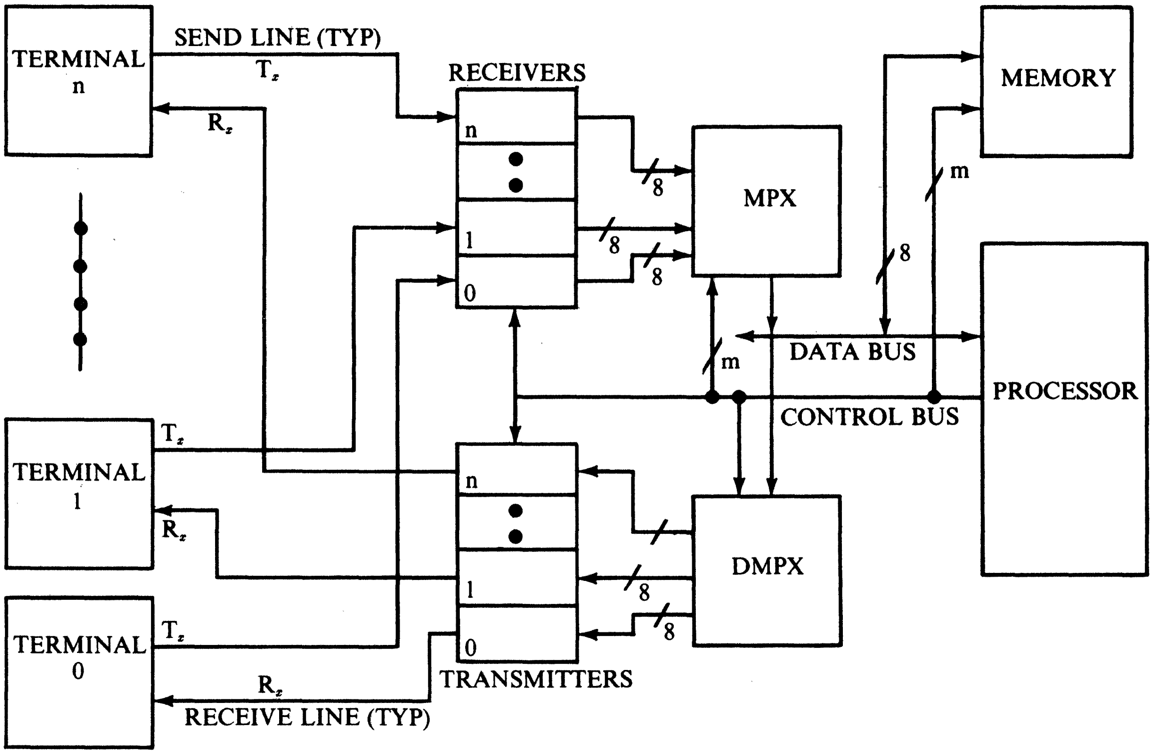


Figure 2-22
MPX and DMPX in Digital Data Transmission

state interface lines of the different sources, or through functionally equivalent methods. Therefore, we see that MPX and DMPX schemes can be quite flexible.

LARGE SYSTEM MPX AND DMPX EXAMPLE

An overall systems example of the use of various means of MPX and DMPX is illustrated in Figure 2-22, showing how frequently we may unknowingly interact with these concepts in familiar situations. This example is a simplified block structure of a system that interfaces a number of user terminals to a remote processor, a common occurrence in interactive time-shared systems. This figure displays the role of busing, MPX, and DMPX within a large-scale system block diagram. Each user terminal has separate transmit (Tx) and receive (Rx) lines. Information is placed on these lines one bit at a time by the user's terminal; this is serial data communication. At the other end of the system, each of these lines is interfaced to receivers/transmitters, one for each terminal. The widely used programmable Universal Asynchronous Receive/Transmit (UART) communications IC module can provide these functions.

This figure displays varied forms of systems uses of MPX and DMPX. The UART receivers are basically serial-to-parallel data converters. The transmitters are parallel-to-serial data converters. Both are usually contained in one IC. The receiver portion, for example, takes incoming serial data from a terminal and shifts it into a parallel register. When a character is thus assembled, the processor is so notified over the control bus. The transmitter section performs the reverse procedure. At this stage, both MPX and DMPX come into play. The processor manages the task, using an 8-bit or larger data bus and an *m*-bit control bus that manages and coordinates the data transfers. Each receiver's assembled characters is MPXed onto the host computer's data bus, one at a time, using TDM. Ultimately, the character from a terminal winds up residing in its appropriate place in memory. Outgoing characters are DMPXed into the appropriate transmitter for each user, over the data bus.

Thus we can see how an understanding of the functioning of common IC's–combined with the concepts of busing–begins to produce an overall picture of what transpires when we sit down at our terminal to use a remote computer. Busing, along with its related selection and distribution techniques, certainly is a large part of any processing system. To really appreciate the behavior of programmable UARTs and related communications devices, the reader should obtain and read a data sheet or application note from a vendor or manufacturer.

The principles for MPXing and DMPXing that we have just reviewed are applied within a processor in many places. While we may not know, in a particular case, exactly which method was used, it is helpful to appreciate in a general way what is being done within an architecture to move information about. Our major goal is to penetrate the mystique of processors and remove the fear factor. These topics resurface again soon, as we start to look at the internal systems structure of the CPU we shall construct and microprogram. Extensive use of MPX and DMPX will be made in this project.

Memory Cells

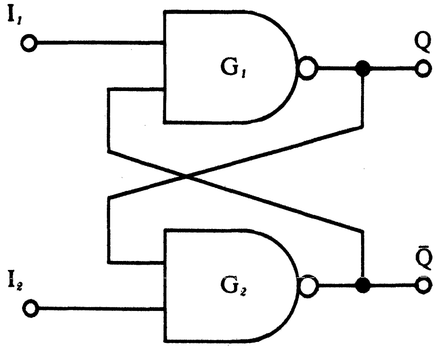
When we study the processor example of this text and its microprogramming, we encounter several types of memory. Included among these are flip-flops, registers, semiconductor memory, and read-only memory. To work comfortably with all these memory devices, we should have an understanding of their fundamental principles of operation. Let us start with the inverting logic gate's role in the formation of a "static" memory cell. *Static* memory cells are characterized by their ability to retain information as long as the power is applied. The inverting logic gate should be viewed as a basic building block of digital systems architecture. Many larger features, such as static memory cells and registers, are formed out of these gates. The memory cell of this type we examine here is formed by cross-coupling two inverting gates (both NAND or both NOR). This type of cell, called a *latch,* is illustrated in Figure 2-23a.

Static Memory Fundamentals

The logical properties of this latch cell are discussed in Chapter 4. Here we shall review the basic mechanisms of its operation. It is widely used as the fundamental part of flip-flops and registers. If not modified, the bare cross-coupled NAND-gate latch has some shortcomings for general usage. These design modifications, which are beyond the scope of this book, lead to the very common JK or D types of edge-triggered flip-flops. Cross-coupled gates, or their functional equivalent, are always found at the output end of these memory cells. They provide the feedback that gives rise to the memory-retention properties of these cells while under power. It is therefore instructive to survey their fundamental operating principles.

As the latch characteristics table in Figure 2-23b shows, latch-output behavior for input state I_1, I_2 = 00 is undefined. This is because, in general, we cannot say in advance which of the two NAND gates is faster and therefore controls the output behavior when both inputs "simultaneously" go from low to high. For this type of input transition, the next state of the outputs is indeterminate. The root of the problem is that, when both inputs are low, both outputs are simultaneously high. To be useful in many applications, this memory cell must *always* have both its outputs in opposite, mutually exclusive logic states. In fact, the difference between a latch and a JK flip-flop is that the latter always meets this criterion through modification of the former. Still, the front end remains the basic latch, as in all static memory cells. One way to force the latch to meet the criterion of mutually exclusive outputs under all input conditions is to resort to pulse-mode operation. A short review of pulsed operation provides insight into why propagation delay is the underlying fundamental form of memory for this class of devices. This is explained below.

In the NAND-based latch in the figure, the memory-retention state occurs when both inputs, I_1 and I_2, are high. Recall that a NAND gate is in reality a device whose output is high when any of its inputs is low. (Contrast this with the behavior of the so-called NOR gate.) Also, we recognize that an input change that affects the output state requires some time to propagate to the output. This time is referred to as the gate's *propagation delay* and is basic to the operating principles behind

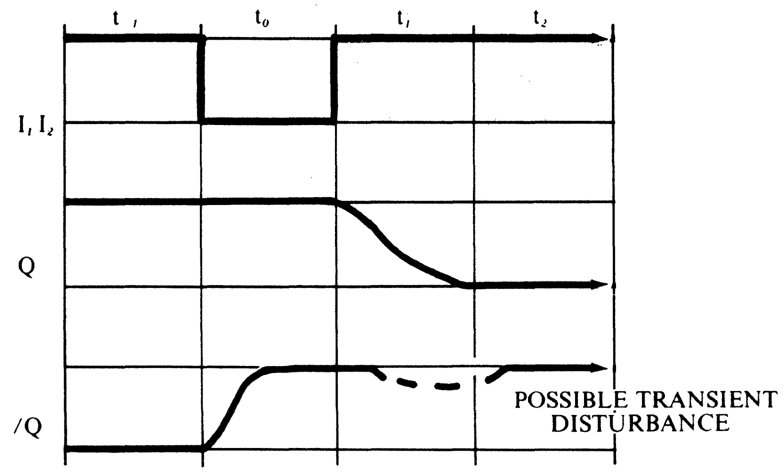


a. BASIC STATIC MEMORY CELL

INPUTS		OUTPUTS		
TIME t		TIME t + Δt		OUTPUT BEHAVIOR
I_1	I_2	Q	$\bar{Q}$	
0	0	1	1	NOT DEFINED
0	1	1	0	Q IS TRUE
1	0	0	1	Q IS FALSE
1	1	Q_t	$\bar{Q}_t$	MEMORY-RETENTION STATE

Note: Positive logic conventions used.

b. LATCH-CHARACTERISTICS TABLE
(A SIMPLIFIED TRUTH TABLE)



c. PULSE-MODE OPERATION TIMING DIAGRAM

Figure 2-23
Basic Static Memory Cell

static memory. The following analysis of pulsed-latch behavior starts with the assumption that both the inputs are tied together and have been high for a time greater than the propagation delay. (A timing diagram of pulsed operation is given in Figure 2-23c.) Therefore the outputs will have stabilized and become mutually exclusive.

This must occur when the inputs are both high, because a low at the output of, say, G_1 is fed back to the input of, say, G_2. This low creates a high output on G_2, which, when fed back to G_1, helps to maintain its low. The condition of having all inputs high forces G_1 low. The feedback paths maintain this static situation. For the NAND latch, the memory-retention state occurs when both inputs are high. Assume static conditions with both inputs and a high Q output. The /Q output is low (*/Q* is read "NOT-Q"):

time frame t0:

A low pulse is simultaneously applied. Both inputs are tied together and driven low for this purpose. After a propagation delay, /Q is forced to change from its initial low value, due to its current low-input condition (any low input forces a NAND gate high). The Q output remains high for now, since it is not yet affected by these changes. After the propagation time, /Q assumes the high state and is fed back to G_1.

time frame t1:

The common low pulse is removed. At this point, both external inputs are high, and a low starts to propagate through G_1. Notice now that both G_1's inputs temporarily remain high. That is, its external input is held high and the /Q feedback from G_2 is still high. Remember that a low now is being forced to propagate to the output of G_1. For pulse-mode operation, the input-pulse width is critical. It must be of sufficient duration (impart enough energy) to initiate a change of state but short enough to avoid indeterminate action.

time frame t2:

G_1 goes low. After its propagation delay, the output of G_1 is now low. The internally fed-back input to gate G_2, from the output of G_1, is now low. This forces G_2 to remain high. This high feeds back to G_1, which, combined with the now-high external input, keeps its output low. *The system is now stable, but the outputs have reversed their respective mutually exclusive states.* This is determinate action that is now being exhibited.

The above very approximate illustration of pulse-mode behavior points to one way the indeterminate input state of the latch may be converted to the toggle (complementing) mode of a JK type of flip-flop, where all input states lead to determinate behavior. Note that propagation delay has a fundamental role in the performance of a static memory cell. In essence, the propagation delay is the basic memory of the cell—it maintains a momentary history of past events that are essential to its correct operation. Another way to obtain completely determinate behavior of the edge-triggered JK flip-flop is to use the circuit's internal gates to form a level mode sequential circuit. Their aggregate external behavior is both sensitive to clock transitions (edge-sensitive) and determinate. Remember that the output stage of all flip-flops and static memory cells consists of the basic latch just discussed and that propagation delay plays its essential role in all of them. The types of static memory cells we are interested in are clock-driven. We shall discuss the characteristics and terminology of clock signals later. These same methods of forming static memory cells are also used with MOS and other technologies.

Dynamic Memory Fundamentals

MOS logic, though, has properties that also make it feasible for the formation of "dynamic" memory cells. These properties are extremely high gate-to-channel and off-state resistances and, most important, gate-to-source parasitic capacitance. The last, which in fact is normally a nuisance factor, is put to advantageous use here. Let us see what all of this means. The organization of one type of basic dynamic memory cell is given in Figure 2-24. This particular type of cell was at the heart of the 1103-type of 1024-bit dynamic random access memory (RAM) that appeared on the market in 1970. Its success led to the present-day large-scale use of dynamic RAM IC's, in which 256K bits of storage in a single IC are common. Even larger storage capacities are under development at this writing. The trend is such that semiconductor memory systems now predominate as the main (random access) mem-

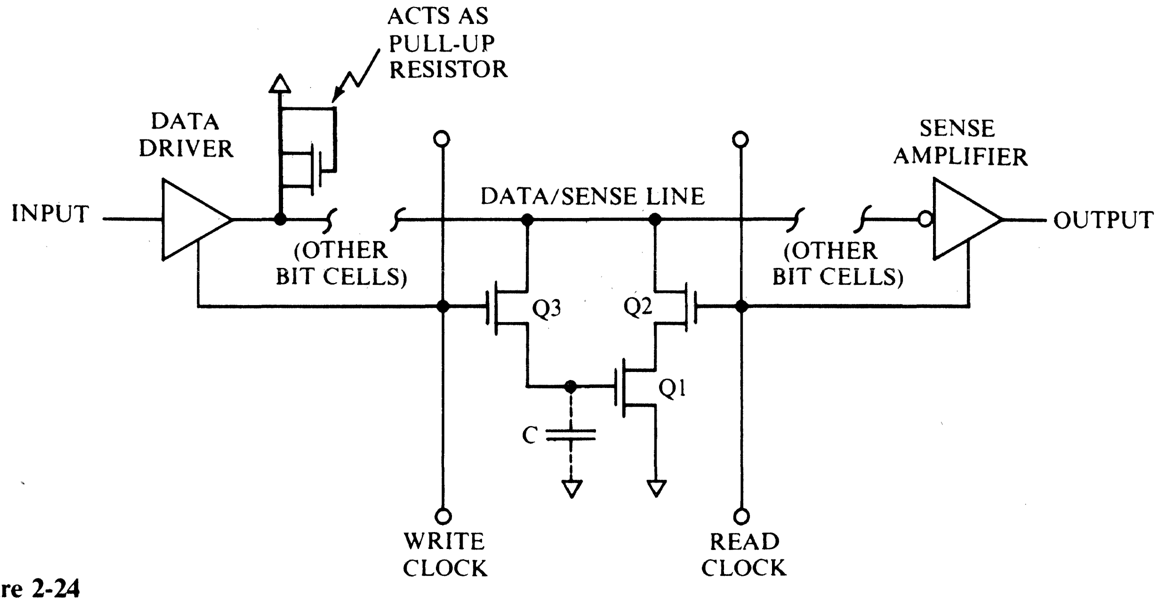


Figure 2-24
Three-Transistor MOS Dynamic Memory Cell

ory for most computers. The storage principles illustrated here probably apply to your personal computer.

In Figure 2-24, Q3 is utilized as a transmission gate. This means that, when the voltage on the *write* line enhances the channel of Q3, a current flows through the channel in the direction dictated by which side has the higher potential. Recall that the MOS transistor–when used as a transmission gate–has bilateral current-flow properties. The data driver is shown as being, in effect, enabled by the write clock signal, too. If the driven *data/sense* line has a higher potential than the storage capacitor, then the capacitor receives charge via Q3; if the potential is lower, then the capacitor loses charge. Actually, the capacitor consists of the purposefully increased gate-to-substrate parasitic capacitance, which "remembers" the last transmitted information in the form of an electrical charge. Once it is charged or discharged, Q3 attempts to isolate this charge when it is turned off. Nothing, such as practical matters, can ruin a good scheme.

While the gate of Q1 has extremely high input impedance, the rest of the circuit is less perfect. Thus the state of the charge in C is subject to slow leakage between itself, Q3, and the substrate. While the state of a fresh charge can be remembered through millions of computer clock periods, "amnesia" eventually sets in. The net result is that the state of the charge on C must be refreshed to prevent the loss of stored information. This is done on the order of every two milliseconds. This explains why refresh is *required* in a dynamic memory system. The refresh consists of restoring the state of the charge on C. Although refresh techniques are beyond our present scope, we will remark that refresh is usually automatically accomplished during use whenever a given number of address lines are all addressed in a given time period. In this type of memory, then, the physical basis for information retention is the amount of charge temporarily stored in a parasitic capacitor.

In this dynamic memory system, reads and writes occupy disjoint time slots. When it is desired to *read* the information stored in C, Q3

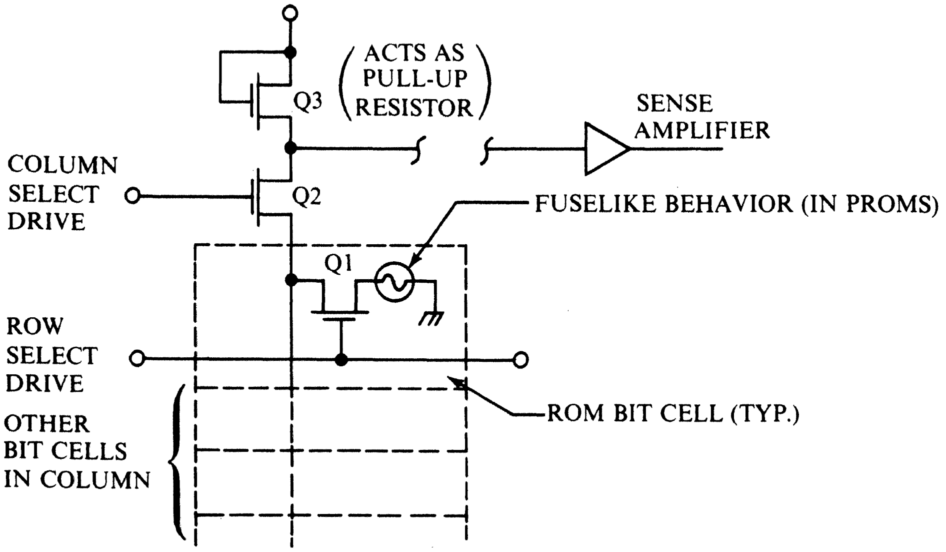


Figure 2-25
MOS Read-Only Memory Cell Structure

will be off (the write clock line is in the inactive state), and the input data driver is disabled. Under these conditions, the *data/sense* line is pulled high through an internal resistive path and is also connected to the input of a sense-amplifier. At the moment a read operation is in effect, the *read clock* line enhances the channel of Q2, so that it acts as a closed switch. The output stage of the sense amplifer is also enabled by this line, where the memory system has tri-state outputs. The pull-up high voltage of the *sense* line is now subject to the state of Q1. If there is a charge in C, Q1's channel is enhanced. It therefore becomes conductive, and the *sense* line is pulled low because Q1 is behaving as a closed switch. If C has no appreciable charge, Q1 is off and the sense amplifier input senses the high created by the MOS transistor equivalent of a pull-up resistor. The behavior of Q1 inverts the sense of the information–a small matter taken care of by another inverter, like an inverting sense amplifier.

There are many other types of MOS dynamic memory cells, even single-transistor ones. This cell, though, illustrates the main features common to most: MOS transistor transmission gate and switch behavior, purposefully used parasitic capacitance, the need to refresh, and the use of the MOS transistor as an inverting buffer gate and as a pull-up resistor–previously discussed fundamentals are all at work here. The memory cells described thus far are also called *volatile.* That is, information is maintained only while power is on. Turn off the power, and all information is lost–be the cell static or dynamic.

Nonvolatile (ROM) Fundamentals

Certain applications require permanent, nonvolatile storage. Nonvolatile cells retain their information regardless of whether the power supply is on. The control system of a processor is a case in point. It is convenient to have the system defined the instant it comes up, without the bother of loading its control memory. Permanent application programs are another case in point. We do not expect to reprogram a processor-controlled vending machine to obtain a cup of coffee after a power interrupt.

Nonvolatility can be achieved through the use of read only memory (ROM) cells, such as the MOS version shown in Figure 2-25. If a memory system contains 256 words of eight bits each, we can imagine the system as consisting of eight of the columns illustrated. The number of memory words would then be equal to the number of ROM cells in each column (only one is illustrated by the dashed box). Where a high level is to be permanently stored in a particular cell, the source-to-substrate connection of Q1 is deleted in the production mask or by burning out its fuse equivalent, shown in Figure 2-25. Thus, when this column and row are selected, the cell is not pulling the column low. The result is a high maintained at the sense-amplifier input when this row is selected. Had the cell remained intact, the row select's going high would turn on the transmission gate, which in turn would pull the sense line low (that is, ground it).

A popular variation is to make the ROM user-programmable. This is called a Programmable Read Only Memory, or PROM. In this case, a fusible link is inserted into the ROM cell in line with transistor Q1. This device is user-programmed by "burning" or "blowing" the fuse. When it is being programmed, a momentary very high voltage applied to any column destroys the fuse of the cell selected by its transmission gate. Yet another variation in MOS static memory cell types is the *Electrically Programmable Read Only Memory,* or EPROM. These are widely used in the experimental stages of developing software that is to be permanently resident in a system. The most widely used of these are erased by ultraviolet light and then electrically programmed by the user. Since this can be done many times, they are very convenient for development work. The microprograms we shall create for the control sytem of our example processor will be stored in EPROM's.

Thus, some of the concepts behind nonvolatile MOS memory are illustrated. ROM memory is used extensively in processor control and operating systems, as we shall see, because we want it to be ready when the power comes up. When used for the permanent storage of operating system, control system, and other applications software, these permanently enshrined types of software in a ROM are called *firmware.* The last variation in nonvolatility deserving of mention now is *bubble memory.* Bubble memories are *serially* accessed read/write memory systems, which also retain their contents when the power is off. This, though, is transient—though sometimes long-term—storage. Since users have access to it, this memory normally does not remain the same for the life of the system, as is expected of ROM.

We have only touched the broad subject of memory systems to create an intuitive feel for a few of the most widely used underlying principles. An overall view of the *world of memory* (Figure 2-26) will show how these techniques fit into a tree-structured perspective of memory systems. In general, we can expect to find incorporated—even within a *single* processing system—many of the types of memory technologies discussed. At the root of the tree structure is memory per se. At the next level down, the permanence of memory retention may depend on the applied power. This separates memory into two broad categories: *volatile* and *nonvolatile.* As noted, volatile memory systems lose their contents when power is removed. One more level down, *directionality* is shown as an attribute of memory systems. Some memories transfer information both into and out of their cells. Others are of the type that may only be read. The next level below this deals with

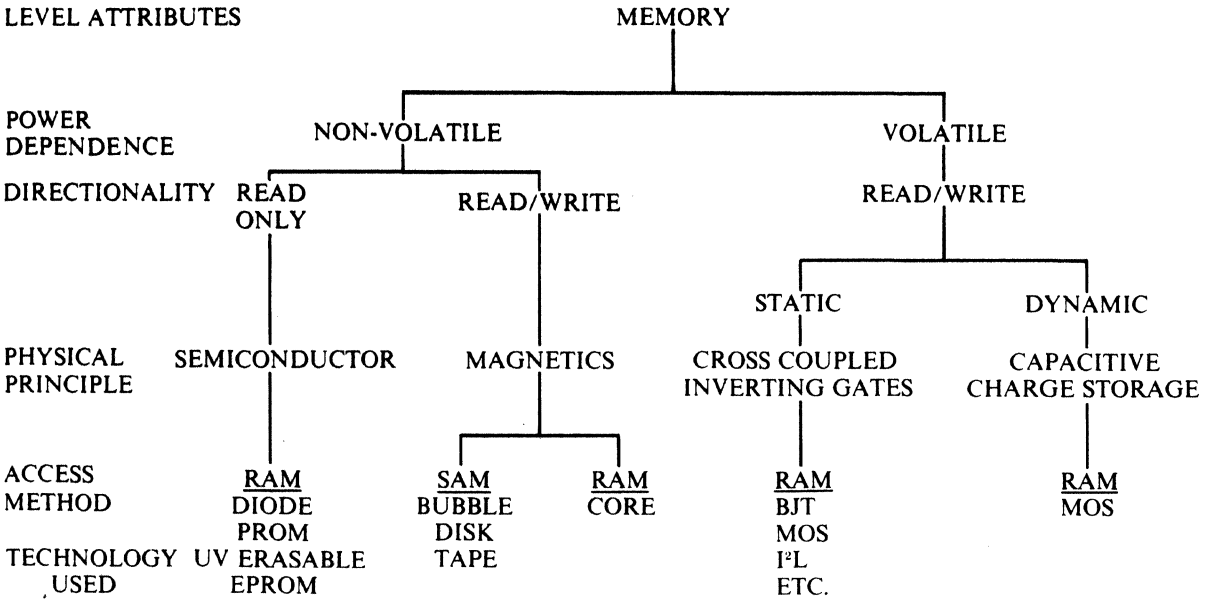


Figure 2-26
The World of Memory: Tree Diagram

access methods. Access can be to any location we choose at random (RAM), or it can be constrained to a serial march through all locations to get to the desired location. This is *Serial Access Memory* (SAM). Finally, typical *fabrication technologies* are indicated. As we work with an actual example of a processor, we shall gain first-hand familiarity with several of these types of memory.

BIBLIOGRAPHY

Anderson, G.A., and Jensen, E.D. "Computer Interconnection Structures: Taxonomy Characteristics and Examples." *ACM Computing Surveys,* Vol. 7, No. 4, December 1975, pp. 197–214.

Bowen, B.A., and Buhr, R.J.A. *The Logical Design of Multiple Microprocessor Systems.* Englewood Cliffs, New Jersey: Prentice-Hall, 1980.

Boylestad, R., and Nashelsky, L. *Electronic Devices and Circuit Theory.* Englewood Cliffs, New Jersey: Prentice-Hall, 1982.

80/85 Family User's Manual. Santa Clara, California: Intel Corporation, 1983.

Fletcher, W.I. *An Engineering Approach to Digital Design.* Englewood Cliffs, New Jersey: Prentice-Hall, 1980.

Siewiorek, D.P., Bell, C.G., and Newell, A. *Computer Structures, Principles and Examples.* New York: McGraw-Hill, 1982.

TTL Data Book, The. Dallas, Texas: Texas Instruments, Inc., 1976.

TTL Data Manual. Sunnyvale, California: Signetics Corporation, 1984.

VAX Architecture Handbook. Manard, Massachusetts: Digital Equipment Corporation, 1983.

PROBLEMS

1. a. Explain why the multiplexing of sources onto a bus is useful common practice.

   b. Explain the functions of the load-enable and clock signals in controlling the transfer of information to one or more sinks connected to a bus system.

   c. Describe two methods of multiplexing.

2. Using simple analogies, explain the essential characteristics and functions of the following:

   a. Diode

   b. Transistor

   c. Multiple-emitter TTL input

   d. Totem-pole output structure

   e. Tri-state operation

3. Explain why Q3 of the logic gate of Figure 2-4 is off when any of the inputs (A, B) is tied low.

4. Convert the logic table of Figure 2-5 into a truth table, using 0s and 1s, for:

   a. Active-high logic convention

   b. Active-low logic convention

5. Derive the logic table, in terms of L and H, for the two-input logic gate of Figure 2-7a. Via a truth table, verify the logical functions that this structure performs for both active-high and active-low logic conventions.

6. Sketch a cross-sectional view of a MOS p-channel transistor. Refer to Figure 2-6 for guidance. Using the MOS transistor symbol for this construct, show the biasing that results in channel enhancement.

7. Sketch the switch-equivalent circuit of the totem-pole output structure used in logic gates.

   a. What is the function of the resistor?

   b. When the totem pole is used in a simple NOR or NAND gate, how do the switches behave?

   c. Describe why the totem pole is capable of isolating its output pin from the data-input behavior in a tri-state gate.

8. Draw the totem-pole-switch equivalent of two tri-state outputs connected to the same line of a bus. Explain, with respect to the drawing, the following:

   a. How the ENable inputs can be used to enforce the rule that only one source is to be master of the bus at a time.

   b. How the third state of electrical isolation, the hi-Z state, is achieved at the output.

9. Draw the logic diagram of the 74LS138 decoder/demultiplexer as it would appear when used as a data demultiplexer.

   a. Explain how the selected output-logic level can be made to carry either the input data or its complement. Hint: Consider how best to utilize the enable inputs for this purpose.
   b. What is the quiescent electrical level of an unselected output pin?

10. a. Define *fan-out.*

    b. Using the above definition, calculate the number of low-power Schottky TTL gates that can be driven by a standard TTL gate.

11. a. Locate an 8080 microprocessor data sheet or catalog. Compare the current drive capacity of the 8080 microprocessor with that of the 74LS240 tri-state buffer. Calculate the drive-current multiplication factor that can be achieved when the buffer is used as a front-end interface to the processor.

    b. Typically, where would one expect to find the application of unidirectional rather than bidirectional tri-state buffers at the interface to a microprocessor? Under what conditions and for which types of signals are these buffers most applicable?

12. Discuss the differences between common and separate I/O interfaces, using bidirectional tri-state buffers.
13. Draw the logic diagrams of the following circuits, using NAND gates:

    a. A four-line into a one-line multiplexer.
    b. A one-line into a four-line demultiplexer.

14. Refer to Figure 2-19 for guidance.

    a. Draw a diagram of four sources, each containing four bits of information, connected to a four-bit bus via combinational-logic multiplexers. Explain the system's operation.
    b. Select components from the TTL data catalog you would use to build the circuit. Draw a logic diagram of the actual circuit using these components.

15. Refer to Figure 2-20. Draw two open collector gates, each with one data input, interfaced to a bus line with a pull-up resistor. Derive the electrical-level truth table for the bus line as a function of the two inputs. What logic functions are performed for active-low and active-high conventions?
16. If NOR gates are used to form a basic memory cell or latch:

    a. Draw the characteristic table for this latch.
    b. What input conditions produce undefined output behavior?
    c. What input conditions cause the latch to "remember" its last state?

17. For the dynamic memory cell of Figure 2-24:

    a. Which transistor serves as a bilateral transmission gate?
    b. Describe the physical mechanism for information storage in this cell.
    c. Why is it necessary to refresh this cell periodically?

18. Why are ROM cells useful in initiating a computer's operation on power-up?
19. Find three types of memory not now shown in Figure 2-26 and place them in their proper place on this chart.

CHAPTER 3
ASSEMBLING AN ARCHITECTURE

REGISTERS, ARRAYS, AND STACKS

From the inverting logic gate to the memory cell of Chapter 2, we shall proceed to their use in building the next-higher orders of processor organization—registers, arrays, and stacks. Registers are simply a coordinated group of some type of basic memory cells. They are coordinated by signals emitted by the control system; we shall gradually become acquainted with the terminology of this system. For example, when a computer receives an instruction, it is placed—in parallel—into a grouping of memory cells called the Instruction Register (IR). Typically, this consists of 8, 16, or 32 bits of memory, depending on the size of the processor, each bit of information residing in one cell. An IR's input comes from the selected instruction in external memory and goes to its parallel inputs over the data bus clocked in during the Instruction Fetch phase of operation. The parallel outputs of the IR inform the control system of the current instruction to be performed, during the EXecute phase of operation. The arrival of information, the simultaneous loading or clocking of these memory cells, and other activity are all controlled by signals from the control system.

In some microprocessors, these cells of a register are even refreshable dynamic cells, thus saving on the amount of silicon "real estate" needed to implement this feature. In any case, the user treats this structure as a logical entity having the storage properties associated with the term *register*—which is, generally, a parallel array of coordinated memory cells. A CPU contains a number of these discrete registers, some of which hold information only temporarily during a cycle of operation. These important "temporary" registers are often transparent to the programmer, and their presence is often unsuspected. They are critical, however, to the functioning of an architecture. We start to illustrate their use and placement in this chapter.

In addition to parallel-loading registers, there are other varieties, such as shift registers and counters. We shall encounter all these types in more detail later,too. The basic registers will be incorporated into larger constructs called *arrays* and *stacks*. Once we add the Arithmetic Logic Unit (ALU) to this assemblage of processor features, we will be able to discuss some basic architectural arrangements for CPU's. As we shall note later, the architectural features of a CPU most prominently displayed are its buses, registers, arrays, and ALU.

REGISTER FEATURES

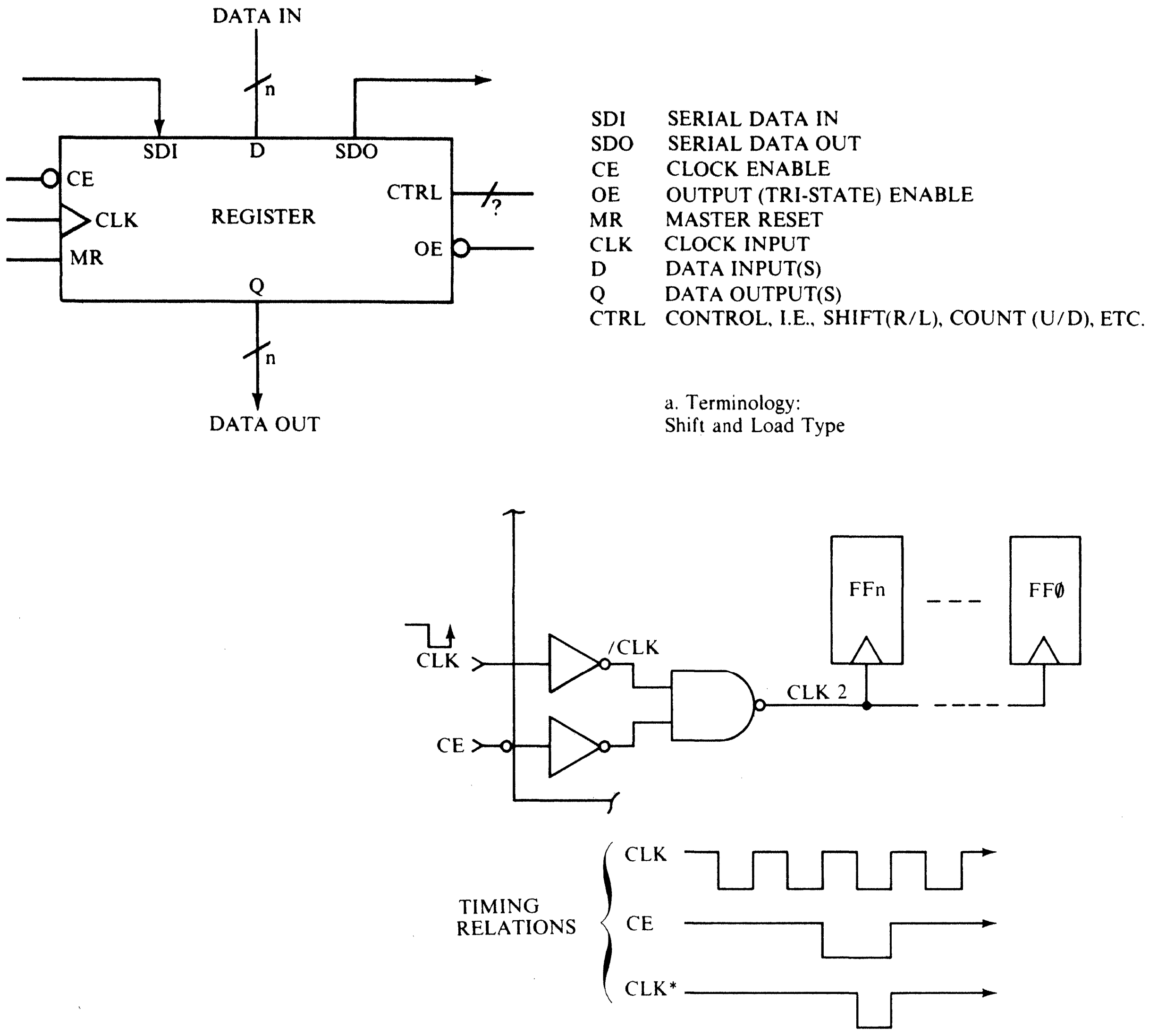
The terminology, including control signals, associated with general shift and load registers is presented in Figure 3-1a. This figure displays the common data and control-signal names for many other registers as well. Naturally, the names of control-system signals are similar. Shift registers contain internal multiplexers used to select the proper bits of data for the shift-right and shift-left operations. That is not apparent from this form of logic diagram, but, since we previously discussed the logic of data selection, there should no mystery about the basic mechanisms involved. Counter registers are generally of the parallel-loading, parallel-output type, whose memory cells are based around the combinational logic associated with a counter. They may be quite flexible in that they can count up or down and emit a control signal when some Terminal Count (TC) is reached. The term *TC* often appears in operations affecting the programming of timers. One example of a counter register from the TTL catalogs is the 74LS161 4-bit binary counter. Thus we see that the memory cells of a register are often provided with a considerable amount of associated combinational logic, consisting of data selection, enabling, and other control functions, such as that presented in Chapter 2.

The Clock Enable (CE) and Output Enable (OE) controls shown in Figure 3-1 are of special interest to us. CE affects the loading of data at inputs of a register, and OE affects the tri-state enabling of stored data onto a bus at the outputs. An understanding of the coordinated use of the CLK and CE signals is important to grasp, since similar system signals, whose behavior we wish to understand, control the selective loading of a processor's registers. The output enabling of a register is frequently done via the tri-state techniques explained in Chapter 2. The simple but important type of circuitry employed in clock enabling is shown in Figure 3-1b. A control system utilizes this type of logic to control when a clock pulse is transmitted to the cells of a register. Let us examine its operation.

The clock and its enabling inputs are fed into separate inverting buffers. The buffer outputs are then fed into a NAND gate. The output of this NAND gate is distributed to each and every flip-flop's clock input of the cells of the register. As the timing diagram indicates, each cell of a register "sees" a clock pulse *only* when CE is active (low, in this case). The external system clock, CLK, runs continuously, but the individual cells of a specific register receive the signal CLK2 *only* when enabled by the control system to store information. The control system maintains the proper phasing of the CE signal with the system clock.

ARRAY CHARACTERISTICS

Registers are a major feature in the structure of processor systems. Actually, there is no conceptual difference between an on-board register and an external memory location. Both serve the purposes associated with data movement and storage. In most architectures, it is desirable to maintain a coordinated group of registers within the CPU, both to facilitate quick access and to simplify construction. One convenient means of achieving this is to arrange the registers into a highly organized structure called an *array,* as shown in Figure 3-2. In an integrated circuit, the array structure is regular and therefore easy to produce. Moreover, it reduces complexity by sharing the bus-interfacing circuitry insofar as possible. Registers are selected in an array through the use of decoders, and communication is accomplished with multiplexers and demultiplexers over internally supplied buses. MPX and DMPX can be implemented through combinational logic, tri-state interfaces, or open collector interfaces—as discussed previously.



SDI	SERIAL DATA IN
SDO	SERIAL DATA OUT
CE	CLOCK ENABLE
OE	OUTPUT (TRI-STATE) ENABLE
MR	MASTER RESET
CLK	CLOCK INPUT
D	DATA INPUT(S)
Q	DATA OUTPUT(S)
CTRL	CONTROL, I.E., SHIFT(R/L), COUNT (U/D), ETC.

a. Terminology:
Shift and Load Type

b. Clock-Enabling Logic and Timing

Figure 3-1
Register Features

The decoders—one for writes, the other for reads—select the registers to be read or written into. The decoders are in fact an integral part of the MPX and DMPX logic. With separate decoding, the simultaneous reading of one register while writing into another is common. Apparently "simultaneous" reads and writes of the same register of the transparent type of latch is more complex. The transparent latch is characterized by its ability to also serve as a buffer gate would. Since they are of simple construction, they are widely used in IC arrays. This important special situation is dealt with in detail, both in our discussion of the clock signal later in this chapter and in Chapter 5. We frequently find transparent latch registers used in IC designs. The edge-triggered memory cells, such as the JK flip-flop, can always undergo true simultaneous reads and writes of the same cell. Signals from the control system manage the action of loading or outputting the contents of a register within an array, coordinated by the system clock. These same signals are often called WRite (WR) and ReaD (RD) pulses and are functionally equivalent to the foregoing CE and OE signals. As

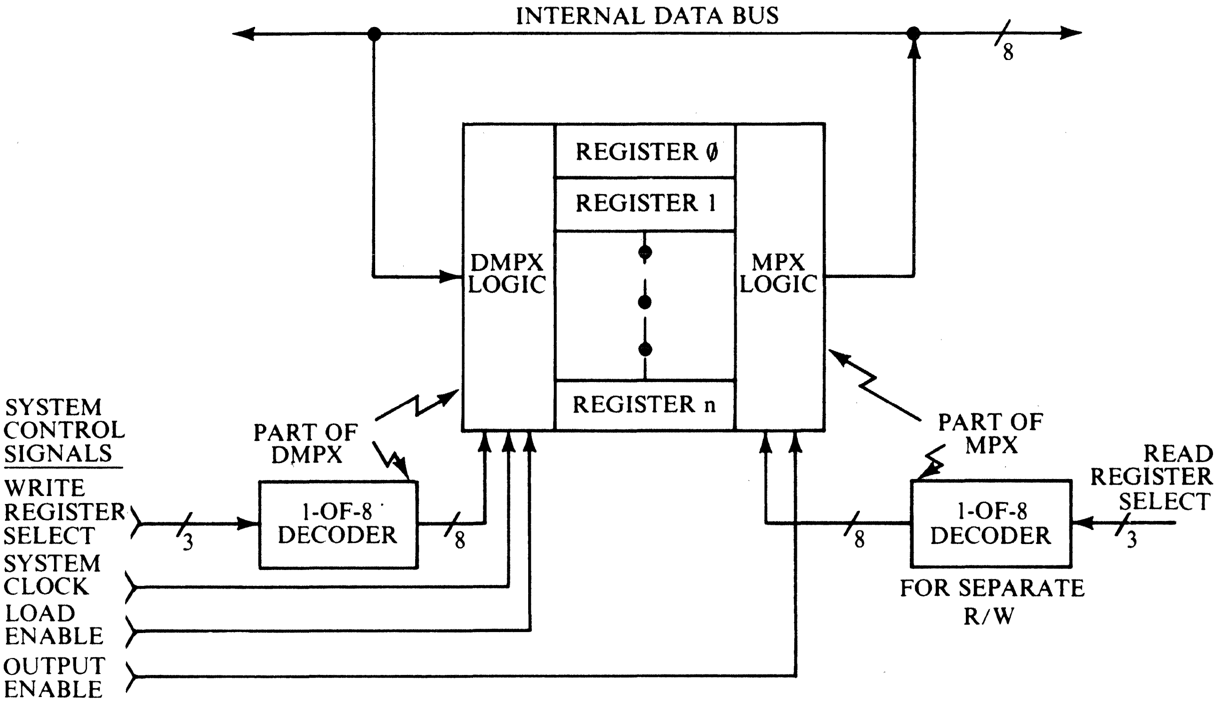


Figure 3-2
Register-Array Organization

shown in Figure 3-2, the register-selection inputs come from the control system. If separate registers are to undergo a simultaneous read and write operation, then the second decoder requires its own separate set of register-select signals from the control system. In this manner, we can move data from one register into another within an array (or from one register into itself), over internal data buses, by output enabling one while input enabling another (or the same one). This is the usual form of the register-to-register MOVE operation.

A register array structured along the general principles above has many uses. One often finds register arrays organized into separate groups or banks. That is, the processor can contain more than one array. For example, in a technique called *bank switching*, the control system uses only one of several register arrays available to it at a given time. Bank switching between arrays is an extremely fast way of saving the status of one process while rapidly turning to serve another. This saves the time otherwise required to tuck the current states of registers safely away into external memory when attempting to respond rapidly to, say, an interrupt request. The PDP 11/70 minicomputer employs bank switching to implement its executive mode. Bank switching among register arrays may be accomplished under either program or hardware control, as when responding to interrupts. The unused bank simply holds its state constant until control returns to it. In effect, a bank is just a segmented register array.

STACK CHARACTERISTICS

There are other ways register arrays are put to work besides banks, such as general accumulator registers, memory pointers, and stacks. Let us first examine *stacks*. There are two types of stacks, both operating on the same set of principles. The difference is that one is a distinct single entity, organized around a register array, often called an *on-*

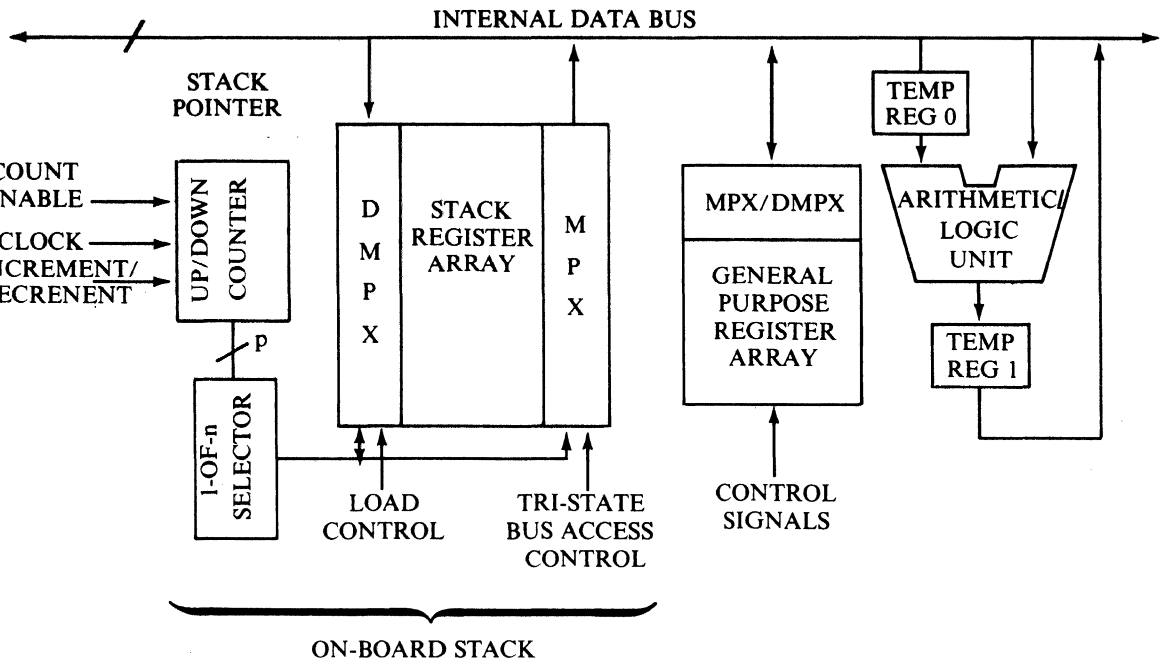


Figure 3-3 Microprocessor On-Board Stack Features: Block Diagram

board (or *hardware*) stack. The other form of stack is actually the same, but *distributed* throughout a computer's total system. Both are widely used. Those who program in high-level languages are more likely to encounter the distributed form of the stack; microprogrammers often encounter the on-board type. To promote a larger systems perspective, the structural features of the on-board stack are portrayed in Figure 3-3, along with some other central-processor features. In general, stacks are used to store both addresses and data. The on-board variety, however, lends itself to the address-stack type of application, important in the handling of subroutine calls and interrupts. In these applications, the stack is an *array of program counters,* only one register of which may serve as the current PC at any moment. The size of the array (the number of registers) is usually very limited. Early microprocessors, such as the Intel 8008 and the Signetics 2650, contained an on-board stack of eight registers in the CPU. They were implemented so as to facilitate the handling of subroutine calls, interrupts, and the returns from these.

Notes on Calls, Interrupts, and Returns

These events are of such conceptual importance to us that it is helpful to explain at this point what physically occurs when they are invoked. Later, we shall have to microprogram calls and returns. Starting with the basics, both subroutines and Interrupt Service Routines (ISR) are bodies of code likely to be used many times, either by the operating system or by the programs they support. As an example, take the code that defines how the operating system inputs a character each time a key is struck. Let us call it CIN, for Character INput. This is an example of a body of code that could be invoked by an interrupt-driven mechanism or by a procedure call. A viable operating system should support only one copy of each such system code, for all users. A programmer, on the other hand, has no particular desire to write identical bodies of code repeatedly every time they are needed. Their repeated use may be invoked through the medium of the subroutine call. Therefore, we need to look at the *common related mechanisms* behind *calls* and *interrupts*—particularly with regard to how the hardware supports a safe return to the program in operation before interrupts or calls occur.

The program flow in a call or interrupt is graphically shown in Figure 3-4. Here, the main program proceeds along on its path (the upper level) until the occurrence of an interrupt or subroutine call, such as the line of assembly language code:

```
CALL CIN
```

The *call* machine instruction evokes a branch to a new routine. However, that is not all the hardware does: the machine must "remember" the value of the program counter in the present calling program's instruction stream. If this were not done, it would be impossible to return to the calling program later. CIN is the symbolic name of the address where the called subroutine starts. When jumping to this new routine, we must save the current value of the PC.

Note that the on-board stack of Figure 3-3 now has a new feature added to its register array, an up/down counter, called the *stack pointer.* It drives a 1-of-*n* decoder that selects the register that is to serve as the current PC. Processors that have an external stack operate similarly to the following explanation of the embedded type. When the call instruction is encountered or an interrupt occurs, the processor must *always* save at least the current PC before branching to the routine CIN. The current PC, as discussed in Chapter 1, contains the address of the *next* instruction during the execution phase, at the current program level. More sophisticated processors authomatically save more of the current state of the machine, such as the system flags, the contents of the other registers, or both. In the case of interrupts, acknowledgment of further interrupts is usually disabled automatically until changed by an instruction. Where it is necessary to save the contents of the flag and other registers, too, this must be programmed into the start of each ISR or subroutine.

In this on-board example, the contents of the current PC are saved simply by leaving the PC alone. Instead of affecting the PC, the stack pointer counter is decremented by one, and the actual value of CIN (the new address of the first instruction in the new routine) is deposited in this freshly designated PC register, which is one level down from the old and now inactive PC. The next instruction is fetched, using the new PC with its just-loaded address. Since the old PC is one level away in the stack, we say that we have "pushed" it onto the stack. In reality, we have merely counted down the stack pointer and reloaded the new PC register, so that it now points to the address at which we wish to start fetching instructions. The essential difference between interrupts and subroutines is that interrupts are often initiated by a hardware-generated signal and subroutine calls are initiated by a line of program code. Also, the interrupt hardware must supply the means of determining the starting address of the interrupt service routine. This can be accomplished either by arranging for the system to "vector" to a fixed predetermined location or by requiring the interrupting device to place an address onto the data bus to be used in reloading the PC.

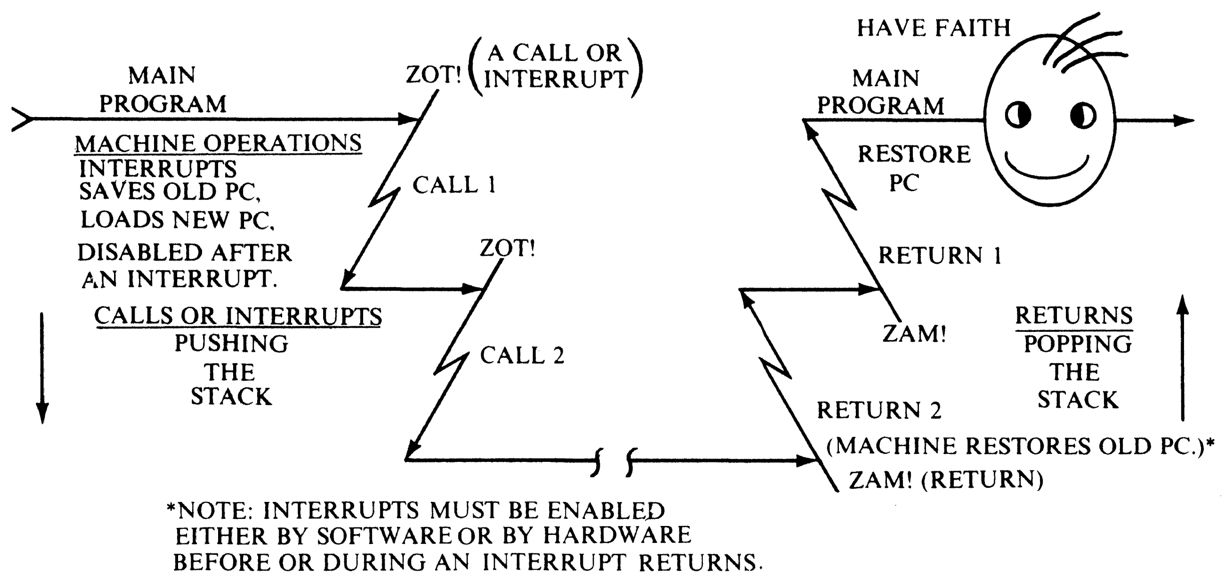


Figure 3-4
Program Levels and Hardware Actions for Calls, Interrupts, and Returns

With the reloading of the PC, the processor completes the call instruction (or response to an interrupt) by entering the next Instruction Fetch (IF) cycle. All instructions at the current level are successively fetched by first using the contents of the present PC as a pointer to where the instruction resides in memory and then by incrementing the PC to "look ahead" for the address of the next instruction–which is normal PC behavior. In the course of executing a subroutine or ISR, it may become necessary to respond to yet another subroutine or interrupt. The stack pointer is decremented yet again, and the newly designated PC is again loaded with the new routine's starting address. Two old PC's are now saved on the stack. The processor again enters the IF phase of its operation for this new level. Obviously, this can go on until the counter goes through a complete cycle and starts overwriting old PC's to which a return has not yet been made. Small size is the very serious limitation of on-board program stacks. Even so, they are used in bit-sliced types of microprocessor architectures because, being on board, the stack is capable of very fast operation.

How does one return to a higher level after an interrupt or a call? The last instruction in a proper ISR or called subroutine is a *Return.* The return is always generated by a line of software code. Its execution evokes a physical transfer back to the next-higher level of the currently used set of program steps. The on-board stack simply increments the stack pointer (referred to as "popping" the stack) and then enters the IF state again. The incrementation of the stack pointer of the on-board stack reselects an old PC, the one left behind after the last call or interrupt. Since it was pointing to the next desired instruction before the call or interrupt, the previous level is rejoined at exactly the correct point. Returns are made until control is back to the original main program. At this point we too return to examine some more of the details of Figure 3-3.

ORGANIZING AN ARCHITECTURE

As a preliminary means of introducing larger system-organizational relationships among its structural blocks, Figure 3-3 contains more than just an on-board stack. It has two register arrays, one organized as a stack, the other serving as a bank of general registers. An ALU and associated temporary registers are also shown. The ALU will be examined in the next section. The temporary registers are "temporary" only in the sense that they are used to hold intermediate results for a short time. The reader may wonder how the current program counter of the stack can be incremented when only a parallel-loading register in the stack serves as the PC. In this organizational structure, which is far from optimum, the current PC register in the stack (selected by the counter-pointer) can be placed onto the data bus, passed through the ALU, where it is incremented, and stored in temporary register 1. The next clock cycle restores this incremented value to the current PC. This structure requires data and address words of the same size. In most cases, however, they are of different sizes. Better solutions are to equip the stack with its own incrementer/decrementer or to make the address word, say, twice as large as the data word. This last solution is not a good one since it requires that the PC be incremented in steps.

DISTRIBUTED STACKS

The severe size limitations of on-board stacks has led to the current widespread use of *distributed* types of stack organizations, along with the introduction of dual-sized register arrays, discussed below. The architectural features of the distributed stack are presented in Figure 3-5, which represents a hypothetical dual 8/16-bit microprocessor. In this case, one or more 16-bit stack pointer (SP) registers, as well as the single 16-bit PC, are kept on board with the CPU. The SP registers fulfill the role of the counter-pointer of the on-board stack just discussed. In this case, *both* the PC and SP now serve as counters and as pointers to memory locations. The PC points to the location from which the next instruction will be fetched. The SP points to the current Top of the Stack (TOS), where old PC's and data are to be saved. The computer's register array, which contains the stack pointer(s), is now a more complex structure, utilizing a number of subarrays. As shown here, a dual-bank 8-bit Register Pair (RP) array is combined and coordinated with a 16-bit register array and its associated busing system, all forming a single complex structure. Note the system's large number of bus paths and its great flexibility to move or alter information. All the previously discussed architectural features of bus organization, data selection, and distribution, as well as sink selection, are fully utilized here.

DUAL-REGISTER-SIZED PROCESSORS

Modern processors often employ dual 8/16 or dual 16/32-bit architectures. These provide the typical environment within which the distributed stack operates. Since they profoundly affect the manner in which stack operations are performed, they are illustrated now. The nature of

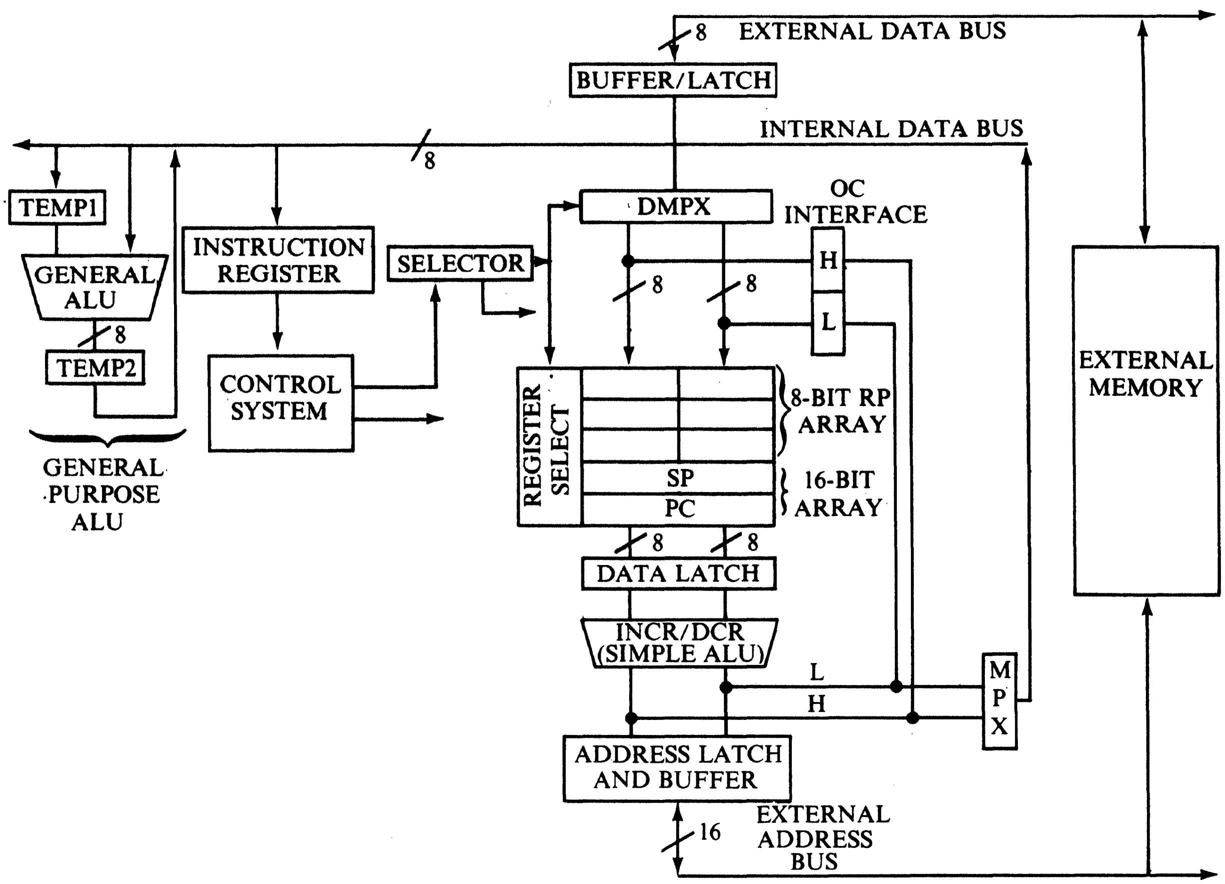


Figure 3-5
Internal Control Features of External Stack for a Hypothetical 8/16-Bit Microprocessor

this arrangement is also shown in Figure 3-5. The logical arrangement of the two 8-bit register arrays permits *pairs* of corresponding registers to behave as a single 16-bit register. This is accomplished by interfacing them to a single 16-bit ALU. This ALU's performance is usually limited to simply incrementing, decrementing, or just passing along, unchanged, the data presented to it. These few operations are far less than is expected of the often smaller general-purpose ALU of the system. Note that the 16-bit incrementer/decrementer ALU shown and the data paths available to it can conveniently handle 16-bit PC, SP, and RP simple arithmetic operations and information transmittal without resort to the system's smaller 8-bit general-purpose ALU. One of the registers of the 16-bit array is designated as the PC. One or maybe more registers of this array serve as the SP's.

The stack proper now resides in external memory, where it can be made as large as memory management allows. This space is controlled by operating-system considerations, but it is far larger than before. The severe size limitations of on-board stacks are now greatly improved upon. Further, more than one stack can be maintained. These can be used for data storage as well as for the type of system-stack usage associated with calls, interrupts, and returns. Let us differentiate between the use of the PC and the use of the SP. PC *alone* points to the instruction stream in memory; SP does not. SP points to *old* PC's, process-status information, and data that is saved on a stack for subsequent retrieval. Now the PC and SP registers serve as *both* counters and pointers to locations in memory. They can now be incremented/decremented (INCR/DCR) in one or two clock pulses for fast operation, as can the RP's. Note that the processor now also has dual-word-size operational capabilities.

These principles of operation may be configured in many different ways. Thus, we have introduced multiple-sized operations along with the array, as well as multiple ALU and stack constructs, in our architectural considerations. Note that these new structures, as illustrated here, permit the placement of any 8-bit register or half of a 16-bit one onto the 8-bit internal bus. The contents of 16-bit registers can now be latched and used as memory-address drivers. Displaying all the required open collector or tri-state interfaces would only clutter up the figures. The reader should intuitively recognize where they may be needed, by now, as well as the types of logic used for implementations. In Chapters 5 and 6, we will put these organizational concepts to work in the very real sense of creating the microcode that controls stack operation.

The INCR/DCR logic provided in the simple 16-bit ALU of Figure 3-5 can, as noted, only add 1 to, subtract 1 from, or transfer through unchanged its sourced data. This ALU has associated with it the indicated buses, data, and address latches that are transparent. Not only does this flexible group serve to increment the PC, SP's, and RP's, it is used to decrement SP's and RPs as well. The output of a register pair may be simply passed through the data latch or latched into it. From here it feeds the simple ALU. Alternatively, its output can be fed through the ALU to the address latch/buffer, where it is held to drive the address bus as long as this address is needed. Any half of a RP may be MPXed onto the internal data bus at a time, using the system's feed-through ability. The difference in data and address word sizes is accommodated by the fact that one is a multiple of the other. This means that addresses may be conveniently stored in memory in multiple steps (two, in this case) via the MPX logic. The data-flow paths support this. Further, any 8-bit data register (or half of a 16-bit one) can carry on exchanges with any single half of the registers in the 16-bit array. This type of architecture is common to many microprocessors. The register-select logic also controls the loading of any RP, or the selected half of one.

DISTRIBUTED STACK OPERATION

Let us make an introductory examination of the steps involved in the operation of such a total system structure. Storing the PC on the stack in external memory after a call or interrupt is the case in point. Referring back to Figure 3-5, this may be carried out as follows:

1. The stack pointer register is stored in the data latch.
2. The ALU decrements the stored image of SP. The decremented result is stored in both the SP and the address latch.
3. The program counter is transferred through the transparent data latch *and* the ALU. Program Counter Low (PCL) is MPXed onto the internal data bus and presented to external memory for storage on the stack at the location specified by SP. The image of SP's contents in the address latch now drives the address bus.
4. Step one is repeated.

5. Step two is repeated. Note that the initial value of SP has now been decremented twice.
6. The PC is again transferred through the data latch and ALU. Now, Program Counter High (PCH) is MPXed onto the internal data bus to external memory, where it is stored on the stack.

Depending on architectural variations, some of the above steps may be performed in parallel, so the process need not be as lengthy as it seems. The point is that, while we have discussed two types of stacks, not all stacks are the same. Obviously, there are some important choices to be made in selecting a system stack architecture, such as the desired amount of direct memory-address space desired, so even more complex addressing schemes, based on these principles, are often found in 16-bit and larger microprocessors. The use of *segmented addressing* methods is a case in point. Here, the program counter forms the low-order part of the total address. A special segment register extends the addressable memory space by supplying the bits that are beyond the range of PC or SP.

USER/EXECUTIVE MODES

As noted, data as well as PC and status information can now be placed on a stack. This has led to the maintenance of more than one stack and the use of several stack pointers. Newer microprocessors possess a *user* data SP as well as the EXECutive or SYStem SP. The system SP is used by the operating system to keep track of where the machine is when a subroutine call or an interrupt is serviced. System stacks may store, in addition to old PC, saved system-status information, such as the flags and other vital statistics. Older architectures use only one SP for both purposes but, as a result, are not as suitable for multiprogramming operations. The more advanced systems contain both USER and EXEC (or SYS) SP's and operate in true user and executive modes. In this case, only the operating system may manipulate the executive stack; as a consequence, the users are protected from each other. In USER mode, the user can perform stack operations on data, *but not with the system's stack.*

Thanks to R.S. Barton's invention of stack computing, stack operations have become a solidly entrenched feature of modern architectures–though true stack machines are rare. His stack-machine ideas were first implemented on the Burroughs B5500 mainframe computer in 1963. The author, then employed as a mechanical engineer, was privileged both to observe and to participate as this machine became a reality–and he became motivated toward a new career by this involvement in its creation.

THE ARITHMETIC/LOGIC UNIT: OPERATIONS AND FLAGS

The structural features of a processor's architecture has been likened to functional building blocks. Familiarity with these blocks helps us both understand and create microprograms and processors. One or more ALU's are an integral part of *every* computer. In fact, the ALU and its associated buses are the *essential* part of a CPU. Most registers could reside in external memory. The number of ALU's used within a design can have a profound effect on how *parallel* the machine is–that is, how many *compatible* operations can be transacted in the same time frame. The expression "compatible operations" alludes to the important question of how much can be accomplished in one clock period of a processor. Within these considerations lie many answers to such problems as the required number of clock pulses per cycle, the instruction-set power and sophistication, and the complexity of the design. In studying the ALU's detailed behavior, we begin to appreciate the nature of many of the types of operations that can be part of a compatible set. The ALU itself, being a purely combinational-logic circuit, never receives a clock pulse. It operates within an environment of clocked devices and interconnecting buses. The total set of these operations must remain mutually compatible.

It is presumed that the reader has already been exposed to the design of the combinational-logic *full adder.* In Chapters 2 and 3, we stressed the functional nature of logical blocks (not their design) to review fundamental principles of operation and to promote an understanding of how they are used, as well as their effect on system behavior. Now may be the time for the less hardware-oriented to review a basic text on Boolean logic design. A common form of the ALU IC contains four full adders in one IC, operating in parallel. Associated with these adders of the ALU are data selection, complementation, and combinational logic. Relatively simple though the ALU may be, knowledge of its behavior is central to an understanding of a processor. The ALU not only transforms information but also *originates the signals that control the flow of machine states and programs.* In particular, all the conditional instructions have their origins in information producéd by the ALU. Let us now explore its detailed behavior.

ALU'S INTERFACE

The major interface features associated with an ALU are shown in Figure 3-6. The ALU contains three main data ports. The two ports called A and B in the figure handle *n* lines of parallel-input data, usually from separate sources. The F, or Function, port also handles *n* lines, but these are called a data-output port of the ALU. What appears on these lines is the combinational result of the arithmetic/logic operations the ALU is capable of performing. It is important to realize that the ALU is solely a *combinational-logic* device. It is fed data information as well as a *control word* at the start of a clock period. It then combinationally operates on the two sets of input data and–after the required settling time–produces a stable result at the F port. The propagation and settling time involved in these operations is an important design parameter that limits the clock frequency at which a given system may be driven.

Shown here are $k + 2$ control lines. The electrical level of the *mode* control line establishes whether the combinational operation to be performed is arithmetical or logical in nature. The Carry-In line, referred to either as *CI* or C_n, is ignored during logic operations but is an essential part of arithmetic operations. For this reason, it is treated here as a control input. The *k* lines of the function-control subset select exactly which operation is to be performed within an established mode. All these above-mentioned lines interface to the control system, which of course controls the operations. The present instruction and

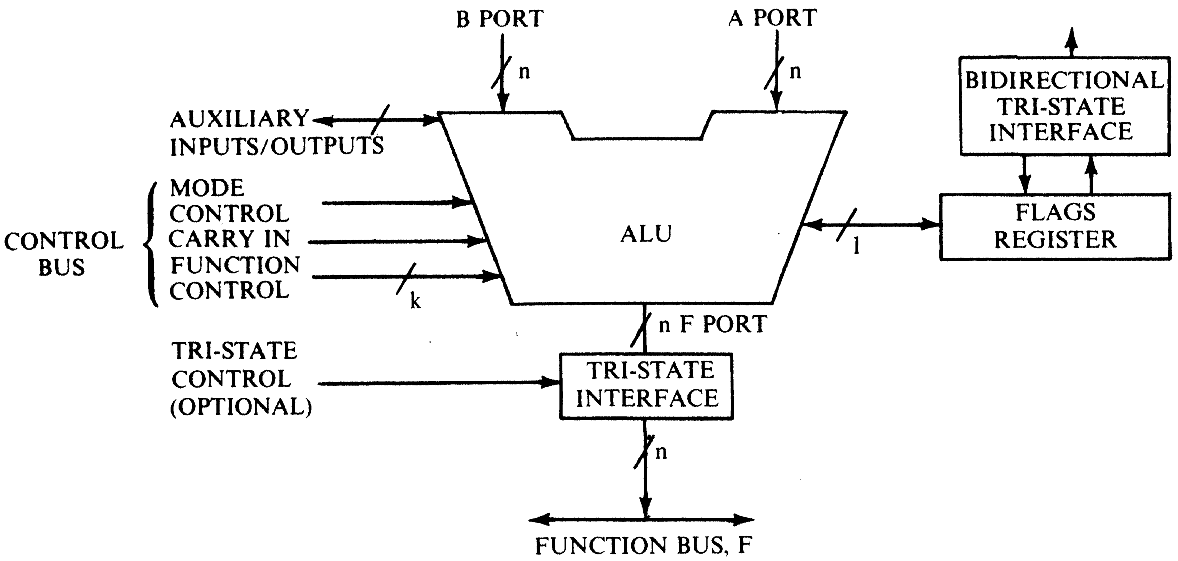


Figure 3-6
ALU Features (Interface Diagram)

the present point in the cycle of computation are interpreted by the control system, which establishes the desired levels on these control lines at the *start* of the current clock period. The data results are stored in some sink at the *end* of the current period. The tri-state interfacing of the F port is necessary only when more than one entity needs to share the F bus with the ALU.

The signals of the auxiliary inputs/outputs include the carry look-ahead, the carry generate, and the propagate signals used to speed up ALU operations when several are joined in parallel. These functions yield significant speed gains, but they are beyond our current scope. The reader should consult a good TTL data catalog, which will give the important application details. If a typical ALU based on a 4-bit module of full adders is used in the construction of a 16-bit or larger CPU, speed gains on the order of thirty percent may be realized by using these auxiliary lines and the available logic IC's associated with them—namely, the fast carry look-ahead generators.

Generation of System Flags

An extremely important function of a computer system's main ALU is its relationship to the associated *Flag Register.* Any time an ALU operation occurs, information is also generated by the ALU, regarding the *nature* of the results. This auxiliary information is stored in a flag register when it is critical to subsequent operations. These flags become the programmer's decision-making mechanism. The importance of this stored flag information lies in their subsequent use in selecting the future path of computation, based upon their contents. The programmer may later (after the flags are loaded) specify conditional instructions, which reference the stored flag values, to establish the direction the program is to take for subsequent operations. For example, conditional branches and multiple-precision arithmetic instructions are based upon the use of flags. The basic idea is that current information about an ALU operation is recorded for future use. Among other things, the stored flag results of an ALU operation can indicate that the results of a previous operation were

Even (Odd)

Negative (Positive)

Zero

Carry (Borrow) generated

Interdigit Carry generated

Overflow (Underflow) occurred

Parity Even (Odd)

A port data Greater (Less than or Equal) than B

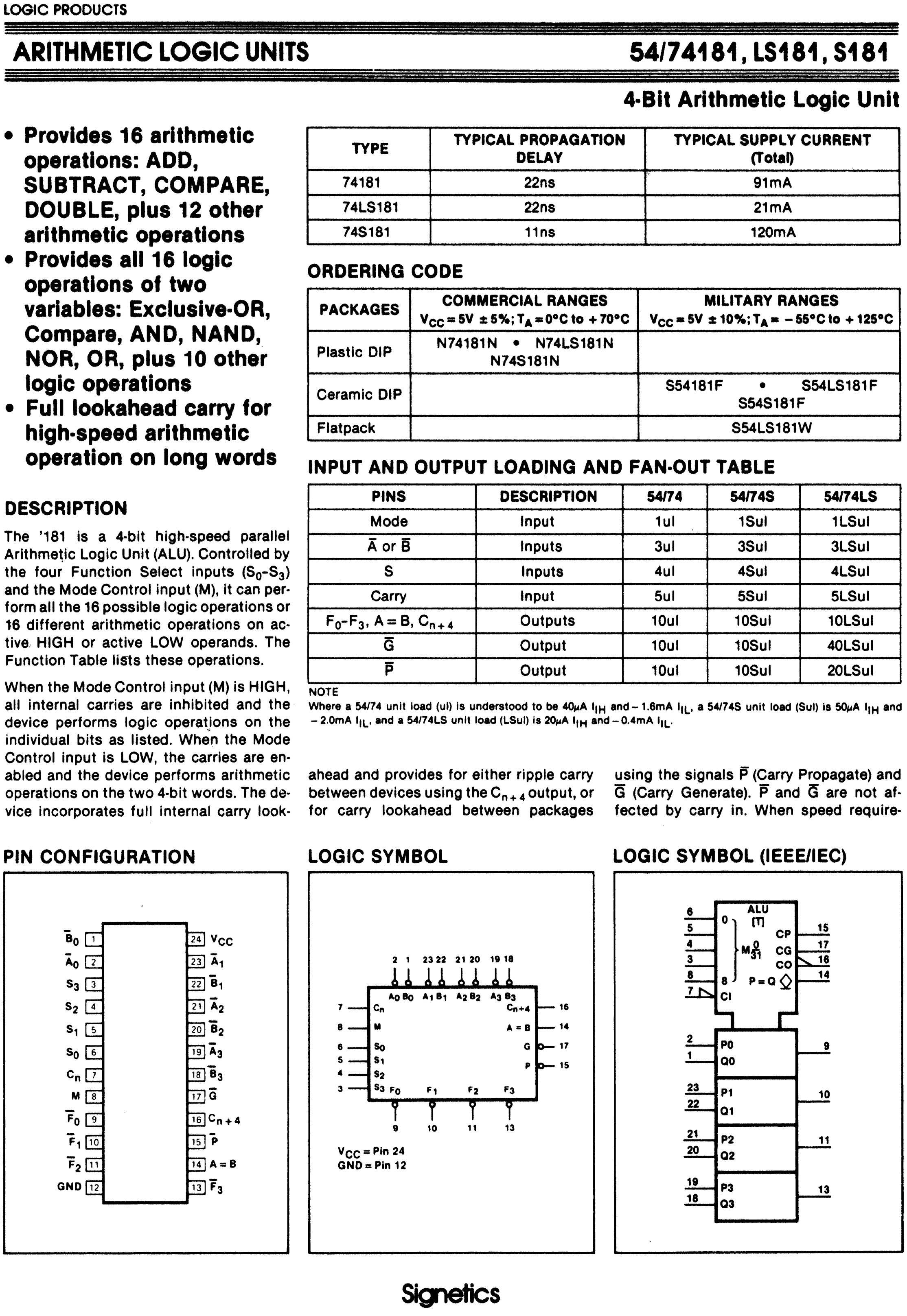
A commonly used minimal set of ALU flags (also called *condition codes*) is Zero, Negative, Carry, and Overflow. The logical inequalities may be deduced from this information. The list above includes condition-code terms that may be derived from a basic set of flags.

A bidirectional interface between the flag register and a system's data bus is necessary for saving the present status of the flags on a stack when servicing calls or interrupts and for setting them to some desired initial value. The flags are a part of the total state of a program that has been subjected to a call or interrupt. They must be preserved when it is put to rest, to correctly recall it again in the future. The programmer, however, may wish to use the flags for, say, multiple-precision (multiple-word) addition. In this case, the first addition is specified as addition without a carry-in. Subsequent additions employ machine instructions that perform an add with carry-in to obtain multiple-precision results. The carry-in is the carry-out resulting from the previous addition, which was stored in a carry-flag cell. Thus the precision of an arithmetic operation can be extended from one word to as many as desired. From these examples we see the necessity for exchanges between the flag register and the rest of the system, as well as for the existence of instructions that affect the flags.

At this point, let us clarify the meaning of the terms *flags* or *condition codes, sense cells*, and *status words*. We have just discussed flags and status information as they are involved in the operation of an ALU. In general, a flag is a memory cell (flip-flop) whose state is affected by the occurrence of an event under conditions established by the control system. There are two contexts in which the term *flag* is used—one external and the other internal. Some processors have flag flip-flops that present their state to the outside world for the use of other devices. These are generally set or reset under program control. A sense line generally refers to an internal flag cell set by the external devices for the internal use of a processor. This combination of the internally generated and externally derived flags is of critical importance to the operation of an advanced system, and the flags are often referred to collectively as the *status word.* A status word or status register can contain some combination of condition codes (internal flags), interrupt status, priority status, external flags, etc. It is probably helpful to think of all flags, sense bits, or status bits simply as generalized status information. They play a critical role in both program and system operation.

74181 ALU INTERFACE

Among the ALU's that can be used for experimentation are the 74LS181 and the pin-compatible 74181. There are also CMOS versions



LOGIC PRODUCTS

ARITHMETIC LOGIC UNITS 54/74181, LS181, S181

4-Bit Arithmetic Logic Unit

- **Provides 16 arithmetic operations: ADD, SUBTRACT, COMPARE, DOUBLE, plus 12 other arithmetic operations**
- **Provides all 16 logic operations of two variables: Exclusive-OR, Compare, AND, NAND, NOR, OR, plus 10 other logic operations**
- **Full lookahead carry for high-speed arithmetic operation on long words**

TYPE	TYPICAL PROPAGATION DELAY	TYPICAL SUPPLY CURRENT (Total)
74181	22ns	91mA
74LS181	22ns	21mA
74S181	11ns	120mA

ORDERING CODE

PACKAGES	COMMERCIAL RANGES $V_{CC} = 5V \pm 5\%; T_A = 0°C$ to $+70°C$	MILITARY RANGES $V_{CC} = 5V \pm 10\%; T_A = -55°C$ to $+125°C$
Plastic DIP	N74181N • N74LS181N N74S181N	
Ceramic DIP		S54181F • S54LS181F S54S181F
Flatpack		S54LS181W

INPUT AND OUTPUT LOADING AND FAN-OUT TABLE

PINS	DESCRIPTION	54/74	54/74S	54/74LS
Mode	Input	1ul	1Sul	1LSul
$\overline{A}$ or $\overline{B}$	Inputs	3ul	3Sul	3LSul
S	Inputs	4ul	4Sul	4LSul
Carry	Input	5ul	5Sul	5LSul
F_0-F_3, A = B, C_{n+4}	Outputs	10ul	10Sul	10LSul
$\overline{G}$	Output	10ul	10Sul	40LSul
$\overline{P}$	Output	10ul	10Sul	20LSul

NOTE
Where a 54/74 unit load (ul) is understood to be 40μA I_{IH} and −1.6mA I_{IL}, a 54/74S unit load (Sul) is 50μA I_{IH} and −2.0mA I_{IL}, and a 54/74LS unit load (LSul) is 20μA I_{IH} and −0.4mA I_{IL}.

DESCRIPTION

The '181 is a 4-bit high-speed parallel Arithmetic Logic Unit (ALU). Controlled by the four Function Select inputs (S_0-S_3) and the Mode Control input (M), it can perform all the 16 possible logic operations or 16 different arithmetic operations on active HIGH or active LOW operands. The Function Table lists these operations.

When the Mode Control input (M) is HIGH, all internal carries are inhibited and the device performs logic operations on the individual bits as listed. When the Mode Control input is LOW, the carries are enabled and the device performs arithmetic operations on the two 4-bit words. The device incorporates full internal carry lookahead and provides for either ripple carry between devices using the C_{n+4} output, or for carry lookahead between packages using the signals $\overline{P}$ (Carry Propagate) and $\overline{G}$ (Carry Generate). $\overline{P}$ and $\overline{G}$ are not affected by carry in. When speed require-

PIN CONFIGURATION

LOGIC SYMBOL

V_{CC} = Pin 24
GND = Pin 12

LOGIC SYMBOL (IEEE/IEC)

Signetics

Figure 3-7
Four-Bit ALU Data Sheet
(Courtesy of Signetics Corporation © 1984 Signetics Corporation.)

ARITHMETIC LOGIC UNITS 54/74181, LS181, S181

ments are not stringent, it can be used in a simple ripple carry mode by connecting the Carry output (C_{n+4}) signal to the Carry input (C_n) of the next unit. For high-speed operation the device is used in conjunction with the '182 carry lookahead circuit. One carry lookahead package is required for each group of four '181 devices. Carry lookahead can be provided at various levels and offers high-speed capability over extremely long word lengths.

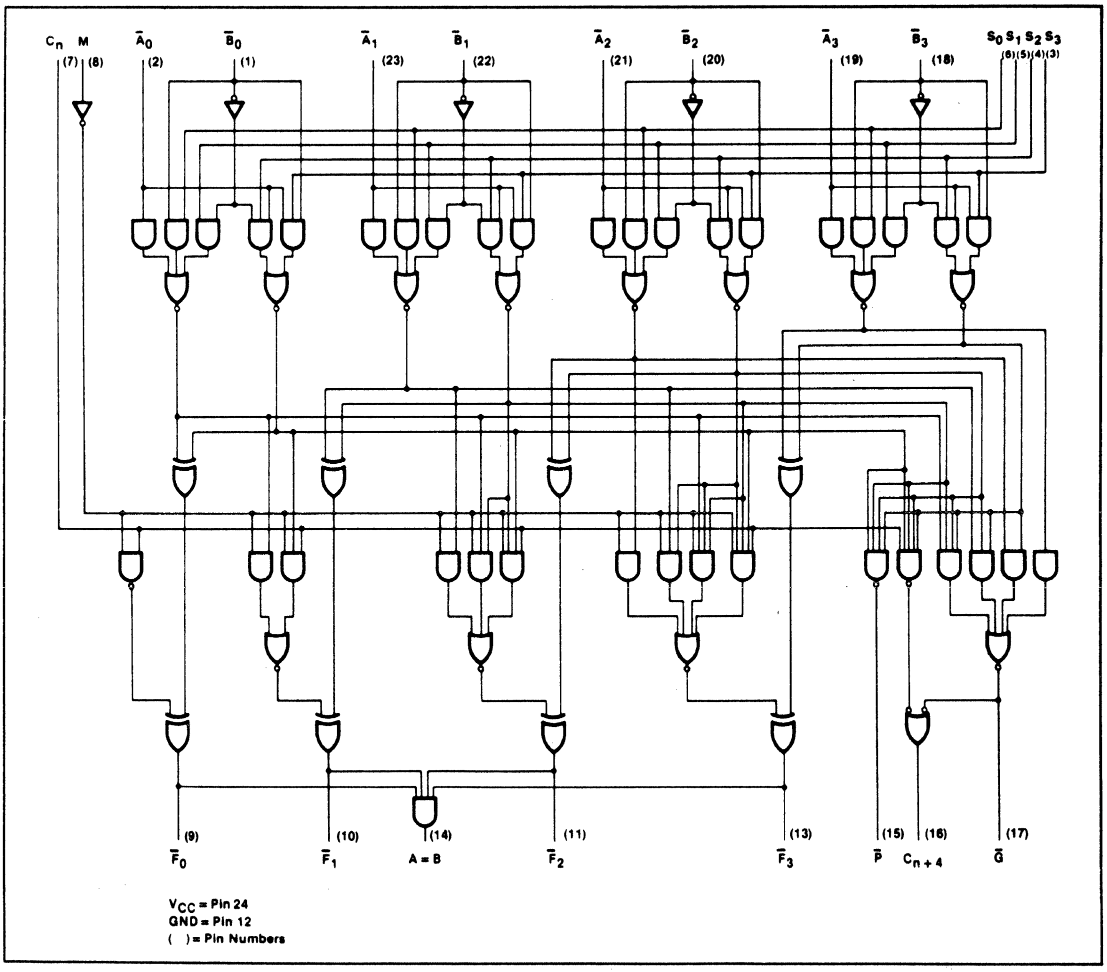
The A = B output from the device goes HIGH when all four $\overline{F}$ outputs are HIGH and can be used to indicate logic equivalence over 4 bits when the unit is in the subtract mode. The A = B output is open collector and can be wired-AND with other A = B outputs to give a comparison for more than 4 bits. The A = B signal can also be used with the C_{n+4} signal to indicate A > B and A < B.

The Function Table lists the arithmetic operations that are performed without a carry in. An incoming carry adds a one to each operation. Thus, select code LHHL generates A minus B minus 1 (2s complement notation) without a carry in and generates A minus B when a carry is applied.

Because subtraction is actually performed by complementary addition (1s complement), a carry out means borrow; thus, a carry is generated when there is no underflow and no carry is generated when there is underflow.

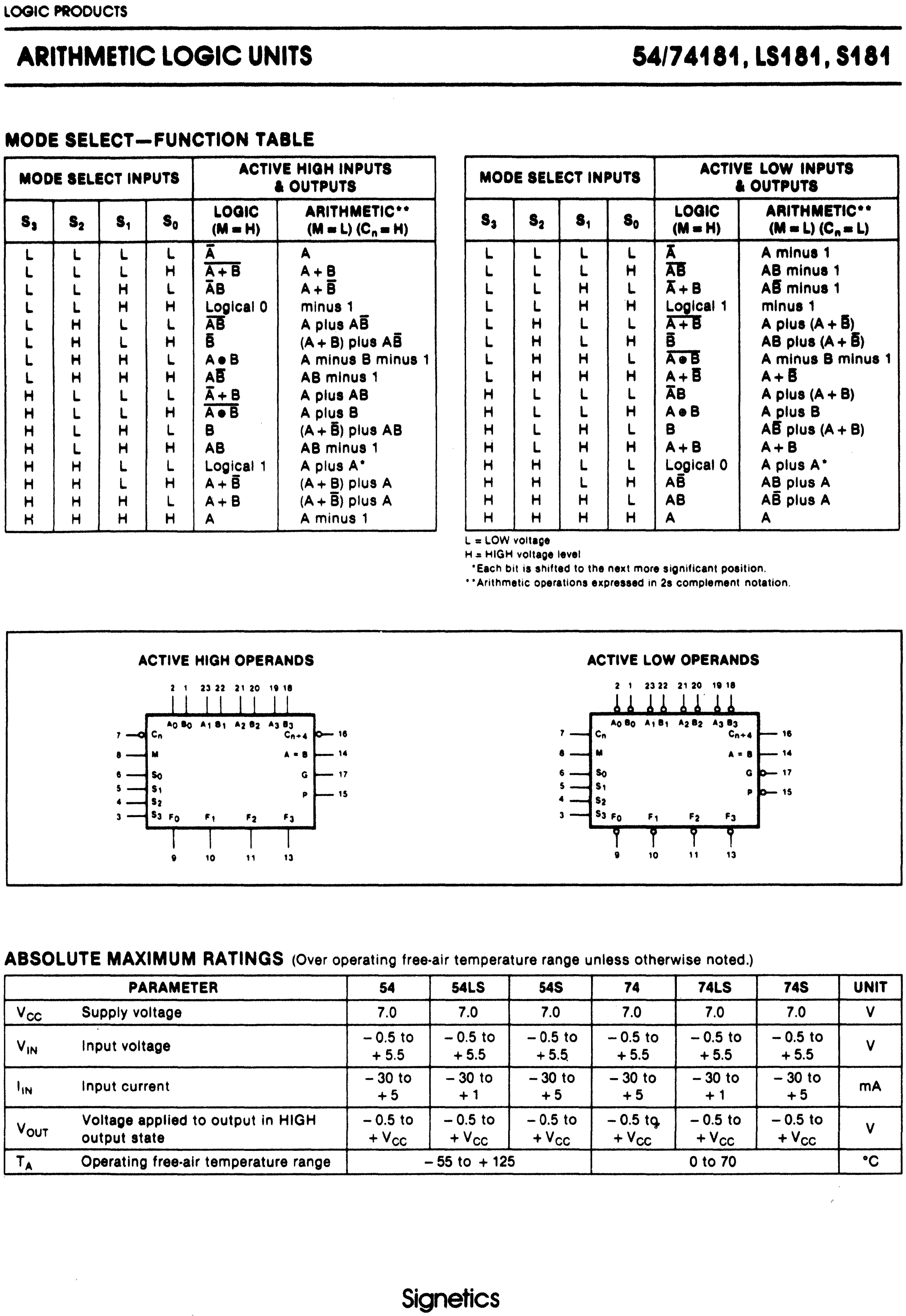
As indicated, this device can be used with either active LOW inputs producing active LOW outputs or with active HIGH inputs producing active HIGH outputs. For either case the table lists the operations that are performed to the operands labeled inside the logic symbol.

LOGIC DIAGRAM



Signetics

Figure 3-7
Four-Bit ALU Data Sheet
(Courtesy of Signetics Corporation © 1984 Signetics Corporation.)



LOGIC PRODUCTS

ARITHMETIC LOGIC UNITS 54/74181, LS181, S181

MODE SELECT—FUNCTION TABLE

MODE SELECT INPUTS				ACTIVE HIGH INPUTS & OUTPUTS	
S_3	S_2	S_1	S_0	LOGIC (M = H)	ARITHMETIC** (M = L) (C_n = H)
L	L	L	L	$\overline{A}$	A
L	L	L	H	$\overline{A+B}$	A + B
L	L	H	L	$\overline{A}B$	$A+\overline{B}$
L	L	H	H	Logical 0	minus 1
L	H	L	L	$\overline{AB}$	A plus $A\overline{B}$
L	H	L	H	$\overline{B}$	(A + B) plus $A\overline{B}$
L	H	H	L	$A \oplus B$	A minus B minus 1
L	H	H	H	$A\overline{B}$	AB minus 1
H	L	L	L	$\overline{A}+B$	A plus AB
H	L	L	H	$\overline{A \oplus B}$	A plus B
H	L	H	L	B	$(A+\overline{B})$ plus AB
H	L	H	H	AB	AB minus 1
H	H	L	L	Logical 1	A plus A*
H	H	L	H	$A+\overline{B}$	(A + B) plus A
H	H	H	L	A + B	$(A+\overline{B})$ plus A
H	H	H	H	A	A minus 1

MODE SELECT INPUTS				ACTIVE LOW INPUTS & OUTPUTS	
S_3	S_2	S_1	S_0	LOGIC (M = H)	ARITHMETIC** (M = L) (C_n = L)
L	L	L	L	$\overline{A}$	A minus 1
L	L	L	H	$\overline{AB}$	AB minus 1
L	L	H	L	$\overline{A}+B$	$A\overline{B}$ minus 1
L	L	H	H	Logical 1	minus 1
L	H	L	L	$\overline{A+B}$	A plus $(A+\overline{B})$
L	H	L	H	$\overline{B}$	AB plus $(A+\overline{B})$
L	H	H	L	$\overline{A \oplus B}$	A minus B minus 1
L	H	H	H	$A+\overline{B}$	$A+\overline{B}$
H	L	L	L	$\overline{A}B$	A plus (A + B)
H	L	L	H	$A \oplus B$	A plus B
H	L	H	L	B	$A\overline{B}$ plus (A + B)
H	L	H	H	A + B	A + B
H	H	L	L	Logical 0	A plus A*
H	H	L	H	$A\overline{B}$	AB plus A
H	H	H	L	AB	$A\overline{B}$ plus A
H	H	H	H	A	A

L = LOW voltage
H = HIGH voltage level
*Each bit is shifted to the next more significant position.
**Arithmetic operations expressed in 2s complement notation.

ACTIVE HIGH OPERANDS

ACTIVE LOW OPERANDS

ABSOLUTE MAXIMUM RATINGS (Over operating free-air temperature range unless otherwise noted.)

PARAMETER		54	54LS	54S	74	74LS	74S	UNIT
V_{CC}	Supply voltage	7.0	7.0	7.0	7.0	7.0	7.0	V
V_{IN}	Input voltage	– 0.5 to + 5.5	– 0.5 to + 5.5	– 0.5 to + 5.5	– 0.5 to + 5.5	– 0.5 to + 5.5	– 0.5 to + 5.5	V
I_{IN}	Input current	– 30 to + 5	– 30 to + 1	– 30 to + 5	– 30 to + 5	– 30 to + 1	– 30 to + 5	mA
V_{OUT}	Voltage applied to output in HIGH output state	– 0.5 to + V_{CC}	– 0.5 to + V_{CC}	– 0.5 to + V_{CC}	– 0.5 to + V_{CC}	– 0.5 to + V_{CC}	– 0.5 to + V_{CC}	V
T_A	Operating free-air temperature range	– 55 to + 125			0 to 70			°C

Signetics

Figure 3-7
Four-Bit ALU Data Sheet
(Courtesy of Signetics Corporation © 1984 Signetics Corporation.)

of the same device. This particular IC was fundamental in the development of many minicomputers and controllers during the 1960s and 1970s. The manufacturer's specification for this device, which comes in a 24-pin dual in-line package (DIP), are presented in Figure 3-7. The first step in using it is to adopt a convention for treating the *data* either as *active high* or as *active low.* Active-low data problems are presented in the exercises. The use of active-low data conventions in ALU implementations and interfacing is widespread in practice. Some manufacturers' data-sheet explanations emphasize only active-low data—that is, the low level is the logically true level on the data lines. This is a reflection of the frequent use of active-low conventions. In this text, as in most, we use active-high data as the usual convention. Even so, we cannot avoid dealing with active-low conditions. The reader may encounter difficulties later, unless the ALU problems that use active-low conventions are mastered now.

The ALU interface contains *both* active-high and active-low interface pins. The carry-in and carry-out pins (7 and 16, respectively) are *always* of the opposite activity level from the data ports. For example, in the case where active-high data is the selected convention, the carries are active low. That is, a carry-in will be asserted by a low level on pin 7 when the data is treated as active high. Had we chosen to consider the data pins of this ALU as being active low, then these two carry lines would have been treated as active high. Under these last conditions, a high on either pin 7 or 16 implies the existence of a carry-in or a carry-out, respectively. The confusion is caused by the improper use of the overbar symbol on all the manufacturers' data sheets: the overbar symbol is correctly used for complementation, but it is incorrectly used when it also represents the active-low condition. Later in the text, we shall use the symbol @ to indicate that a named signal is active low. For now, the remedy is to think of the pin names of the ALU just in terms of their functional names: *data* and *carry* pins. If the data ports are designated to you as active high, then the carries are all active low, and vice versa.

ALU RELATIVE MAGNITUDES

An output of this ALU, pin 14, is termed the A = B output. It too has special characteristics. This pin is used to ascertain whether bit-for-bit equality exists between the two sets of input-data lines. This equality detection is performed when the ALU is in the subtract mode, while the carry-in is high. Pin 14 is an *open-collector* output and therefore requires a pull-up resistor between itself and the power supply, V_{CC}, to be useful. As the data-sheet specifications indicate, the combination of carry-out (C_{n+4}) and A = B may be used to establish the logical relationships between the A and B ports. C_n is also utilized to control these relative-magnitude operations. Table 3-1 summarizes the pin relationships for these relative magnitude operations. As discussed, the control over the flow of software originates in the ALU. These are the signal types that, when stored as flags, enable conditional instructions to make decisions. From the equivalent logic diagram, note that the A = B function is produced when all the inputs to an AND gate are high. Therefore the A = B output is *always* active high, regardless of the data activity-level convention used. Further, A = B has the character-

Table 3-1
Relative Magnitude Tests

C_n *Input*	*Active-Low Data*	*Active-High Data*	C_{n+4} *Output*
H	$A \geq B$	$A \leq B$	H
H	$A < B$	$A > B$	L
L	$A > B$	$A < B$	H
L	$A \leq B$	$A \geq B$	L

Notes: $M = L$
$S_3 \ldots S_0 = L\,H\,H\,L$

istic that it is high whenever all F lines are high—a useful fact that has some innovative applications.

The ALU has a central role in both the organization and the operation of any processor that transforms data. There is an ALU at the heart of every computer, though it is not necessarily the 74181. A study of the equivalent logic diagram of the ALU confirms that it contains four combinational-logic full adders arranged in parallel, fed by data selection and combination circuitry. A close scrutiny of this chip's logic (IC's are also often referred to as chips) is well worth the time.

To specify arithmetic/logic operations, refer to the *mode-select function* table in Figure 3-7. When the mode line is high, this ALU is in the logic mode. In this mode, the four S lines ($S_3 \ldots S_0$) specify the generation of *all 16 functions of two variables.* When binary logical operations are selected, these are performed on the basis of matched pairs of data. Carries never enter into the logic-function formation process. That is, A_0 and B_0 are logically operated on, independently of other bit positions or carries, to form F_0, and so forth. When the mode bit is low, the arithmetic operations specified by $S_3 \ldots S_0$ depend on the state of the carry-in, C_n. Not all of these arithmetic operations yield useful results, but there are some valuable operations, too. For example, when selecting the function F = A (in the arithmetic mode) the C_n line can be used to produce either this or F = A plus 1, depending on the value of the carry. This provides a means of either passing the contents of A port through the ALU intact or incrementing it by one, before handing it over to the F bus. *Therefore, the carry-in affects which arithmetic function is actually being performed,* by adding one to the function produced when there is no carry. Some data sheets display two arithmetic function columns. The second, missing here, is simply the function shown here plus one. Note that the data sheets use the word *plus* to indicate addition and use the symbol + only to indicate the logical OR operation.

In the logic mode, we can also produce an F = A and an F = B function. These are most useful when we simply wish to transfer the contents of one input bus or the other to the output, or F bus. ALU's are frequently used for bus-transfer operations. Since carries cannot influence the results, this is a less mistake-prone method of effecting data transfers from one bus to another when specifying microcode. Also note that, when the ALU is in the arithmetic mode and performing F = A plus A, the quantity is being doubled. In binary, this is tantamount to shifting left one place. Thus this ALU shifts left, too. In practice, however, shifting is usually accomplished with the aid of auxiliary registers and multiplexers. The power that can be packed into this relatively simple device is impressive. The ALUs used in a monolithic processor design do not require all 16 logic functions or the useless arithmetic ones. They are therefore streamlined versions of this ALU, tailored to meet the specific needs of a particular computer's design.

ALU WORKSHEET

Figure 3-8 presents a work sheet for solving 74181 ALU problems. The *Data Activity Level* column information must be given to work a problem. An AH in this column implies that the electrically high level is to be considered as the logically true, or 1, level for that row. An AL implies that the electrically low level is to be treated as the logical true or 1 level for the current problem. Regardless of the given or chosen activity level, all arithmetic computations are to be performed in binary, using two's-complement arithmetic conventions. If active-low data is being used, an L in a given bit position signifies that an arithmetic or logic 1 applies to that data bit. Keep in mind that the C_n and C_{n+4} pins of the ALU have an activity level opposite that of the data. This is a frequent source of errors. For a given truth value (1 or 0), the electrical level (H or L) of the carries that is either applied or produced, respectively, is the reverse of the level of a data line for that same truth value.

In the exercise problems, the *ALU Function Performed* column either is given or must be derived from a specified control word, with the aid of the data sheets of Figure 3-7. The control word may be specified in the next six columns, or it must be derived from the given function named through the use of the tables. One must be given, the other derived. The M or mode bit controls whether the operation to be performed is arithmetical or logical in nature. The device data-sheets format species this, too. Referring to them again shows that, when the M bit is high, this ALU is always in the logic-operations mode, regardless of the data-activity level chosen. If the M bit is low, then an arithmetic operation is specified. In either mode, the four S lines select the exact function to be performed. Remember that, in the logic mode, both the carry-in and carry-out bits are treated as irrelevant. This irrelevance is signified by an X in the appropriate place(s) on the work sheet.

As noted, in the arithmetic mode, the C_n bit controls the function to be performed, in a simple way. If the carry-in is a logical 0, that is, no carry-in, then the arithmetic operation is as specified in the arithmetic column of the data sheet. If the carry-in is active (true), then the operation is this same operation plus one. The 16 functions of the arithmetic mode are not all sensible or useful. Still, incrementation or decrementation of the A port (A_3 . . A_0) data, one's- and two's-complement addition of A and B port data, and subtraction are all quite useful. Even shifting of the A-port data, which in practice is performed outside the ALU by other hardware, may be demonstrated. This ALU shifts only to the left. The results of all data operations appear at the F, or function, port pins. The auxiliary information is also output on the C_{n+4}, A = B and F_3 pins for possible storage as flags. The most significant bit of the output is the *sign* bit in arithmetic operations. That is F3 in this case. It, too, may be stored as a status flag. The flags

	DATA ACTIVITY LEVEL	ALU FUNCTION PERFORMED	FUNCTION						A BUS DATA LINES				B BUS DATA LINES				F BUS DATA LINES				C OUT
			MODE	SELECT LINES				C IN													
			M	S3	S2	S1	S0	C_N	A3	A2	A1	A0	B3	B2	B1	B0	F3	F2	F1	F0	C_{N+4}
A)																					
B)																					
C)																					
D)																					
E)																					
F)																					
G)																					
H)																					

Figure 3-8
74181 ALU Worksheet

METHOD: FILL IN BLANKS WITH APPROPRIATE H OR L AFTER CONSULTING ALU TABLES. FILL IN MISSING ALU FUNCTIONS. SHOW CALCULATIONS BELOW.

we will use that originate in this ALU are the carry, equal, and sign flags.

Two's Complement ALU Problems

A sample problem set and their solutions are given in Figures 3-9a and 3-9b. In the first problem, the data activity level is given as AL, and the function required is A plus 1. Consulting the active-low data table for the 74181, we find that, since this is an arithmetic operation, the mode line must be low. For active low, the four select lines for the function f = A are specified as $S_3 \ldots S_0$ = HHHH. Note that this is f = A in the arithmetic mode. C_n is specified as an L. We can rationalize this as follows: If the data is active low, the carries are active high, implying that C_n must be low to be inactive. The function we desire, however, is f = A plus 1. To add 1 in the arithmetic mode, we specify C_n as active, or H. The A data is given as −2, decimal equivalent. In two's-comple-

	DATA ACTIVITY LEVEL	ALU FUNCTION PERFORMED	FUNCTION						A BUS DATA LINES				B BUS DATA LINES				F BUS DATA LINES				C OUT
			MODE	SELECT LINES				C IN													
			M	S3	S2	S1	S0	C_N	A3	A2	A1	A0	B3	B2	B1	B0	F3	F2	F1	F0	C_{N+4}
A)	AL	A pl 1								−	2			+	5						
B)	AH		H	L	L	L	L	L													
									L	H	L	H	L	L	H	H					
C)	AL	A mi B								−	3			−	4						
D)	AH		L	H	H	L	L	L													
									L	L	H	L	H	L	H	L					
E)																					
F)																					
G)																					
H)																					

Figure 3-9a
74181 ALU Worksheet:
Example Problem

METHOD: FILL IN BLANKS WITH APPROPRIATE H OR L AFTER CONSULTING ALU TABLES.

ment (TC) arithmetic, negative numbers are represented by their TC form, which is arrived at by the following simple procedure:

1. Write the binary equivalent of the absolute magnitude of the number: For example, the absolute magnitude of −2 in binary notation is 0010. We are using a 4-bit ALU size, the most significant bit being the sign bit. If the sign bit is a 0, the number is positive. If the sign bit is a 1, the number is negative, and the balance of its bits (the magnitude bits) are in TC form.

2. Starting with the least significant binary digit, copy down all digits, as they are, up to and including the first 1. That is, in the example, copy the digits *10*.

3. Now write down the complement of all the remaining digits. That is, the lefthand 0s become 1s, and vice versa. In the example here, change the leftmost *00* to *11*.

4. This result, *1110*, is the TC form of the original number. Taking the TC form of this number reproduces the original number again.

	DATA ACTIVITY LEVEL	ALU FUNCTION PERFORMED	FUNCTION						A BUS DATA LINES				B BUS DATA LINES				F BUS DATA LINES				C OUT
			MODE	SELECT LINES				C IN													
			M	S3	S2	S1	S0	C_N	A3	A2	A1	A0	B3	B2	B1	B0	F3	F2	F1	F0	C_{N+4}
A)	AL	A plus 1	L	H	H	H	H	H	1	1 –	1 2	0	0	1 +	0 5	1	1	1	1	1	0
									L	L	L	H	X H	X L	X H	X L	L	L	L	L	L
B)	AH	F = Ā	H	L	L	L	L	X L													X
									L	H	L	H	L	L	H	H	H	L	H	L	
C)	AL	A min B	L	L	H	H	L	H	1	1 –	0 3	1	1	1 –	0 4	0	0	0	0	1	1
									L	L	H	L	L	L	H	H	H	H	H	L	H
D)	AH	A pl 1 pl 1	L	H	H	L	L	L													
									L	L	H	L	H	L	H	L	L	H	L	H	H

METHOD: FILL IN BLANKS WITH APPROPRIATE H OR L AFTER CONSULTING ALU TABLES.

A) 0010
1110
+1
(0)1111 = -1

B) FROM DATA SHEETS
F = Ā LHLH
HLHL

C) 0011 0100
1101 1100 = -4
0100 0100
(1) 0001

D) A = LLHL 0010
SHLA = (H) LHLL (0)0100
+1 H +1 1
H (0)0101

Figure 3-9b
74181 ALU Worksheet: Completed Problem

The B data in the first problem is irrelevant because it is not involved in the specified function. To complete the operation, we add 1 to the A data, as follows:

1110 = −2
plus 1
(0)1111 = −1

The logical carry-out is enclosed in parentheses. To convert this result to the F-port electrical levels, convert all the true data bits to L's, to denote that, in this result, they are all active. The carry-out (for active-low data) is active high. Since the carry-out of 0 means this line is not to be active, we specify an L for $C_n + 4$.

The third problem of the set illustrates the fact that the ALU does not actually perform a physical subtraction—it contains only an adder. In TC arithmetic, subtraction is accomplished within the ALU by *adding* the subtrahend's TC to the minuend. The A data (-3) is 1101. The B data (-4) is 1100. To subtract -4, we first find its TC form, and then add it to the minuend. The arithmetic appears as follows:

$$\begin{array}{rcr} -3 & (-3) = & 1101 \\ \underline{\text{min } -4} & \text{TC of } (-4) = & \underline{\text{plus } 0100} \\ ? & & (1)0001 \end{array}$$

Note that the carry-out is now active and that C_{n+4} is therefore an H. The F-port data is now HHHL, in electrical-level terminology. The advantage of the electrical-level symbology is that it expresses the behavior of components without ambiguity.

We have spelled out terms in the arithmetic operation above, to avoid confusing the two's-complement arithmetic operation with the sign of the value to be operated on. The data sheets reflect this practice, too. Differentiating between ALU operations and the signs of the data values avoids a great deal of confusion.

The neat thing about TC arithmetic is that the results of all arithmetic addition and subtraction operations are correct in both sign and magnitude and that the carry-out is used only in the detection of overflow or underflow. Since we are not discussing these special overflow situations here, we should mention that any TC arithmetic problem formulation that causes an overflow or underflow to occur produces erroneous results. It may come as a surprise to see that ordinary processors have only the capability to *add.* Operations such as multiplication are performed by an algorithm involving successive additions. We will microprogram one of these later, using this ALU. The high-performance microprocessors speed up multiplication and addition by providing special logic for these purposes. Since we make extensive use of this ALU in microcoding instruction sets for a processor, the practice problems provide an opportunity to gain proficiency in using this ALU.

EFFECTS OF ARCHITECTURAL VARIATIONS ON OPERATIONS: PUTTING IT ALL TOGETHER

Let us now examine several basic systems architectures containing the major constructs we have studied so far—buses, registers, arrays, and ALU's. As noted, the arrays and many registers, being memory, need not be part of a CPU. To obtain speed, they generally are included in the CPU. The ALU and associated buses are essential to the formation of a CPU architecture. There are several CPU's that employ external memory for registers and arrays. Here, we put all these major structural elements of the traditional CPU together in a system, examining several architectural variations of bus-organized systems. Specifically, we shall observe how the ALU and BUS organization impacts on their *register transfers,* as well as on the number of clock periods required to accomplish a selected task. The number of steps required to effect a transfer, along with the architectural sophistication of the system, is an important subject in establishing an architecture for a system. As so often happens, the considerations of cost versus speed of a system are also involved. Because this subject is at the heart of how we perceive and understand computer operations, a little experimentation with architectural variations should go a long way toward providing insight into *why* a particular machine is constructed or behaves in a given manner. In Chapters 5 and 6, we will study a specific implementation of a basic 4-bit CPU, where this type of perceptual background is applied to the microprogramming of instruction sets.

ARCHITECTURAL COMPARISONS

To compare the effects of architecture on operation, we use the following generalized situation for three selected architectural examples:

Problem Statement

The overall problem consists of adding the contents of a register R in an array to that of a memory location M. All arithmetic/logic operations are to be performed with the use of an Accumulator (ACC) register.

Next, we define the operation of addition, as follows:

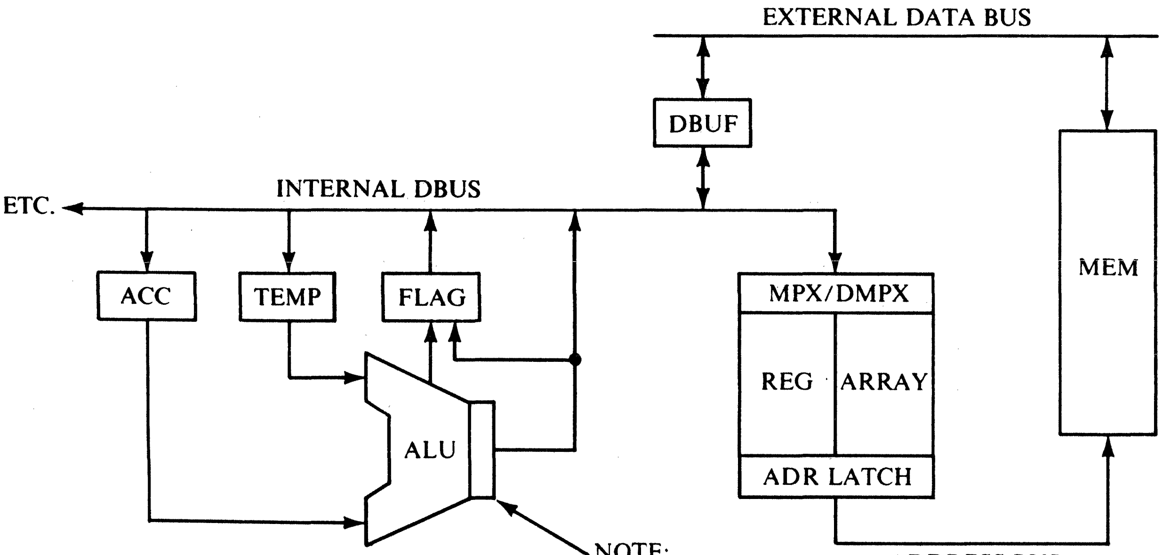
Definition (ADD)

The operation of addition is defined as the summing of the present contents of an ACC register with itself, or with the contents of another register, or with the contents of a memory location. The results of the operation are to be stored in the ACC register, generally changing its contents. Source operands, other than the ACC, are to remain unchanged. Note that tha ACC register always serves as a source of one operand.

EXAMPLE 1: THE SINGLE-BUS SYSTEM ARCHITECTURE PERFORMANCE

Figure 3-10 displays a *single-bus system,* defined as such because the ALU's inputs and outputs interface to the same internal data bus as the register array. In this example, the register array is to be used only for data retrieval and storage, *not* as an arithmetic/logic accumulator. The results of all arithmetic/logic operations are to be deposited in the special accumulator provided. In practice, this restriction is often applied where an architecture lacks parallelism. Coincidentally, similar constructs were utilized in the design of some very successful 8-bit microprocessors, such as the 8080. For comparison purposes, we shall examine the execution of an overall operation that includes the step of addition, as presented in the problem statement.

In Figure 3-10, two registers feed the ALU. These are the ACC and TEMP registers. The ACC is shown as a single register, presumed to have edge-triggered JK-type clocking, for simplicity. Actually, two transparent latches are often used here. In the next section, we shall start to clear up these unexplained references to this widely applied type of latch and its clock characteristics. The overall addition operation involves a register, say R1, of the array in the total transaction. One of the numbers to be added comes from R1. The other number comes from memory. The steps to add these numbers and store the results in R1 follow. In this process, R1 is treated as the destination



NOTE:
ALL INTERFACES
TO DBUS ARE TRI-STATABLE
OR OPEN-COLLECTOR TYPES.

ADDRESS BUS
(EXTERNAL)

Figure 3-10
Single-Bus System (Partial)

storage area for the final results. The special ACC register is the sole recipient of the results of all arithmetic/logic steps in the total operation. It would be nice if we could treat *any* register as a generalized accumulator; this will come later. For now, the sequence of steps required to add under these conditions is given below–*each step is synonymous with a single clock period:*

1. Transfer the contents of R1 into ACC over the internal DBUS.
2. Bring in the contents of the desired memory location, assumed to be present at DBUF, and store them in TEMP. These two steps are preliminary to the actual operation of addition, next.
3. Place the ALU in the ADD state, and transfer its output to ACC over DBUS. The ALU is fed by the ACC and TEMP registers. Coincidentally, the FLAG register would be loaded at this point. It would record only status information about this operation, such as the state of the sign and carry bits, not the results of the addition.
4. Place the ALU in the transfer state, such that the previously summed contents stored in ACC are placed onto the DBUS. At the same time, have R1 capture these contents at the end of this current clock period.

This entire procedure cannot be performed in fewer than these four steps, given the constraints of this architecture. One cannot drive the DBUS simultaneously from two different sources–a case of operations that would be incompatible. Therefore we see that the single-bus system requires a rather large number of steps to perform the given task. This design approach is economical in its use of components, but, due to a lack of parallelism, the trade-off penalty is the time required to achieve the end result.

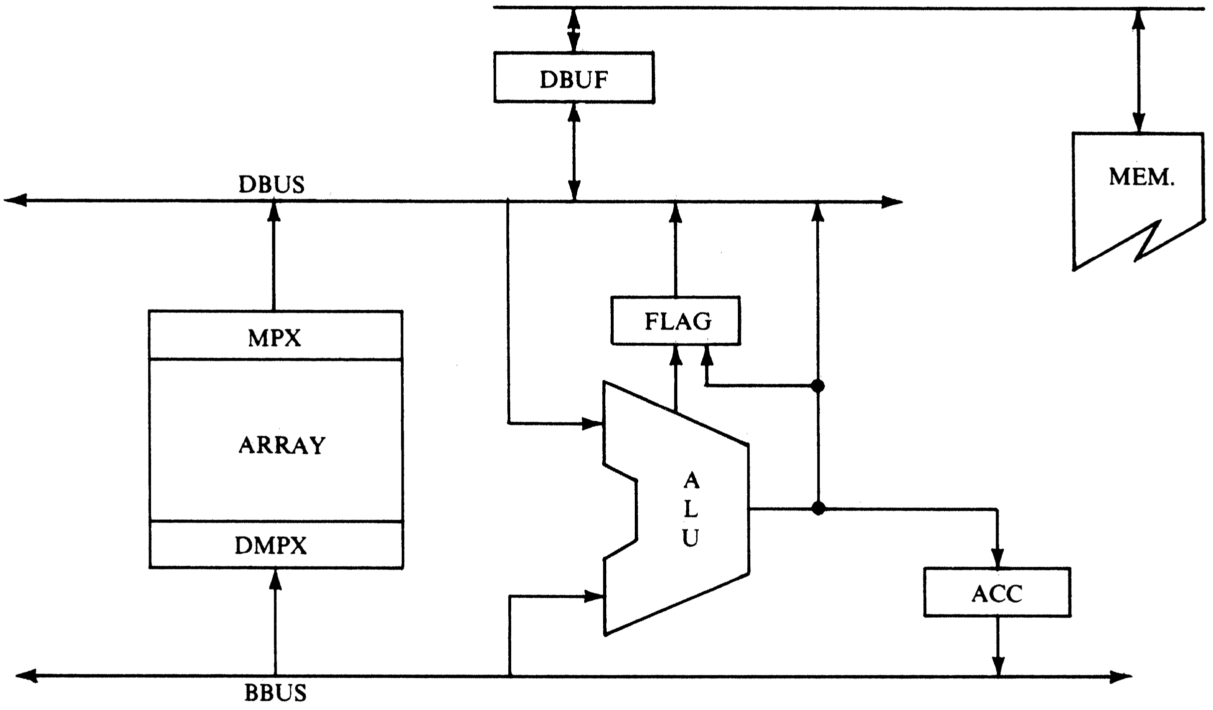


Figure 3-11
Two-Bus System (Partial)

EXAMPLE 2: THE TWO-BUS SYSTEM ARCHITECTURE PERFORMANCE

Let us now compare the case above to the more sophisticated two-bus architecture of Figure 3-11. In this new hypothetical case, which still uses a single special ACC register, the following steps are necessary to accomplish the same task:

1. Place the contents of R1, in the array, onto the DBUS, and deposit it into the ACC.
2. Place the contents of the desired memory location onto the DBUS. With the ALU in the ADD state during this time frame, the ACC captures the sum of the operands. All these operations are compatible, since no data clashes occur on the buses.
3. Place the contents of ACC onto the BBUS and store the result in R1.

At the expense of an additional bus and interfaces (tri-state or functional equivalents), it is now possible to perform this computation in three steps. This is a more parallel architecture. Can you invent a better architecture by slightly rearranging this one? Some of the embellishments added in this example are:

a. The ACC is now interfaced to the DBUS via an appropriate interface, which is seldom shown explicitly.
b. The contents of any register in the array may be directly presented to the ALU over the BBUS.
c. The contents of memory can directly reach the ALU over the DBUS.

There are limitations on the designer in the selection of architectures, of course. These include cost, state of the fabrication art, and optimization of an entire instruction set, as opposed to a single operation, as we are doing. Yet building blocks are meant for building with, so let's carry on.

EXAMPLE 3: THE THREE-BUS SYSTEM ARCHITECTURE PERFORMANCE

Our last hypothetical example is the three-bus architecture, displayed in Figure 3-12. In this example, we loosen the original constraints and dispense with an explicit accumulator. The accumulator can be any of the 8-bit registers in the dual-bank array. Notice, too, that we are now definitely mixing dual-sized architectures together—a common practice. Now, all that is required to perform the given overall task is to:

1. Simultaneously place the contents of R1, which may exist in any one of the two 8-bit arrays, via the MPX/DMPX logic, onto the BBUS. The contents of memory are also placed onto the DBUS in this same time frame. No conflicts here. In the current time frame, the control system also orders the ALU to ADD. The combinational result of the addition appears on the FBUS before the end of this clock period. It can be stored in any general register of the 8-bit array desired—we assume even R1. Thus, with this architecture, the general registers in the arrays can serve as accumulators. These registers in the array are called *general registers,* as most of the PDP-11 registers are, to differentiate them from some special registers—such as a possible SP or PC within the array (not the case here).

The assumption that R1 can support true simultaneous reads and writes is not always carried out in practice but could be in principle. Chapter 5 more fully explains these practical concerns about the proper use of the widely applied transparent latch. If we assume the straightforward use of edge-triggering for now, no principles are violated. We are careful to point out that actual usage may vary.

Depending on our choice of architectures and the parallelism of their organization, we see that the number of steps to transform information vary greatly. Maximizing the set of operations that can be performed simultaneously is very advantageous. We apply this topic again in the subsequent chapters that develop microprogramming skills. One sobering thought is that architectural design is not an end in itself. What really counts is the set of instructions that are available to to the programmer for the development of sophisticated software. In the final analysis, good architectural design can only serve the best interests of software sophistication—that is, it does not exist in a vacuum. When we get around to practicing microprogramming for a limited architecture, we can observe the frustrations attached to hardware/software trade-off considerations. First, though, we need to grasp the system coordination of the hardware. That involves a deeper understanding of these architectural constructs, presented here in an introductory way to provide motivation and orientation.

A few more remarks on the sophisticated three-bus approach of Example 3 are relevant. Besides the mixing of 8-bit and 16-bit architectures, notice the second autonomous simple ALU, which is an integral part of the Address Bus (ABUS) structure of the system. This enhances the concept of compatibility, in that even more can be performed in one time frame than before. These ideas can be extrapolated to the simultaneous overlapping of the Instruction Fetch and Instruction Execute operations, using segmented memories. This has been done in practice. As it is, the inclusion of the program counter and one or more stack pointers, as well as an operand address register, in the 16-bit array—along with their own basic ALU—significantly increases the

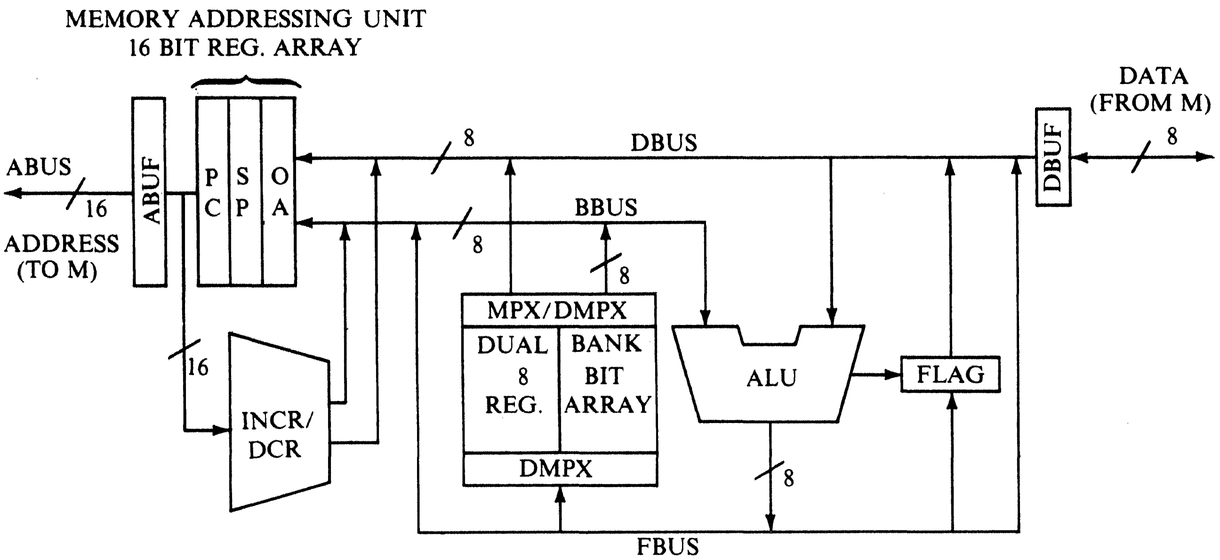


Figure 3-12
Three-Bus System (Partial)

speed and power of the system. From the previous constructs, we have actually created a separate, larger scale functional block, which is applied to VLSI designs. In terms of block structure, some modern CPU systems are described as consisting of an Instruction Prefetch unit and an Instruction Execute unit. Carried even further, we can begin to visualize multiple ALUs, register arrays, and sophisticated busing schemes, leading to the *array processor* form of architecture. Few if any new functional blocks are required.

The point of the above is that we have already seen most of the functional blocks of which an architecture consists, and we have studied some of their organizational usage within a system. We have delved into circuit-design details only enough to gain an appreciation for functioning of *blocks* of logic. It is the innovative use of these blocks that is involved in systems organization. These have been only partial views of the anatomy of the architecture of a processor–micro or otherwise. It is an important topic. The relationships between the ALU's, registers, and the external environment are critical to the success of a system. Most important, as noted, these features must promote the expeditious performance of the software algorithms, for which the machine is designed in the first place.

An important point to note in following the examples was our need to *visualize* the data paths available to us at any step in solving a data flow and transformation problem. Development of this ability of visualizing highly parallel and compatible data flow, from sources to sinks, is vital to our mastery of the operation and microprogramming of a given architecture.

In conclusion, note that we started with a very elementary construct–the logic gate–and demonstrated that it could be used as part of a building-block organizational scheme, to arrive at a basic understanding of the structure of bus-organized processors. The final results of this section began to stress the functional relationships between these blocks of interface gates, buses, memory elements, register arrays, ALU's, and MPX/DMPX schemes. From an understanding of these systems organizations, we may start to visualize possible successful patterns for data flow within a processor. All of these will be explored in specific detail in Chapters 5 and 6, where we will find that a working instructional CPU can be organized (and constructed, too) using only approximately ten types of IC's. In learning about these few IC's, we will see most of the features and functions of present architectures at work. It is to be emphasized that we are beginning to view the processor as a *system* that utilizes the basic concepts discussed here. We may also begin to understand the innovative nature of the tasks available to us, using relatively few concepts, which can yield significant computing power.

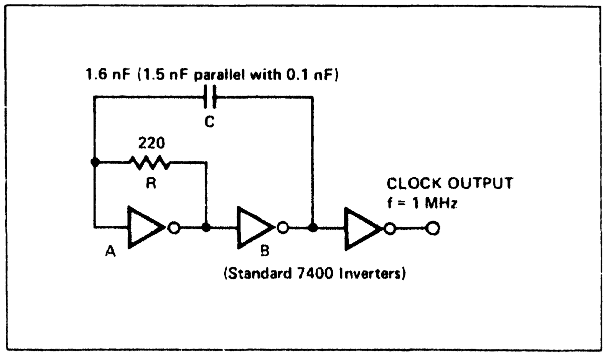
CLOCK CHARACTERISTICS AND SYSTEM-CLOCK PERIODS

In this section, we expand on the clock as a basic feature of a state machine's system architecture. The system clock is the timing element that causes the processor to proceed from one state to another and provides the coordinating signal that causes those memory devices, enabled by the control system, to accept information. It causes the microcoded control signals to take effect system wide. Experience has shown that the beginner often has difficulty in distinguishing between the nature of the control signals that exist *during* a clock period and events that are *initiated* at the end of it. It is necessary to perceive the differences between control signals, which exist in this clock period, and the state of the information, which is available in the next clock period, to grasp the significance of system timing. Information is captured in memory devices, many of which respond differently to the same clock signal.

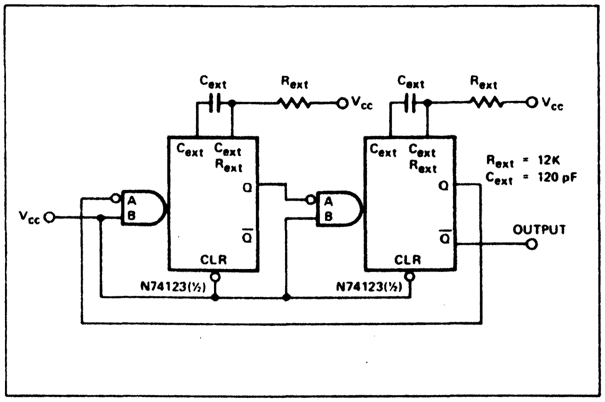
There are several types of sequential machines, but we deal only with the *synchronous* (clock-controlled) machine—the category into which almost all processors fall. In this section, we introduce the fundamental concepts and terminology associated with system clocks, to underpin what follows. The clock is the feature that paces the events within a system. The relationship of these preliminary clock details to the *system's* total behavior—and the clock is an *essential* part of it—are important.

CLOCK PHASES

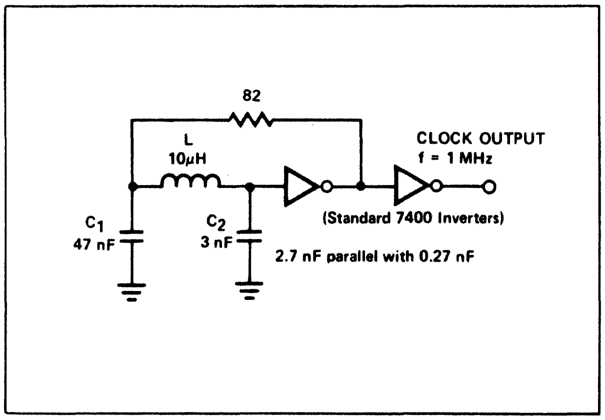
There are single-phased as well as multiphased clocking systems. Internally, most MOS devices require more than a single clock signal to proceed through a basic cycle of operation. This becomes apparent if we review Figure 2-24, which illustrates a dynamic memory cell. The relationships between the transmission gate Q3 and the inverting gate Q1 are such that Q3 must first have established useful information in the form of a charge on C before Q1 can pass on valid information. This multistep approach to first gating and then passing on information gives rise to the use of multiphased clocks in MOS devices. The tendency now is to have a TTL voltage-compatible *single-phase* clock drive a system and then to derive multiphased signals from it internally, if required. Let us look at a single-phase clock signal first. This is the type of clock signal that we shall use later. We shall also indicate how multiphased clocks may be derived from it.


1.6 nF (1.5 nF parallel with 0.1 nF)
C
220
R
A
B
CLOCK OUTPUT
f = 1 MHz
(Standard 7400 Inverters)

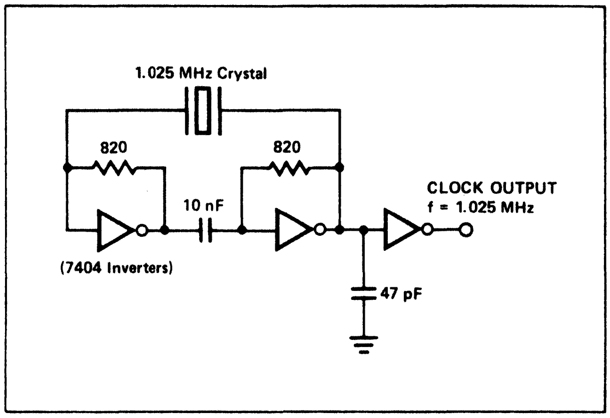
RC Clock Generator



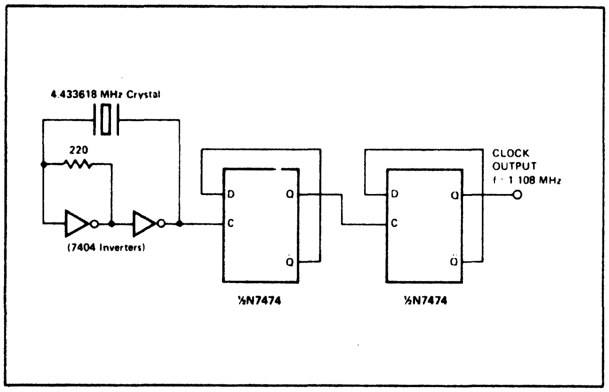
RC Clock Generator with Monostable Circuit N74123



LC Clock Generator



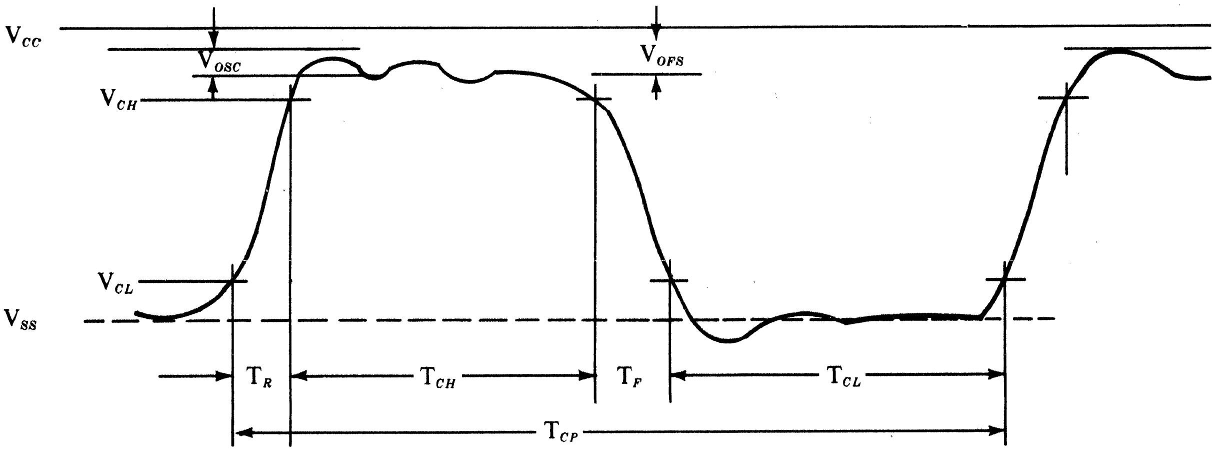
Clock Generator Using a Non-TV Standard Crystal



Low Cost Color TV Crystal Clock Generator

CIRCUIT TYPE	STABILITY					
	(4.75V to 5.25V)			(0°C to 70°C)		
	0°C	25°C	70°C	4.75V	5.0V	5.25V
RC	+0.12% -0.42%	+0.52% -0.98%	+0.2% -1.1%	+1.51% -2.96%	+1.91% -3.24%	+2.48% -3.35%
RC MONO-STABLE	+0.00% -0.014%	+0.276% -0.373%	+0.826% -0.833%	+1.62% -0.53%	+2.01% -0.98%	+2.29% -1.53%
LC	+0.05% -0.08%	+0.07% -0.07%	+0.03% -0.04%	+0.92% -1.31%	+0.95% -1.26%	+0.94% -1.24%
CRYSTAL	+0.0003%	-0.0001%	+0.0002%	+0.001%	±0.0001%	+0.0004%

Figure 3-13
Typical Clock Circuits and Relative Stabilities
(For example only. Data may be old and obsolete. © 1976 Signetics Corporation.)



TERMS

V_{SS} SUBSTRATE VOLTAGE

V_{CC} SUPPLY VOLTAGE

V_{CL} MAXIMUM CLOCK LOW VOLTAGE

V_{CH} MINIMUM CLOCK HIGH VOLTAGE

V_{OSC} VOLTAGE OSCILLATION RANGE

V_{OFS} VOLTAGE OFFSET FROM V_{CC}

T_R RISE TIME

T_F FALL TIME

T_{CH} CLOCK HIGH TIME

T_{CL} CLOCK LOW TIME

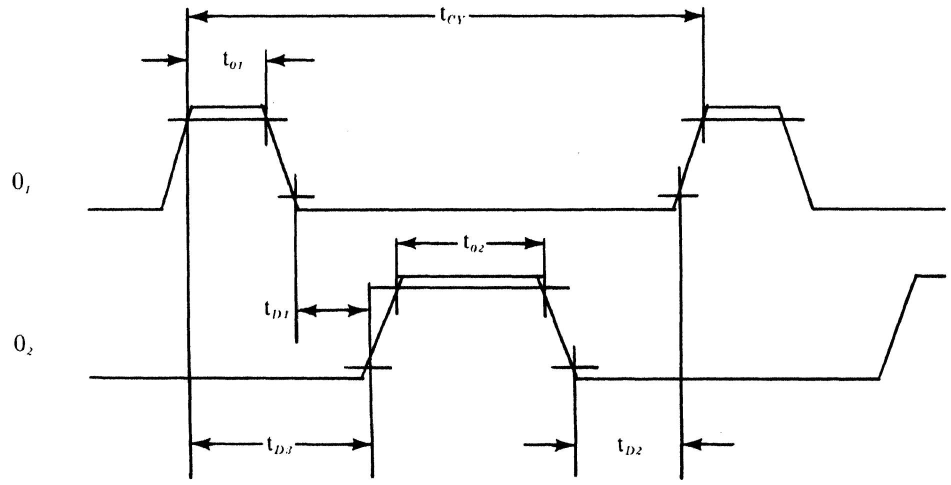
T_{CP} CLOCK PERIOD

Figure 3-14
Single-Phase Clock Waveform and Terminology

SINGLE-PHASE CLOCKS

Single-phase clocks are often made from cross-coupled inverters, acting within a Resistor/Capacitor oscillator circuit. If the capacitance in the circuit is supplied by a crystal, then excellent voltage, temperature, and time stability are achieved. Figure 3-13a shows some typical clock circuits, and Figure 3-13b summarizes their stability characteristics. Note that the stability of the crystal-controlled oscillator is far superior than the others; it is the most widely applied type for this reason. These characteristics are particularly important where the system clock not only drives the processor, but provides timing-reference signals for communication devices and CRT-display systems as well. Where this is done, the high system-clock frequency is divided down through the use of counters (sometimes called *baud-rate dividers*) to produce the desired reference-frequency from the primary source.

A typical periodic waveform for a single-phase clock is presented in Figure 3-14, along with some of the associated terminology. This figure is a guide to clock terminology and clock features that a designer might most be concerned with. Semiconductor fabrication technologies may be critically sensitive to different aspects of the basic clock features. For example, V_{OSC}, the voltage oscillation range of the clock, can cause spurious internal clocking in processors, if not kept within



t_{CY} = CLOCK PERIOD
t_{01} = 01 PULSE WIDTH
t_{02} = 02 PULSE WIDTH
t_{D1} = DELAY–01 TO 02
t_{D2} = DELAY–02 to 01
t_{D3} = DELAY–01 to 02 LEADING EDGES

Figure 3-15
Two-Phase Clock Waveform

tight limits. In the field of microprocessors, the tendency now is to provide on-board oscillators, letting the consumer provide only the crystal and resistor to modify the frequency of operation. This is the result of many expensive and painful experiences in the search for a clean clock signal. Another (expensive) alternative is to use commercial, proven, oscillators where on-board oscillators are not provided. Figure 3-14 does not present industry standards, just some commonly used terminology for describing clock wave forms. For instance, a specific manufacturer may measure the clock period T_{CP} from corresponding V_{CH} to V_{CH} over the waveform, instead of as shown. Becoming familiar with this terminology may induce culture shock among the nonhardware-oriented. Still, it helps to acquire a practical awareness and vocabulary in the age of personal computing–in which we are *all* involved in both hardware and software.

Two-Phase Clocks

An idealized waveform for a two-phase clock is shown in Figure 3-15. This illustration indicates that some older systems must be supplied with two separate but coordinated, nonoverlapping clock signals. The 8080 microprocessor utilizes a clock of this type. The pain of working with multiphased clocks has led to the predominant use of single-phase clocks and internal generation of multiphased signals from them. Most important, though, is to realize that the effective transfer of data can occur only if it is timed properly with respect to this basic system waveform–the clock. The complex timing waveforms presented in microprocessor data manuals for a variety of system operations are based upon this signal, which drives the entire system. When our system is in trouble–and usually as a last resort–we refer to the timing diagrams to begin to grasp how the system cycles. At that point, we will observe if any of the timing relationships are incorrect.

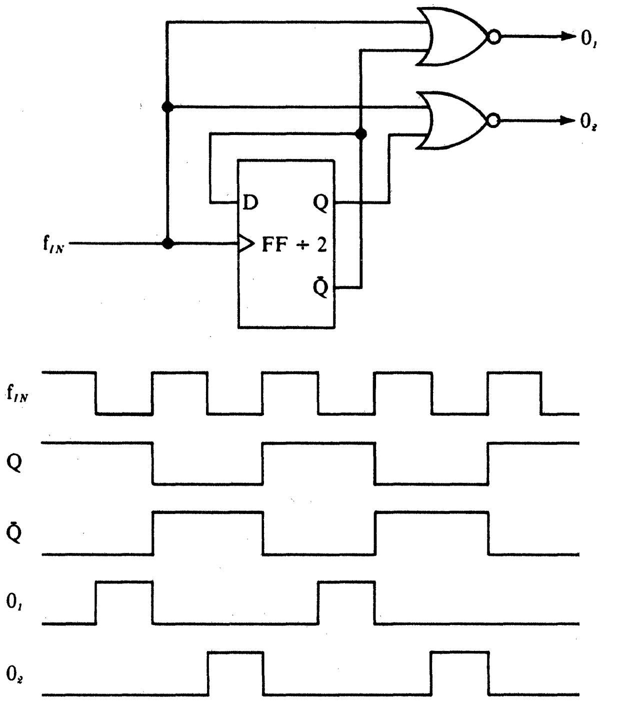


Figure 3-16
Counter-Phase Generation

COUNTER–DECODER CLOCK PHASE GENERATION

How are MOS devices multiphased internally when the system clock is single-phased? The internal generation of multiple phases can be achieved through shift-register counters and decoding or through edge detection. An example of a counter-decoder is presented in Figure 3-16. The input frequency is some multiple of the system-clock frequency in this case, and it is used to define the basic system period, which includes more than one phase. If we follow the signals of the timing diagram in the figure, we observe that the flip-flop alternately enables first one gate and then the other. Recall that the NOR gate is a device whose output is clamped low whenever any of its inputs is high. When one of the gates is enabled by the flip-flop, its output still depends on the level of the input frequency signal. The net result is a multiphased clock, where each phase pulse is separated in time from those of the other phase. This is referred to in the industry as *nonoverlapping phases.*

EDGE-DETECTION CLOCK PHASE GENERATION

The generation of multiphased clock signals by edge detection is interesting because the method has other applications, as in data-communications circuits, and also displays a basic property of real gates—the memory property associated with propagation delay. Figure 3-17 illustrates the generation of a pulse from an input-signal level change. Edge detection is a case where a "glitch" due to propagation delay can be an ally.

Normally the NAND gate of Figure 3-17, if analyzed ideally, would always have one input low. If this were actually true, then its

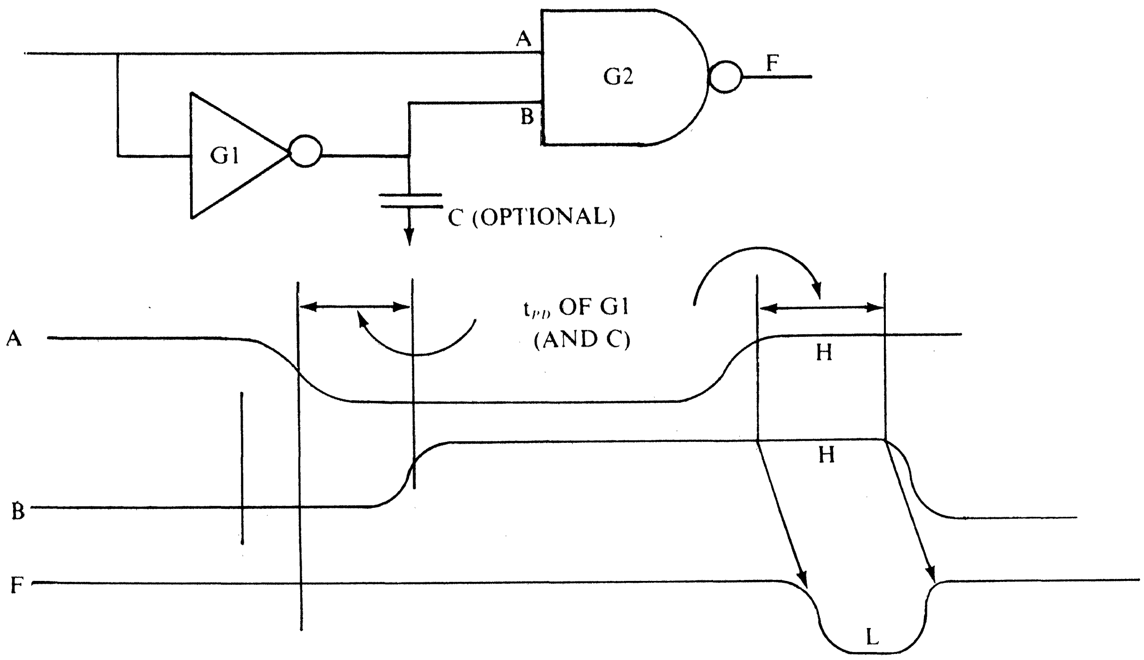


Figure 3-17
Positive-Edge-Detecting Pulse Generator

output would always be high. Instead, the input and stray capacitances (or one deliberately provided) of the inverting gate G1 creates a time delay between the level changes occurring at the A and B inputs to the NAND gate G2, as illustrated in the waveforms. In this case, note that, when the external input to the A line goes high, the B line will remain high for a finite time related to the propagation delay and capacitances associated with gate G1. Momentarily, G2 finds both inputs high and, as a result, goes momentarily low. A pulse is emitted for each rising edge detected. If an EXCLUSIVE-OR gate were used, both rising and falling edges would be detected. In this case, two internal pulses would be emitted for each input clock cycle–a simple derivation of internal two-phase pulses from a single-phase clock or a frequency doubler. These effects, based on propagation delay, can be used to create multiphased clocks from an external single-phased one. For communications circuits, encoded data may be decoded through the use of such edge-detection circuitry.

General Edge-Triggered Clocking Characteristics

Many of a system's memory elements may be so-called edge-triggered flip-flops. These types of memory element capture and subsequently display at their outputs the information present at an input *only* when the clock signal it "sees" goes from one given level to another. *Output information remains static outside the zone associated with these clocking edges.* Depending on the gating used, we say that a memory element captures information on the low-to-high clock transition (positive edge-triggered) or on the high-to-low one (negative edge-triggered). We shall now learn about the phenomena related to the "edge" of a clock. This has an important bearing on *how* and *when* and control signals are set up and on *when* information is captured within a system.

The terminology associated with the clocking of memory elements is *always* extremely important to us. Figure 3-18 presents the

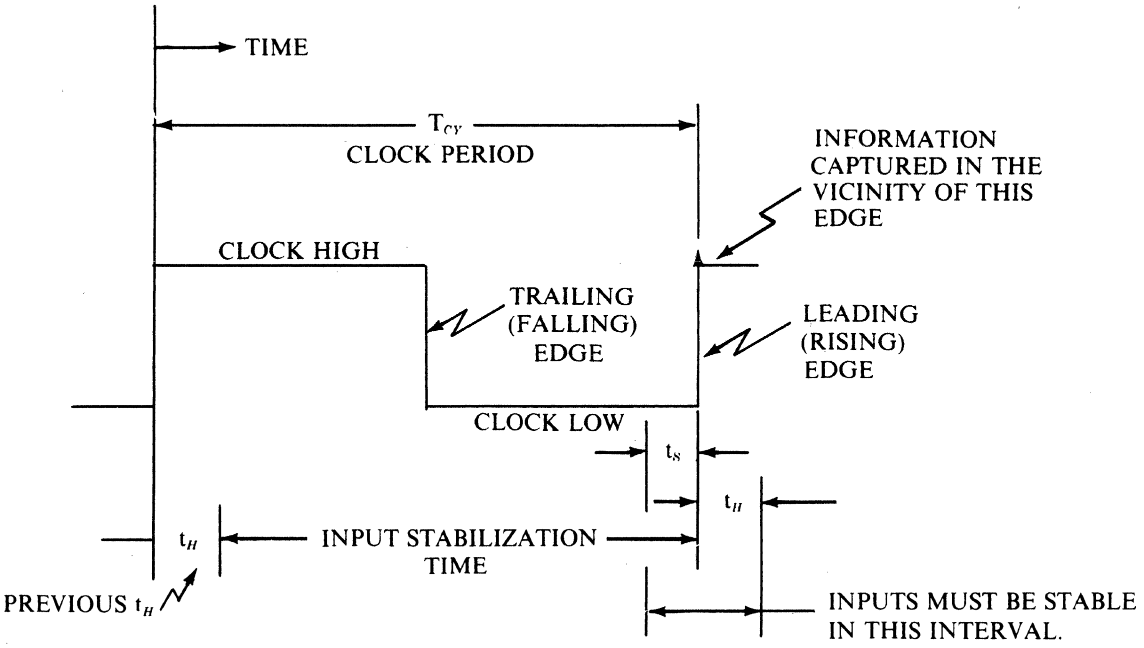


Figure 3-18
Clock Characteristics for 74LS74 Flip-Flop

idealized waveform of the clock as it applies to the 74LS74 D-type edge-triggered flip-flop. Its specifications, found in the data catalog, provide insight into how a processor moves from one state to another. Use of the TTL data catalogs helps the software-oriented reader begin to understand the nature of computing machinery. Data catalogs are as important as any text–including this one. Eventually, one graduates from the text, which provides the initial basic systems insight, to becoming literate in the concepts found only in the advanced data catalogs. In these, a computer system's processor and programmable support IC's are described in industrial terminology. Both hardware and software details, essential to the effective use and development of processing systems, are presented.

Clock Setup and Hold Times

One way to begin is by looking at the clock features of the 74LS74, as derived from the very basic TTL data sheets. The waveform shown in Figure 3-18 introduces additional terminology associated with clocking. We now define some new parameters, including the set-up (T_S) and hold (T_H) times:

Definition (T_S)

The set-up time is defined as the amount of time *before* a referenced clock edge, by which time the input data must have become stable. It is to remain stable until the end of the T_H period, next. These times and the associated clock transition (edges) are specified in the data catalog for a particular flip-flop.

Definition (T_H)

The hold (sometimes referred to as *release*) time is defined as the amount of time *after* a referenced clock edge by which the input data (being held stable) may now be released.

Definition (Stable Time)

This definition does not appear in the data catalogs, but it is in effect. The input data must be held *stable* from the beginning of the set-up to the end of the hold time.

Definition (End of Current Clock Period)

Again, this definition is not given in the date catalogs but must be understood. The end of the current clock period for a given device is identified by the clock edge that T_H is referenced to in the data catalog.

The waveform of Figure 3-18 is based on the 74LS74 flip-flop. The input set-up time T_S for this flip-flop is specified in the data catalogs with respect to the rising edge of the clock. It is the time by which the input data must have become stable *before* the clock signal goes high, to be reliably entered (stored) into this memory element. Note that, for this flip-flop, this time period is defined with respect to the rising edge of the clock. The story is complete only when the hold time T_H is accounted for. In this case, T_H is specified as the time the input data must continue to be held stable *after* the clock goes high–if the data is to be reliably recorded. Data-input changes that take place outside of the region defined by the total $T_S - T_H$ time period *and the edges they are referenced to* do not affect the integrity of recorded data. Signals can safely vary in this time zone without destroying the desired sequential behavior of the system. Note that T_S and T_H need not be referenced to the same clock edge, as in this case. The input data, of course, had better be both stable and correct at the point T_S is reached. The data catalogs often display either an up or down arrow to signify whether a referenced clock edge is the high- or low-going one.

Edge-Triggered Clock Characteristics

There are a number of variations on this theme. In the exercises, the reader is asked to sketch the basic clock waveforms for a variety of flip-flops. It is necessary to understand this phenomenon, in all of its varieties, to understand clocked synchronous sequential behavior of a system containing elements displaying different clock behaviors. What happens in the operation of a sequential circuit is this: The control system's data-path and register-clock-enabling signals, as well as the data to be recorded, all generate inputs to a system's memory elements. The input data must stabilize in the time period *beyond* previous T_H but *before* the T_S of the current period is reached. At this point, they are properly entered into those memory elements that the control system has enabled to receive the system clock pulse. In the case of the 7474, this takes place with respect to the rising edge of the clock. The input data signals *must* also be *held stable* for the duration of the total T_S – T_H time interval. Keep in mind that the clock edge to which T_H is referenced also *defines the end of the current clock period.* The newly entered information becomes available for observation, at a clock-enabled memory element's outputs, *only* in the *next* clock period. There it, too, must stabilize before the next T_S zone is reached.

This, then, is a two-step process:

1. Control (enabling) and data signals are stabilized in the time interval between the past period's T_H and the current period's T_S
2. The memory elements will properly record data and system-state transition information held stable *during* the $T_S - T_H$ time interval beginning at the *end* of the current period. Note that this stored information may be viewed at the *outputs* of these flip-flops (or other type of memory storage) only in the *next* period.

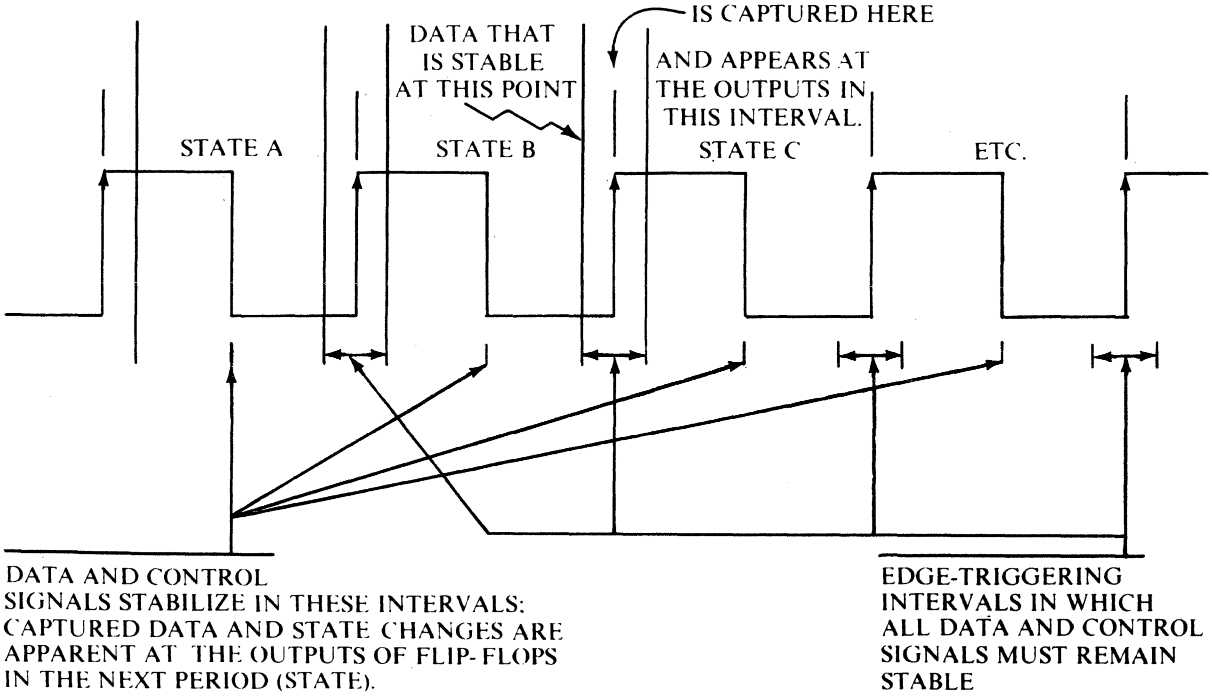

IS CAPTURED HERE
DATA THAT IS STABLE AT THIS POINT
AND APPEARS AT THE OUTPUTS IN THIS INTERVAL.
STATE A
STATE B
STATE C
ETC.
DATA AND CONTROL SIGNALS STABILIZE IN THESE INTERVALS: CAPTURED DATA AND STATE CHANGES ARE APPARENT AT THE OUTPUTS OF FLIP-FLOPS IN THE NEXT PERIOD (STATE).
EDGE-TRIGGERING INTERVALS IN WHICH ALL DATA AND CONTROL SIGNALS MUST REMAIN STABLE

Figure 3-19
Data and Clock-Period Timing Relationships

T_S and T_H are not the only signals referenced to a clock edge. Timing diagrams of processors, including the programmable peripheral support IC's, contain many such types of relationships that must be maintained for proper operation to occur. This has been just a bare introduction to the essentials of timing diagrams.

This is the manner in which sequential devices record state transitions and information that is entered into memory devices, as orchestrated by the control system's logic. The dual concept of, first, a period for input-data and control-signal stabilization, which is coupled to a memory-element-recording interval, into, second, the next period, where stored results are used again is what makes the system perform. For some reason, there is a tendency to fight accepting the concept that data or desired next states being entered as *inputs* in this time frame will be available at the *outputs* of memory elements *only* in the next period. We are dealing with an input-output relationship. It is to be hoped that we will never be confused again with respect to this important timing concept of sequential behavior. This pattern is illustrated again in Figure 3-19 for changing data being entered into a single 7474 D-type of flip-flop over several clock periods. Note that changes of data outside of the $T_S - T_H$ stable interval have no effect on the output of this device.

LEVEL AND EDGE CLOCK CHARACTERISTICS

However, some other major variations in memory element clock behavior require mention. The first is that some devices are designed to capture information when the clock goes from high to low. The 7476 flip-flop is a case in point. An inversion of the system clock used before, driving this device, would cause it to change state at the same time as a 7474 would change state if driven by the noninverted clock. Not all versions of the 7476 have the same $T_S - T_H$ specifications. The

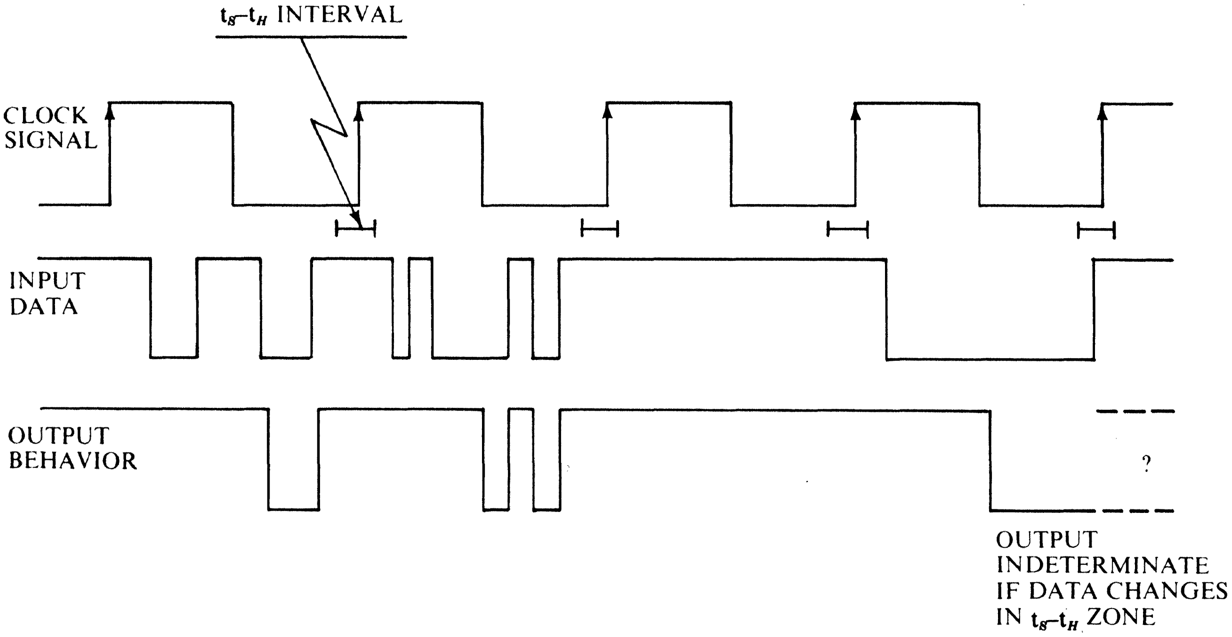


Figure 3-20 Clock-Data-Output Relationships (Transparent Latch)

74LS76 is a strictly edge-triggered device. That is, both T_S and T_H are measured with respect to the same (falling) clock edge. The 7476 and 74H76 devices require that the input data be stable *while* the clock is high. Consulting the data sheets shows that this is the same $T_S - T_H$ stability criterion we described earlier–except that T_S and T_H are now separately measured with reference to two different clock edges, the rising and falling clock edges, respectively. In these cases, the information must be stable during the *entire* interval that the clock is at one level, as defined by the T_S–T_H time interval. This is not strict edge-triggered behavior, but rather both level and edge behavior.

Transparent Clock Characteristics

Yet another important variation in clock behavior is the type of clocked memory element referred to as the *transparent latch.* MOS memory, internal microprocessor registers, and the 74LS670 register array all exhibit this type of behavior. We discuss and use both the 670 and MOS memory later. The behavior of these devices with respect to the clock signal is summarized in Figure 3-20, which is representative of the 74LS670. Here, when the clock is high (*inactive* in this case), the memory element remains stable regardless of the data-input variations. When the clock is low (*active* in this case), the output *follows* input changes, separated in time only by the propagation delay of the device. This gave rise to the name *transparent latch.* In this case T_S and T_H apply to the same edge. For the 74LS670, this is the low-to-high clock transition. Data is reliably captured so long as it does not change during the $T_S - T_H$ interval associated with the rising clock edge. This type of device is simpler to build in the silicon IC, which accounts for its frequent use. The transparent behavior, when the clock signal is at the *active level,* creates problems in applying these devices to synchronous systems. We shall complete our look at these problems in Chapter 5, where the behavioral characteristics of the transparent latch will finally be resolved in a systems context.

The three major variations in clocked behavior of memory elements that we have studied are true *edge*-triggering, *level* and *edge*-triggering, and the *level* and *edge-capture* behavior of the transparent latch. All three variations of these clocked behaviors may be observed in practice, even within one system. The data sheets and problems provide insight for our future study of the operation of a CPU. More abstract topics, such as pipe-lined operation, require an understanding of the effects of the clock-signal behaviors briefly described here. In all cases, the relation between the enabling and input-data signals of the *current* period and the subsequent availability at the outputs in the *next* period (after the T_H edge of reference) is the essence of synchronous sequential performance.

BIBLIOGRAPHY

Barton, R.S. "A New Approach to the Functional Design of a Computer." *Proc. WJCC,* 1961, pp. 393–96.

Cleary, J.C. "Process Handling on the B6500." *Proceedings of the Fourth Australian Computer Conference,* Adelaide, South Australia, 1969, pp. 231–39.

80/85 Family User's Manual. Santa Clara, California: Intel Corporation, 1983.

TTL Data Book, The. Dallas, Texas: Texas Instruments, Inc., 1976.

TTL Data Manual. Sunnyvale, California: Signetics Corporation, 1984.

PROBLEMS

1. Refer to the register array of Figure 3-2. Assume that the DMPX function employs the clock-enable technique for selectively writing into a register and that the MPX function is performed using tri-state interfacing. Draw a logic diagram of a four-register array with four bits per register, using IC's from the TTL data catalog.
2. Make a list of the different types of registers found in a TTL data catalog. List the names of any new interface signals not already presented in Figure 3-1a. State the function of each such signal.
3. Draw the circuit of a NOR-gate-based clock-enabling circuit that has, as inputs, the system clock waveform and an active-low clock-enabling signal. The resulting enabled clock pulse is to have the same waveform as the system clock of Figure 3-18.
4. List both the similarities and the differences in machine operation between a CALL and an INTerrupt. Assume the machine uses a stack pointer.
5. Describe how the program counter is saved on a stack in memory during a CALL instruction.

6. Verify the operation of the CALL instruction by consulting a processor's hardware manual, such as the Intel *80/85 User's Manual* or the DEC *PDP-11 Architecture Handbook*. The PDP-11 computer uses the Assembly-language notation *JSR PC, SUB* for the form of the CALL instruction discussed here.

7. Describe why bank switching capabilities can be useful in the development of instructions and software for subroutine calls.

8. Complete the problems shown on the ALU work sheet of Figure 3-21. Do any of these problems produce incorrect results because of overflow?

9. Define both the control words for the 74181 ALU and the resulting output signals that would be used in establishing the existence of the following inequalities. Do this for both active-low and active-high data conventions:

   a. $A \leq B$

   b. $A = B$

   c. $A > B$

10. The introductory architectural examples in this chapter display a limited number of functional blocks, out of which an architectural block diagram of a computer is drawn. Make a list of these functional blocks.

11. Suppose that the special ACC register were eliminated from the architecture of Figure 3-11. How would this affect the manner in which the register array must be used? How many clock periods would now be required to perform the problem in the third section of this chapter?

12. Modify the data-flow paths (bus directionality and connections) of Figure 3-11, for the purpose of improving the performance shown in Example 2 of the third section in this chapter. Can the problem be performed in fewer steps?

13. In the architectural examples of the second and third sections in this chapter, data-selection interfaces are not explicitly described. They seldom are. Identify which functional components of Figure 3-5 should have tri-state or open-collector type of multiplexer interfaces. Discuss the reasons for your choices.

14. The architecture of Figure 3-12 contains an Operand Address (OA) register, with auxiliary simple ALU, in the memory-addressing unit. The OA register contains the address of the operand during the execute phase of operation. Describe how this could be advantageously used for the transfer of *blocks* of data. Would you wish to enhance the simple ALU's performance to create an instruction for block moves? Discuss.

15. The memory-addressing unit of Figure 3-12 is a simple example of the current trend toward the formation of separate semiautomatic functional units within a CPU's architecture. Cite examples of this practice found in modern 16/32-bit architectures.

	DATA ACTIVITY LEVEL	ALU FUNCTION PERFORMED	FUNCTION						A BUS DATA LINES				B BUS DATA LINES				F BUS DATA LINES				C OUT
			MODE	SELECT LINES				C IN													
			M	S3	S2	S1	S0	C_N	A3	A2	A1	A0	B3	B2	B1	B0	F3	F2	F1	F0	C_{N+4}
A)	AH		L	L	H	H	L	L		–	1			–	3						
B)	AL		H	L	H	H	L	H	0	1	1	0	1	1	0	0					
C)	AH	A pl A pl 1								–	5			+	4						
D)	AL	$\overline{A \cdot B}$							1	0	1	1	1	1	0	1					
E)																					
F)																					
G)																					
H)																					

METHOD: FILL IN BLANKS WITH APPROPRIATE H OR L AFTER CONSULTING ALU TABLES. FILL IN MISSING ALU FUNCTIONS. SHOW WORK METHOD BELOW.

Figure 3-21
74181 ALU Worksheet (Filled In)

16. Assume a clock frequeny of 1 MHz for a system clock. This clock has a 50-percent duty cycle (high 50 percent of the time). From the data shown in your TTL data catalog, calculate the stabilization times for a 7473 flip-flop and for a 7474 flip-flop. Take into account the effects of the set-up and hold or release times given there. Which of the two flip-flops is capable of the greater speed of operation?

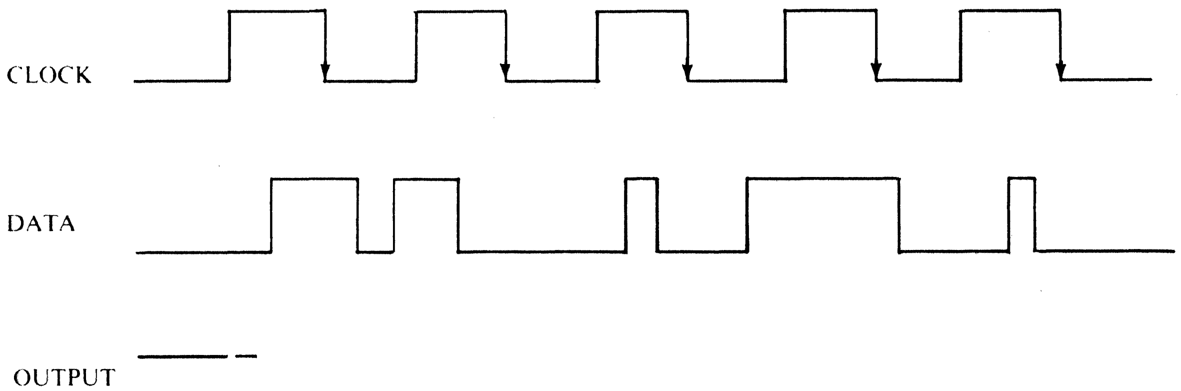
17. Using the TTL data catalogs, determine the greatest speed of operation allowed for the 7476 and 74LS76A flip-flops. This requires that you account for all the timing parameters related to a clock cycle for each of these devices.

18. In the TTL data catalog, find and list three IC's that exhibit transparent-latch behavior.

19. Given the clock and input signal behavior shown in Figure 3-22, determine the output response of a hypothetical transparent latch whose clock is active when high.

20. A major theme of chapters 2 and 3 has been that the functional blocks of an architecture are derived from simpler basic components. Starting with an inverting logic gate, list

the successively higher level components that can be created with its use, to arrive at the functional structure called a *general-register array.*



NOTE: ASSUME CLOCK IS ACTIVE WHEN HIGH FOR THIS TRANSPARENT LATCH.

Figure 3-22
Clock-Data Timing Waveforms

CHAPTER 4
SEQUENTIAL-MACHINE FUNDAMENTALS

MICROPROGRAMMING'S BASIS IN STATE-MACHINE THEORY

In chapters 2 and 3, we reviewed many of the hardware-related fundamentals required to understand the *functional* behavior of the devices that make up a computer architecture. System organization was stressed, but primarily from the viewpoint of how actual hardware elements are combined in a structured, rational way to form a system. Part of the understanding of a processor, it was pointed out, strictly depends on a growing familiarity with the functional behavior of a very few integrated circuits–and how they are used together when we form a CPU system. Here, we begin a review of the fundamental concepts that *control* the behavior of the CPU–that is, the *state machine* that forms the control system of a processor.

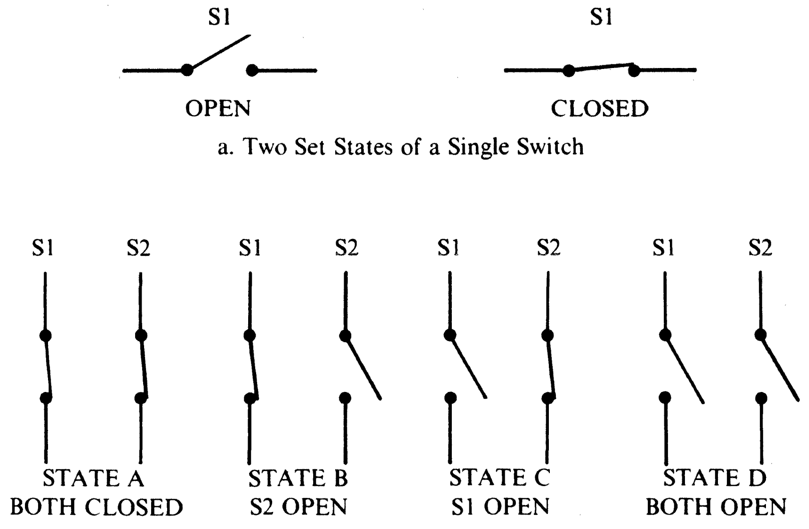
PROCESSOR = CPU + CONTROL STORE

Actually, the conceptual behavior of processors is often presented as if there are two entities. The first is the CPU, and the second is its control system, or Control Store (CSTR). Taken together, they really form the single entity we call the processor, or computer. The CPU half relates to how the hardware registers that the programmer can explicitly affect actually function and how information is physically moved and altered within the CPU. This half of the system generally forms the *user's* perspective of what a particular machine consists of. The science of computing requires that we also understand the control-system portion of the total machine and, further, how the software capabilities relate to the instruction set that the CSTR implements. This is the area dealt with when a machine is microprogrammed. Our knowledge of the CSTR and instruction set's potential capabilities depends in turn on our understanding of the first principles of sequential machine behavior. The term *microprogramming* is often used to imply the means by which modern computers are controlled or (equally) how instruction sets are implemented. Let us therefore state at the very outset that *a microprogram is nothing more or less than a state table for a sequential machine.*

STATE-MACHINE ASPECTS OF MICROPROGRAMMING

The goals of this chapter, then, are to provide an understanding of what a state table is and to discuss the implementation of a simple state machine. We will examine the physical details of both a CPU and its control system in the following chapters, covering how to construct them and how to create (microprogram) the instruction set of a processor. In this chapter, we review the theoretical system principles neces-

Figure 4-1
Concept of State and Switch Settings



a. Two Set States of a Single Switch

b. Four Set States of Two Switches

sary to an understanding of processor control-system behavior. *Automata theory* is the theory of state machines. It is fundamental to both computer hardware and software sequential-machine structures. For example, both compiler and processor design utilize its tools to implement state systems rationally. We shall examine a small, hard-wired, sequential circuit in this chapter. Its state-machine principles of representation and operation are the same as for the model computer we shall study in chapters 5 and 6. The methods of implementation differ, but these underlying principles form a unifying picture for the organization of control systems. These principles are also the basis of microprogramming.

SEQUENTIAL-MACHINE REPRESENTATION

CONCEPT OF STATE

Sequential-machine (SM) representation requires a good deal of abstract visualization of the concept of *state*. We know and feel the state of our emotions or health. Each of us knows from experience that a single entity (oneself) can experience more than one state of being. While a simple switch does not have emotional states (unless it is broken or unstable–and the emotions are our own), it does possess two states, shown in Figure 4-1a. A switch is either *open* or *closed*. Given two switches, we can produce four unique states of the total system of switches, as indicated in Figure 4-1b: both are closed; both are open; switch A is closed, and switch B is open; and vice versa.

We noted in Chapter 2 that, logically speaking, a transistor, as used in a logic gate, behaves as a switch. It has two states, *on* and *off*. It was also shown that, by cross-coupling two inverting transistor gates, we could produce a cell, the latch or flip-flop, that had the ability to "remember" (record) its past state or to change its current one, depending on the external inputs. This we termed a *memory element*. We saw too that there are many types of memory elements. A register is simply a linear array of some type of read/write memory elements. Each cell in the array is a binary device, with the memory-retention property, under external control, of being able both to accept and to retain information in the form of its own state. The primary output, often labelled Q, has only two states. Q is either electrically high (H) or electrically low (L), depending on whether the transistor switches associated with it are *on* or *off. Within a sequential machine there is always a special set of these memory elements whose outputs define its current state.* These are called its *State-Sequencing Register* (SSR).

State-Sequencing Register

At the heart of every SM are two classes of hardware: *Combinational Logic* (CL) and the memory element(s) of its SSR. Combinational logic has the property that its outputs are *always* an immediate consequence of its present inputs. Further, for a given input, the output response is always the same. This does not mean that an input change always produces an output change. It means that, if the logic design is such that, for a change in input state, the output state is to change, then it does so immediately. The basic elementary latch behaves in this manner, too, since it is not clock-controlled. This form of sequential behavior is often termed *level-mode,* or *asynchronous,* operation. Yet it is different from CL, in that it has an input state that causes it to retain its last output state and, in particular, in that the same external input combinations do not always produce the same next state. The pulsed-mode complementation of the latch illustrated this. In the latch, this memory property arose out of the propagation delay of its internal gates and their manner of use. Thus, we observe that level-mode sequential circuits, while different from CL, can also change state as an immediate consequence of input changes. This is not a convenient mode of operation for sequential machines.

Need for Synchronous Operation

Level-mode operation, while it is the internal basis for the operation of every flip-flop, is very difficult to work with in practical machines. To avoid the problems encountered in level-mode operation, the clocked, or *synchronous,* flip-flop was developed. Clocked or synchronous operation means that, if a flip-flop (memory element) is to change state—as dictated by its present inputs—it does so *only* in coordination with the clock waveform. The term *synchronous* does not imply that the clock signal consists only of a waveform with regular intervals. It means only that, if a flip-flop is to change state, it does so only when the clock signal *triggers* it—which could be aperiodic or periodic in nature. That is, changes in state are synchronized to changes in the clock-signal level and not to changes in inputs. External inputs influence the choice of the next state, but not when they occur. In synchronous operation, the clock signal establishes when a state change is to occur.

Because synchronous memory elements are simpler to control and far more reliable in practice, they are widely used in computers. As noted previously, for reliable operation the rate of state changes controlled by the clock must be slow enough to permit the input signals to a flip-flop to stabilize completely. This is why we studied the clock characteristics of T_s, T_h and the associated required stabilization time in the previous chapter. Also, recall that any input condition that causes a change is observed at the output of a flip-flop only *after* the clock has triggered it, or in the *next period.* For practical reasons, then, we will deal with synchronous sequential circuit behavior in the following material. Fortunately, except for details, the same principles are found in all SM, both synchronous and asynchronous. These behavioral characteristics of synchronous systems form the foundations for an understanding of microprogramming operations.

In our discussion here, the SSR of a sequential machine consists of one or more synchronous flip-flops. Each flip-flop possesses two states. Therefore the total number of states that an SSR can possess (some may not be used) is called the *state set* of the machine. It is often symbolized by the letter Q; it is dependent on–but not to be confused with–the Q output of an individual flip-flop. The maximum number of individual states m in a state set is

$$m = 2^n; \text{ where } n = \text{number of cells in the SSR}$$

Physical Significance of States

While we have been dealing with a mechanistic approach to the concept of state in an SM, showing that a particular state is related both to the *number* of *memory elements* (switches) in the SSR and to their *current setting,* it is extremely important to realize that, for a given problem, *these switch settings have physical significance.* That is, the current switch settings (state) are a reflection of the real events that have occurred in the past. We will emphasize this important point again soon. The goal will be to focus on the *meaning* of a current state.

Sequential-Machine Representation

First, let us look at some classical models for SM's. The most general model for an SM is the Mealy model, shown in Figure 4-2. The presence of a clock signal in our illustration indicates that we are interested in synchronous operation. Any SM is completely described by five characteristics, or *5-tuple,* of an SM. That is, we say

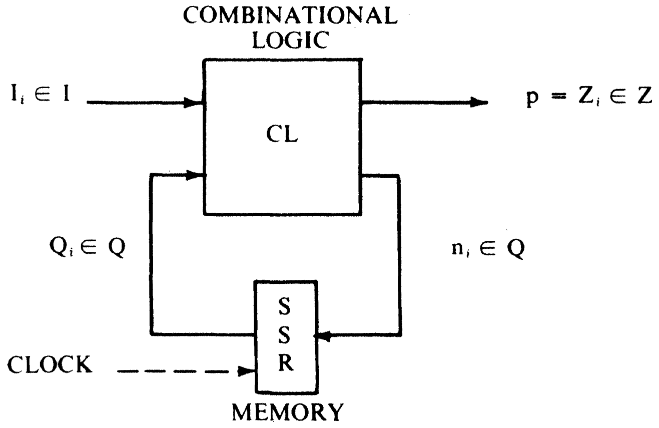
$$\text{SM} = <\text{I, Q, Z}, n, p>.$$

Five-tuple SM Representation

These characteristics may be associated with the features of Figure 4-2. They are

1. A finite set I of external input symbols.
2. A set Q of internal states. Finite sets are considered here.
3. A finite set Z of output symbols.
4. A mapping n of I × Q into Q called the *next state* function.
5. A mapping p of I × Q onto Z called the *present output* function.

Figure 4-2
Mealy Model of an SM



Cartesian Product

Note: I × Q is called the *Cartesian product* of the sets I and Q. The state tables we shall work with are displayed in Cartesian-product form. It is defined in the following way. Let S and T be any two subsets of a universe U. The set

$$S \times T = \{(s,t) \mid s \in S, t \in T\}$$

is the Cartesian product. The order of the elements is important, and the resulting product operations are therefore not commutative, but they are associative. This product operation can generate sets that are not in U. It is useful because it provides a way to generate new sets. Our concern here is to generate both the set of next states in Q and the new set of present outputs in Z for an SM, from the elements of I and Q. Formalities aside, we eventually want to produce these sets, using the tabular method of representation natural to Cartesian product representation.

The 5-tuple above completely describes a machine's behavior when all the characteristics in it are fully specified. The set I of inputs in the figure is the set of all possible inputs. The set Q is the set of all possible states the machine possesses. The number of cells in the SSR determines the maximum size of this set, though not all elements need be used. The finite set Z is the set of output symbols the machine is capable of generating. One mapping, I × Q onto Q, produces *n*, the next state table. Its symbols are next state prediction that will excite (or steer) the cells of the SSR along the path of desired behavior when we are specifying its state behavior. The results of this mapping are always an element of Q that is realized in the next period (after the excitations of the current clock period have taken effect). The second mapping, I × Q onto Z, produces the present output, an element of Z. This output is utilized by the SM that is the control system of a processor to manage the behavior of the slave CPU. Through this mapping we may specify what we want the CPU to do now, in the current period.

Mealy Machine

The structure of the Mealy-type machine of Figure 4-2 reveals that an SM can be reduced to a block of combinational logic CL, interacting

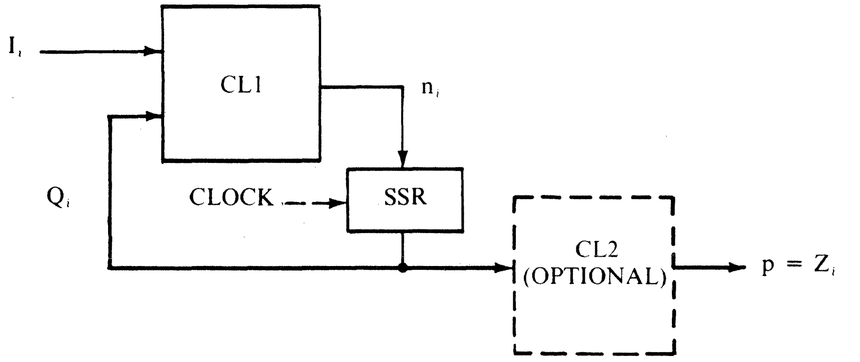
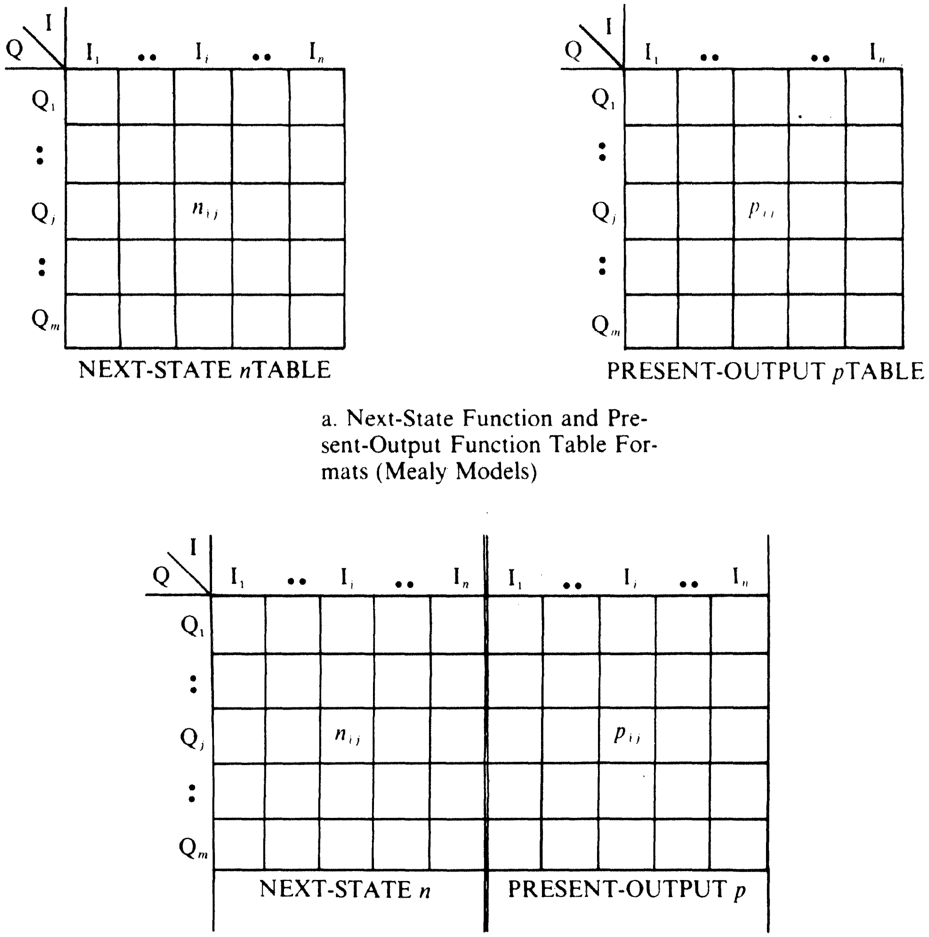


Figure 4-3
Moore Model of an SM

with a block of memory cells, the SSR. From this we see that the present output, an element of Z, say Z_i, is a combinational function of the present input, an element I_j in I, and the present state, an element Q_k in Q. An interesting and sometimes troublesome facet of this organization is that, should the external input change while its internal state remains constant, it is possible for the output to change. This is a result of the fact that the machine's outputs are produced by a block of combinational logic, whose output is always an immediate consequence of the present inputs. The present state behaves differently, since we are dealing with synchronous clock-controlled machines. It changes its value only when ordered to do so by the clock. This means that, if an input changes, the next state excitation presented to the SSR's inputs may change. This change, however, is not entered into the flip-flops until the clock waveform authorizes it. The predicted next state becomes a real present state only in the *next* period. In clocked-state machines, the clock affects *only* the SSR of the state machine, not its combinational logic. Since the present state consists of the outputs of the SSR, the clock controls when these state changes occur.

Moore Model

The fact that the present output of a Mealy machine may react to input changes as they occur can be avoided. A modification of the Mealy model, called the Moore model, is often used so that both changes in state and in output are synchronized with the clock waveform. This pattern of organization is shown in Figure 4-3. The change that this organization produces is that the present output can only be some function of the present state. There may be combinational logic present in the present output path (CL2, if it exists), but it acts only as an encoder of the SSR's present-state value. Since the change in present state is clock-controlled, this pattern of organization results in clock time-coordinated changes to both the present output, *p*, and the SSR. This is a desirable and often used pattern of organization. Here, only the next state is controlled by the block of combinational logic (CL1) that has elements of I and Q as its inputs. Now the present output can only be a function of the present state. The net result of this is that there can be only one output symbol per machine state, regardless of input changes that may occur while within the state.



a. Next-State Function and Present-Output Function Table Formats (Mealy Models)

b. Combined State-Table Format (Mealy Model)

Figure 4-4
Mealy Model: State-Table Format

State-Table Representation

The cross product mathematical model and the Mealy/Moore architectural models of the state machine naturally lead us to the state-table form of representation for sequential machines, Figure 4-4a and b. In Figure 4-4a, the next-state and present-output tables are shown as independent entities. Since both of these function tables are derived from cross product operations on elements in the same sets, $I_j \times Q_k$, Figure 4-4b seemingly combines them into a single table. This form of combined representation is most common and is referred to as the *state table,* which consists of two tables: the next-state (*n*) function table and the present-output function (*p*) table. Any box in *n* or *p* represents the cross product of a particular element I_j in I with a particular element of Q_k in Q. The entries in these boxes are the specific functional values that specify the next state in Q (for *n*) or the present output symbol in Z (for *p*).

When all the boxes in the combined state table are specified, we say that the state table describes a completely specified SM. While there are procedures for analyzing incompletely described machines, the end result of any design process must be a state table

Figure 4-5
Moore Model: State-Table Format

Q \ I	I_1	••	I_i	••	I_n	$p(Q)$
Q_1						$z_1 = p(Q_1)$
⋮						⋮
Q_i			n_{ij}			$z_i = p(Q_i)$
⋮						⋮
Q_m						$z_m = p(Q_m)$
	NEXT-STATE n					PRESENT-OUTPUT p

for a completely described machine. The form of the state table of a Mealy machine was presented in Figure 4-4. The Moore model is more representative of the type of table to be found in a computer's control system, because the generated final value of the single present-output symbol produced each clock period is not affected by input-signal changes (or noise) while within the state. These signals are reacted to only after they have settled down, at the end of the current period. The required settling time required for this output symbol to stabilize after a change in state is what limits the speed of a computer system. The specificity of this form of design is easier to follow. For a given state, the present-output symbol tells the slave CPU portion of the machine what it is to do *now;* the next-state symbol specifies what state its SSR will be during the *next* period. An example of the Moore form of the state table is presented in Figure 4-5.

The Moore form of the state table representation of a sequential machine in Figure 4-5, then, is very important to us for a number of reasons:

First: it is capable of completely describing a sequential machine.

Second: A computer's control system is a sequential machine.

Third: The accepted definition of *microprogramming* is "a rational approach to the design of a computer's control system" (Wilkes).

Last: A microprogram is nothing more or less than a state table for a sequential machine. It most often has the form of representation of a Moore state table.

Later we will develop the instruction set of a simple computer and implement it as a state machine. When we learn how to do this (and it is not hard), we will have mastered the elements of microprogramming. It will be seen that the essence of the form of a microprogram is the form of the Moore state-table representation for a sequential machine. At the end of this chapter, we will design a simple actual Moore circuit. This simple circuit should be constructed and operated as an aid to developing a practical intuitive feel for the operation of a sequential machine (or computer).

SEQUENTIAL-MACHINE VISUALIZATION

STATE DIAGRAM

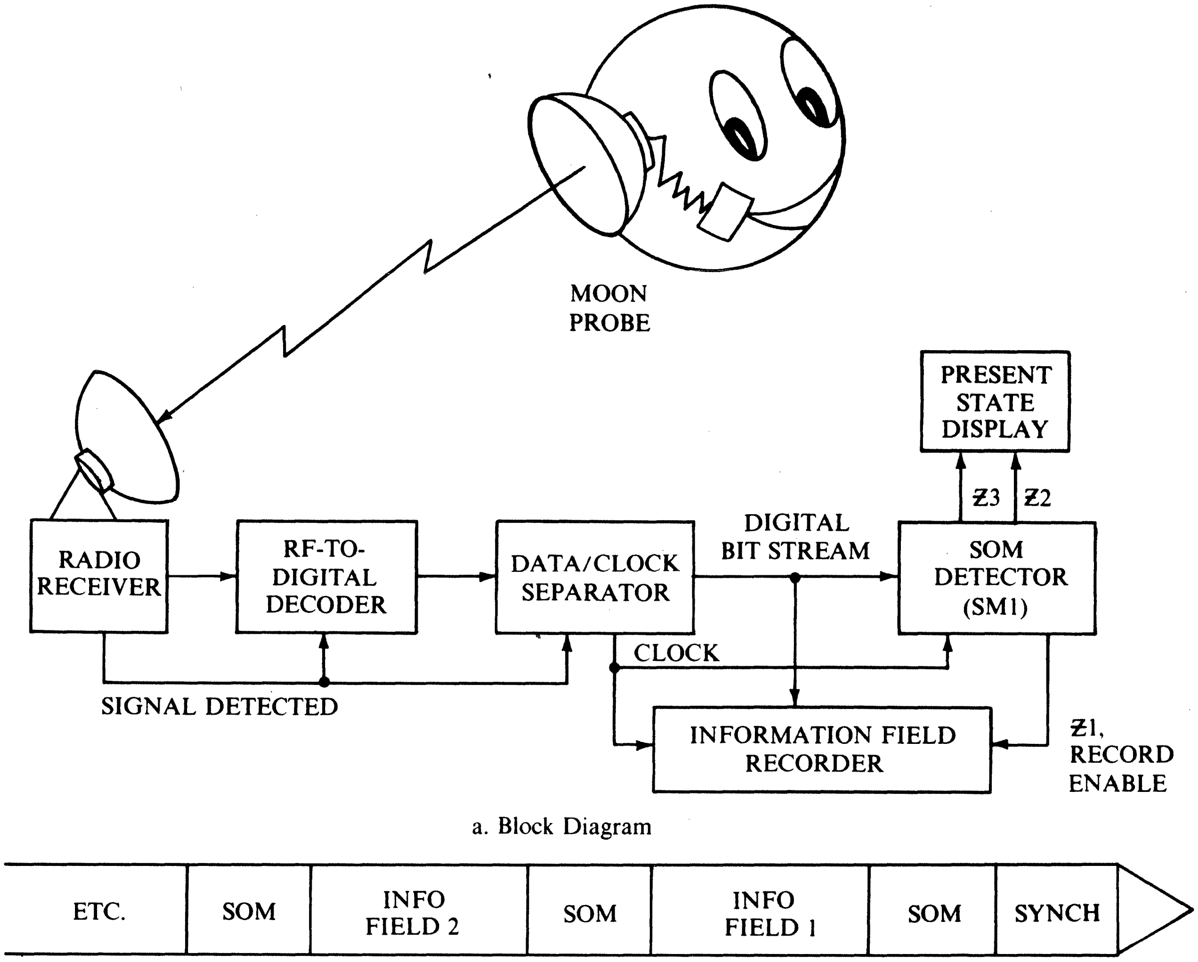
In practice, the need to visualize a sequential machine (SM) usually must be satisfied before we can represent it in the form of its state table. The *state diagram* is a useful tool to assist us in visualizing the desired *behavior* of an SM. Once the behavioral pattern of an SM is fixed–in the form of a state diagram–then we are able to proceed on to its state-table representation and actual implementation. Before we can begin the process, a word specification for the machine must be either supplied or invented. This is the necessary human link in the chain, for sequential machines–such as computers–simply are not (yet) capable of creativity. A convenient simple problem of practical importance is the binary sequence recognizer problem, below. In a classroom, someone usually points out that this particular problem can be more conveniently implemented using a shift register. True, but we use the full SM design approach, with separate flip-flops, to illustrate the general case while keeping the presentation simple.

DEFINITION OF REMOTE COMMUNICATION SYNCHRONIZATION SM PROBLEMS

Suppose we want to receive digital-communication signals from an experiment located on the moon. When the moon probe has collected sufficient data to be transmitted to earth, it transmits this information in the form of a "packet" of radio-frequency-encoded digital information. Transmissions occur at random, so that the receiver on earth must always be in operation, while waiting for the radio transmission of the encoded binary data to commence. The usual practice is to send out a string of bits (say, all 1s) to give the receiver a chance to *synchronize* itself with this incoming transmission. The purpose of this *synch* field in the packet is to allow the receiving station time to lock on to the signal and start to separate the encoded clock and data bits. When a sufficient number of *synch* bits have been transmitted, the moon probe then–without pause–sends the following bit sequence, bit T_0 being the first to arrive:

time period: T_2 T_1 T_0
transmitted data: 0 1 0

Further, this sequence is to be recognized *every time it occurs* after synchronization by an SM you are to design. "Every time occurs" is a critical portion of the system specification. Whenever this critical sequence is detected, it is used to cause recorders to store the bits following this string separately, as the "information" part of the packet for a number of different experiments. In other words, this bit string is to act as an information-field separator or *start-of-message* (SOM) field. Since the earth station must always recognize this bit string, we specify that the SM is to recognize it *whenever* it occurs in the incoming stream of bits. The overall system block diagram is shown in Figure 4-6a, and the format of a packet is shown in Figure 4-6b. This type of problem frequently occurs in networking and floppy-disk communications and is selected for the valuable insight it can begin to offer.



a. Block Diagram

b. Digital Data Packet

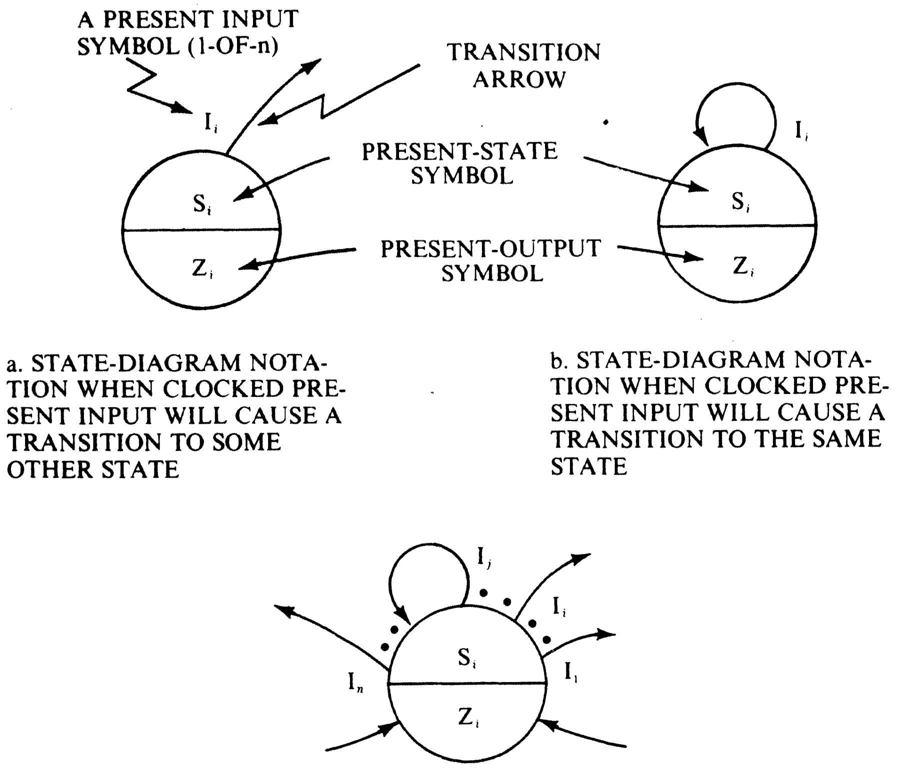
Figure 4-6
Remote Data Collection

FORMAL STATEMENT OF PROBLEM

Design a synchronous Moore-type SM for the detection of the serial bit string *010* whenever it occurs in a stream of digital data. You will receive each data bit and its associated clock pulse together. Arriving input bits are not reacted to until clocked in. The output of this SM is to indicate the current state of the machine and whether or not an SOM sequence has just been clocked in.

To solve the problem, we use a Moore-type state diagram. Each state of our SM is visually represented by a circle, as in Figure 4-7. Remember that this circle must always be associated with two other concepts:

a. Each circle (state) in the total state diagram must be associated with an event of physical significance that is to be "remembered" by the SM.



a. STATE-DIAGRAM NOTATION WHEN CLOCKED PRESENT INPUT WILL CAUSE A TRANSITION TO SOME OTHER STATE

b. STATE-DIAGRAM NOTATION WHEN CLOCKED PRESENT INPUT WILL CAUSE A TRANSITION TO THE SAME STATE

c. THE TOTAL NUMBER OF ARROWS n LEAVING A STATE S_i MUST EQUAL 2^m, m BEING THE NUMBER OF INPUTS. ANY NUMBER OF ARROWS CAN ENTER A STATE.

Figure 4-7
Moore SM: State Visualization Symbology

b. Each circle (state) in the total state diagram will eventually be associated with a unique setting of the memory elements in an SM's SSR. This means that there must be a one-to-one correspondence between circles (used states) and specific settings of the SSR. As noted, the number of circles (required states) determines the required size of the SSR.

The circles in Figure 4-7 are divided by a line. Above the line, we record the name of a state, say S_i. Below the line, we record the present output symbol of the SM, say Z_i, where Z_i is some function of the present state, S_i. Also associated with each circle are arrows representing transitions to the same or to another state. These are separately shown in the figure. At the base of each arrow is the associated external input symbol I_i, $i = 1, 2, \ldots, n$; where n is the total number of input symbols. That is, the total number of transition arrows *leaving* each state must be equal to the number of input symbols before the actual design can proceed. Put another way, we must have completely specified all state-diagram transitions for each state, before it becomes realizable. Therefore, the number of departing arrows required to completely specify each state has been related to the number of different input lines within the machine, say m. That is, n above is determined by

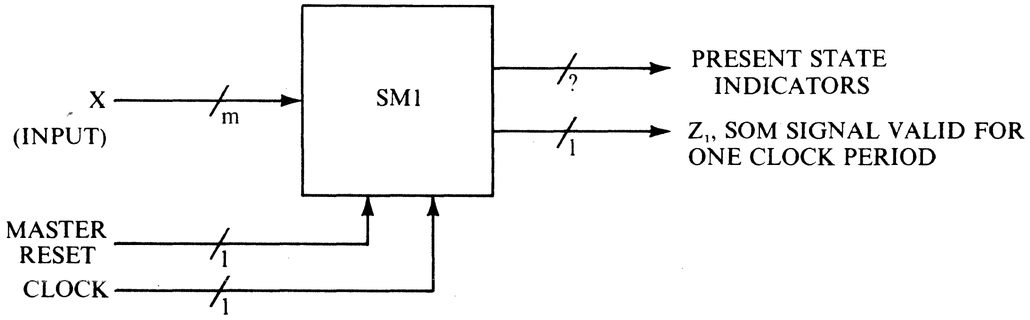


Figure 4-8
Block Diagram of SM1

$$n = 2^m$$

for each synchronous SM.

In recognition of the foregoing, we start each problem solution with a block diagram that represents the basic machine configuration, shown in Figure 4-8. The clock input is not a data input and therefore does not figure in establishing *m*. In fact, the clock input is not explicitly referred to in the design process, but its use as the signal that synchronizes state and output changes must not be forgotten. Neither is the master reset signal a data input. Its use is to initialize the system to some known state. Since our machine, which we will call SM1, has only one data input line, X, which may be in one of two states, then each state (circle) must have two associated transition arrows departing from it to be completely specified. Any number of arrows may enter a given state, but the number leaving is rigidly controlled by the number of inputs. If there were two inputs, four departing arrows would be required, etc.

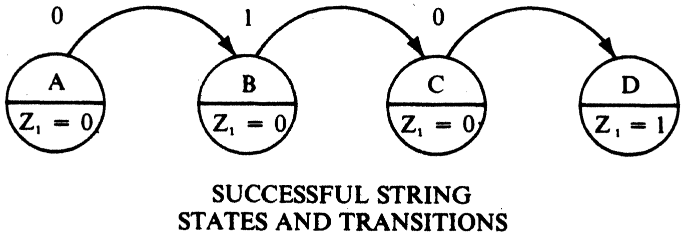
The number of outputs of SM1 is fully specified when we decide how many states are required to solve the problem and how we want to represent the present state. In this example, some of the outputs will be used to display the present state, which is instructive but not necessary. One required output, though, is to be used to signify whether a complete start-of-message (SOM) sequence has just been clocked in. It is valid only for that one clock period. Let us call this particular output Z1.

SM DESIGN PROCEDURES

STATE DIAGRAM CREATION

A convenient way to start the analysis of a problem is to recognize that some initial state should exist. Typically, this state is entered into after a *reset* signal on power-up or other initialization procedures. For example, a loss of detected signal could be used to reset SM1. Reset causes all the flip-flops of SSR to be set to, say, low. A master *clear* input achieves this result and is also shown in the block diagram of Figure 4-8. Realize that *the physical significance of this state is that it represents the state in which **none** of the bits in the SOM sequence have been clocked in.* Thus, we have introduced an example of physical significance. We begin the solution in Figure 4-9, our first pass at forming a state diagram for SM1, by naming the first circle state A and by rec-

Figure 4-9
Partially Specified State Diagram of SM1



ognizing that it is our home state. Its physical significance is established.

Successful String

State A's associated outputs will be fully dealt with soon. For now, let us recognize that output Z1 must be 0, since we cannot have clocked in any bits in the SOM sequence (by definition) after reset or whenever we are in the initial home state. Now, how do we proceed? The answer is simply to recognize that the events of physical significance our machine should remember are the clocking in of each proper successive bit in the SOM sequence. Let us refer to this as the *successful string*, which in the present case is our SOM sequence. As each bit along the successful input string is clocked in, SM1 will proceed to another state to physically record this progress.

This is shown in Figure 4-9, too. State B has the physical significance of recording the fact that the first bit in our successful string has been clocked in; state C, the second; and State D, the complete successful string SOM. Each data bit of SOM that creates a state transition after it is clocked in appears at the base of the transition arrows that have so far been associated with the state progress of the successful string. For each state, the present input bit applied to that state causes the indicated transition *when it is clocked in.* Remember, this is a synchronous circuit. Therefore we can say that a *0* applied to SM1 in state A, if clocked in, will cause it to transition to state B, that a *1* clocked in while in state B will cause SM1 to proceed to state C, etc.

What of the inputs not along a successful string? They too must be specified if SM1 is to be completely specified. This portion of the problem solution can be difficult to produce if the worded statement of the problem is not clear. Our specific problem was to recognize the SOM sequence *whenever* it appeared in an incoming stream of bits. This precise enunciation of what is to be performed is added to the fully specified state diagram of Figure 4-10, the defined *state paths* of SM1. We notice that, in state A, the clocking in of a *1* input results in a transition into the same state. The reasoning for this transition is that the SOM string must start with the clocking in of a *0.* We must therefore, for now, remain in the state that asserts that no bits along the successful sequence have been recorded. State A, having two departing arrows, is now completely specified.

In state B, we have specified that a *0* input, when clocked in, causes SM1 to remain in B. The reasoning applied here is that we wish to recognize SOM whenever it occurs. Take, for example, the string below, applied after a reset. Note that the first applied input is at the far right end of the sequence. Embedded in it is the successful string:

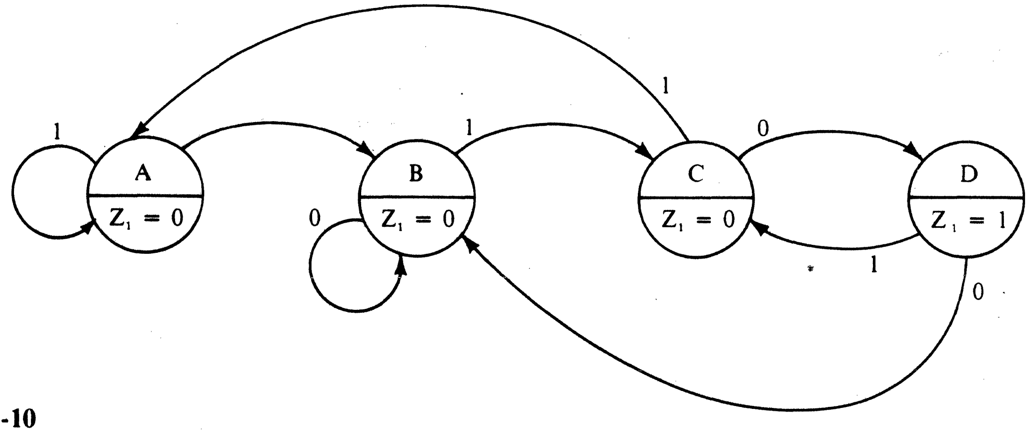


Figure 4-10
Completely Specified State Diagram of SM1

Period	T_n	T_{n-1}	T_{n-2}	..	T_2	T_1
Present Input:	?	0	1	all 0s	0	0
Present State:	D	C	B	. . B . .	B	A
Present Output Z1:	1	0	0	. . 0 . .	0	0

The first clocked *0* in this particular string put SM1 into state B in time period T2, recording the fact that the first bit on the successful string was noted. Any of the succeeding *0* inputs could *also* have been the first *0* on a successful string. As noted above, any number of *0*s could be received but, if they are then followed by a *1* and then a *0*, the machine still records the fact that SOM has been fully clocked in, by reaching state D. It indicates this by outputting a Z1 = 1 whenever SOM has occurred.

COMPLETELY SPECIFIED SM

Readers should satisfy themselves that all other state transitions in the state diagram conform to the rule that an embedded successful string is recognized *any time* it occurs. The procedure of forming a state diagram ends when:

a. All successful strings specified are realized in a state diagram representation that reflects the physical significance of all such strings (there can be more than one), and when

b. Every state has as many transition arrows leaving it as there are input symbols. This means that the effects of all input symbols *not* in the successful strings must be accounted for, too.

Once all the states are visualized in this manner and each state is completely specified, as in Figure 4-10, we are prepared to represent SM1 in the form of its state table. Two small items must be clarified first: How to represent the present state of SM1 at its outputs and how to encode its state names. Notice that the completed state diagram contains four states. These four states can be encoded with two extra outputs, which we shall call Z3 and Z2. This particular assignment is not a necessary one. It is chosen both to display the state sequence of the successful string in straight binary order and to enlarge the size of the present output. The present output-word size is purposefully enlarged to resemble the appearance of a microcoded control system more closely. Z1 remains the output bit, signifying when SOM has just been clocked in. In Table 4-1, we arbitrarily assign the representation significance to bits Z3 and Z2 of the output:

Table 4-1
Binary Assignments of Present Output Symbols for SM1

State	*Z3*	*Z2*	*Z1*
A	0	0	0
B	0	1	0
C	1	0	0
D	1	1	1

Thus we have established the nature of our three *present output* lines. Notice that the output symbol set {Z3, Z2, Z1} has eight symbols available, only four of which are actually used here. This means that only two output symbols are really required to encode all the desired outputs of this problem. For illustrative purposes, we are separately indicating both the present state and the signal that SOM has been received as the present output's fields of SM1. The present output now explicitly contains these two fields in its structure. The present-output portion of a line of microcode is similarly divided into explicit fields. So our state table crudely approximates a line of microcode.

State Table

Encoding State Assignments

We still need to assign binary values to the names of the states of SM1. Our four states imply that there are two memory elements in the SSR of SM1. Let us call these memory cells Q1 and Q0. Q1 and Q0 are our *state variables.* At the possible expense of some added gate requirements, we straightforwardly assign binary values to the state names in the encoding sequence of the Karnaugh logic map's (K-map's) reflected Grey Code order shown in Table 4-2 below. In conveniently using the K-map order for these state assignments, we reduce the amount of work later required to derive and simplify the state machine's logic equations. No attempt is made here to optimize either the binary state or the binary output assignments. The techniques for doing this are beyond our scope.

Table 4-2
Binary Assignments of State Variables for SM1

	Binary Assignments	
State Name	Q1	Q0
A	0	0
B	0	1
C	1	1
D	1	0

Figure 4-11
Binary-Encoded State Table: SM1

STATE DIAGRAM STATE SYMBOL	I Q (x)		$x = 0$	$x = 1$	z		
	Q_1	Q_0			z_3	z_2	z_1
A	0	0	0 1	0 0	0	0	0
B	0	1	0 1	1 1	0	1	0
C	1	1	1 0	0 0	1	0	0
D	1	0	0 1	1 1	1	1	1
	PRESENT STATE		NEXT-STATE PREDICTION n		PRESENT OUTPUT p		

Completion of SM1

With the above output and state-variable binary assignments, we can now complete the state table for SM1, presented in Figure 4-11. The method of completing the next-state portion of the state table is to note in its state diagram—for a given state and a given input—what the next state name is. This information is then entered into the appropriate box of the state table of the figure in its binary-encoded form, as assigned in Table 4-2. For example, consider state C of the state diagram. When the input X is equal to *0*, then SM1's next state is to be state D. In state C, we also see that, when the clock is applied to the input X = 1, the next state is to be state A.

Now go to the row of the state table named state C. Here, under the $X = 0$ column, we entered the binary code for state D, which is *10*. Likewise, under the column headed $X = 1$, note that *00* was entered in the row for state C, which is the binary encoding of state A. In this manner, by transferring the information from the state diagram to the state table, in binary encoded form, we have completely specified, in the state table, the desired *state path* behavior of the state diagram for SM1.

State Path Analysis

This defined state path(s) concept is an important way to visualize what possible sequences an SM follows. For the software-oriented, the next-state entries can be compared to a linked-list structure containing one or more linkage paths. The entire structure, however, is closed. This implies that state-machine behavior may be simulated with the use of linked list data structures.

State Table/Microprogram Relationship

Again, we want to emphasize that this *state table* contains the same underlying form as that of a *microprogram.* This fact will be amply illustrated subsequently. At this point, this should be no surprise, since *microprogram* has been defined previously as "a rational approach to the design of a computer's control system," which we now understand to be itself an SM. The elements we see here in the 5-tuple SM1 = <I, Q, Z, *n, p*> that characterizes this machine also are present in a microprogram. Let us next investigate the implementation of SM1, as an aid in developing a clearer appreciation of the clocked sequential nature of SMs and microprograms.

Logical Properties of Flip-Flops (FF's)

Before we can implement a sequential circuit from a state table, using flip-flops, we must first derive the logical properties of these elements. We shall also refer to a flip-flop as an FF in the following discussions. When the implementation of the SM we are discussing is completed, it is a hard-wired machine. Prior to the 1970s, most computers fell into this category. Microprogrammed design techniques are the alternative to the hard-wired approach. Since the latter evolved from the former and both are currently in use, it is instructive to complete the design of a hard-wired circuit that uses flip-flops and combinational-logic elements in its control system. The timing relationships to be observed will also help enlarge our appreciation of a computer's operations. To reach this implementation goal with understanding, we must briefly review the derivation of the design tools required in the logical application of flip-flops. This step is necessary because not all flip-flops of a given type have the same logical properties.

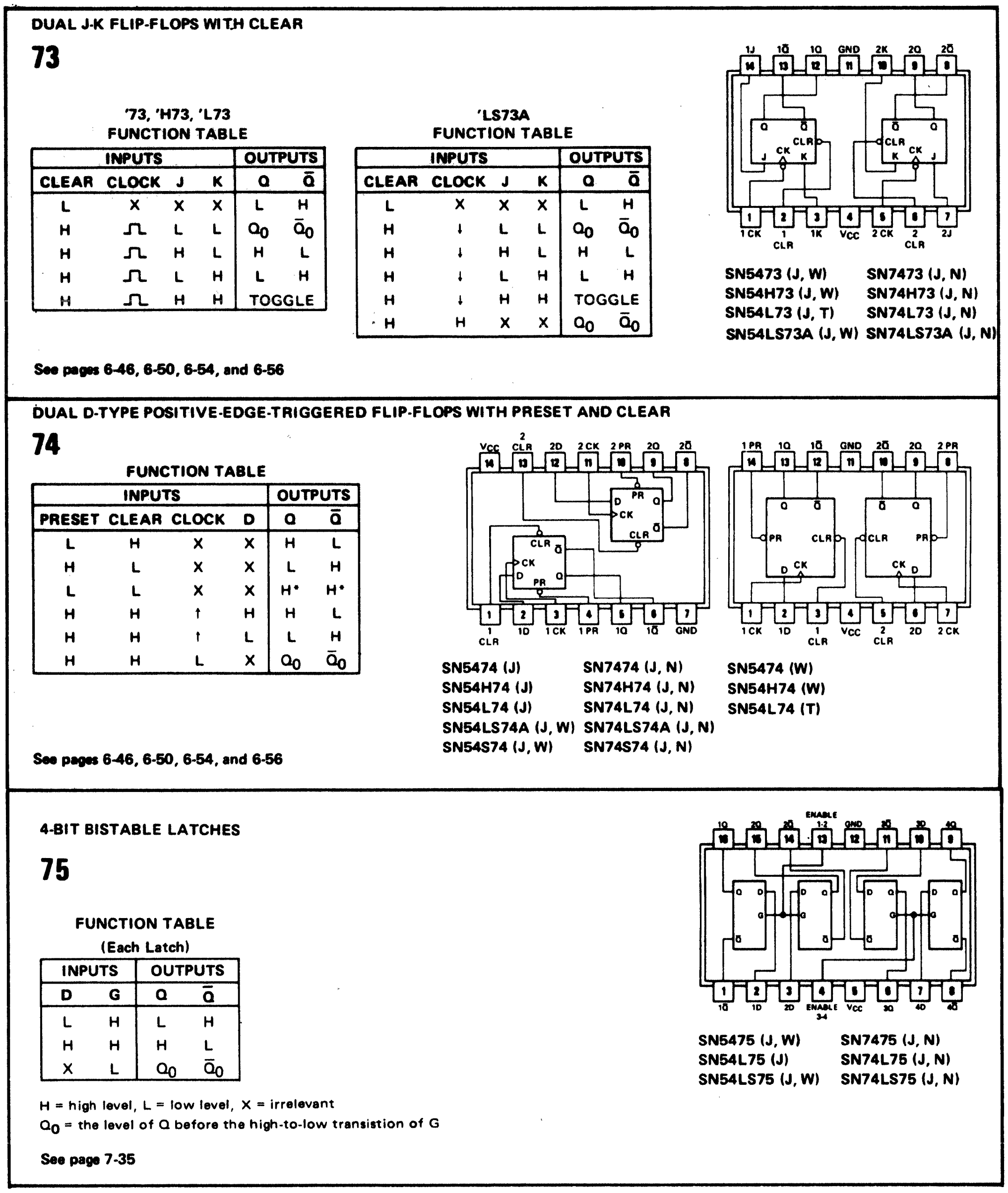
Popular Flip-Flop Types

SC Latch

The most frequently used types of latches and flip-flops are referred to as the *Set-Clear* (SC), the JK, and the *Delay* (D) varieties. The SC (also called the *Set-Reset* or SR) latch was discussed in Chapter 2, where it was shown to be fundamentally composed of cross-coupled inverting-logic gates. This structure is used as the output stage for all other types of flip-flops. It was shown to possess the basic memory-retention property of static semiconductor memory cells. In practice, the SC is difficult to use as a flip-flop, for two reasons. First, it does not have a clock input to synchronize changes in state and, as a result, it is difficult to achieve precise coordination in large systems. Second, for one input state, both the Q and the /Q outputs can be in the same state, a situation we cannot allow in practice, due to the resulting indeterminate behavior of the memory element. An example of an SC latch package (included in Figure 4-12) is the 74LS279 Quad Latch IC, which presents to the external world only the Q output of its four internal latches. Notice that its operation is undefined for one input state. A JK flip-flop, on the other hand, is designed always to have mutually exclusive Q and /Q outputs and no undefined states of operation for *all* input conditions.

JK and D Flip-Flops

The JK flip-flop comes with only one guarantee: *all* input combinations lead to logically useful results and therefore are allowed. The commercial varieties are clocked, making them a good basis for synchronous SM designs. Examples of the JK flip-flops are the 74LS76 and the 74LS109 IC's, also part of Figure 4-12. The 74LS109 is interesting, in that it can be converted to the last of the popular types, the D flip-flop, by using a single input line, tied to both the J and K inputs. The *delay* characteristic means that the present input appears at the Q output after the clock has taken effect after the current period. That is, present inputs appear as outputs delayed by one clock period. An example of a purely D-type IC is the 74LS74, also



DUAL J-K FLIP-FLOPS WITH CLEAR

73

'73, 'H73, 'L73
FUNCTION TABLE

INPUTS				OUTPUTS	
CLEAR	CLOCK	J	K	Q	$\bar{Q}$
L	X	X	X	L	H
H	⎍	L	L	Q_0	$\bar{Q}_0$
H	⎍	H	L	H	L
H	⎍	L	H	L	H
H	⎍	H	H	TOGGLE	

'LS73A
FUNCTION TABLE

INPUTS				OUTPUTS	
CLEAR	CLOCK	J	K	Q	$\bar{Q}$
L	X	X	X	L	H
H	↓	L	L	Q_0	$\bar{Q}_0$
H	↓	H	L	H	L
H	↓	L	H	L	H
H	↓	H	H	TOGGLE	
H	H	X	X	Q_0	$\bar{Q}_0$

SN5473 (J, W) SN7473 (J, N)
SN54H73 (J, W) SN74H73 (J, N)
SN54L73 (J, T) SN74L73 (J, N)
SN54LS73A (J, W) SN74LS73A (J, N)

See pages 6-46, 6-50, 6-54, and 6-56

DUAL D-TYPE POSITIVE-EDGE-TRIGGERED FLIP-FLOPS WITH PRESET AND CLEAR

74

FUNCTION TABLE

INPUTS				OUTPUTS	
PRESET	CLEAR	CLOCK	D	Q	$\bar{Q}$
L	H	X	X	H	L
H	L	X	X	L	H
L	L	X	X	H*	H*
H	H	↑	H	H	L
H	H	↑	L	L	H
H	H	L	X	Q_0	$\bar{Q}_0$

SN5474 (J) SN7474 (J, N) SN5474 (W)
SN54H74 (J) SN74H74 (J, N) SN54H74 (W)
SN54L74 (J) SN74L74 (J, N) SN54L74 (T)
SN54LS74A (J, W) SN74LS74A (J, N)
SN54S74 (J, W) SN74S74 (J, N)

See pages 6-46, 6-50, 6-54, and 6-56

4-BIT BISTABLE LATCHES

75

FUNCTION TABLE
(Each Latch)

INPUTS		OUTPUTS	
D	G	Q	$\bar{Q}$
L	H	L	H
H	H	H	L
X	L	Q_0	$\bar{Q}_0$

H = high level, L = low level, X = irrelevant
Q_0 = the level of Q before the high-to-low transistion of G

See page 7-35

SN5475 (J, W) SN7475 (J, N)
SN54L75 (J) SN74L75 (J, N)
SN54LS75 (J, W) SN74LS75 (J, N)

1J 1Q̄ 1Q GND 2K 2Q 2Q̄
14 13 12 11 10 9 8
Q Q̄ CLR CK J K
1 2 3 4 5 6 7
1CK 1 CLR 1K VCC 2CK 2 CLR 2J
VCC 2 CLR 2D 2CK 2PR 2Q 2Q̄
D PR Q CK CLR Q̄
1 CLR 1D 1CK 1PR 1Q 1Q̄ GND
1PR 1Q 1Q̄ GND 2Q̄ 2Q 2PR
1CK 1D 1 CLR VCC 2 CLR 2D 2CK
1Q 2Q 2Q̄ ENABLE 1-2 GND 3Q̄ 3Q 4Q
16 15 14 13 12 11 10 9
D Q G Q̄
1 2 3 4 5 6 7 8
1Q̄ 1D 2D ENABLE 3-4 VCC 3Q 4D 4Q̄

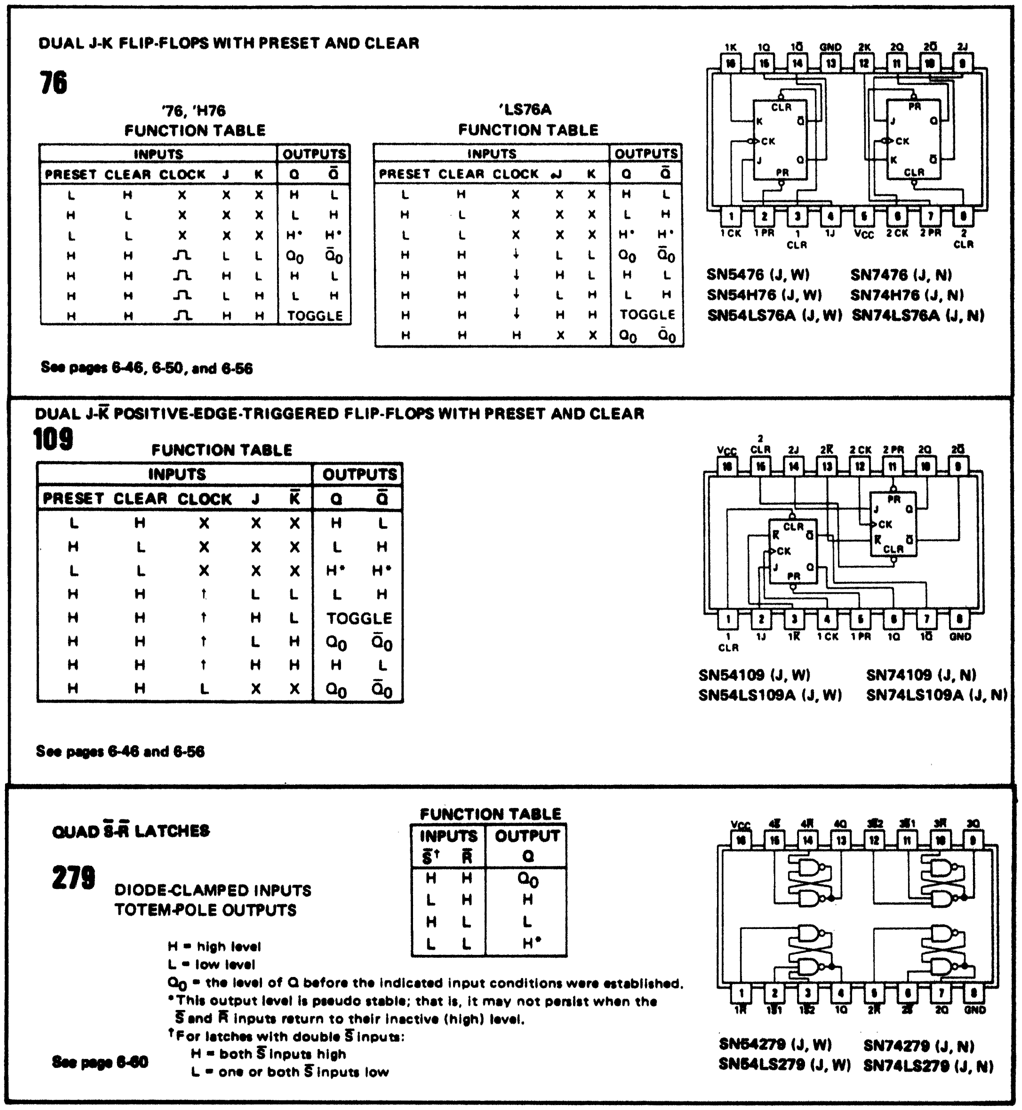
Figure 4-12 Flip-Flop and Latch Characteristics Data Sheet *(For educational purposes only. Data may be old and obsolete. Courtesy of Texas Instruments, Inc. © 1984, Texas Instruments, Inc.)*

illustrated in Figure 4-12. The TTL data sheets on all these types should be studied.

DERIVATION OF LOGICAL PROPERTIES OF FF'S

Manufacturers' Characteristics Table of FF's

The wording above on the guarantees for flip-flops is a simple warning that type names do not imply the logical properties of a given flip-flop. The user must derive these logical properties, for they depend on activity level, design, and logic-gate type, i.e., NAND or NOR. The starting point for establishing logical properties is the manufacturer's table of characteristics of a chosen flip-flop. Figure 4-12 presents this information for the latches and flip-flops we shall discuss here. The first thing to note is that the left side of the table contains the names of the two



DUAL J-K FLIP-FLOPS WITH PRESET AND CLEAR

76

'76, 'H76
FUNCTION TABLE

INPUTS					OUTPUTS	
PRESET	CLEAR	CLOCK	J	K	Q	$\overline{Q}$
L	H	X	X	X	H	L
H	L	X	X	X	L	H
L	L	X	X	X	H*	H*
H	H	⎍	L	L	Q_0	$\overline{Q}_0$
H	H	⎍	H	L	H	L
H	H	⎍	L	H	L	H
H	H	⎍	H	H	TOGGLE	

'LS76A
FUNCTION TABLE

INPUTS					OUTPUTS	
PRESET	CLEAR	CLOCK	J	K	Q	$\overline{Q}$
L	H	X	X	X	H	L
H	L	X	X	X	L	H
L	L	X	X	X	H*	H*
H	H	↓	L	L	Q_0	$\overline{Q}_0$
H	H	↓	H	L	H	L
H	H	↓	L	H	L	H
H	H	↓	H	H	TOGGLE	
H	H	H	X	X	Q_0	$\overline{Q}_0$

See pages 6-46, 6-50, and 6-56

SN5476 (J, W) SN7476 (J, N)
SN54H76 (J, W) SN74H76 (J, N)
SN54LS76A (J, W) SN74LS76A (J, N)

DUAL J-$\overline{K}$ POSITIVE-EDGE-TRIGGERED FLIP-FLOPS WITH PRESET AND CLEAR

109

FUNCTION TABLE

INPUTS					OUTPUTS	
PRESET	CLEAR	CLOCK	J	$\overline{K}$	Q	$\overline{Q}$
L	H	X	X	X	H	L
H	L	X	X	X	L	H
L	L	X	X	X	H*	H*
H	H	↑	L	L	L	H
H	H	↑	H	L	TOGGLE	
H	H	↑	L	H	Q_0	$\overline{Q}_0$
H	H	↑	H	H	H	L
H	H	L	X	X	Q_0	$\overline{Q}_0$

See pages 6-46 and 6-56

SN54109 (J, W) SN74109 (J, N)
SN54LS109A (J, W) SN74LS109A (J, N)

QUAD $\overline{S}$-$\overline{R}$ LATCHES

279 DIODE-CLAMPED INPUTS
TOTEM-POLE OUTPUTS

FUNCTION TABLE

INPUTS		OUTPUT
$\overline{S}$†	$\overline{R}$	Q
H	H	Q_0
L	H	H
H	L	L
L	L	H*

H = high level
L = low level
Q_0 = the level of Q before the indicated input conditions were established.
*This output level is pseudo stable; that is, it may not persist when the $\overline{S}$ and $\overline{R}$ inputs return to their inactive (high) level.
†For latches with double $\overline{S}$ inputs:
H = both $\overline{S}$ inputs high
L = one or both $\overline{S}$ inputs low

See page 6-60

SN54279 (J, W) SN74279 (J, N)
SN54LS279 (J, W) SN74LS279 (J, N)

Figure 4-12 (cont.) Flip-Flop and Latch Characteristics Data Sheet *(For educational purposes only. Data may be old and obsolete. Courtesy of Texas Instruments, Inc. © 1984, Texas Instruments, Inc.)*

present inputs and, beneath these, their applied voltage-level combinations. The righthand column is the output condition of Q *after* the applied input-voltage levels have taken their effect. Notice that the right column uses symbols other than H or L. This is merely a compressed form of data representation.

The symbol Q_0 in the right side column means that the Q output is the same as it was before the input excitation of that row was applied. This is the memory-retention state. If Q was low (L), it remains L. If it was high (H), it remains H. The next two input excitations force Q to an H and then to an L, if applied in that order, regardless of the prior state. These external input combinations cause the indicated results whenever applied. The last input combination shown (LL) leads to an indeterminate result when removed. It is symbolized here as the H* entry in the righthand column, which implies that this output condition is valid only while the present inputs are applied and that the next state is indeterminate. This input excitation forces *both* gates of the internal latch high, whenever applied. To use this latch as a flip-flop, we must guarantee that the LL input combination will not occur. The reason that this particular excitation leads to indeterminate results is that we do not know in advance which of the two logic gates is faster and therefore how the circuit will recover if the next input excitation is HH.

Characteristic Equation (CE)

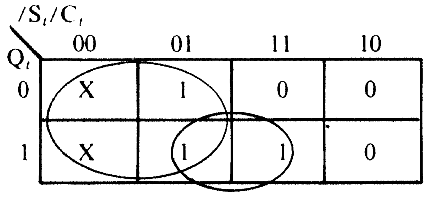
The manufacturer's characteristics table, in fact, is an encoded truth table. Due to the feedback of the Q output to one of the inputs, the circuit really contains two external inputs and a single internal one, Q. Since /Q is always the complement of Q when flip-flop action is enforced, its state does not convey information not already provided by Q. A three-input K-map must therefore be used to characterize this device's logical behavior. To do this, the data from the manufacturer's characteristics table is entered–first decoded if necessary–into the K-map. The equation derived from such a K-map is called the *characteristic equation* (CE) of the mapped flip-flop. If active-high logic conventions are applied, the CE of the 74279 latch of Figure 4-12 is obtained, as presented in Figure 4-13a. Double logic inversions, applied to the terms of Figure 4-13a, may be confusing to the reader. Figure 4-13b provides clarification of the CE and *Auxiliary Equation* (AE) derivations by adding inverting gates to the inputs of the basic latch. While the industrial choice of terminology for naming the inputs is unfortunate, we simply have to learn to live with it. Calling the inputs I_1 and I_2 or names that imply neither their function nor their supposed activity level is far less confusing to the beginner, when all we want to do is derive their logical behavior. We shall do so in the next example. For now, let us use the manufacturer's input signal names as used above in deriving the CE of the 74279.

The CE is the *only* feature that logically characterizes a flip-flop. In Figure 4-13a, the use of the subscript t implies a *present* input or output condition. The subscript $t+$ implies the next state, the time period after the present, when all present-input conditions have taken effect. The column headed by the $/S_t/C_t = 00$ external excitation will not be allowed to occur in practice and therefore can be used for the "don't-care" entries, which are marked *X*. Consulting the manufacturer's characteristics table for the 74LS279, we see our *01* column contains only *1* entries, because the manufacturer's table declared unequivocal high output behavior for that particular excitation. In the *11* column, we find that the entries correspond to the state of Q *before* this excitation was applied. This is the memory-retention state of this flip-flop. This state means that, if the output state was *0,* then it is to remain *0,* and, if *1,* it is to remain *1* after the input *11* is applied. Finally, the application of the excitation *10* enforces the *0* results shown in that column. The CE derived from the K-map plot of the manufacturer's characteristics for a chosen logic activity level is the expression for the *next* state (Q_{t+}) of a flip-flop, in terms of both its present external inputs (say, in general, I_{1t} and I_{2t}) and its present internally fed-back input (Q_t). Thus, it truly characterizes the device's logical behavior.

Auxiliary Equation

The equation that, if enforced in usage, prevents the *00* excitation from being applied is called the Auxiliary Equation (AE). It too is shown in Figure 4-13a. Obviously, if the condition specified in the AE (that $/S_t + /C_t = 1$) is observed during usage, then they cannot both be *0* at the same time. This assures us that the latch can be used as a flip-flop if we observe the conditions of the AE while applying the CE. We will

Figure 4-13
Derivation of CE and AE for 74279

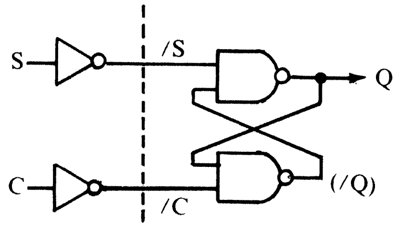


Q_t \ $/S_t/C_t$	00	01	11	10
0	X	1	0	0
1	X	1	1	0

CE: $Q_{t+} = /(/S_t) + /C_t \cdot Q_t$
$= S_t + /C_t \cdot Q_t$

AE: $/S_t + /C_t = 1$
OR $S_t \cdot C_t = 0$

a. Derivation of the Characteristic and Auxiliary Equations of the 74279 SC Latch, Using Active-High Logic Conventions from Manufacturer's Table in Figure 4-12a



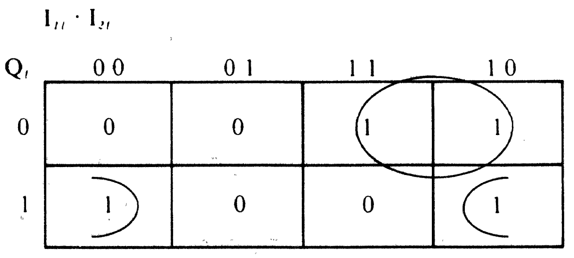
b. Clarification of Logic Inversions in CE and AE

not pursue the SC-type of flip-flop further, here, since it is beyond our scope to do so. The text by Phister listed in the References, though old, contains one of the better explanations of the derivation of logical properties of flip-flops. We need the ability to derive CEs, so let us look at a JK flip-flop we will use. The advantage in practice of the JK class of flip-flops, besides their being clocked, is that *all* external excitations lead to determinate output behavior. The net result is that the AE is not applicable to this class and was acknowledged here essentially as a rounded-out foundation for the following.

CE Derivation of JK Type

Figure 4-14 presents the derivation of the characteristic equation of the 7476 group of JK flip-flops from the information contained in its manufacturer's table in Figure 4-12. Our interest in this CE-derivation procedure is more than casual, since we will use the 7476 in the logical design example to follow. The tables of Figure 4-12 often contain more information than needed for the CE derivation. The upper halves of the manufacturer's characteristics tables in Figure 4-12 relate to the overriding effects of the Set and Clear inputs. This behavior is due to the existence of an SC latch at the front end of this device. The lower halves of these tables contain the manufacturer's specified clocked input/output transformations needed in the derivation of the CE. Also note that there is a reason for the existence of two 7476 tables, a side point of interest. The 7476 group contains flip-flops with two types of clocking characteristics (see Chapter 3, for explanations of T_s and T_h). The 74LS76 is strictly trailing-edge-sensitive, while the others of this group are both level- and falling- (i.e., trailing-) edge-sensitive. This does not

Figure 4-14
CE Derivation for 7476 Flip-Flop (Active-High Logic)



Q_t \ $I_{1t} \cdot I_{2t}$	00	01	11	10
0	0	0	1	1
1	1	0	0	1

Note:
$J_t = I_{1t}$
$K_t = I_{2t}$

$$\text{CE: } Q_{t+} = I_{1t} \cdot /Q_t + /I_{2t} \cdot Q_t$$
$$= J_t \cdot /Q_t + /K_t \cdot Q_t$$

affect their logical properties, since all share the same manufacturer's characteristics. It only affects the duration of time input signals must remain stable. If any changes in state do occur, they will be apparent to the user only after the clock goes low, in any case.

In Figure 4-14, we have generalized the input names to $J_t = I_{1t}$ and $K_t = I_{2t}$, as an indication of a better use of names in working out a problem. The final substitution of the manufacturer's terms for the generalized input symbol names avoids the confusion when multiple inversions are encountered in the process. It also focuses our attention on the innate effects of the inputs without the risk of forming misconceptions about names that can imply function. These names can lose their meaning, if we apply to the device an activity level other than the one used in selecting the functional name in the first place.

As noted, the JK-type of flip-flop does not need an AE. Internal modifications of the design have converted the not-allowed input state into a toggle mode. The toggle mode occurs when the external inputs will simply cause a flip-flop to change to the other output state when clocked. The pulsed-mode example of the latch, in Chapter 2, demonstrated one form of this type of action. Notice that the *11* external input-state column of Figure 4-14 indicates that, if the present output is $Q_t = 0$, then it will become $Q_{t+} = 1$, and vice versa. In the *outputs* column of the manufacturer's characteristics table, Figure 4-12, the encoded behavior of Q_{t+} is described by calling it Q and /Q (we have used Q; the other is redundant), from which the foregoing Q_{t+} is derived. Further, the inputs take effect under the T_s and T_h specified conditions, for proper clock-driven operation to occur. Consideration of the effects of T_s and T_h has led the manufacturer to use the two different symbols in the clock column of the characteristics tables for each flip-flop. The 74LS76 uses an arrow in the table to indicate that its clock response is strictly edge-sensitive. The hat-shaped symbol used with the 7476 flip-flop, _⎍_, or its inversion denotes both level and edge-clocking characteristics apply. If a change is to occur, it will be noticed only within the *next* clock period. As we noted in Chapter 3, output changes are synchronized to the clock edge associated with T_h, which, by our definition, establishes the end of each clock period.

EXCITATION TABLE DERIVATION

The CE (and AE, where applicable) thus obtained for any flip-flop is used to derive an *excitation table*. This is the sought-after design tool that is required to implement a hard-wired SM. Figure 4-15 illus-

Figure 4-15
Derivation of 7476 Excitation Table (Active-High Logic)

$Q_t \rightarrow Q_{t+}$	J_t	K_t	$Q_{t+} = J_t \cdot Q_t + /K_t \cdot Q_t$
$0 \rightarrow 0$	0	X	$0 = J_t \cdot 1 + /K_t \cdot 0$
$0 \rightarrow 1$	1	X	$1 = J_t \cdot 1 + /K_t \cdot 0$
$1 \rightarrow 0$	X	1	$0 = J_t \cdot 0 + /K_t \cdot 1$
$1 \rightarrow 1$	X	0	$1 = J_t \cdot 0 + /K_t \cdot 1$
DESIRED STATE CHANGE	REQUIRED INPUT EXCITATION		CE CALCULATIONS
7476 EXCITATION TABLE–ACTIVE HIGH			DISCARD WHEN DONE

trates how the excitation table is derived. It is divided into three major columns. The lefthand column contains all possible desired transitions of the two variables from Q_t to Q_{t+}. If, for a given state, one can control the desired transition of an SM to some next state, then the implementation of a machine's finite state path is realizable. The control of the state path of an SM, as expressed by its SSR, is the essence of its state behavioral design. The second major column grouping contains these state controlling external inputs to the flip-flop. The logic levels applied to the present inputs (J_t and K_t, in this case) "excite" the device, to produce the desired transition of the leftmost column. They can only be specified after we have substituted the desired transition states into the CE, as is performed in the rightmost column.

The rightmost major column of Figure 4-15 is headed by the CE of the flip-flop. Since the CE expresses the future state in terms of its present inputs and present state (both external and the internal feedback inputs), it contains all the information required to generate the entries of the center group–the present-input columns that excite the desired transitions of the leftmost column. In the first row, the desired transition specified is that $Q_t = 0$ is to become $Q_{t+} = 0$ (remain 0). If these values are substituted in the equation that characterizes this flip-flop's behavior, we get the result shown under the CE major column for this 0 → 0 transition row. Now, both sides of a Boolean equation must always agree, if the expressed characteristics are going to be observed. Since the right side of this CE, with the present transition values for Q_t and $/Q_t$ substituted in it, must also agree with the desired left-side transition value of 0 for Q_{t+}, then the Boolean expression on the right side must also reduce to 0.

Excitation Table For 7476 FF

Notice that $/K_t \cdot 0$ will not affect the balance between the right and left sides of the CE in this row. Regardless of the logical state of $/K_t$, its ANDing with 0 always produces 0. The net result is that we do not care what K_t is, and so an advantageous "don't-care" *X* entry is made in this row for K_t. J_t presents an entirely different situation, because this part of the Boolean expression is $J_t \cdot 1$. If J_t were allowed to be 1, then the rule that both sides of the equation must agree would be violated. The solution is simple–enter 0 in the J_t box for this row. We have now specified the required input excitation for the desired 0-to-0 transition of the 7476 series of flip-flops, using active-high logic conventions. We have also gained don't-care entries in the process. This is an important result, in that it leads to very simplified circuit implementation.

In the last row of Figure 4-15, notice that the $J_t \cdot 0$ portion of the substituted CE can never satisfy the 1 on the left side of this equation. For this row, J_t is a don't-care entry. Therefore $/K_t \cdot 1 = 1$ is the only condition that can properly balance the equation. Therefore $/K_t$ *must* equal 1. If that is the case, then the K_t entry *must* equal 0, since it is the complement of $/K_t$. In this manner, the entire excitation table is completed, after which we can discard the CE calculations portion. The excitation table proper (the two left-hand portions) is our sought-after valuable design tool. The great advantage of this process is that it provides us with the ability to logically apply *any* flip-flop with *any* chosen logic activity level to the solution of implementation problems—with the greatest possible number of don't-care entries.

To complete the general methodology for deriving an excitation table, we ask: "What would one do if there were an AE associated with a particular flip-flop?" Fortunately, a simple general answer is incorporated into the derivation of the excitation table. In the presence of an AE, we make sure that we do not violate the rule of the AE each time we select external input excitations for a desired transition of the output. That is, if the AE requires that both I1t and I2t *not* simultaneously assume a particular total state, then our choices for the logic values these external excitations can assume are restricted to those that meet this condition. Fewer don't-care conditions occur in excitation tables with AEs, due to the application of the AE rule. As an exercise, derive the excitation table of the 74279 latch. It will contain only two—instead of four—don't-care entries when the CE calculations on each row also factor in the restriction of the AE rule.

Remember—an excitation table is applicable only to the particular flip-flop for which it was derived. It is also dependent on the logic activity level used in its derivation. The type names, such as JK, SC, etc., convey no significant information about the logical properties of flip-flops. These must be derived from the manufacturer's characteristics and a chosen activity level, as illustrated here.

SEQUENTIAL-MACHINE IMPLEMENTATION

Due to its synchronous clocking characteristics, the JK type of flip-flop is most advantageous in SM design problems using discrete components. This is attested to by their abundant availability in manufacturers' data catalogs and electronics parts stores. Integrated semiconductor IC designers may not wish to restrict themselves to the sole use of JK types to reduce the required transistor-equivalent count of the IC or to reduce propagation delay-time. They have their own set of specialized problems to face. Whatever the choice of FF type, the basic design approach does not change. Except for minor detail, what we have to say for the JK applies to other types when used in SM design.

Application Equations

The JK has two external inputs, J and K. With two inputs, one gains flexible control over the next state of the FF. It can be excited to output an unconditional value of either 0 or 1, to toggle (change its present value), or to retain its present contents. This means that we need to develop a logic equation to excite *each* input of every memory element in an SSR. These logic equations for each input are called the *application equations.* This is the logic we *apply* to each input to "steer" it along a desired state path. The net result of this is that we require two K-maps for obtaining the application equations of each FF in an SSR. Each FF represents a single state variable in the state table of an SM. The role of the application equations, then, is to enforce strict observance of the state paths defined in the state table by applying the logic levels to the flip-flop inputs that will steer them along these paths.

The specific implementation example we pursue here is the synchronous code sequence recognizer (SM1) problem earlier in this chapter. We also require an excitation table for the specific flip-flop to be used in the design. This shall be the 7476 type, discussed in the previous section. For convenience, the excitation table of the 7476 and the state table of the code sequence recognizer, SM1, are repeated here in Figure 4-16a and b. Figure 4-16c shall serve as our logic-design worksheet for SM1. In the design process, we must scan parts a and b together. In addition, we must draw up a set of K-maps, one for the J input and one for the K input, for each state variable. The forms of these application equation K-maps are also presented in Figure 4-16c. Each state variable is represented by a single FF that requires two separate application equations. When these K-maps are complete, the design of the SSR's control (excitation) logic is complete. Note that the state variables of SM1 are named Q_I and Q_0 and that the present input is called X. (Do not confuse this with the don't-care symbol X.)

The task before us is simply this: For a given state, a given state variable in that state, and a given input symbol, establish the next state of that variable. To do this:

1. Select a present state in the present-state column of the state table of an SM. This establishes the row of the state table that we will reference in the following steps.
2. Note the present value of only one of the state variables at one time within the present state.
3. For a given input symbol, note its corresponding desired future value in the next-state portion of the state table. This is located under the column heading of a selected input symbol, in the corresponding box (beneath it) of the row we are working in. Be certain that corresponding variables are compared within both present-state and next-state entries. This establishes the *required transition* of that state variable, if it is to follow the state path defined in the state table. This is tantamount to to saying that, for a given input and given state, this state variable is to go from its assigned present value to the desired new value after the clock takes effect.
4. In the excitation table, note the required values for J and K that enforce the required transition.

$Q_t \rightarrow Q_{t+}$	J_t	K_t
0 → 0	0	X
0 → 1	1	X
1 → 0	X	1
1 → 1	X	0

a. 7476 Excitation Table

X			Z
Q_1Q_0	X = 0	X = 1	Z3, Z2, Z1
0 0	01	00	0 0 0
0 1	01	11	0 1 0
1 1	10	00	1 0 0
1 0	01	11	1 1 1
PRESENT STATE	NEXT STATE		PRESENT OUTPUT

b. SM1 State Table

Q_1Q_0	X = 0	X = 1
0 0		
0 1		
1 1		
1 0		

$JQ_0 =$

Q_1, Q_0	X = 0	X = 1
0 0		
0 1		
1 1		
1 0		

$KQ_0 =$

Q_1, Q_0	X = 0	X = 1
0 0		
0 1		
1 1		
1 0		

$JQ_1 =$

Q_1, Q_0	X = 0	X = 1
0 0		
0 1		
1 1		
1 0		

$KQ_1 =$

c. Application Equation K-Maps

Figure 4-16
SM1 Logic Design Worksheet (Blank)

5. Enter these values into the application equation K-maps for J and K of that state variable. These entries are to be made in the boxes of the K-maps that agree with the present state and input symbols used in each determination above.

6. Repeat the above until the K-maps of all the application equations are completely filled in.

Figure 4-17
Some Application-Equation K-Map Entries for Q_0

Q_1Q_0 \ x =	0	1
0 0	1	0
0 1		
1 1	X	
1 0		

JQ_0 =

Q_1Q_0 \ x =	0	1
0 0	X	X
0 1		
1 1	1	
1 0		

KQ_0 =

7. Extract the simplified logic equations from the application equation K-maps for each input of each state variable. When this is done, the SSR design portion of the design of an SM is complete.

Design Application

Let us apply the above procedure by filling in the empty K-maps of Figure 4-16c. A copy of these blank forms may be used as a worksheet in following the discussion. An example of the procedure is presented in Figure 4-17, which displays partially filled-in K-maps for JQ_0 and KQ_0. Let us start with state Q_1Q_0 = 11 of the state table. The state variable Q_0 has the present value of 1 in this state. If SM1 is to clock in an external input of X = 0, then the box of the next state portion of the state table that forms the intersection of the Q_1Q_0 = 11 row with the column headed by X = 0 contains the next state prediction that Q_1Q_0 = 10. By comparing the corresponding state variable in each of these pairs, we observe that Q_0 is to transition from 1 to 0 after the clock "strikes." Note, too, that Q_1 is to undergo a different state transition. What is it? Returning to Q_0, we have established that the desired transition is 1 to 0 for this present-state and present-input combination.

The 7476 excitation table informs us that, to obtain 1-to-0 transition, J_t may be the don't-care symbol X, but K_t must be 1. Figure 4-17 shows that X has been entered in the box of the present-state and present-input intersection above in the K-map for JQ_0, but that 1 was entered into the corresponding intersection of the K-map for KQ_0. We have thus "steered" the J and K inputs of the FF representing state variable Q_0 for a specific present-state and present-input combination. The process is repeated until the entry combinations are exhaustively entered into the application-equation K-maps.

Let us now consider the present state Q_1Q_0 = 00. For the input X = 0, Q_0 is to undergo a 0-to-1 transition. The excitation table tells us that J_t must be 1, but K_t can be X. When the input is X = 1, a 0-to-0 transition is specified. For this case, J_t must be 0, but K_t may be X. Check these entries in Figure 4-17 for JQ_0 and KQ_0. The other state variable, Q_1, is processed in the same manner.

The completed worksheet is presented in Figure 4-18. Here, the completed K-maps are simplified to obtain the set of application equations for SM1. Note again that the application-equation K-maps, as well as the state table, are organized around the same

$Q_t \longrightarrow Q_{t+}$	J_t	K_t
$0 \longrightarrow 0$	0	X
$0 \longrightarrow 1$	1	X
$1 \longrightarrow 0$	X	1
$1 \longrightarrow 1$	X	0

a. 7476 Excitation Table

X Q_l, Q_0	X = 0	X = 1	Z Z3 Z2 Z1
0 0	01	00	0 0 0
0 1	01	11	0 1 0
1 1	10	00	1 0 0
1 0	01	11	1 1 1
PRESENT STATE	NEXT STATE		PRESENT OUTPUT

b. SM1 State Table

Q_l, Q_0	X = 0	X = 1
0 0	1	0
0 1	X	X
1 1	X	X
1 0	1	1

$JQ_0 = /X + Q_l$

Q_l, Q_0	X = 0	X = 1
0 0	X	X
0 1	0	0
1 1	1	1
1 0	X	X

$KQ_0 = Q_l$

Q_l, Q_0	X = 0	X = 1
0 0	0	0
0 1	0	1
1 1	X	X
1 0	X	X

$JQ_l = X \cdot Q_0$

Q_l, Q_0	X = 0	X = 1
0 0	X	X
0 1	X	X
1 1	0	1
1 0	1	0

$$KQ_l = X \cdot Q_0 + /X \cdot /Q_0$$
$$= /(X \oplus Q_0)$$

c. Application Equation K-Maps

Figure 4-18
SM1 Logic Design Worksheet (Filled In)

K-map format. This greatly reduces unnecessary work. The control-logic design phase for the SSR of SM1 is now complete. Before implementing the design, let us first consider the design of the present-output logic.

Derivation of Present Output

The present-output equation can be extracted *directly* from the state table. It is given there as a combinational-logic function of two variables (Q_l and Q_0 in K-map order) that has three output functions, Z3, Z2, and Z1. By observation and simplification, we find that:

$$Z3 = Q1$$

$$Z2 = /(Q1)\cdot(Q0) + (Q1)\cdot/(Q0)$$

$$Z1 = (Q1)\cdot/(Q0)$$

How is one to handle output equation problems that are more complex than this? Since the outputs of a Moore machine are a function only of the present state, K-maps based on the state variables can be drawn up. Then the output vector for each Z can be entered into its own map for simplification and implementation. We just performed this procedure by visual inspection for our simple problem. The expression for Z2 above can also be expressed as the eXclusive OR (XOR) of Q_1 and Q_0.

The astute reader may note that a different order of the binary encoding of the names of the states in the state diagram and/or a rearrangement of the choices for output symbols could result in even simpler logic equations than we have attained here. There are special techniques for state and output assignments that can minimize the logic of these equations. An easy observation is that the only output symbol we required was Z1, as noted. The present outputs that display the current state could just as well have come directly from the outputs of the SSR. This was deliberately avoided so that our final result would more closely resemble the large output fields of the state-table format for a microprogram. It also explicitly displays all the combinational-logic blocks of the classic Moore model shown in Figure 4-3. This simplification is left as a project for the reader.

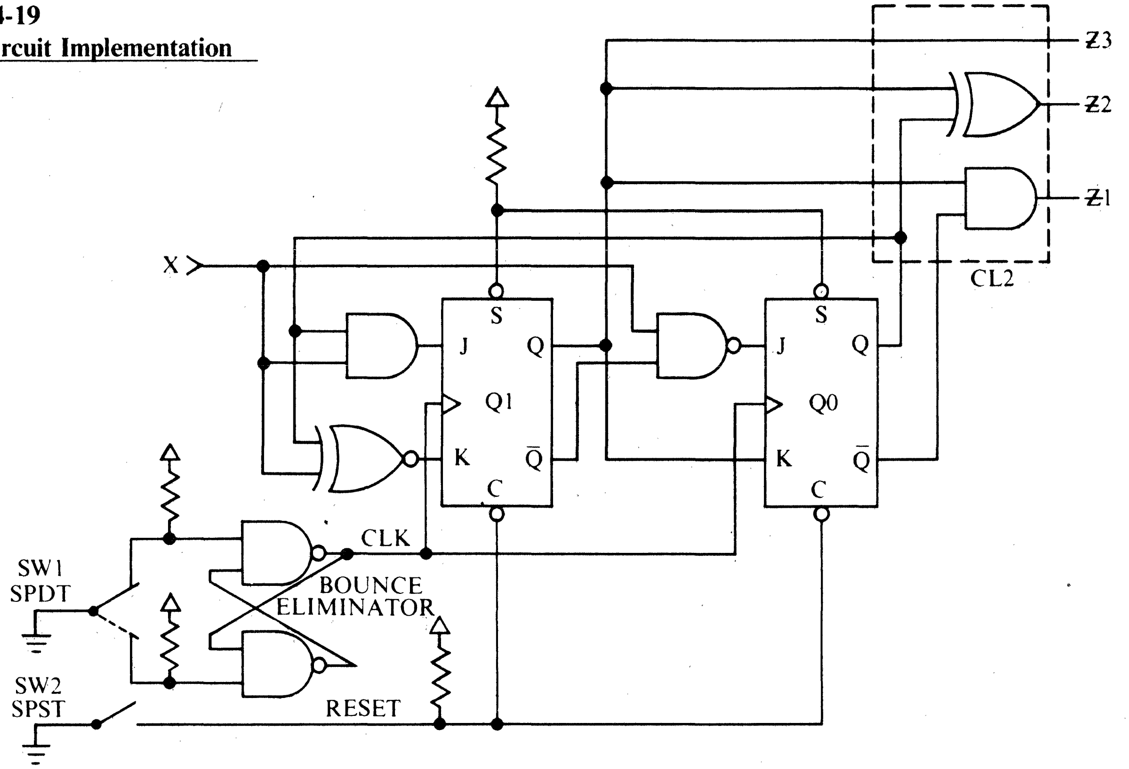
Implemented Design of SM1

Finally, the implemented design is presented in Figure 4-19. For simplicity, mixed logic-gate types are used. The block of logic labeled CL2 translates the present-state into the present-output function. The balance of the combinational logic (except for the switch-bounce eliminator) is CL1 of the Moore-machine block diagram of Figure 4-3. Its inputs are the present state and the present input, and it produces the next-state excitations that are—in reality—the next-state function. The FFs Q_1 and Q_0 are of course the SSR of SM1. Every SM has an SSR whose size establishes the maximum number of states available for possible use in its state set. When people say that a computer can have a number of states that approaches the number of stars in the universe, they are exaggerating greatly. The fundamental states of a computer are those of its SSR, and this is of very tractable size. Some might wish to add to this figure the state of every FF in the data registers and that of every transistor in the entire system, but this is not a useful approach for comprehending a system. It is the SSR alone that defines the basic sequential behavior of a computer.

Latch Bounce—Eliminator Clock

The switch-bounce eliminator circuit and the reset line are shown in Figure 4-19 to provide the necessary information for breadboarding a circuit. The 74LS279 IC provides four such circuits in one IC. The pull-up resistors shown provide better noise immunity and are a must in commercial practice. Typical values are 1KΩ for standard TTL and about 5KΩ for the LS varieties of TTL. All the components, including IC-prototyping boards or kits, are readily available at electronic hobbyist shops. The home-brew approach is strongly recommended to those who wish to *really*

Figure 4-19
SM1 Circuit Implementation



understand what a computer is. After all, your livelihood may depend on one, so why not philosophically master an understanding of its hardware? The construction details of the next chapter present more information on home breadboarding. For example, a 6-volt camping lantern and two or three silicon diodes make a perfectly adequate power supply for these projects.

A final word in summary: We have reviewed here some of the major principles behind the physical implementation of a circuit for a state machine. It is one of the pieces in the mosaic of state-machine theory. Computers are state machines, and the programs they implement are state machines. The theory of compilers and their implementation often resorts to state-machine tactics to reach its goals. Operating systems are state machines. Finite-state automata theory and its applications to finite-state machines are very pervasive. They are among the fundamental tools in computing. If we have dispelled the mystique surrounding basic details of their circuit implementation, so much the better. We shall return to this topic when we implement a microprogram—a state machine by another name. There, we shall find that it indeed has the same format and fundamental behavior as exhibited here.

BIBLIOGRAPHY

Bartee, T.C., Lebow, I.L., and Reed, I.S. *Theory and Design of Digital Machines.* New York: McGraw-Hill, 1962.

Booth, T.L. *Sequential Machine and Automata Theory.* New York: John Wiley & Sons, 1967.

Caldwell, S.H. *Switching Circuits and Logical Design.* New York: John Wiley & Sons, 1959.

Denning, P.J.; Dennis, J.B.; and Qualitz, J.E. *Machines, Languages and Computation.* Englewood Cliffs, New Jersey: Prentice-Hall, 1978.

Mealy, G. H. "A Method of Synthesizing Sequential Circuits." *Bell System Technical Journal* 34, No. 5 (September 1955), pp. 1045.

Moore, E. F. "Gedanken Experiments on Sequential Machines." *Automata Studies, Annals of Mathematical Studies* No. 34, 129–53, Princeton: Princeton University Press, 1956.

Phister, M., Jr. *Logical Design of Digital Computers.* New York: John Wiley & Sons, 1959.

Unger, S.H. *Asynchronous Sequential Switching Circuits.* New York: John Wiley & Sons, 1969.

PROBLEMS

1. A synchronous sequential-code recognizer, called SM2, is to recognize the string "110." The least significant bit arrives first. If an error is made in the proper sequence, then the machine is to return to the initial (home) state and start looking for the full sequence again. The output is to be 1 only in the clock period following the clocking in of the successful string.

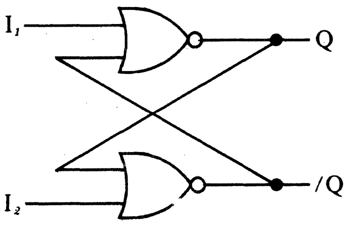
   a. Draw the Moore-type state diagram of SM2.

   b. From this state diagram, construct the state table of SM2. Assign binary codes to the names of the states in K-map Grey Code order, such that the states visited along the successful string follow the sequence of the reflected Grey Code used in K-maps.

2. Derive the application equations for Problem 1 above, simplify them, and draw the circuit. Use the excitation table of the 7476 flip-flop derived in the text.

3. Derive the characteristic equation and the excitation table of the 74109 Flip-Flop from a manufacturer's TTL data manual. Use active-high logic conventions. In deriving the CE, it may help to rename the J and /K inputs I1 and I2 temporarily, until the last step.

Figure 4-20
NOR-Gate-Based SC Latch



4. Derive the characteristic and auxiliary equations of the SC-type latch shown in Figure 4-20. Use active-high logic conventions. Why do its logical properties differ from those of the 74279? Hint: First ascertain the *not allowed* input state. Why must this input combination be prevented?

5. Derive the characteristic equation of the 74279 SC latch, using active-low logic conventions. Is it the same as the CE derived in the text, using active-high logic conventions? Compare this result with the CE result of Problem 4.

6. Simplify the design of SM1 by eliminating outputs Z3 and Z2. Instead, we can directly use the outputs of its SSR to indicate the state path. Output Z1 is to be decoded from the states of the SSR.

   a. Derive the logic circuit design.
   b. Implement it as breadboard experiment.

7. Show that SM1 follows the prescribed path for its successful string by demonstrating that its excitation logic design "steers" Q_1 and Q_0 correctly. Use a table of the form shown in Figure 4-21 and consult the manufacturer's 7476 characteristics table as necessary, to determine the next states from the present excitations derived in the current state. The initial state is the home state $Q_1Q_0 = 00$.

8. Outline the automation of a state-machine design procedure for finding the application-equation K-maps as described in this chapter. Do this by drawing the flowchart of a procedure for systematically scanning a given state table to detect state transitions of each state variable as a function of Q and I. Then find the prescribed values of inputs I_1 and I_2 from a given excitation table, and finally post these values in a set of application-equation K-maps. Include the steps required to print out the K-maps.

9. Write a computer program to implement the flowchart of Problem 8 above, using as your data base the state table for SM1 and the 7476 excitation table given in the text.

10. Implement the machine SM2 described in Problem 1 above, using 74109 flip-flops.

Figure 4-21
SM Excitation Logic-Verification Table

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
$Q_1 =$	0								
$Q_0 =$	0								
$JQ_1 =$									
$KQ_1 =$									
$JQ_0 =$									
$KQ_0 =$									

11. A sequential-code recognizer called SM3 is to recognize the bit stream "0110" whenever it occurs. Its output, Z, is to be 1 only in the period following the clocking in of the successful string.

    a. Find the application equations of SM3. The corresponding next-state and present-output entries of unused states may be treated as don't-care conditions. Base the design on the 7476 flip-flop.

    b. Check the correctness of your design by using an SM excitation logic verification table similar to that in Problem 7 to determine whether SM3 correctly steps through the states of its successful string.

CHAPTER 5 STUPIDD V– A MICROPROCESSOR ANALYSIS AND CONSTRUCTION PROJECT

COMMON IC'S IN A SYSTEMS CONTEXT

In this chapter, we analyze the design of the CPU portion of a microprogrammable processor and then explore its functional behavior. At this stage, we will develop microprograms to control the CPU, which may then be manually demonstrated using the associated construction project. Computer-description language notation is introduced for use in the microprogramming activities. We are about to study the fundamental use of actual, commercially available IC's in the creation of a functional processor *systems* context. Understanding and being at ease with the overall systems aspects of a digital processor design is a necessary prelude both to the use of modern programmable peripheral processor IC's and to creating the logical systems design of one. This has become mandatory of late, due to the economic availability of sophisticated controller and processor IC's to be programmed or otherwise utilized in design projects. It should not come as a surprise at this point that the key concept here is the application of *chips*, or IC's, in a *systems context.* We have stressed in this text that a limited number of IC's exhibit the functional patterns of behavior found within any processor. Forming a processor-system from IC's, analyzing and operating it, teaches one a good deal about all the others—for the features used in their organization are common to all.

Analyzing a given design can be a valuable learning experience for the beginner. While we have studied related material on logic design and devices, we have yet to see how they are integrated into a processing system. This task is actually an easy one. Far more creative is the task of inventing instructions for demonstration on the construction project associated with the text. Microprogramming is also the art of creating instruction-set algorithms that we use in programs. To specify the operations we want to implement on the processor, we shall resort to the use of symbolic notation of a design language. This material is introduced in the microprogramming section of this chapter. At this stage, the symbology is used as a form of register-transfer notation. Computer-hardware and instruction-set reference manuals use this type of notation to describe concisely exactly what an instruction does during its execution. We will be analyzing a design and creating its instruction sets. In the context of a design language, the notation can be used to specify a processor's design. It is possible to use this background in the synthesis of processors, but that requires another text.

The material in this chapter was the result of a number of projects designed to explain, through lecture and laboratory activities, the inner workings of a microprogrammed computer architecture at the actual hardware level. These projects were referred to as Student Projects In Digital Design–the acronym *STUPIDD* appealed to the students, and the name stuck. The concept presented here was the fifth in an evolving series, resulting in the title *STUPIDD V.* It was kept simple–stupid, if you wish–to provide the reader with a project that can be constructed with readily available IC's. STUPIDD permits one to explore–on paper as well as by construction–the system control and execution of most types of instruction found in current computers. We will learn how to implement new instructions and practice the art of microprogramming in this process. Even though short demonstration programs may be executed, STUPIDD was never intended to be used as a microprocessor–only as a learning tool. The chip count accelerates rapidly when one wants to turn it into a complete system–the added increment of learning about basic computer operation is small compared to the effort involved. This time would be better spent on, say, studying the application of microprocessors or bit-sliced machines using the 2901 type of IC's.

COMMONALITY OF FEATURES OF ALL PROCESSORS

STUPIDD's CPU is archetypical of von Neumann types of architecture. Most modern minicomputers and microprocessors that we encounter in normal practice fall into this category. The block diagram of STUPIDD's CPU is presented in Figure 5-1. Compare this with the block diagrams of the commercially available Intel 8080 and the AMD 2901 bit-slice processors, shown in Figures 5-2a and 5-2b. Note that all three, like any CPU, have certain features in common–a register array (R), an arithmetic logic unit (ALU), one or more general-purpose registers (T) for temporary storage, and, last, interconnecting bus paths. These four features, outlined or self-evident in the figures, are the common core of most modern von Neumann architectures. STUPIDD also typifies the architecture used in the basic PDP-11 minicomputer, with a similar organization of the same features, as shown in Figure 5-2c. For the beginner, the hope is that an appreciation of STUPIDD demystifies the *intrinsic* nature of the majority of the more sophisticated real-world processor systems likely to be encountered. In fact, there is a particularly close correspondence between the features of STUPIDD's CPU and the essentials of the 2901 4-bit-slice processor, from which many minicomputers and mainframe emulators were constructed.

RATIONALE

One could directly use the 2901 4-bit-slice IC to replace all of STUPIDD. That would be the right thing to do if one already comfortably understood the concepts we are about to study, all of which are contained in the 2901. Computer science students, who generally were lacking in previous hardware exposure, have stated that the wire-wrap project to build the system provided a true appreciation for IC behavior they would not otherwise have gotten. Experience has shown that it is more effective to start on the path to understanding the overall systems behavior of these processors by first considering the separate embodiment of many of their internal functions, which are already

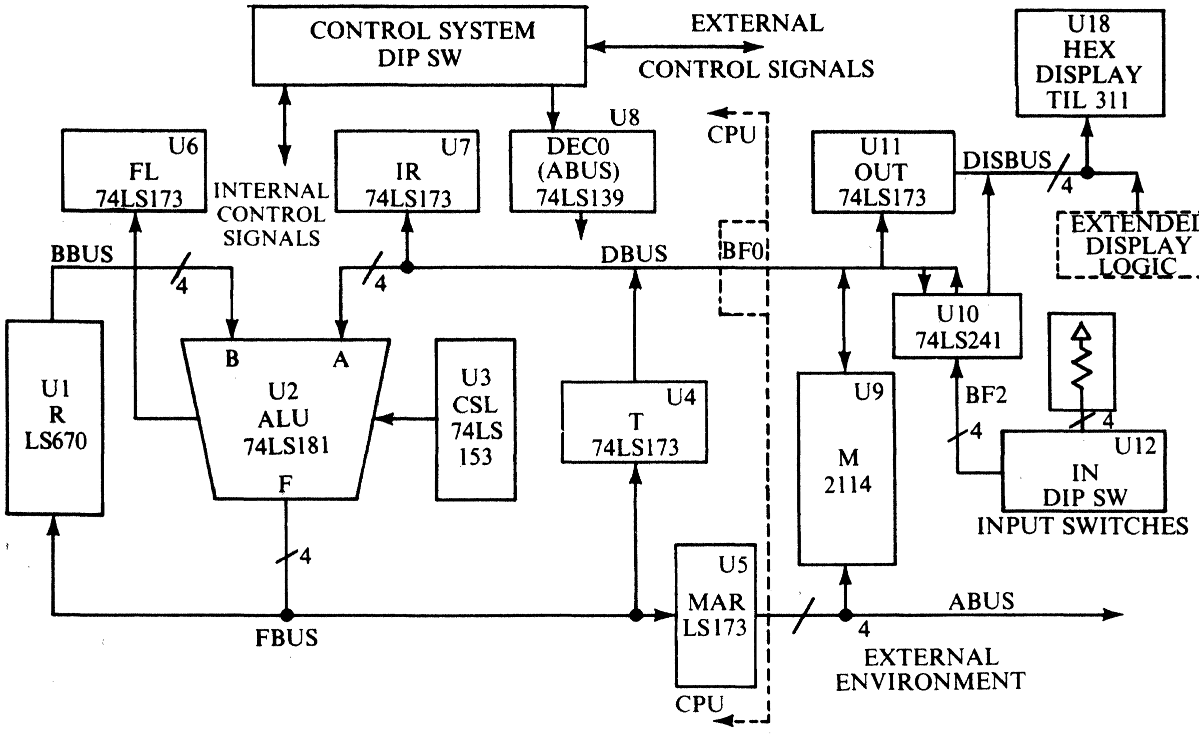


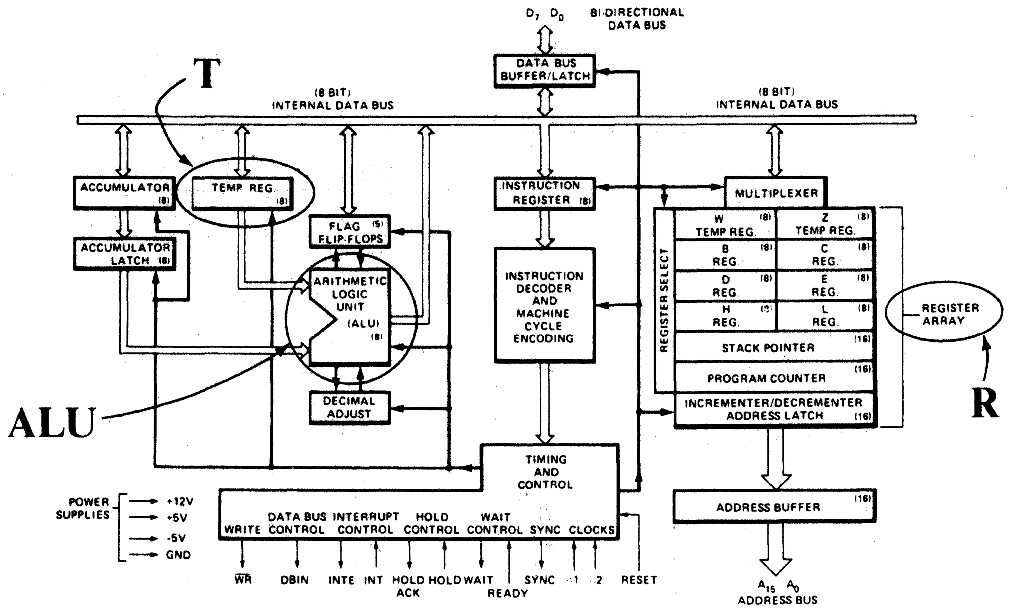
Figure 5-1
CPU Block Diagram:
STUPIDD V

observable, in readily available IC's. Next comes an understanding of the system coordination of these functions. Finally we arrive at the means of specifying the control system of a microprogrammed processor. At that point, we should be reasonably familiar with the inner workings of commercial programmable chips and therefore ready to work with them. In fact, the Very Large Scale Integrated (VLSI) circuit microprocessor construction know-how evolved out of the ability to incorporate more and more of the smaller IC's, such as those we shall explore, into a single LSI or VLSI design. The techniques used here, however, were selected to display systems operation principles, as simply and clearly as possible, rather than to show the most effective way they are actually implemented by industry.

Processor Interface Signal Categorization

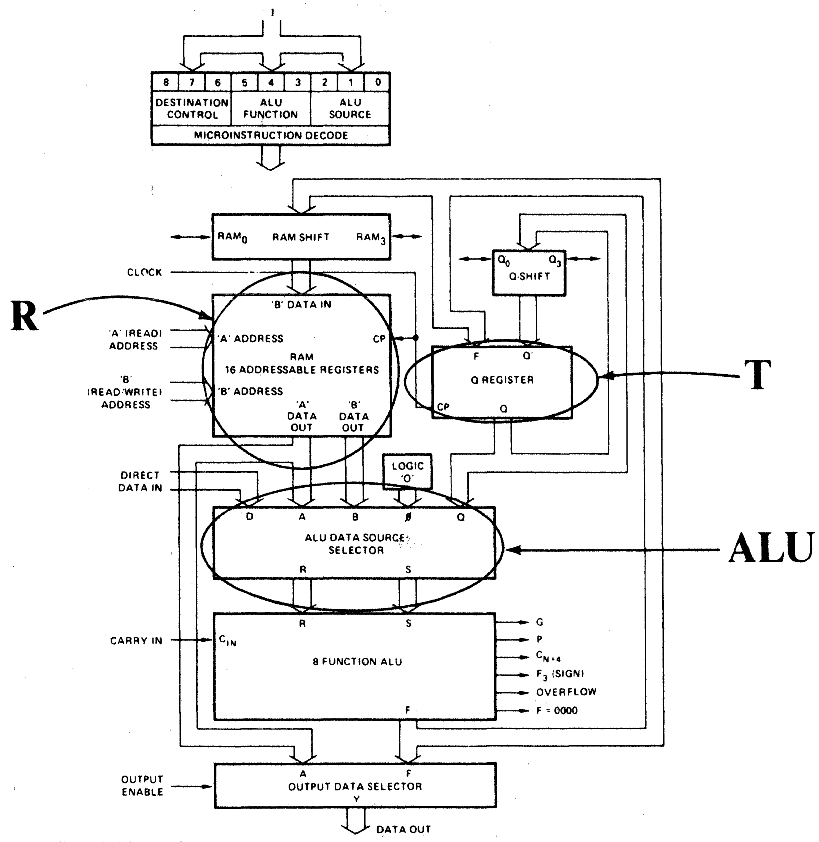
The 4-bit slice, such as the 2901, contains only the minimal features of a CPU: i.e., R, ALU, T, and bus paths. The complete microprocessor CPU's have a few other embellishments, which are also included in Figure 5-1. Three categories of signals are to be found at their interfaces to the external environment. First is the data bus, DBUS, buffered by a bidirectional buffer, such as a 74LS242, represented here by the dashed block, BFO. This chip is not necessary for the current design, so it is not used—its symbolic presence is a reminder of the existence of the bidirectional interface buffers at this place in the architecture of commercial processing systems, such as microprocessors. Second is the address bus, ABUS, buffered by the memory-address register, MAR. The third category of interface signals is the control system's external interface-control signals. The control system itself will be dealt with in the next chapter. These three categories of signals are what is commonly found at a processor's interface, which may be confirmed by categorizing the 8080's

Figure 5-2 Common Essential Features of Typical CPU's



A. 8080 CPU
(Reprinted by permission of Intel Corporation. © 1983, Intel Corporation.)

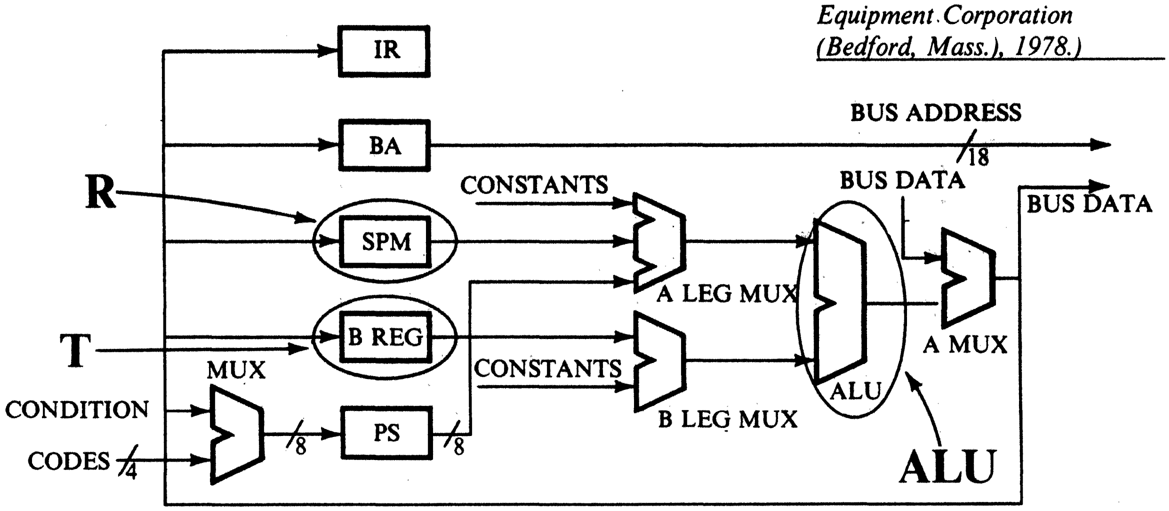
MICROPROCESSOR SLICE BLOCK DIAGRAM



B. AMD 2901
(Copyright © 1979, Advanced Micro Devices, Inc. Reprinted with the permission of the copyright owner. All rights reserved.)

Figure 5-2c
Common Essential Features of Typical CPU's

(Reprinted with permission from Computer Engineering: A DEC View of Hardware Systems Design, *by Bell Mudge McNamara. Copyright © Digital Press/Digital Equipment Corporation (Bedford, Mass.), 1978.)*



NOTE: ALL DATA PATHS ARE 16 BITS WIDE UNLESS OTHERWISE NOTED.

signals, displayed in Figure 5-2. These classifications of the interface signals will also become evident in the tripartite presentation of a processor that we are about to pursue. The total system consists of a CPU, a control system, and an external environment (memory, input/output, and a display). The interfaces of the system are analyzed in terms of data, address, and control-bus signals.

Specialized Register and Logic Usage

Internally, there are additional blocks in the typical CPU, besides the aforementioned ALU, register array, T register, and bus paths team. An instruction register, IR, holds and presents the instruction word to the control system during the execution phase of the computer's cycle. Properly considered, it is a part of the control system that is generally inaccessible to the user, but it is customarily displayed in the CPU block diagram. An internal flag register, FL, is also shown. This register stores status information supplied by the ALU, such as the present sign or carry status, when directed to do so by the control system at critical points of operation. The final internal block we shall deal with in this bare-bones system is the carry select logic, CSL. As we shall see, the carry input to the ALU can come from one of several sources, which the control system selects via CSL.

Externally, our system contains a random access memory, M, a set of input switches, IN, and an output register, OUT. The output register shown can drive a hexadecimal light-emitting diode (LED) display. Additional display logic is provided as a convenience to the user, providing for the visual display of the current state of the buses and registers of interest in the hex LED IC. The input-switch connections to the DBUS are buffered by half of the 74LS240 IC, denoted as BF2. The IN and OUT facilities represent only the most fundamental of I/O communications with external devices.

We are now ready to study the detailed operation of microprocessor features, through the use of common IC devices. What is so intriguing about STUPIDD V is that most of the mysteries of CPU behavior can be revealed by using approximately ten types of logic IC's, in a systems context. It is simple enough for the reader to construct and verify the operational principles. To this end, construction details are presented at the end of the chapter. While paper and pencil exercises alone may provide sufficient intellectual insight into computer operation, those with insufficient hardware experience will find that conquering a construction project provides an added measure of satisfaction. For others, seeing is believing. Building the project is simple, because all the pertinent details are given. Microprogramming it requires creativity based on a knowledge of the hardware details and function.

DEVICE CONTROL AND NOMENCLATURE

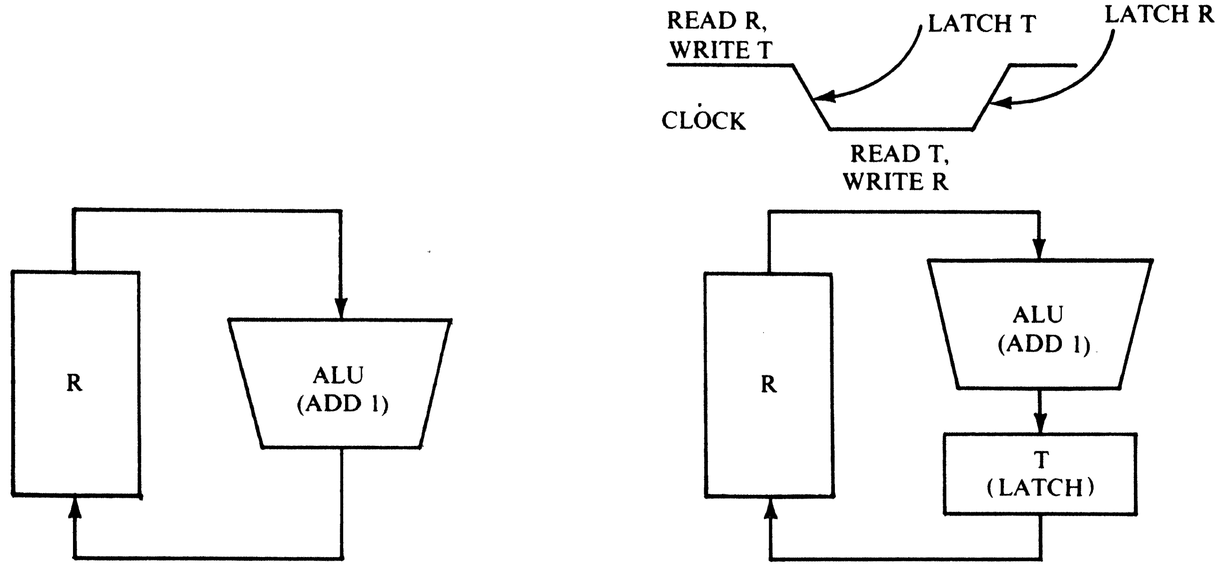
STUDY OF INTERFACE SIGNALS FOR CONTROL AND COORDINATION OF IC'S

Our study of the characteristics, behavior, and control of processors begins in earnest with a survey of the nature and control of the IC's utilized in STUPIDD's design. The system is built upon their characteristics. A TTL data manual, of course, must be referred to while reading this material, to gain a true appreciation of the logical and electrical characteristics of these devices. The major goals of this section are to help you become familiar with the *interface* signals of the logical devices we shall use and to provide understanding of the methods used for their control and coordination. Figure 5-1, the block diagram of STUPIDD, presented the relationships of these IC's to each other within the system. The U-numbered designations of the IC's are simply short names by which each may be conveniently recognized. We begin our survey of these devices and their manner of application in STUPIDD with the register array, R (U1), as typified by a 74LS670 IC.

R, THE REGISTER ARRAY (U1)

Transparent Latch
Simultaneous Read and Write

The 670 is a 4 × 4 register array, possessing separate read and write enables. While, to keep things simple, we will not take advantage of this property here, virtually (i.e., not actually) concurrent reads and writes of an array of transparent latches are fully utilized in the 2901 bit slice. The transparent latch was introduced in Chapter 3. An important constraint of a transparent latch is that one cannot simultaneously read and write the same register. Simultaneous read and write operations on the 670 are restricted to separate addresses on the read and write address lines, respectively. The fundamental reason for this constraint lies in the nature of the transparent-latching characteristic of this IC. When write enable (WE) is active–low in this case–the behavior of the transparent latch is such that the data output will follow the input.



a. Purely Combinational Loop (Oscillates)

b. Interposing Transparent Latch (2901 Latch Is at Output of R.)

Figure 5-3
Simultaneous Read, Write of Transparent Array

There are two aspects to the problem presented by this characteristic of transparent latches. First, the early modification of the output permitted by the active level of the clock signal (as opposed to edge-triggered) can cause incorrect data to be clocked in elsewhere, where the unmodified register contents are expected to be held available at least until the start of the next clock period. Fortunately, this situation can be avoided entirely with proper system design. The more important second case occurs when transparent latches are used within closed loops of combinational logic. Under these conditions, oscillatory changes of data may occur. Fixes for this condition use a pair of separately clocked latches. For example, the 2901 4-bit slice IC utilizes an internal latch, separate from its register array, which captures read information before writes commence. That is, reads are latched into an auxiliary register on the clock edge in the middle of a system period, while writes are recaptured by the original source in the transparent register array on the clock edge at the period's end. Using this intermediate register prevents data oscillation.

Let us further clarify the second point above, since the use of transparent registers in processors is widespread and this common situation *must* be avoided in practice. Figure 5-3 shows two partial architectures, each performing a simultaneous read, modify, and write into the same register. Let us presume that the ALU in each loop is adding 1 to the output of the register array. Further, the T register and the array are of the transparent-latch type. In Figure 5-3a, the incremented output of the ALU is fed directly back to the inputs of the transparent latch. While both the read and write clocks of the same register are simultaneously active, information coming out of the selected register is fed to the ALU, modified, and then fed right back into the same register. Naturally, it passes through the same combinational loop again and again, resulting in continuous oscillatory modification of the data until the read and write clocks cease their simultaneous active mode. Where it stops, no one knows.

The architecture shown in Figure 5-3b is somewhat analogous to the 2901 situation, as well as the pair of registers that form the

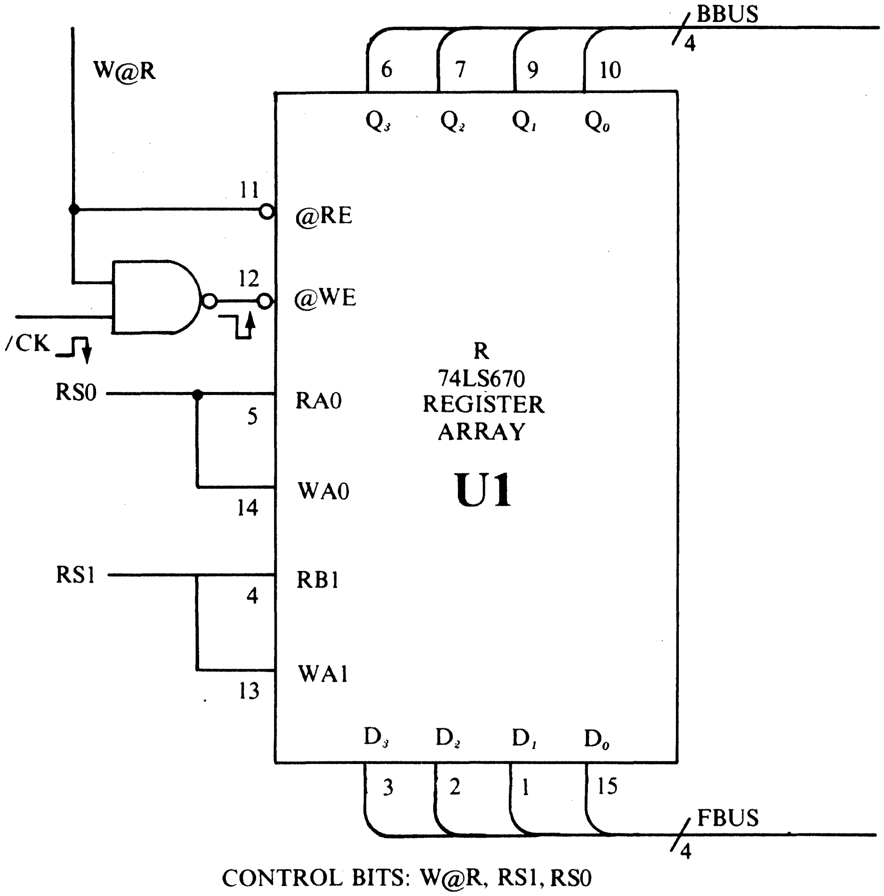


Figure 5-4
Register Array R Logic Diagram

accumulator in the 8080. Data from a transparent-latch type of source, which may be modified in passing through an ALU, is first locked into the separate T latch on, say, the falling edge of the clock. The transparent read is now terminated, and the transparent write into the originally read register commences. Figure 5-2a explicitly displays this relationship for the 8080's accumulator. Consider how you would increment this accumulator by one. The write data could come from anywhere, but the simple closed path in Figure 5-3 illustrates how potential oscillations are blocked by this technique. This solution uses time separation (an application for two-phase clocking) of the active levels of the read and write clocks applied around the data loop. It only appears to be simultaneous to the outsider, because it all takes place within one system clock period. STUPIDD's simplified architecture avoids all these problems, but it is less flexible as a result.

CONTROL SIGNALS OF R REG.

Let us examine the control signals for the register array, R. Figure 5-4 is the logic diagram of the 670, as we shall employ it. The read enable (RE) input to this IC is an active-low control line coming from the control system. The write enable (WE) input is also active low. To avoid simultaneous reads and writes, we make the level of these two signals mutually exclusive by employing the NAND gate at pin 12,

@WE. (The @ symbol means the logic name following it is active low.) The signal from the control system driving these now mutually exclusive inputs is called W@R in the figure. When it is low, we are reading the addressed cell of the array. When it is high, we are writing under clock control, as provided for by the complemented system-clock input to the NAND gate. The R register array's tri-statable data-output interface is controlled by @RE. When RE is low, the addressed register is enabled onto the BBUS.

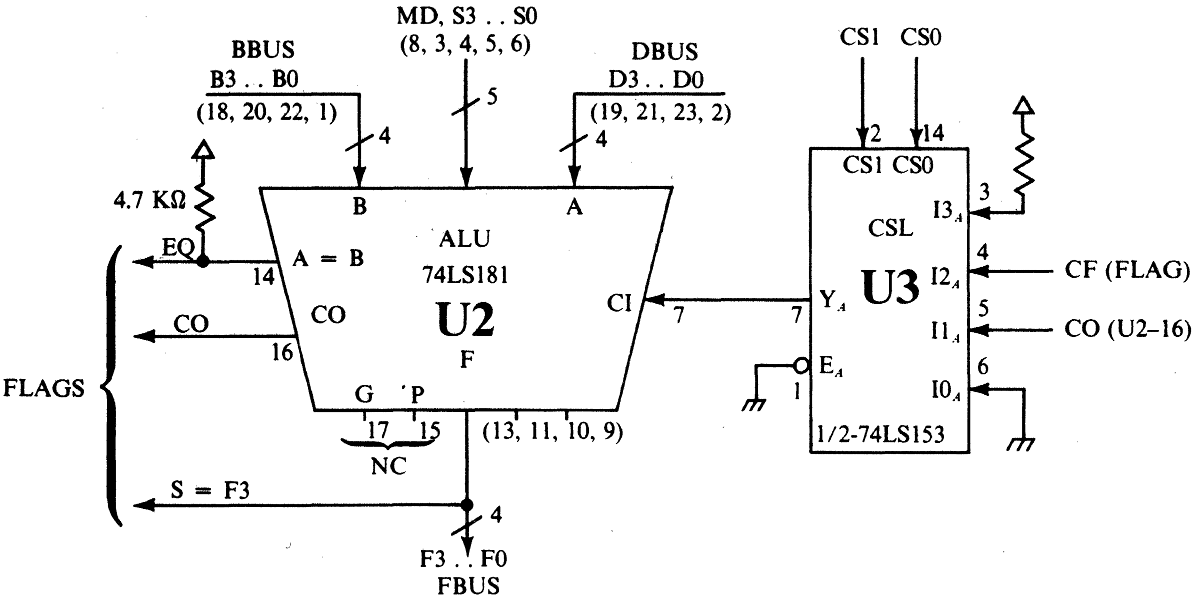
Write/Read Control

Reading is "safe," in that it cannot modify a register's contents. Writing can be hazardous to data if spurious transitions of the control signals occur during state changes of the system. The potential hazard of accidental writes is overcome by the filtering action of the NAND gate driving @WE. Noise spikes on the control signals during state transitions settle down by midperiod. With the use of the NAND gate, the system clock's complement, /CK, now coordinates the production of an active-low WE signal *only during the last half of the present period.* It masks possible noise during the first half. When /CK is low, @WE is clamped high. Only when /CK is high (in the last half of the system-clock period) AND W@R is also high can @WE go low.

Register Select Control

The register-select control lines, RS0 and RS1, also come from the control system. They simultaneously control the corresponding read (RA0, RA1) and write (WA0, WA1) register selection address lines of this IC. Concurrent reads and writes are precluded in this application by the mutual exclusivity of @WE and @RE, described above. Therefore we have no need to be concerned with the fact that these read and write addresses are always the same.

Three bits coming from the control system are required to manage the 670 register array R properly, namely: W@R, RS1, and RS0. We may manipulate these signals, via microprogramming, to make the 670 register array perform the tasks we choose for it. The four internal four-bit registers may be used in any general way we prescribe when we invent instruction-set algorithms, or *macros,* for the control store. One register of the four, let us say R3, shall be reserved for use only as the program counter, the PC. All future references to the PC shall therefore imply R3. The remaining registers of R are used in different ways to illustrate register-usage variations that occur in practice. The 2901 contains an array of 16 registers similar to those in this IC; commercial microprocessors vary widely in the number of their registers. Often, they exhibit transparent-latch behavior.



CONTROL BITS: M, S3, S2, S1, S0, CS1, CS0

Figure 5-5
ALU and CSL Logic Diagrams

THE ALU (U2) AND CSL (U3), CARRY SELECT

The Schottky version of the 74181 ALU was the device that drove much of the early minicomputer industry. We shall use the low-power Schottky version, to reduce power, in the manner shown in Figure 5-5. This chip contains carry-generate and propagate pins that can increase the speed of arithmetic operations on the order of 30 percent when four or more ALU's are used in parallel. These pins are not relevant in the current application, but they merit independent study, along with the 74182 fast-carry look-ahead IC. The 181 was discussed earlier, so its features will be only briefly reviewed now.

ALU Control Signals

Six signals control its behavior: mode (MD), carry-in (CI), and the four select lines (S3 .. S0). The MD line chooses between the logic (MD = H) and the arithmetic (MD = L) modes. In the logic mode, the device ignores the carry-in line. Logic operations are performed bit by corresponding bit, and the carries are irrelevant under these conditions.

Carry In/Out Activity Level

For arithmetic operations, the carry-in *is* relevant, and CI is presented to the ALU by the carry select (CSL) IC. An aspect of this ALU's carry lines, both CI and CO (carry-out), is that they *always* possess the opposite activity level ascribed to the data ports—that is, when data is treated as active high, carries *must* be treated as active low, and vice versa.

Open-Collector Nature of A = B

The A = B equality-detection output is of the open-collector type, to facilitate wire-ANDing of these outputs when several ALU's are used in parallel. Naturally, it requires the use of a pull-up resistor to function correctly. This output can also be used, in conjunction with the CO line, for the detection of inequalities. The sign flag is the most significant output bit of an array of ALU's. Since we are using only one ALU, this is F3 in our case.

Carry-In Selection Control

As noted in Figure 5-5, the signals we use to control the ALU are MD, S3, S2, S1, S0, CS1 and CS0. The last two indirectly affect the ALU during arithmetic operations, by controlling the selection of the signal level to be applied to CI and therefore the ALU behavior in the arithmetic mode. CSL is a 74LS153 dual 4-line-to-1-line data selector, fully described in your TTL data catalog. As employed here, its selection-control (address) lines may choose one of the four following items for presentation at its output: An unconditional logic high, an unconditional logic low, the conditional value of carry-out, or the conditional value of the previously stored carry flag. The one of these four actually selected by the control system is presented to the ALU's carry-in line, CI. This repertoire permits both the absolute control of the value of CI, as well as its *conditional* control, as shown in Table 5-1. This CI selection process plays a major role in the creation of instruction sets.

Table 5-1
Carry In (CI) Selection Control Signals

CS1	*CS0*	*Source of CI*
L	L	Low logic level
L	H	CO of the ALU
H	L	CF bit of FLag register
H	H	High logic level

ALU Shift Operation

Left-shift operations, either circular or through the previous carry, may be demonstrated. While far from the flexibility of a full-blown microprocessor, these operations provide sufficient illustrations of real practice to be educationally useful. Shift operations are not generally performed with the ALU in practice. We shall discuss shift operations further when we come to the T register.

ALU Flag Generations

While we are able to derive several important flags with the simplified design used here, the overflow flag is omitted to reduce the IC count. In two's-complement (TC) notation, an overflow may be detected if the carry into and the carry out of the sign bit are different. An Exclusive-OR gate can detect this difference nicely, but we simply do not need another IC here for tutorial purposes. The flag signals generated by the ALU are the Carry Out, the Sign, and the A = B flag signals. These are stored in the Flag register.

T, THE TEMPORARY REGISTER (U4)

Need for a Load/Shift T Register

A 74LS173 is selected as the T register because its built-in controls for tri-stating the output and for clock-enabling register loads simplify our tasks. Actually, a generalized load and shift register, such as the 74LS194, would better illustrate the type of capabilities utilized at this location. The 2901's Q register is comparable to the 194, which unfortunately does not possess output and clock-enabling controls. In conjunction with an internal multiplexing scheme, generalized shift operations are thus effected, external to the 2901's ALU. These generalized shift and load features are found in the 194. Exercises at the end of this chapter deal with this more generalized case, implementation of which would increase STUPIDD's chip count too much.

T Register Control

Figure 5-6a shows the 173. Its two clock-enable lines are tied to the signal from the control system, @LT (Load T). The output-enable inputs, OE, are tied to the control line, @ET (Enable T). The outputs Q3 .. Q0 are thus fed onto the time-shared internal portion of the DBUS, under system control. Input data is presented by the FBUS to T, which will be parallel loaded when it is clock-enabled.

MAR, THE MEMORY ADDRESS REGISTER (U5)

MAR Register Control

Figure 5-6b presents the logic diagram of the memory-address register, MAR. MAR's function is *only* to select the location (address) in primary memory the system wishes to communicate with. It does not handle any data. This, too, is a 173 parallel-load register, with the clock-enabling line, @LMAR, coming from the control system. The restrictions we have placed on STUPIDD are particularly evident here. Four address lines provide an address space of only sixteen memory locations–barely adequate to illustrate the execution of basic instructions and programs of a microprocessor. Yet, if we can generate these, there will be very little that we don't understand about the operation of a commercial processor.

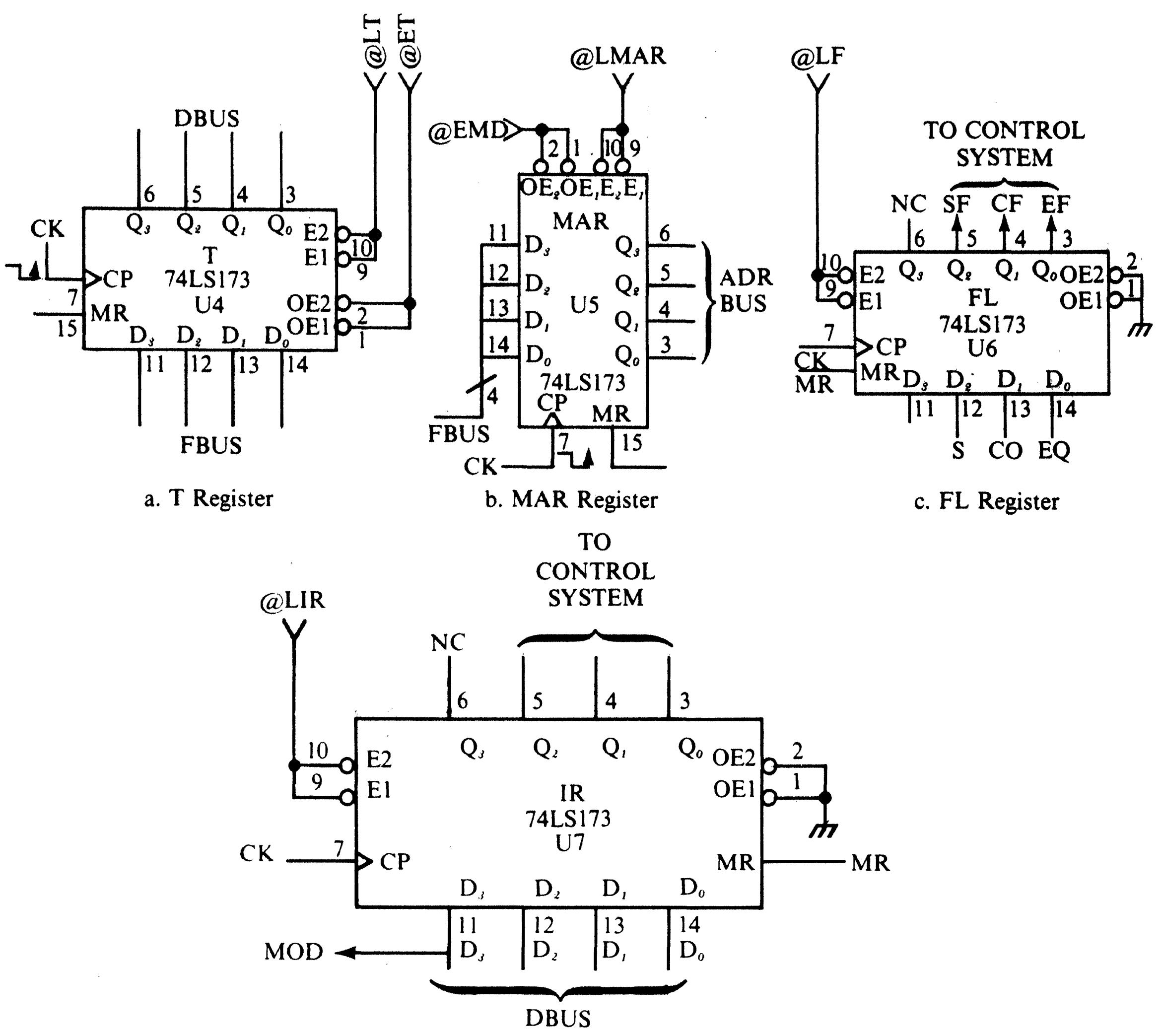
DMA-like Operational Aspects of MAR

An interesting side point is that the tri-state control of MAR is enabled by the @EMD signal during normal operation. In a production model, this line will be controlled by the control system. This feature is useful in direct memory access, or DMA, operations. The DMA procedure lets a processor electrically isolate its own data, address, and appropriate memory-control lines from the interface buses, thus permitting another external device to utilize the memory system. The external device must now provide data, address, and control signals to the memory. Thus, here we see fundamental operational features of a processor system that supports DMA operations are present in this system. If the control system is also built, MAR can be disabled by @EMD for down-loading demonstration programs into the memory system. This is not normal operation, but it will reveal some important aspects of actual DMA operation.

FL, THE FLAG REGISTER (U6)

Flag Storage

The stored status flags used in STUPIDD are the carry, sign, and equal flags, referred to as CF, SF and EF, respectively. During some ALU operations, it is important to save the status of the present value of these flags in the flag register, FL. These stored values may subsequently be tested and used to control the flow of either an instruction or a program; this important aspect of systems operation will be illustrated later.



a. T Register

b. MAR Register

c. FL Register

d. IR Register
CONTROL BITS: LIR

Figure 5-6
Register Logic Diagrams

Program-Status Words

This is not an optimal selection of flagged conditions (flags are sometimes referred to as condition codes or are part of the so-called *program-status word*), but it is adequate to illustrate the operational use of flags.

Minimal Sets of Flags

An often-used minimal set of flags consists of the 2901's sign, carry, zero, and overflow flags. FL is implemented with yet another 173 IC. The control system provides the clock-enabling signal @LF to cause the flag information to be stored, as illustrated in Figure 5-6c.

Need for Flag-Bus Interfaces

In a marketable system, the FL register would be interfaced to the data bus. The reason for this is that, after a call or interrupt, these flags may have to be saved on the stack in memory as part of the vital statistics of the currently running process. The ways to accomplish this are the same as in saving the PC, already discussed in Chapter 3. While we do not do this here, it is important to recognize the real need for these additional capabilities.

IR, THE INSTRUCTION REGISTER (U7)

Function of the Instruction Register

The instruction register, IR, captures the current instruction from the program stream in memory at the end of the instruction-fetch phase of the cycle of computation. During the entire execution phase of the complete cycle, the information in IR provides the control system with its current instructions. IR, it was noted, is actually a part of the control system but is generally portrayed as belonging to the CPU. Another 173 is utilized to implement the OP code portion of IR, whose logic-connection diagram is presented in Figure 5-6d. The data inputs to IR are the internal DBUS, bits D3, D2, D1, and D0.

IR Register Control

The IR does not use bit D3; this bit really goes to the MOD flip-flop in the control system, to be discussed later. This MOD bit, though, is the part of the instruction word that controls operand address modification (i.e., the indirect or indexed addressing modes) whose systems operation will be demystified later. The control-system signal that affects IR is @LIR. This signal enables the instruction word from memory, which sits on the DBUS, to be clocked into the IR register during the IF phase.

LIR Signal and Major Changes of State

An important aspect of how STUPIDD is organized (mentioned now to avoid misunderstanding later) is that @LIR is also used to enable the loading of the IF flip-flop in the control system, at the same time as IR, only to reduce the required amount of hardware. Therefore, switching between the IF and EX phases of operation is compatible with–and occurs simultaneously with–the loading of IR in this system. It should be emphasized now that the OP and MOD fields of IR are *ignored* by the control system during the IF phase of operation. Therefore the loading of garbage into the IR at the end of the EX phase has no effect on our system's operation during IF.

These devices constitute the minimal set of registers and an ALU found in most portrayals of a CPU's block diagram (Figure 5-2). Their coordinated system behavior is typical of what occurs in *any* CPU, yet this information can be studied using only these seven IC's as a foundation. Our challenge, then, is to gain an understanding of how the typical CPU behaves by learning the *systems* behavior of only seven IC's. With the help of the TTL data catalog, we can do this.

THE EXTERNAL WORLD

Memory-Mapped versus Hardware I/O

The next few devices represent a *minimal* external I/O interface to a CPU. Expanded descriptions of actual interfaces are given in the material that follows, as appropriate. External devices fall into two categories: the memory system and the peripheral devices (other external physical devices). Some architectures, notably those of the PDP-11 and the 68000 class of CPU's, treat all the external environment as memory addresses. That is, an external device responds to the same signals as main memory, when addressed. This method of handling I/O is referred to as *memory-mapped I/O.* The second approach, called *hardware I/O organization,* supports separate I/O addresses and operations for memory and peripheral devices. This is the method used here. It requires the control system to provide the control signals that distinguish between memory and peripheral operations. Further, the instruction set must now contain specific I/O instructions for peripheral communications, not just memory reads and writes.

In what follows, our input port, IN, can be thought of as information coming from a keyboard, from a floppy disk, or whatever. The single output, OUT, represents information being transferred to a CRT display, to a printer, or to a disk system. A production CPU would have far more sophisticated addressing, test, handshake, and control signals to support data transfers. At this point, however, our purpose is to show, in the simplest manner possible, how a CPU handles the flow of information. Therefore, we mention again that buffer BF0 (shown in phantom in Figure 5-1) is omitted. BF0 would be a bidirectional bus-driver interface, implemented with circuitry such as that in the 8T26/28 IC, studied in Chapter 2, or in the more modern 74LS243 IC. Consult your TTL data book for further details on this IC. The following IC's are external to the CPU but represent the type of environment that it interfaces to.

OUT, THE OUTPUT REGISTER (U11)

OUT Register Bus Interfaces

The output register, OUT, consists of another 74LS173 4-bit register. Its inputs are attached to the portion of the DBUS that is external to the CPU. Its outputs, which are tri-statable, feed the display bus, DISBUS, as shown in Figure 5-7a.

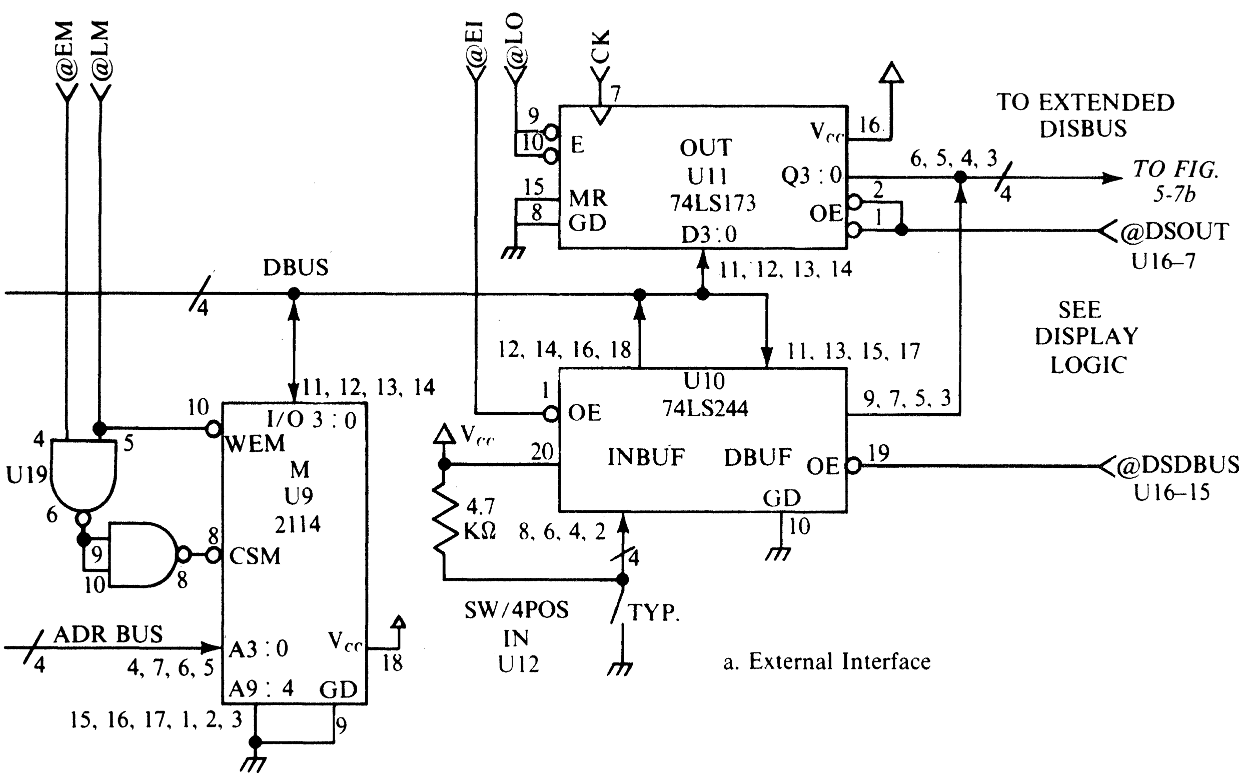
OUT Register Control

The clock-enabling control signal, @LO, governs the clocking of data. The tri-state output is controlled by the signal @DSOUT, which emanates from the one-of-eight display-select decoder in Figure 5-7b. Here, OUT implements a simple output operation whose results may be presented to an LED display. This display (U18) is also used to observe the status of many other key areas within STUPIDD. The complete display circuitry is presented in Figure 5-7b.

IN, THE INPUT INTERFACE (U10, U12)

IN Switch Control

The input consists of a set of switches (U12) buffered by half of a 74LS244 tri-state buffer (U10), with the signal, @EI, provided by the control system (via a decoder) to grant it access to the external data bus, DBUS. It too is a device external to the CPU. The DIP switches used are fed through a resistor pull-up package from the power supply. These DIP switches are single-pole, single-throw (SPST) devices that either present the supply voltage to the buffer or shunt it to ground, using the current-limiting resistors in the typical manner illustrated in Figure 5-7a. Note that the other half of the 244 is used to present the signals of the DBUS to the DIS-



a. External Interface

Figure 5-7a
Logic Diagrams: External World

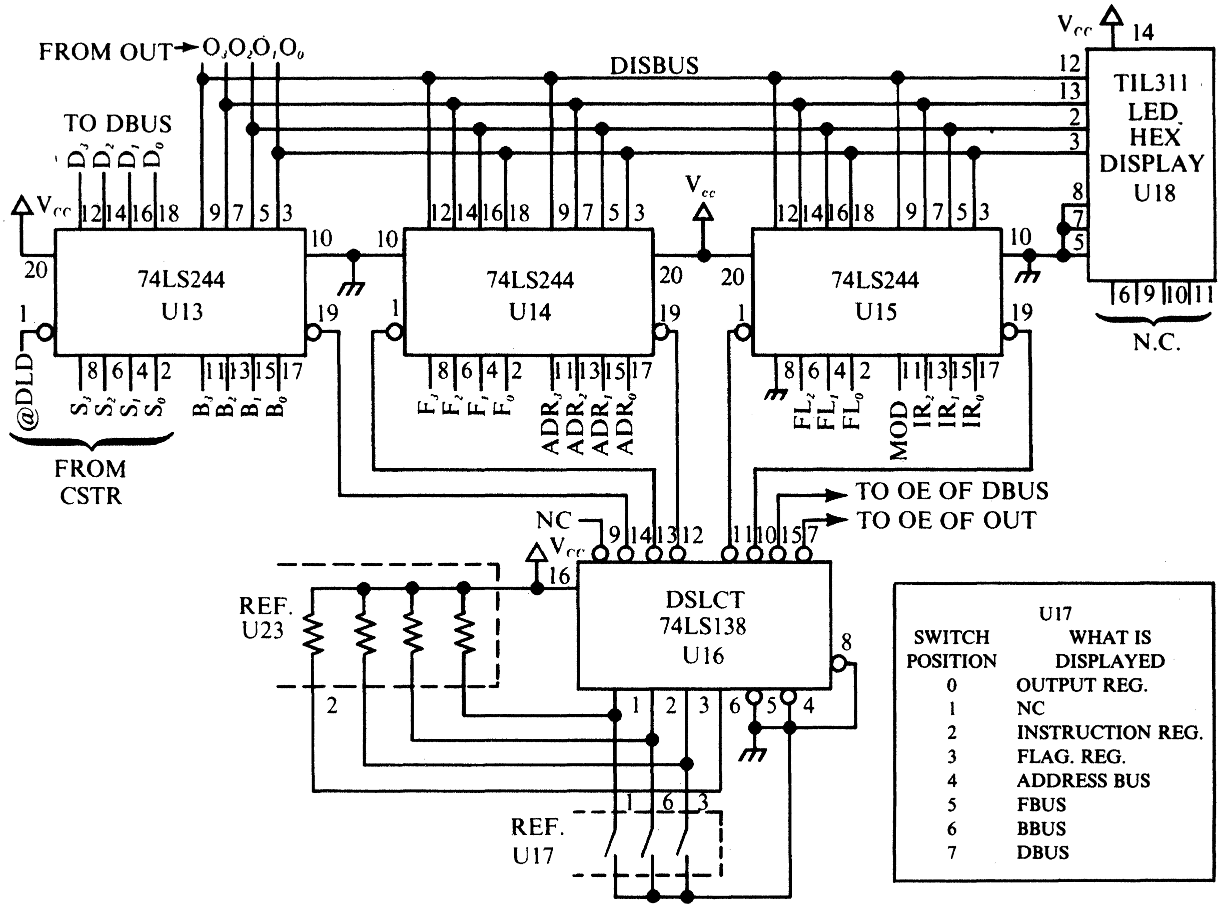
BUS for display. This action is controlled by the signal DSDBUS. U12 is an eight-position DIP switch, half of which is used by an operator to set inputs that can be stored either in memory or in the CPU's registers. Thus, in this basic CPU, *program loading and register initializations are established by the operator*, by hand operation of these switches. If the control system is constructed, demonstration programs conveniently stored in PROM may be downloaded automatically with far greater ease.

M, THE MEMORY SYSTEM (U9)

Main memory is embodied in an Intel 2114 4 × 1024 static-memory-array IC, with a common input/output port, as shown in Figure 5-7a. Why use a 1024-word memory in a system with 16 words of address space? First, to save money–this IC is readily available and economical, too. Second, this permits expanding the address space as a project.

Memory Control

The load-enable line @LM, which controls the @WEM memory-input pin, is clock-qualified by the U8 decoder IC, soon to be discussed. The reasons for this arrangement previously presented in the case of the register array R apply here as well–to prevent accidental noise-inspired writes that may occur during system state transitions, when critical control lines may not be stable. As shown in the figure, memory chip select (CSM) of this IC is controlled by *either* @EM or the clock-qualified @LM with the use of the NAND gates. If CSM is not active, the device does not respond to any of the other control signals. The memory system also uses @LM and @EM to control the writing and reading of memory and to control tri-state access to the DBUS.



SWITCH POSITION	U17 WHAT IS DISPLAYED
0	OUTPUT REG.
1	NC
2	INSTRUCTION REG.
3	FLAG. REG.
4	ADDRESS BUS
5	FBUS
6	BBUS
7	DBUS

b. Extended Display Logic

Figure 5-7b
Logic Diagrams: External World

Note particularly, in studying the data sheets for this IC presented in Figure 5-11, that this IC tri-states its own internal outputs during writes and when the chip is not selected. This is what makes the common I/O port of this IC so practical. A carefûl study of the specifications for this IC is in order. It is typical of a basic semiconductor static RAM. Its simple timing diagrams serve as stepping-stones to learning about the timing relationships of the more complex and larger dynamic RAMs.

Extended Display Logic (U13 through U18)

Display Bus Interfaces

The previously mentioned display control logic is presented in Figure 5-7b. U13 .. U15 are 74LS244 tri-state buffers, consisting of two separate sets of four buffers each. Two separate tri-state enables provide independent output control for each set of four. These display enables come from the 74LS138 one-of-eight decoder (U16) that is driven in turn by a user-operated binary coded decimal (BCD) switch (U17).

Display Selection

The user selects the particular set of displayable signals that are desired. The observable sets of signals are the aforementioned DBUS

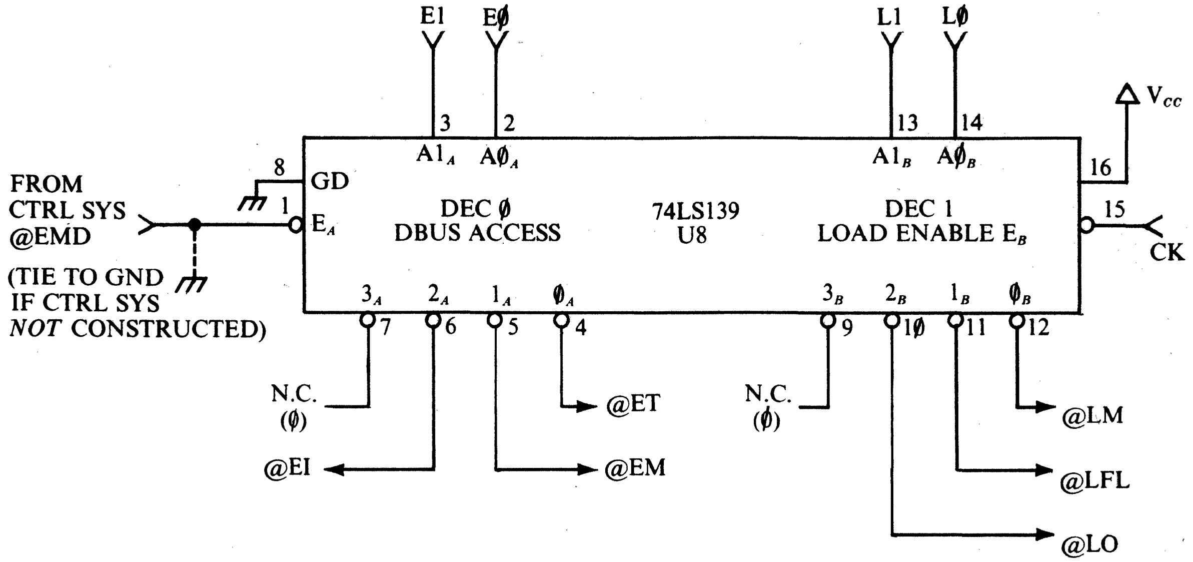


Figure 5-8
DBUS Access and Load-Enable Decoders

and OUT register, as well as the BBUS, the FBUS, the outputs of MAR, the FL register and IR register. The IR register consists of the MOD and OP code fields.

TIL 311 LED Display

The MOD field is not displayed unless the control system is built. All of these may be examined, one at a time, in the TIL 311 hex display, U18. The Texas Instruments TIL 311 is a latch, decoder, driver, and hexadecimal LED display in a single package, as shown in the data sheets of Figure 5-10. Here, its clock is made permanently active, so that whatever is on the DISBUS is passed through its transparent latch and is immediately observable on the display. U13 has a spare set of inputs and an enable line, which are used only if the control system is constructed in the downloading of demonstration programs stored in PROM.

DEC0 and DEC1: DBUS Source and Clock-enable Controls (U8)

Register Enable Controls

U8 is a dual 2-line-to-4-line decoder with separate enables for each half. It is required to decode the two DBUS-access signals (E1, E0) and the two load-enable controls (L0, L1) that strobe (clock) the memory, the flag, and the out registers. These encoded signals originate in the control system. Notice that, in DEC0, the DBUS access half, register T, the input buffer of the IN switches, and the memory IC all interface to the DBUS. The use of the U8 decoder, which has active-low outputs, is illustrated in Figure 5-8.

DBUS Source Control

Note that the DEC0 half enforces mutually exclusive access to the DBUS. This device is a part of a control system, shown here so that the CPU and its external-environment control signals can be defined and used in their final form. Two signals (E1 and E0) from the control system select which single device will be master of the DBUS. The use of this device is not mandatory, but it avoids any possibility of a bus-access conflict, due to its one-of-four active nature. Equally important, it shortens the required control-word size coming from the control system's ROM. This reduces the word size needed for the control store memory. Its use illustrates the difference between *horizontal* and *vertical* microprogramming.

In vertical microprogramming, the memory size of the control word is shortened and encoded, as here. Subsequent decoders restore the separate control signals required. These important considerations will be enlarged upon later. Table 5-2 summarizes the decoded signal names of DEC0:

Table 5-2
DBUS Source Control Decoding

E1	*E0*	*Signal and Source Names*
L	L	@ET, enable T register
L	H	@EM, enable Memory
H	L	@EI, enable Input
H	H	N.C., enable nothing

DMA Aspects of DBUS Control

The "not connected" (N.C.) condition implies that nothing is being sourced to the DBUS by the CPU. When nothing is enabled onto the DBUS by the CPU, DMA devices may be given control of the data bus by the CPU, using the N.C. line as an external DMA grant signal. This illustrates how a CPU can relinquish use of its data bus by tristate control for a DMA operation when DEC0's inputs are HH. We previously noted how MAR can be made to relinquish use of the address bus. Thus, the stuff that DMA consists of is slowly becoming apparent as we proceed.

Table 5-3 displays the decoding of the original control-system signals L0 and L1 for the vertical control of selected sinks.

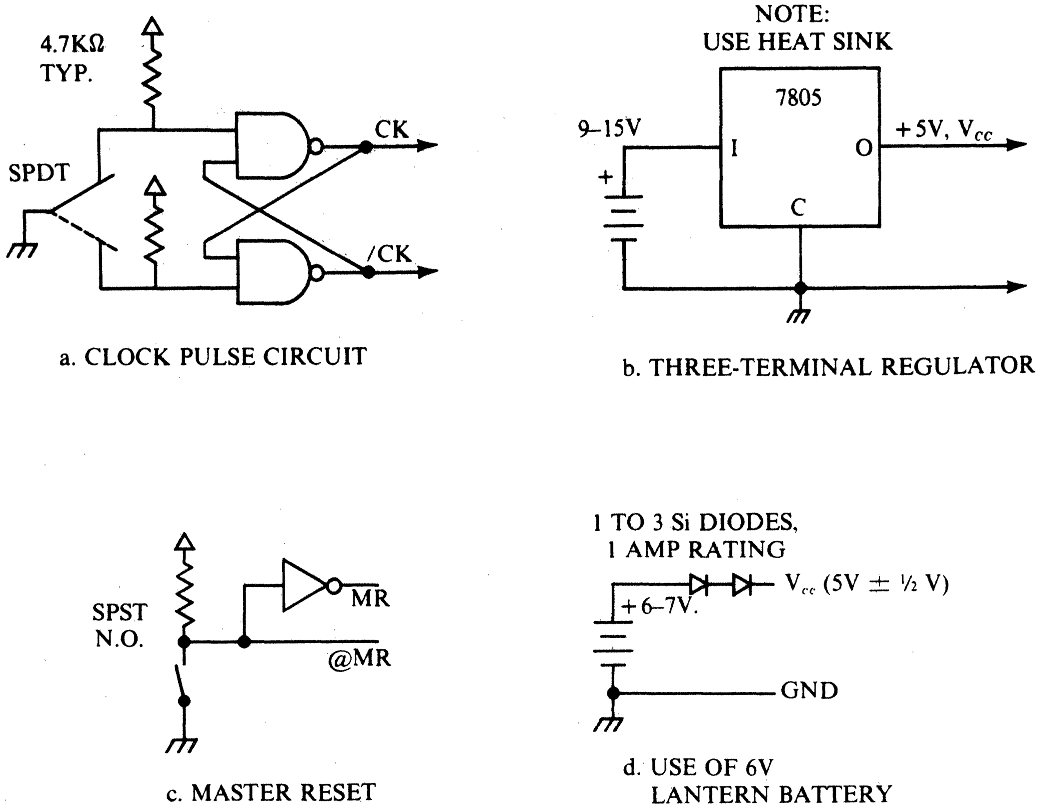
Table 5-3
Load-Enable Decoding

L1	*L0*	*Signal and Sink Names*
L	L	@LM, Load Memory
L	H	@LFL, Load Flags
H	L	@LO, Load Output
H	H	N.C., load nothing

CPU System Summary

System Schematics

We have now discussed the basic nature of the IC's and the associated control signals, to be employed in fashioning STUPIDD's CPU and external environment. The latter consists of a memory system, an input port, an output port, and a display system. A complete schematic of the entire CPU system is shown in the CPU schematic provided with this book. In this schematic, all the CPU control signals emanate from DIP switch packages containing eight switches each. These switches will be replaced by the signals from the control system itself later, when we investigate it. That is, in our first phase of controlling STUPIDD's behavior, we concern ourselves only with the control-store signals, as simulated by our own setting of the switches. We can then observe their effect on the CPU and on the external environment. The control system (shown in the schematic) is *not* complex; we shall



a. CLOCK PULSE CIRCUIT

b. THREE-TERMINAL REGULATOR

c. MASTER RESET

d. USE OF 6V
LANTERN BATTERY

Figure 5-9
Clock, Reset, and V_{CC} Details

study its construction and work with it in the next phase of our progress. In the meantime, *we are the control system,* learning how to understand its behavior.

Manual Clock Circuit

Once the control signals are established either by hand selection of switches or by control system operation, the system can be caused to respond by manually issuing a clock pulse. The circuitry for accomplishing this, shown in Figure 5-9a, is included in the design and wire lists that follow. The clock is a manually operated single-pole, double-throw (SPDT) momentary-contact push-button switch, interfaced to the basic latch formed with two NAND gates and used as a bounce eliminator. The output of this circuit, also shown in the full schematic, gives us the clock signals CK and /CK.

Power Supply

The simplest power supply for operating STUPIDD, shown in Figure 5-9d, consists of a 6V camping-lantern battery in series with about two silicon diodes of 1 amp current rating. Be sure to use the right number of diodes to produce a voltage at the applied output between 4½ and 5½ volts under load; the output voltage of these batteries varies with age; up to three diodes may be necessary. This economical arrangement does not drain the battery soon and has worked very well for home-brew experimenting. Figure 5-9b contains a suggestion for a power-supply schematic consisting of a 9-volt transistor or a car battery and a three-terminal regulator. Γhe current drain on a 9V transistor battery depletes it in short order, but it has worked for quick demonstrations of the CPU, such as at job interviews. The three-terminal regulator requires at least a 9V source—preferably 11 or 12 volts—to operate well. In any event, expensive power supplies are not required.

For those with little or no hardware experience, the greatest amount of learning is derived from the wire-wrap construction project described next. Handling each IC, consulting the data catalog, and the sense of achievement that comes from making a significant hardware system work all add up to a significant educational experience. Those with a good deal of hardware-construction background, of course, may wish to use tne printed circuit (PC) board approach.

TYPE TIL311
HEXADECIMAL DISPLAY WITH LOGIC

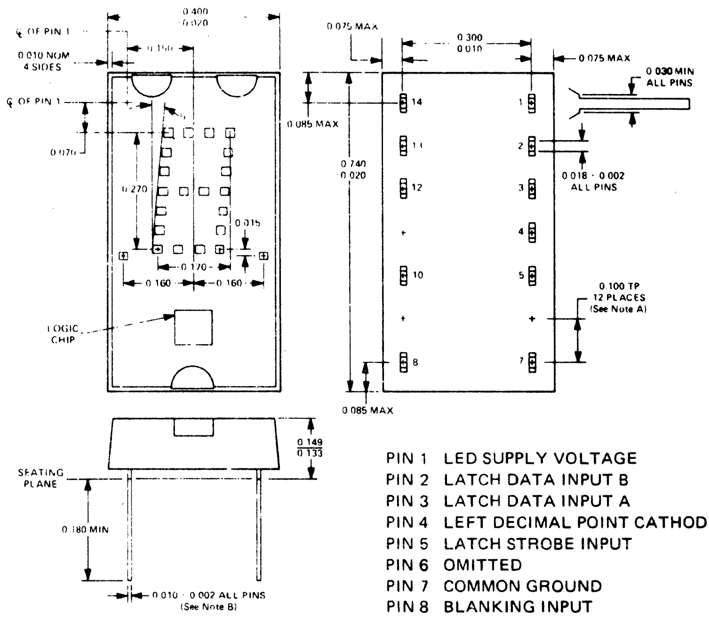
BULLETIN NO. DL S 7611653, MARCH 1972–REVISED MARCH 1976

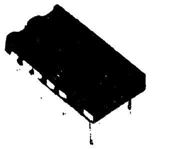
SOLID-STATE VISIBLE HEXADECIMAL DISPLAY WITH INTEGRAL TTL CIRCUIT TO ACCEPT, STORE, AND DISPLAY 4-BIT BINARY DATA

- 0.300-Inch-High Character
- High Brightness
- Left-and-Right-Hand Decimals
- Separate LED and Logic Power Supplies May Be Used
- Easy System Interface
- Wide Viewing Angle
- Internal TTL MSI Chip with Latch, Decoder, and Driver
- Operates from 5-Volt Supply
- Constant-Current Drive for Hexadecimal Characters

mechanical data

The display chips and TTL MSI chip are mounted on a header and this assembly is then cast within a red, electrically nonconductive, transparent plastic compound. Multiple displays may be mounted on 0.450-inch centers.





PIN 1 LED SUPPLY VOLTAGE
PIN 2 LATCH DATA INPUT B
PIN 3 LATCH DATA INPUT A
PIN 4 LEFT DECIMAL POINT CATHODE
PIN 5 LATCH STROBE INPUT
PIN 6 OMITTED
PIN 7 COMMON GROUND
PIN 8 BLANKING INPUT
PIN 9 OMITTED
PIN 10 RIGHT DECIMAL POINT CATHODE
PIN 11 OMITTED
PIN 12 LATCH DATA INPUT D
PIN 13 LATCH DATA INPUT C
PIN 14 LOGIC SUPPLY VOLTAGE, V_{CC}

NOTES:
A. The true-position pin spacing is 0.100 between centerlines. Each pin centerline is located within ±0.010 of its true longitudinal position relative to pins 1 and 14.
B. Lead dimensions are not controlled above the seating plane.
C. Dimensions associated with position of LED's are between centerlines and are nominal.
D. All dimensions are in inches unless otherwise specified.

TEXAS INSTRUMENTS
INCORPORATED
POST OFFICE BOX 5012 • DALLAS, TEXAS 75222

Figure 5-10
TIL311 Hexadecimal Display Data Sheet
(For educational purposes only. Data may be old and obsolete. Courtesy of Texas Instruments, Inc. © 1976, Texas Instruments, Inc.)

TYPE TIL311
HEXADECIMAL DISPLAY WITH LOGIC

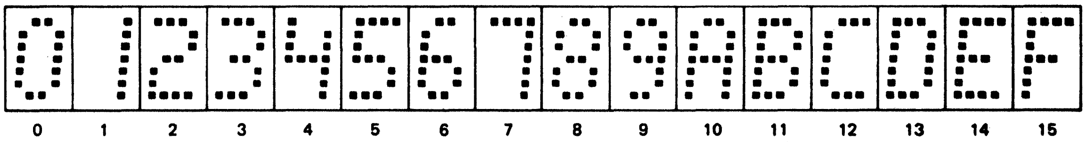
description

This hexadecimal display contains a four-bit latch, decoder, driver, and 4 X 7 light-emitting-diode (LED) character with two externally-driven decimal points in a 14-pin package. A description of the functions of the inputs of this device follows.

FUNCTION	PIN NO.	DESCRIPTION
LATCH STROBE INPUT	5	When low, the data in the latches follow the data on the latch data inputs. When high, the data in the latches will not change. If the display is blanked and then restored while the enable input is high, the previous character will again be displayed.
BLANKING INPUT	8	When high, the display is blanked regardless of the levels of the other inputs. When low, a character is displayed as determined by the data in the latches. The blanking input may be pulsed for intensity modulation.
LATCH DATA INPUTS (A, B, C, D)	3, 2, 13, 12	Data on these inputs are entered into the latches when the enable input is low. The binary weights of these inputs are A = 1, B = 2, C = 4, D = 8.
DECIMAL POINT CATHODES	4, 10	These LEDs are not connected to the logic chip. If a decimal point is used, an external resistor or other current-limiting mechanism must be connected in series with it.
LED SUPPLY	1	This connection permits the user to save on regulated V_{CC} current by using a separate LED supply, or it may be externally connected to the logic supply (V_{CC}).
LOGIC SUPPLY (V_{CC})	14	Separate V_{CC} connection for the logic chip.
COMMON GROUND	7	This is the negative terminal for all logic and LED currents except for the decimal points.

The LED driver outputs are designed to maintain a relatively constant on-level current of approximately five milliamperes through each of the LED's forming the hexadecimal character. This current is virtually independent of the LED supply voltage within the recommended operating conditions. Drive current varies slightly with changes in logic supply voltage resulting in a change in luminous intensity as shown in Figure 2. This change will not be noticeable to the eye. The decimal point anodes are connected to the LED supply; the cathodes are connected to external pins. Since there is no current limiting built into the decimal point circuits, this must be provided externally if the decimal points are used.

The resultant displays for the values of the binary data in the latches are as shown below.

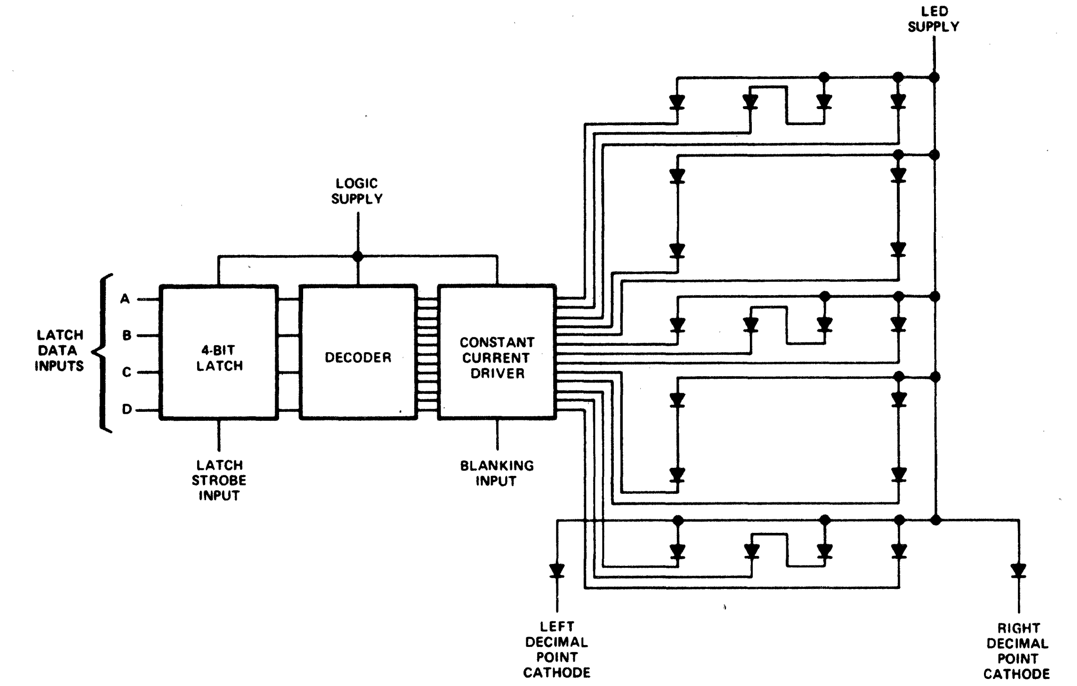


TEXAS INSTRUMENTS
INCORPORATED
POST OFFICE BOX 5012 • DALLAS, TEXAS 75222

Figure 5-10
TIL311 Hexadecimal Display Data Sheet
(For educational purposes only. Data may be old and obsolete. Courtesy of Texas Instruments, Inc. © 1976, Texas Instruments, Inc.)

TYPE TIL311
HEXADECIMAL DISPLAY WITH LOGIC

functional block diagram



absolute maximum ratings over operating case temperature range (unless otherwise noted)

Logic Supply Voltage, V_{CC} (See Note 1)	7 V
LED Supply Voltage (See Note 1)	7 V
Input Voltage (Pins 2, 3, 5, 8, 12, 13; See Note 1)	5.5 V
Decimal Point Current	20 mA
Operating Case Temperature Range (See Note 2)	0°C to 85°C
Storage Temperature Range	–25°C to 85°C

NOTES: 1. Voltage values are with respect to common ground terminal.
2. Case temperature is the surface temperature of the plastic encapsulant measured directly over the integrated circuit. Forced-air cooling may be required to maintain this temperature.

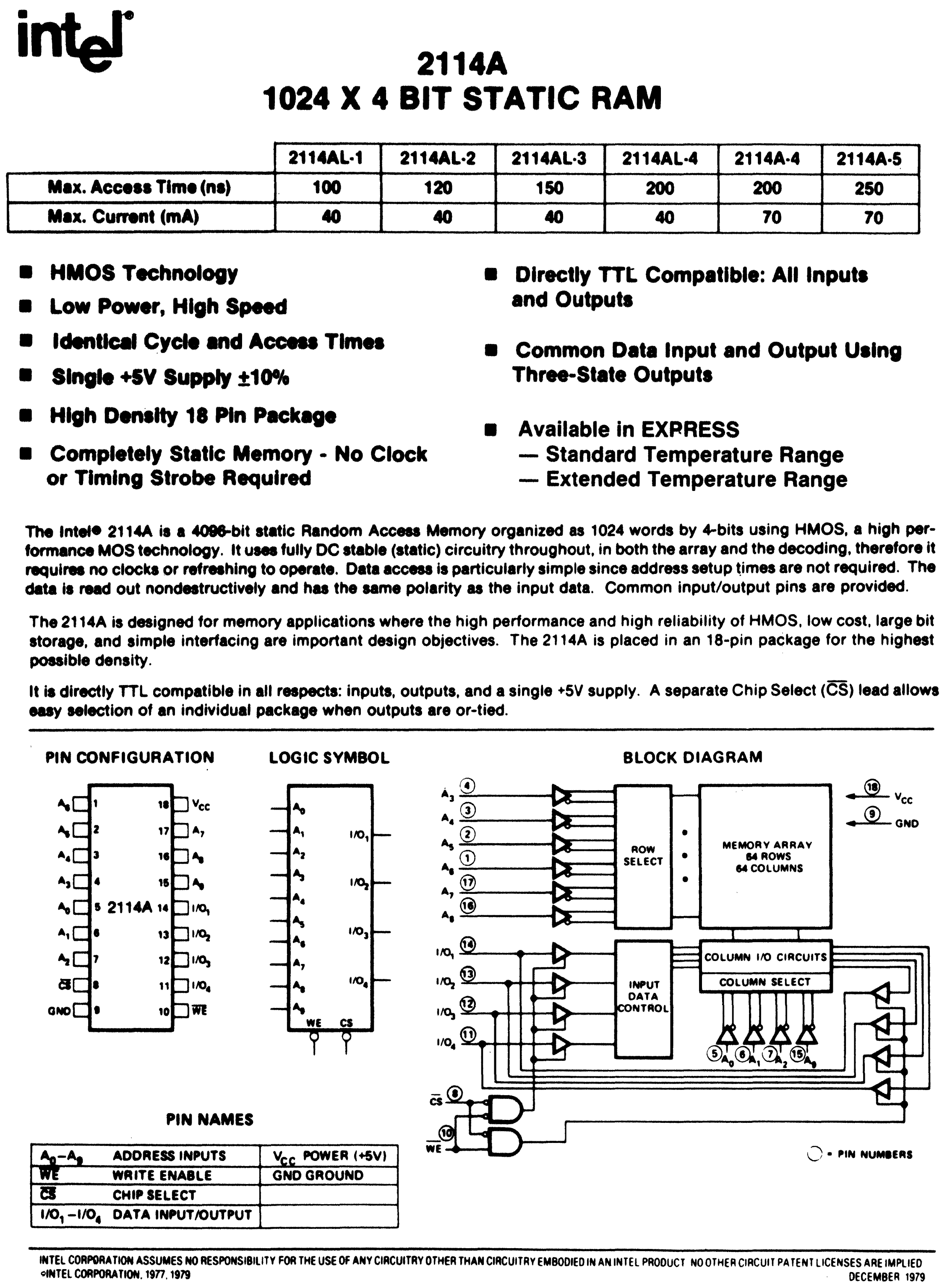
recommended operating conditions

	MIN	NOM	MAX	UNIT
Logic Supply Voltage, V_{CC}	4.5	5	5.5	V
LED Supply Voltage, V_{LED}	4	5	5.5	V
Decimal Point Current, $I_{F(DP)}$		5		mA
Latch Strobe Pulse Width, t_w	40			ns
Setup Time, t_{setup} (See Note 3)	50			ns
Hold Time, t_{hold} (See Note 4)	40			ns

NOTES: 3. Minimum setup time is the interval immediately preceding the positive-going transition of the latch strobe input during which interval the data to be displayed must be maintained at the latch data inputs to ensure its recognition.
4. Minimum hold time is the interval immediately following the positive-going transition of the latch strobe input during which interval the data to be displayed must be maintained at the latch data inputs to ensure its continued recognition.

TEXAS INSTRUMENTS
INCORPORATED
POST OFFICE BOX 5012 • DALLAS, TEXAS 75222

Figure 5-10
TIL311 Hexadecimal Display Data Sheet
(For educational purposes only. Data may be old and obsolete. Courtesy of Texas Instruments, Inc. © 1976, Texas Instruments, Inc.)



intel®

2114A
1024 X 4 BIT STATIC RAM

	2114AL-1	2114AL-2	2114AL-3	2114AL-4	2114A-4	2114A-5
Max. Access Time (ns)	100	120	150	200	200	250
Max. Current (mA)	40	40	40	40	70	70

- **HMOS Technology**
- **Low Power, High Speed**
- **Identical Cycle and Access Times**
- **Single +5V Supply ±10%**
- **High Density 18 Pin Package**
- **Completely Static Memory - No Clock or Timing Strobe Required**
- **Directly TTL Compatible: All Inputs and Outputs**
- **Common Data Input and Output Using Three-State Outputs**
- **Available in EXPRESS**
  - **— Standard Temperature Range**
  - **— Extended Temperature Range**

The Intel® 2114A is a 4096-bit static Random Access Memory organized as 1024 words by 4-bits using HMOS, a high performance MOS technology. It uses fully DC stable (static) circuitry throughout, in both the array and the decoding, therefore it requires no clocks or refreshing to operate. Data access is particularly simple since address setup times are not required. The data is read out nondestructively and has the same polarity as the input data. Common input/output pins are provided.

The 2114A is designed for memory applications where the high performance and high reliability of HMOS, low cost, large bit storage, and simple interfacing are important design objectives. The 2114A is placed in an 18-pin package for the highest possible density.

It is directly TTL compatible in all respects: inputs, outputs, and a single +5V supply. A separate Chip Select ($\overline{CS}$) lead allows easy selection of an individual package when outputs are or-tied.

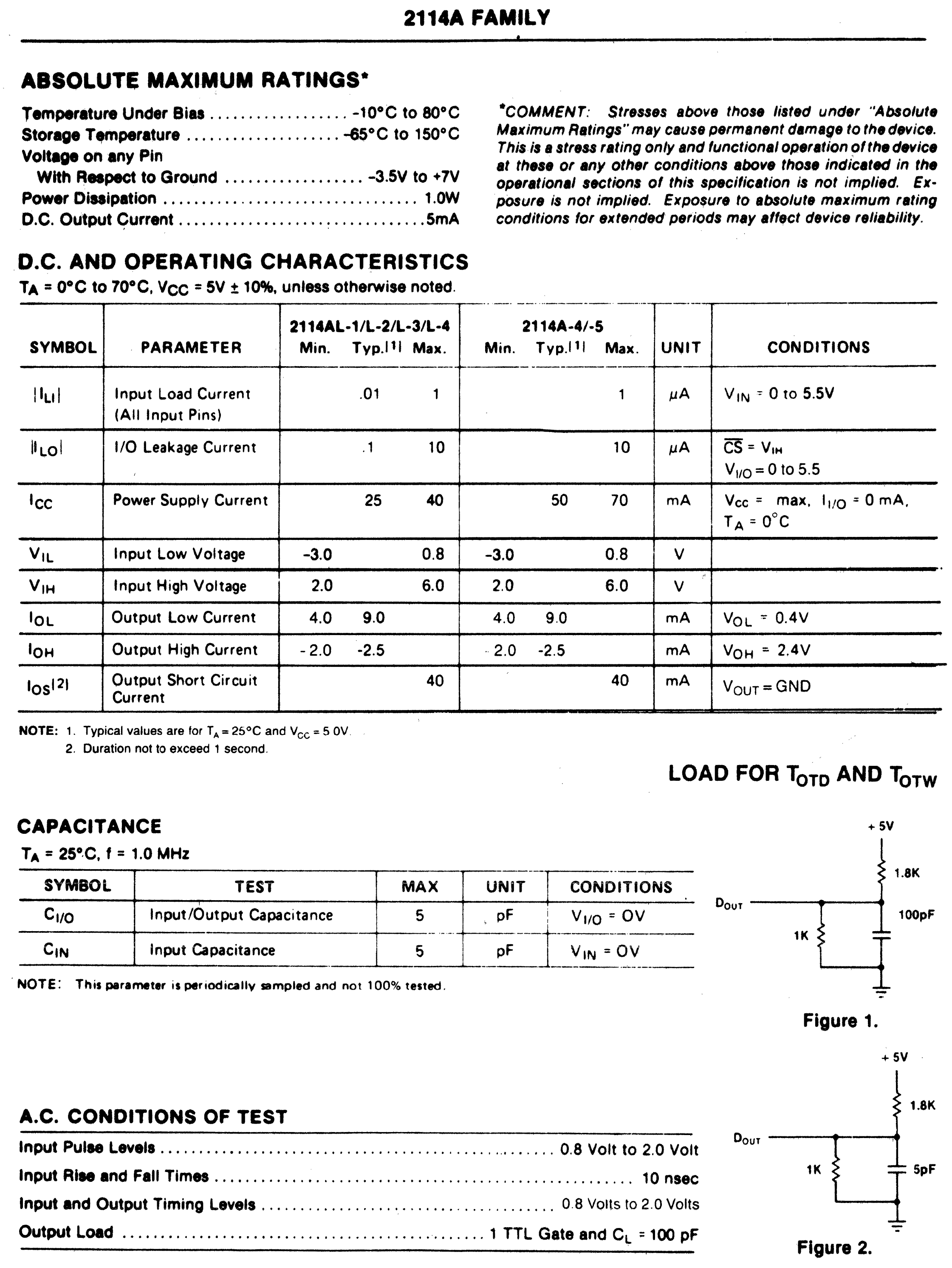
PIN NAMES

A_0–A_9	ADDRESS INPUTS	V_{CC} POWER (+5V)
$\overline{WE}$	WRITE ENABLE	GND GROUND
$\overline{CS}$	CHIP SELECT	
I/O_1–I/O_4	DATA INPUT/OUTPUT	

INTEL CORPORATION ASSUMES NO RESPONSIBILITY FOR THE USE OF ANY CIRCUITRY OTHER THAN CIRCUITRY EMBODIED IN AN INTEL PRODUCT NO OTHER CIRCUIT PATENT LICENSES ARE IMPLIED

DECEMBER 1979

Figure 5-11
2114A (1024 × 4-Bit) Static RAM Data Sheet
(Reprinted by permission of Intel Corporation. © 1983, Intel Corporation.)



2114A FAMILY

ABSOLUTE MAXIMUM RATINGS*

Temperature Under Bias	-10°C to 80°C
Storage Temperature	-65°C to 150°C
Voltage on any Pin With Respect to Ground	-3.5V to +7V
Power Dissipation	1.0W
D.C. Output Current	5mA

**COMMENT: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.*

D.C. AND OPERATING CHARACTERISTICS

T_A = 0°C to 70°C, V_{CC} = 5V ± 10%, unless otherwise noted.

SYMBOL	PARAMETER	2114AL-1/L-2/L-3/L-4 Min.	Typ.[1]	Max.	2114A-4/-5 Min.	Typ.[1]	Max.	UNIT	CONDITIONS
$\|I_{LI}\|$	Input Load Current (All Input Pins)		.01	1			1	μA	V_{IN} = 0 to 5.5V
$\|I_{LO}\|$	I/O Leakage Current		.1	10			10	μA	$\overline{CS} = V_{IH}$ $V_{I/O}$ = 0 to 5.5
I_{CC}	Power Supply Current		25	40		50	70	mA	V_{CC} = max, $I_{I/O}$ = 0 mA, T_A = 0°C
V_{IL}	Input Low Voltage	-3.0		0.8	-3.0		0.8	V	
V_{IH}	Input High Voltage	2.0		6.0	2.0		6.0	V	
I_{OL}	Output Low Current	4.0	9.0		4.0	9.0		mA	V_{OL} = 0.4V
I_{OH}	Output High Current	-2.0	-2.5		2.0	-2.5		mA	V_{OH} = 2.4V
I_{OS}[2]	Output Short Circuit Current			40			40	mA	V_{OUT} = GND

NOTE: 1. Typical values are for T_A = 25°C and V_{CC} = 5.0V.
2. Duration not to exceed 1 second.

CAPACITANCE

T_A = 25°C, f = 1.0 MHz

SYMBOL	TEST	MAX	UNIT	CONDITIONS
$C_{I/O}$	Input/Output Capacitance	5	pF	$V_{I/O}$ = 0V
C_{IN}	Input Capacitance	5	pF	V_{IN} = 0V

NOTE: This parameter is periodically sampled and not 100% tested.

LOAD FOR T_{OTD} AND T_{OTW}

Figure 1.

Figure 2.

A.C. CONDITIONS OF TEST

Input Pulse Levels	0.8 Volt to 2.0 Volt
Input Rise and Fall Times	10 nsec
Input and Output Timing Levels	0.8 Volts to 2.0 Volts
Output Load	1 TTL Gate and C_L = 100 pF

Figure 5-11
2114A (1024 × 4-Bit) Static RAM Data Sheet
(Reprinted by permission of Intel Corporation. © 1983, Intel Corporation.)

2114A FAMILY

A.C. CHARACTERISTICS T_A = 0°C to 70°C, V_{CC} = 5V ± 10%, unless otherwise noted.

READ CYCLE [1]

SYMBOL	PARAMETER	2114AL-1		2114AL-2		2114AL-3		2114A-4/L-4		2114A-5		UNIT
		Min.	Max.	Min.	Max.	Min.	Max.	Min.	Max.	Min.	Max.	
t_{RC}	Read Cycle Time	100		120		150		200		250		ns
t_A	Access Time		100		120		150		200		250	ns
t_{CO}	Chip Selection to Output Valid		70		70		70		70		85	ns
t_{CX}[3]	Chip Selection to Output Active	10		10		10		10		10		ns
t_{OTD}[3]	Output 3-state from Deselection		30		35		40		50		60	ns
t_{OHA}	Output Hold from Address Change	15		15		15		15		15		ns

WRITE CYCLE [2]

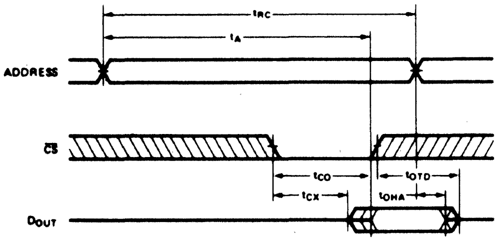
SYMBOL	PARAMETER	2114AL-1		2114AL-2		2114AL-3		2114A-4/L-4		2114A-5		UNIT
		Min.	Max.	Min.	Max.	Min.	Max.	Min.	Max.	Min.	Max.	
t_{WC}	Write Cycle Time	100		120		150		200		250		ns
t_W	Write Time	75		75		90		120		135		ns
t_{WR}	Write Release Time	0		0		0		0		0		ns
t_{OTW}[3]	Output 3-state from Write		30		35		40		50		60	ns
t_{DW}	Data to Write Time Overlap	70		70		90		120		135		ns
t_{DH}	Data Hold from Write Time	0		0		0		0		0		ns

NOTES:

1. A Read occurs during the overlap of a low $\overline{CS}$ and a high $\overline{WE}$.
2. A Write occurs during the overlap of a low $\overline{CS}$ and a low $\overline{WE}$. t_W is measured from the latter of $\overline{CS}$ or $\overline{WE}$ going low to the earlier of $\overline{CS}$ or $\overline{WE}$ going high.
3. Measured at ± 500 mV with 1 TTL Gate and C_L = 500 pF.

WAVEFORMS

READ CYCLE ③



NOTES:

3. $\overline{WE}$ is high for a Read Cycle.
4. If the $\overline{CS}$ low transition occurs simultaneously with the $\overline{WE}$ low transition, the output buffers remain in a high impedance state.
5. $\overline{WE}$ must be high during all address transitions.

WRITE CYCLE

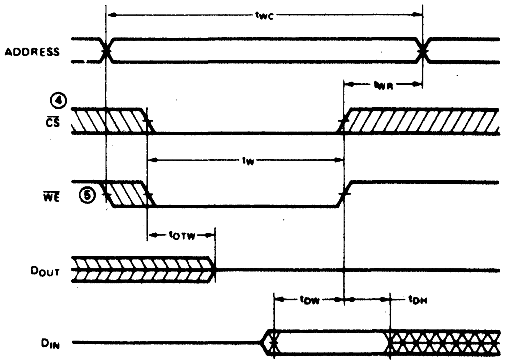


Figure 5-11
2114A (1024 × 4-Bit) Static RAM Data Sheet
(Reprinted by permission of Intel Corporation. © 1983, Intel Corporation.)

CPU CONSTRUCTION PROJECT AND CHECKOUT

RATIONALE FOR CONSTRUCTING STUPIDD

This section contains the details of the materials, wiring lists, and procedures to construct and check out the CPU portion of the Student Project In Digital Design (STUPIDD). The control-system construction details are also, for the convenience of constructing the entire project at once, presented at the end of this section. The control system will be formally studied after we have had a chance to observe the operating principles required to control the CPU's behavior. This material was first developed for electronic engineering students and was then extended so that computer science students could participate, too. Computer science students with no hardware experience have constructed the CPU at the California State Polytechnic University, Pomona, during a ten-week quarter. This was done as a home assignment, with only the check-out and demonstration of created microcode taking place in a classroom or laboratory. Although, as mentioned before, those with hardware experience may prefer the PC board approach, actual construction is pedagogically the better way for most. The main problem of those with only a software background has been "culture shock"–indeed, this is exactly what provided the motivation for establishing STUPIDD as a project. Besides the components, a hand wire-wrap tool, an inexpensive volt-ohmmeter (VOM), a 6V camping-lantern battery, and a kitchen table are all that is required.

Some students who were new to hardware became inspired, making innovative suggestions for the use and improvement of STUPIDD. They became hardware hackers (in a few cases, specialists) who had software expertise. For example, one student greatly simplified the design by constructing it on a prototype board for insertion into a slot in a personal computer. Thus, no display circuitry was required. The state of the registers was displayed on the CRT monitor by trace-like procedures after each step. Another software-oriented student independently developed the concept for downloading a program from PROM into primary memory, so that the system could be brought up and run immediately in an automatic mode. These innovators were computer science students with no hardware experience. The majority of students came away with a thorough understanding of the machine they planned to base their livelihood on. In acquiring this insight, they also deepened their understanding of the basis of *microprogramming.* This could only enhance their job skills.

Students, particularly those who feel out of their element working with hardware, should perform the actual construction of STUPIDD, not despite but because of the labor that leads to mastery of the wire-wrap technique. The world is changing rapidly, and the sweeping success of microprocessors and their programmable peripheral-support IC's in applications for the control of machines, instruments, cars, the medical field, avionics, operating-systems implementation, etc., means that job qualifications should include some appropriate mixture of *both* hardware and software knowledge from all of us. The cost of the components can be minimized by ordering from a mail-order house that advertises in computer and electronics hobbyist magazines. Is the project worth the cost? The answer

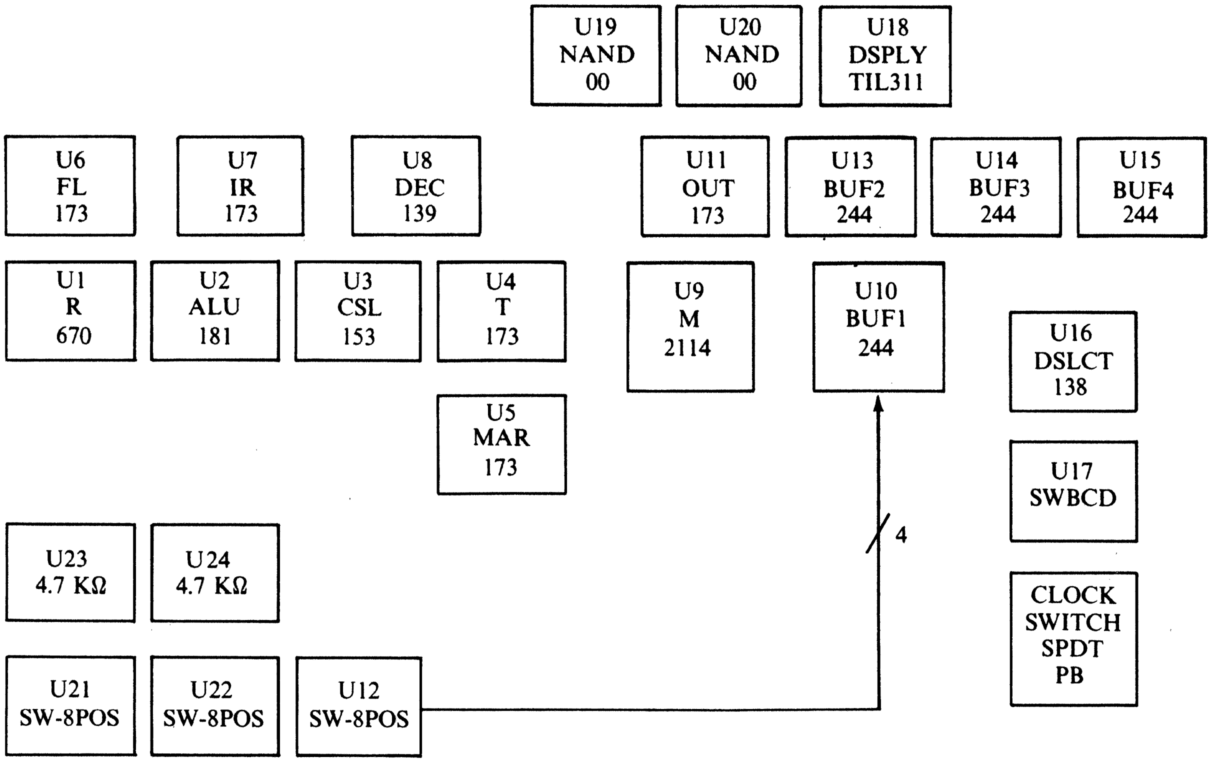


Figure 5-12
STUPIDD V CPU Wire-Wrap Layout and Module Designations

depends on what touching the essence of a computer and of practicing real microprogramming is worth to each individual. We will also find that one wire-wrap project is enough.

CPU Parts List

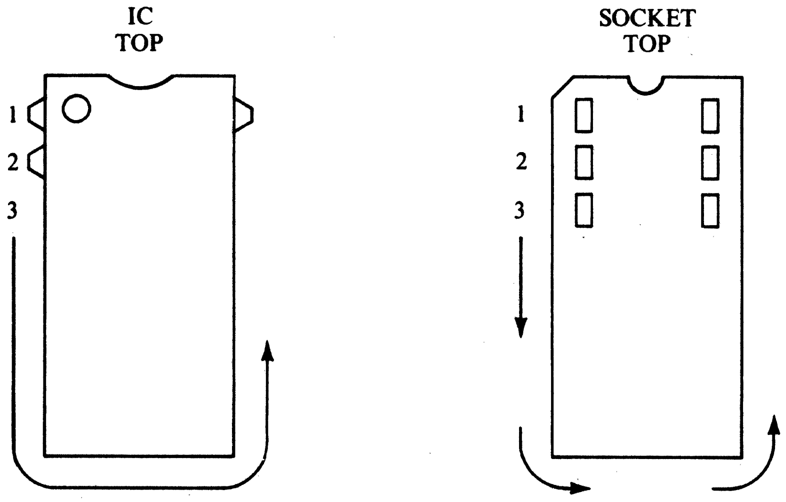
Figure 5-15 presents the parts list for the CPU portion of the STUPIDD project. All these parts are available at any electronics parts store. The first thing to acquire is the wire-wrap prototyping board specified or its equivalent. This is simply a sturdy, plain (no copper) fiberglass PC board, with a rectangular array of small holes on 0.1-inch centers. The pin spacing of IC's and sockets are also based on this 0.1-inch measurement.

Construction Procedures

The board will accept the sockets wherever we choose to place them. Next, drill and mount the six stand-off legs that will protect the wire-wrap pins of the sockets from damage as we proceed. Drill holes large enough to accept the stand-offs. Now the single-pole, double-throw (SPDT) switch (item 16) can be mounted, after the PC board is drilled out. The SPDT switch should be placed close to one of the stand-off legs, to minimize sagging whenever we press the clock button. (The photographs of Figure 5-14, showing two different construction projects, can serve as a frame of reference here.)

The wire-wrap board is now ready to receive the sockets. A suggested general layout is presented by Figure 5-12, although this may be

Figure 5-13
Pin Identification and Numbering Conventions



modified to taste. The major consideration is whether one plans to construct the control system later. If so, then acquire at this time *all* the sockets you plan to use eventually, and place them in the board now, to make sure of their relative spacing. The CPU project alone permits all CPU operations to be performed, but only manually. Completing the control system later is both informative and convenient. With the control system, STUPIDD can be operated automatically and its behavior observed on an oscilloscope. Relatively few sockets are required for the control system; their approximate layout is suggested by Figure 5-40, the control-system block diagram, and by Figure 5-14a as well. The control-system parts and wire lists appear as Figures 5-41 to 5-51 at the end of this section.

SOCKET ORIENTATION AND LAYOUT

All IC's and most sockets make it easy to identify Pin 1 for the purposes of wiring and insertion. Figure 5-13 illustrates this. Look at the IC from above, that is, with the pins down. All IC's have either a dot in the upper lefthand corner or a semicircular depression at the top center; some have both. In this position, the upper lefthand pin is Pin 1. Pin numbering proceeds counterclockwise from this pin, as illustrated in the figure. Sockets are not always so uniformly identified. Generally, there is a chamfer at the upper lefthand corner of the socket to identify the Pin 1 position. Some sockets have a semicircular notch at top center. If your sockets have none of these identifying marks, mark them with model paint or nail polish, using a toothpick so that the paint or polish is not carelessly placed in the contacts, to identify the upper lefthand corner (looking down, from the IC-insertion side). Any paint or polish in the contacts of the sockets will cause a lot of grief, so take care.

With the sockets in place, carefully turn the board upside down. Tape may be used to prevent the sockets from dropping out. Place a small piece of pressure-sensitive label material between the pins of each socket that has a U-number on the wire lists, or you may devise some other method of marking the U-numbers. This will help easily identify the sockets as the wiring proceeds. Now place a drop of nail polish on the board near the V_{CC} pin of each socket. Another color of nail polish can be used to identify all the ground pins of each socket. Again, use a toothpick rather than a brush. Remember that, since you are looking at the pins from the bottom, you are now looking at the mirror image of the pin diagram. For example, in Figure 5-13, Pin 1 is at the upper lefthand corner (seen from the top); seen from the bottom, Pin 1 is at the upper righthand corner of the socket. The numbering of the pins is now clockwise. The photographs in Figure 5-14 should also aid in visualizing the project.

Figure 5-14a is a top view of the author's completed project—both CPU and control system. Figures 5-14b and c are top and bottom views of a former computer science student's CPU-only project, the first hardware project attempted by the student. Note the neat workmanship, particularly the wire wrap. (The author's wire wrap is purposely not shown, but it works.) The student's project uses 9V transistor batteries and a 3-terminal 5V regulator. It is portable and has been to a few successful job interviews.

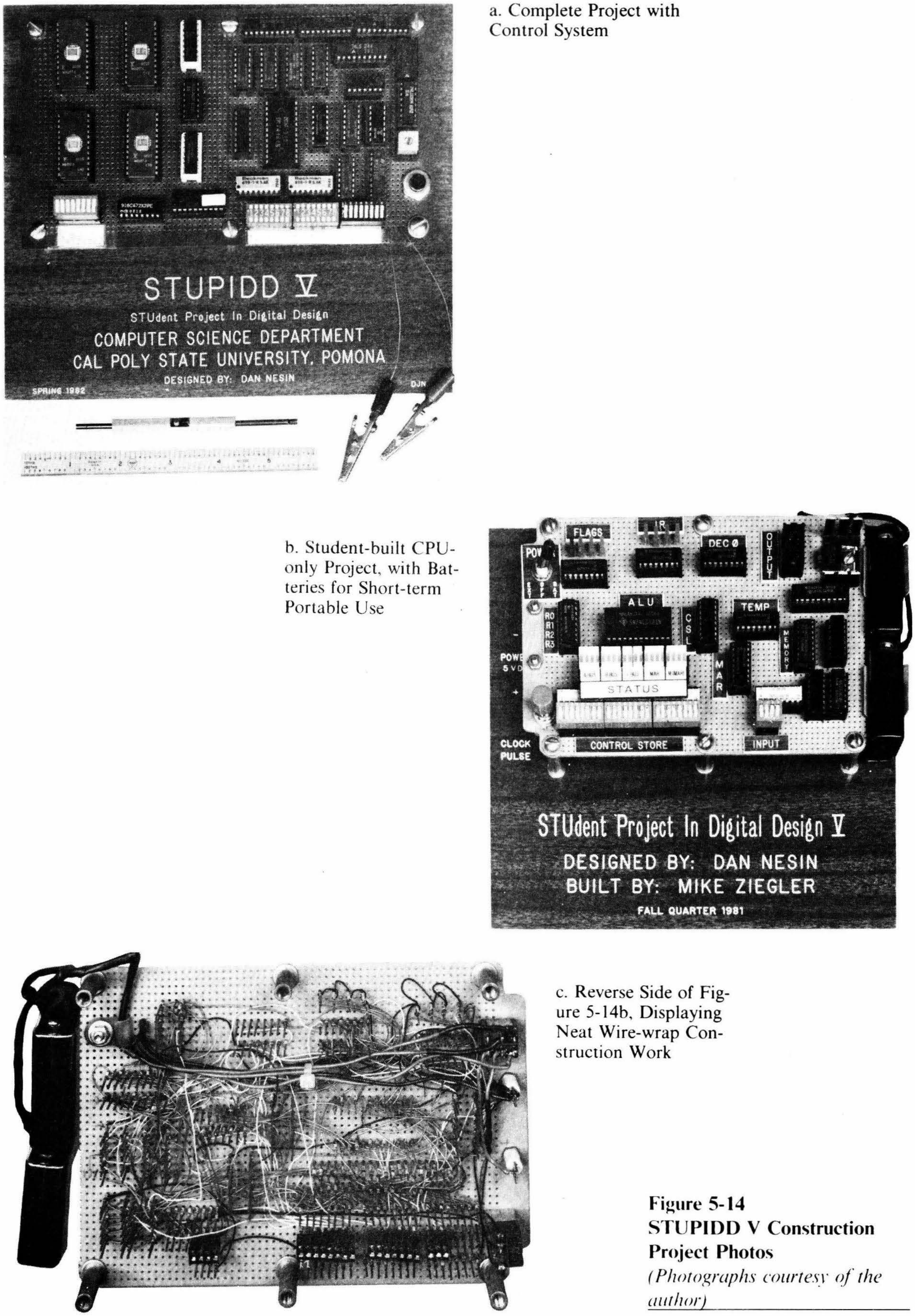
WIRE-WRAPPING PROCEDURE

Practice wire-wrapping a few pins of a socket. The wire comes in colors and in prestripped form in various lengths. The prestripped wire has approximately one inch of insulation conveniently removed for you at each end of the wire. The hand wire-wrap tool, available at your local electronics parts shop, has a bit for wrapping at one end and another bit for unwrapping at the other end. Some even come with an easy-to-use wire-stripping device built into the center. In this case, wire in spools is more economical. Ask for a demonstration. Battery-powered wire-wrap guns are also available at reasonable cost to the hobbyist, in case you have some long-range plans to build wire-wrap boards for use with your personal computer. In wrapping a pin, be sure that all the bare wire is wrapped on the pin. Loose bare wire can cause electrical shorts. A turn or so of insulation wrapped onto the pin will not hurt, so err in this direction.

Wire-Wrapping V_{CC} and Gnd

With your wire-wrap skills in place, wrap all the V_{CC} pins in one continuous chain, following the colored dots. Be sure to place the first wrap on a pin as close to the board as possible, to prevent the socket from being too loose. You may use red insulation wire to identify the V_{CC} line. When this is done, wrap on an extra piece of red wire, leaving one end loose. This should be done at the V_{CC} pin closest to where you want to connect power later. The metal stand-offs can be used for power connections, as can alligator clips, etc. Repeat the above procedure for the ground (Gnd) chain, but use green wire. If these wraps have been made reasonably close to the board, the sockets are now secure and not too loose. The more wraps, the more secure the socket becomes. At this point, however, you should be able to hold the board in any position without fear of losing a socket. Use other colors for the rest of the project. If it helps, use specific colors for identifying such entities as the DBUS.

Checking your wiring as you proceed is a good idea. (Wire-wrap lists appear as Figures 5-16 to 5-51 at the end of this section.) At this point, you can connect the V_{CC} and Gnd leads to a battery and use a low-cost voltmeter to probe these pins at the *top of the sockets,* to be



a. Complete Project with Control System

b. Student-built CPU-only Project, with Batteries for Short-term Portable Use

c. Reverse Side of Figure 5-14b, Displaying Neat Wire-wrap Construction Work

Figure 5-14
STUPIDD V Construction Project Photos
(Photographs courtesy of the author)

sure the connections carry the voltage. Defective sockets occasionally occur.

Wire-wrap the rest of the board, following the *from–to* directions of the wire lists. As you proceed, it is suggested you use a yellow marker to mark out each line of the wire list *after* it is wrapped. A *See* reference on the lists means that a connection to this point has already been specified on another sheet of the lists. *You must wire all non-*See*-referenced connections.* You should use the ohmmeter of your VOM to verify electrical continuity as you proceed with each step; do *not* wait until later. We want to avoid troubleshooting problems. The push button used for the clock requires that the wires that connect to it be soldered to the NO (normally open), NC (normally closed), and Common terminals (each) while the other ends are wire-wrapped as usual. See the U20 wire list (Figure 5-35) and the CPU schematic. The NO terminal is detected by an open circuit between it and Common—use the ohmmeter to test this.

IC Insertion and Power Application

Now we are ready to insert the IC's, being sure to follow the Pin 1 convention established above. Nothing excites an IC like placing it in the socket backwards. Confirm IC orientation before insertion. U-17, the BCD switch, has only six pins but is inserted into an eight-pin socket. Be sure that Pin 1 of the switch matches Pin 1 on the socket. Now apply power—a 6V lantern battery and several series 1-amp silicon diodes will do. The V_{CC} voltage should be between 4½ and 5½ volts under load. Touch the IC's to see if any are getting hot. Some, like the ALU, do get warm, but this is normal. With care, a project can work correctly from the very start. At this point, start to play the role of the control system by executing the microcode for the IF and other macros of the next section. For convenience, label the DIP switches with the names of the control-word signals.

Take care in the wiring, and good luck!

General Procedures for System Checkout

Troubleshooting Philosophy

Based on what we have just studied, a general philosophy of the troubleshooting of digital logic and computers may now be presented. The essence is that all IC's and all systems composed of them should be viewed as a block of logic with an interface. The block simply transforms an input excitation into an output condition, according to a set of known transformation rules. That is all there is to it. In the case of combinational logic, the output state to be checked is assumed to be an immediate consequence of the applied input. In the case of synchronous memory elements, such as registers, the output transformation can be checked only after the clock has been applied. This is the *systems* approach. It relies on our understanding of the functional performance of a block, which can consist either of an IC or of an isolatable collection of IC's, whose group functioning we understand.

With the above systems perspective in mind, let us develop an overview for bringing up a new computer. Essentially, we are on our own, and courage is required. Many computer engineers and scientists actually fear the systems their livelihoods depend upon. Understanding the system through methodical ways of probing a machine's behavior can dispel this insecurity. The following is very brief, since troubleshooting is ηot our main subject–just one we cannot avoid.

Mechanical Checks

Do not insert the IC's into their sockets until you are sure that all potentially harmful wiring errors have been fixed. IC's can take a fair amount of abuse and still work. The worst thing you can do is reverse V_{CC} and Gnd, which can heat things up in a hurry. The next to worst thing is to connect the output of an IC's transistor *directly* to V_{CC} without an intermediate pull-up resistor. This will definitely burn out the transistor. Defective IC's or sockets are occasionally found. Therefore, *all* voltage and resistance checks should be probed at the IC-insertion side of the socket, not just the easier to reach pins. Follow these steps to check the mechanical integrity of the wiring:

1. After wiring V_{CC} and Gnd, connect any battery across the power leads. With a voltmeter, check for the correct voltage between V_{CC} and Gnd at the outputs of *each* socket. This confirms both continuity and the absence of shorts or opens for this circuit.
2. Remove the battery, and switch your meter to the *ohms* position. If the battery were left in, your ohmmeter circuit might be damaged. The meter has a built-in battery for the ohms measurement section. Where the wire lists indicate that there is a string of connections, such as with the data bus, we can check for continuity. Continuity means zero ohms. For example, there should exist essentially zero ohms between the A0 pin of the ALU socket and the Q0 pin of the T register socket. Leaving one lead of your ohmmeter on the A0 pin of the ALU, we can now run this check for the rest of the string.
3. Keep in mind that a missing string connection can show up as a high or infinite impedance. In this manner, check your wired strings and pull-up resistor connections with the ohmmeter for continuity, shorts, and opens (missing wires).

The following collection of tables, which appear at various points in the text, will be useful for the electrical check procedures that follow. These tables contain the signal names and switch settings for controlling and investigating much of the system. They serve as handy references for operating, programming, and debugging the processor.

Electrical Checks of IC Performance

Now insert the IC's into their sockets. Take care with their orientation, so that V_{CC} and Gnd are not reversed. Apply power between V_{CC} and Gnd. Look for "smoke"–shut it down if something gets very hot or if the LED display does not light up. Whether there is a problem or not, this is the time for a little reflection. Perform the following checks:

1. Does the LED display work? Rotate the display bus-selector switch, U17, and observe whether the display's response appears normal. Test the voltage levels at the four IN and other switches to see if they work and to make sure which position producés an H or L. Orient the switches in their sockets so that the position nearest the operator produces, say, a high. If this is not the case, turn the switch around. (For simplicity, all switches should have the same operational behavior and orientation.)

Table 5-4
Summary of Useful Tables

Carry In (CI) Selection Control Signals

CS1	*CS0*	*Source of CI*
L	L	Low logic level
L	H	CO of the ALU
H	L	CF bit of FL register
H	H	High logic level

DBUS Source Control Decoding

E1	*E0*	*Signal and Source Names*
L	L	@ET, enable T register
L	H	@EM, enable memory
H	L	@EI, enable IN
H	H	N.C., enable nothing

Load Enable Decoding

L1	*L0*	*Signal and Sink Names*
L	L	@LM, Load Memory
L	H	@LFL, Load FLags
H	L	@LO, Load Output
H	H	N.C., load nothing

2. Set U17 to display the DBUS. Table 5-4 summarizes signal levels and switch settings. Set control-system simulation switches E1 and E0 to enable the IN switches onto the DBUS. Does the display follow your switch settings as you change them? If not, then it is time for some more reflection. Check the high-low operation of the switch again. Is the IN buffer half of U10 enabled? (Pin U10-1—it should now be low.) Does its output follow input changes? Is the output display of the DBUS enabled? (Pin U10–19, DISBUS, should now be low.) Follow this trail right up to the LED display. Simply ask yourself what each device is supposed to do, then test its performance. If we understand the functional behavior of each device, then we can test it. If no problems appear, carry on.

3. Press the clock push button. With a voltmeter, check to see if the clock signal CK goes low and /CK goes high while the clock is pressed. If not, the push-button leads may be reversed.

4. Set the control-system simulation switches to load MAR and any register in R with some selected input value from IN. You are now microprogramming in a very real sense. You may have to read ahead in this text to understand fully how to set these switches. Memory is addressed by MAR, and both it and a register are about to be loaded with input data.

5. Apply the clock pulse by pressing its push button. Now read and display MAR in the LED hex display. Does the memory address just written correspond to the IN switch settings? Display R, and check that it too was properly written into. If problems occur, is

Branch Control (BC) Flag Selection Codes

BC1	*BC0*	*Flag Selected by MPX*
L	L	Low level
L	H	CF (Carry Flag)
H	L	SF (Sign Flag)
H	H	EF (Equal Flag

LED Display Selection

Switch U17	*Displayed Item*
0	Output Register
1	N.C.
2	Instruction Register (IR)
3	Flag Register (FL)
4	Address Bus
5	F Bus
6	B Bus
7	D Bus

TABLE 1

SELECTION				ACTIVE-LOW DATA		
				M = H	M = L; ARITHMETIC OPERATIONS	
S3	S2	S1	S0	LOGIC FUNCTIONS	C_n = L (no carry)	C_n = H (with carry)
L	L	L	L	$F = \overline{A}$	F = A MINUS 1	F = A
L	L	L	H	$F = \overline{AB}$	F = AB MINUS 1	F = AB
L	L	H	L	$F = \overline{A} + B$	$F = A\overline{B}$ MINUS 1	$F = A\overline{B}$
L	L	H	H	F = 1	F = MINUS 1 (2's COMP)	F = ZERO
L	H	L	L	$F = \overline{A + B}$	$F = A$ PLUS $(A + \overline{B})$	$F = A$ PLUS $(A + \overline{B})$ PLUS 1
L	H	L	H	$F = \overline{B}$	$F = AB$ PLUS $(A + \overline{B})$	$F = AB$ PLUS $(A + \overline{B})$ PLUS 1
L	H	H	L	$F = \overline{A \oplus B}$	F = A MINUS B MINUS 1	F = A MINUS B
L	H	H	H	$F = A + \overline{B}$	$F = A + \overline{B}$	$F = (A + \overline{B})$ PLUS 1
H	L	L	L	$F = \overline{A}B$	F = A PLUS (A + B)	F = A PLUS (A + B) PLUS 1
H	L	L	H	$F = A \oplus B$	F = A PLUS B	F = A PLUS B PLUS 1
H	L	H	L	F = B	$F = A\overline{B}$ PLUS (A + B)	$F = A\overline{B}$ PLUS (A + B) PLUS 1
H	L	H	H	F = A + B	F = (A + B)	F = (A + B) PLUS 1
H	H	L	L	F = 0	F = A PLUS A*	F = A PLUS A PLUS 1
H	H	L	H	$F = A\overline{B}$	F = AB PLUS A	F = AB PLUS A PLUS 1
H	H	H	L	F = AB	$F = A\overline{B}$ PLUS A	$F = A\overline{B}$ PLUS A PLUS 1
H	H	H	H	F = A	F = A	F = A PLUS 1

*Each bit is shifted to the next more significant position.

TABLE 2

SELECTION				ACTIVE-HIGH DATA		
				M = H	M = L; ARITHMETIC OPERATIONS	
S3	S2	S1	S0	LOGIC FUNCTIONS	$\overline{C}_n$ = H (no carry)	$\overline{C}_n$ = L (with carry)
L	L	L	L	$F = \overline{A}$	F = A	F = A PLUS 1
L	L	L	H	$F = \overline{A + B}$	F = A + B	F = (A + B) PLUS 1
L	L	H	L	$F = \overline{A}B$	$F = A + \overline{B}$	$F = (A + \overline{B})$ PLUS 1
L	L	H	H	F = 0	F = MINUS 1 (2's COMPL)	F = ZERO
L	H	L	L	$F = \overline{AB}$	$F = A$ PLUS $A\overline{B}$	$F = A$ PLUS $A\overline{B}$ PLUS 1
L	H	L	H	$F = \overline{B}$	$F = (A + B)$ PLUS $A\overline{B}$	$F = (A + B)$ PLUS $A\overline{B}$ PLUS 1
L	H	H	L	$F = A \oplus B$	F = A MINUS B MINUS 1	F = A MINUS B
L	H	H	H	$F = A\overline{B}$	$F = A\overline{B}$ MINUS 1	$F = A\overline{B}$
H	L	L	L	$F = \overline{A} + B$	F = A PLUS AB	F = A PLUS AB PLUS 1
H	L	L	H	$F = \overline{A \oplus B}$	F = A PLUS B	F = A PLUS B PLUS 1
H	L	H	L	F = B	$F = (A + \overline{B})$ PLUS AB	$F = (A + \overline{B})$ PLUS AB PLUS 1
H	L	H	H	F = AB	F = AB MINUS 1	F = AB
H	H	L	L	F = 1	F = A PLUS A*	F = A PLUS A PLUS 1
H	H	L	H	$F = A + \overline{B}$	F = (A + B) PLUS A	F = (A + B) PLUS A PLUS 1
H	H	H	L	F = A + B	$F = (A + \overline{B})$ PLUS A	$F = (A + \overline{B})$ PLUS A PLUS 1
H	H	H	H	F = A	F = A MINUS 1	F = A

Table 5-4

(For educational purposes only. Data may be old and obsolete. Courtesy of Texas Instruments, Inc. © 1984, Texas Instruments, Inc.)

the clock to the register changing state properly when pressed? Is the correct data present at the inputs? Are outputs enabled? Etc.

6. Display the memory and then some register in R by enabling them onto the DBUS in turn. Make sure that they are in the *read* mode. Observe their contents in the LED display. Now set IN to some other value than those just observed. Enable the writing of both M and R. This confirms the integrity of the two data paths to M and R from the input switches. Check the settings once more, and then push the clock button. The moment of truth is upon us.

7. Display the memory by enabling it onto the DBUS, making sure that it is in the *read* mode. Observe its contents in the LED display. These just-written contents should agree with the IN setting before the clock was applied and should be different from the original contents of this location. Also confirm that R is following orders properly, too. We now know that we can read and write memory and registers.

8. Continue this type of read-write check for all of the other displayable registers: T, FL, MAR, and IR.

In Case of Trouble

1. We are on our own.
2. We have studied IC interface descriptions and internal functioning. Now, we must apply this insight.
3. The general rule:

   a. Stimulate (or just note) the logic levels of the input pins of the system.

   b. Is the output what is expected from observation of the system's functioning?

   c. If not:

      Wiring error?
      Bad IC? This happens on very rare occasions.
      Time for reflection? STUPIDD does work.

4. The most important rule of all: You can win. You are smarter than the machine.

This has been our introduction to applied digital systems philosophy and microprogramming. Success *is* sweet if one has the will to win. Do not hesitate to practice wire-wrapping before you start—or hesitate to tear it all out and start again if it is not a neat job. As noted, students have used some of the neater projects to good advantage in job interviews, as a vehicle for demonstrating their knowledge of computers and microprogramming. For the beginner, though, one good wire-wrap project may last a lifetime.

Item No.	Part No.	Description and Source	Quant.
1	74LSØØ	Quad NAND DIP	2
2	74LS138	Decoder, 1 of 8	1
3	74LS139	Decoder, 1 of 4	1
4	74LS153	Multiplexor, 4-to-1	1
5	74LS173	Register, 4-bit, D type	5
6	74LS181	ALU	1
7	74LS244	Buffer, tri-state	4
8		Volt-ohm meter (low-cost)	1
9	74LS67Ø	Register array, 4 x 4	1
1Ø	2114	Memory, 1K x 4 bit	1
11	TIL311	Hex display, DEC-drive; Texas Instruments, Inc.	1
12		15-resistor array, 4.7 KΩ , DIP	2
13	3Ø gauge	Prestripped wire, assorted colors	---
14		DIP SW, SPST, 8	3
15	23ØØØ2G	DIP SW, DEC, BCD; EECO	1
16		SW, SPDT, push-button	1
17		Socket, wire wrap, 3 high, 8 pin	1
18		Socket, wire wrap, 3 high, 14 pin	3
19		Socket, wire wrap, 3 high, 16 pin	14
2Ø		Socket, wire wrap, 3 high, 18 pin	1
21		Socket, wire wrap, 3 high, 2Ø pin	4
22		Socket, wire wrap, 3 high, 24 pin	1
23	145ØD	Stand-offs, 3/4" x 4-4Ø	6
24		Wire wrap tool, hand	1
25	79P44ELBDP	PC BD., Ø.1 CTRS, 4½ x 6; Vector	1

Figure 5-15 STUPIDD V CPU Parts List

Part No.
and name 74LS67Ø Register array **U1**

FROM pin no.		TO U no.	pin no.	SIGNAL NAME	REMARKS
1		U2	1Ø	F1	
2		U2	11	F2	
3		U2	13	F3	
4		U1	13	RS1	Register select 1
5		U1	14	RSØ	Register select Ø
6		U2	18	B3	
7		U2	2Ø	B2	
8				GND	
9		U2	22	B1	
1Ø		U2	1	BØ	
11		U19	1	W@R	Write/read enable
12		U19	3	@WE	
13	See	U1	4	RS1	
14	See	U1	5	RSØ	
15		U2	9	FØ	
16				Vcc	

Figure 5-16 Register-Array Wire-Wrap List (CPU: U1)

Part No. and name	74LS181	ALU	**U2**

FROM		TO			
pin no.		U no.	pin no.	SIGNAL NAME	REMARKS
1	See	U1	1Ø	BØ	
2		U7	14	AØ	Same as DØ of DBUS
3		U21	14	S3	ALU select line
4		U21	13	S2	
5		U21	12	S1	
6		U21	11	SØ	
7		U3	7	CI	Carry in, ALU
8		U21	15	MD	ALU mode line
9	See	U1	15	FØ	ALU function output
1Ø	See	U1	1	F1	
11	See	U1	2	F2	
12				GND	
13	See	U1	3	F3	
14		U6	14	EQ	A=B, pull-up
15		NC		P	Not used
16		U6	13	CO	Carry out, ALU
17		NC		G	Not used
18	See	U1	6	B3	
19		U7	11	A3	Same as D3 of DBUS
2Ø	See	U1	7	B2	
21		U7	12	A2	Same as D2 of DBUS
22	See	U1	9	B1	
23		U7	13	A1	Same as D1 of DBUS
24				Vcc	

Figure 5-17 ALU Wire-Wrap List List (CPU: U2)

Part No. and name	74LS153	CSL	**U3**

FROM		TO			
pin no.		U no.	pin no.	SIGNAL NAME	REMARKS
1		U3	6	L	to GND
2		U21	1Ø	CS1	
3		U23	12	H	Pull-up for CI
4		U6	4	CF	Carry flag
5		U2	16	CO	ALU carry-out
6		U3	8	L	to GND
7	See	U2	7	CI	ALU carry-in
8				GND	
9		NC			Not used
1Ø		NC			Not used
11		NC			Not used
12		NC			Not used
13		NC			Not used
14		U21	9	CSØ	
15		NC			Not used
16				Vcc	

Figure 5-18 CSL Wire-Wrap List (CPU: U3)

Part No. and name	74LS173	T Register	U4

FROM	TO			
pin no.	U no.	pin no.	SIGNAL NAME	REMARKS
1	U4	2	@ET	Enable temp
2	U8	4	@ET	
3	U7	14	DØ	
4	U7	13	D1	
5	U7	12	D2	
6	U7	11	D3	
7	U5	7	CK	Clock
8			GND	
9	U4	1Ø	@LT	
1Ø	U22	12	@LT	
11	U2	13	F3	
12	U2	11	F2	
13	U2	1Ø	F1	
14	U2	9	FØ	
15	U5	15	MR	Master reset
16			Vcc	

Figure 5-19 T-Register Wire-Wrap List (CPU: U4)

Part No. and name	74LS173	MAR	U5

FROM	TO			
pin no.	U no.	pin no.	SIGNAL NAME	REMARKS
1	U5	2	@EMD	Enable MAR, DECØ
2	U8	1	@EMD	
3	U9	5	ADRØ	
4	U9	6	ADR1	
5	U9	7	ADR2	
6	U9	4	ADR3	
7	U6	7	CK	
8			GND	
9	U5	1Ø	@LMAR	Load memory ADR REG
1Ø	U22	13	@LMAR	
11	U4	11	F3	
12	U4	12	F2	
13	U4	13	F1	
14	U4	14	FØ	
15	U2Ø	8	MR	
16			Vcc	

Figure 5-20 MAR-Register Wire-Wrap List (CPU: U5)

U6

Part No. and name 74LS173 FL (Flags)

FROM		TO			
pin no.		U no.	pin no.	SIGNAL NAME	REMARKS
1		U6	2	L	to GND
2		U6	8	L	to GND
3		U15	2	EF	Equal flag
4		U15	4	CF	Carry flag
5		U15	6	SF	Sign flag
6		NC			Not used
7		U7	7	CK	
8				GND	
9		U6	1Ø	@LFL	Load flag REG
1Ø		U8	11	@LFL	
11		U6	8	L	To GND
12		U1	3	S	F3, sign bit
13	See	U2	16	CO	
14	See	U2	14	EQ	
15		U4	15	MR	
16				Vcc	

Figure 5-21 FL-Register Wire-Wrap List (CPU: U6)

U7

Part No. and name 74LS173 IR

FROM		TO			
pin no.		U no.	pin no.	SIGNAL NAME	REMARKS
1		U7	2	L	OE (to GND)
2		U7	8	L	OE (to GND)
3		U15	17	IRØ	
4		U15	15	IR1	
5		U15	13	IR2	
6		NC			Not used
7		U8	15	CK	
8				GND	
9		U7	1Ø	@LIR	
1Ø		U22	11	@LIR	
11	See	U2	19	D3	
12	See	U2	21	D2	
13	See	U2	23	D1	
14	See	U2	2	DØ	
15		U6	15	MR	
16				Vcc	

Figure 5-22 IR-Register Wire-Wrap List (CPU: U7)

Part No. 74LS139 DECØ & DEC1 **U8**
and name

FROM		TO			
pin no.		U	pin	SIGNAL NAME	REMARKS
		no.	no.		
1	See	U5	2	@EMD	*See Note
2		U12	15	EØ	
3		U12	16	E1	
4	See	U4	2	@ET	
5		U19	4	@EM	
6		U1Ø	1	EI	
7		NC			Not used
8				GND	
9		NC			Not used
1Ø		U11	9	@LO	Load output REG
11	See	U6	1Ø	@LFL	
12		U9	1Ø	LM	
13		U22	1Ø	L1	DEC1 Load Select 1
14		U22	9	LØ	DEC1 Load Select Ø
15		U11	7	CK	
16				Vcc	

*Note: This pin (U8--1) ties to GND if the control
system is not constructed.

Figure 5-23 DEC0 and DEC1 Wire-Wrap List (CPU: U8)

Part No. 2114 M (Memory) **U9**
and name

FROM		TO			
pin no.		U	pin	SIGNAL NAME	REMARKS
		no.	no.		
1		U9	2	L	to GND
2		U9	3	L	to GND
3		U9	17	L	to GND
4	See	U5	6	ADR3	
5	See	U5	3	ADRØ	
6	See	U5	4	ADR1	
7	See	U5	5	ADR2	
8		U19	8	CSM	Chip select memory
9				GND	
1Ø		U19	5	@LM	(@WEM)
11		U4	6	D3	
12		U4	5	D2	
13		U4	4	D1	
14		U4	3	DØ	
15		U9	9	L	to GND
16		U9	15	L	to GND
17		U9	16	L	to GND
18				Vcc	

Figure 5-24 Memory Wire-Wrap List (CPU: U9)

Part No. 74LS244 INBUF & DBUF **U10**
and name

FROM		TO			
pin no.		U no.	pin no.	SIGNAL NAME	REMARKS
1	See	U8	6	@EI	
2		U12	9	INØ	
3		U11	3	DISØ	
4		U12	1Ø	IN1	
5		U11	4	DIS1	
6		U12	11	IN2	
7		U11	5	DIS2	
8		U12	12	IN3	
9		U11	6	DIS3	
1Ø				GND	
11		U1Ø	12	D3	
12		U9	11	D3	
13		U1Ø	14	D2	
14		U9	12	D2	
15		U1Ø	16	D1	
16		U9	13	D1	
17		U1Ø	18	DØ	
18		U9	14	DØ	
19		U16	15	DSDBUS	OE, display DBUS
2Ø				Vcc	

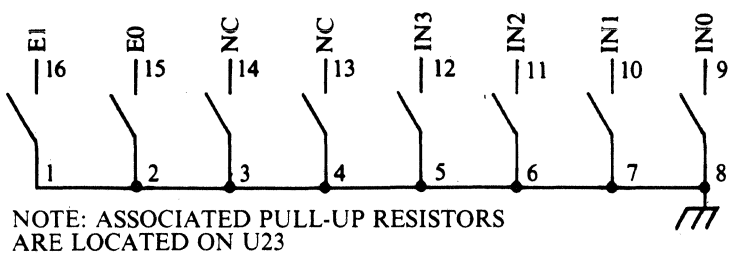
Figure 5-25 INBUF and DBUF Wire-Wrap List (CPU: U10)

Part No. 74LS173 OUT REG. **U11**
and name

FROM		TO			
pin no.		U no.	pin no.	SIGNAL NAME	REMARKS
1		U11	2	@DSOUT	Display out REG
2		U16	7	@DSOUT	
3	See	U1Ø	3	DISØ	
4	See	U1Ø	5	DIS1	
5	See	U1Ø	7	DIS2	
6	See	U1Ø	9	DIS3	
7		U2Ø	3	CK	
8				GND	
9	See	U8	1Ø	@LO	
10		U11	9	@LO	
11		U1Ø	11	D3	
12		U1Ø	13	D2	
13		U1Ø	15	D1	
14		U1Ø	17	DØ	
15		U11	8	L	MR not used here
16				Vcc	

Figure 5-26 Output-Register Wire-Wrap List (CPU: U11)

U12



NOTE: ASSOCIATED PULL-UP RESISTORS
ARE LOCATED ON U23

Part No. and name ASRC & IN DIPSW

FROM		TO			
pin no.		U no.	pin no.	SIGNAL NAME	REMARKS
1		U12	2	L	to GND
2		U12	3	L	to GND
3		U12	4	L	to GND
4		U12	5	L	to GND
5		U12	6	L	to GND
6		U12	7	L	to GND
7		U12	8	L	to GND
8				GND	
9	See	U1Ø	2	INØ	
1Ø	See	U1Ø	4	IN1	
11	See	U1Ø	6	IN2	
12	See	U1Ø	8	IN3	
13		NC			Not used
14		NC			Not used
15	See	U8	2	EØ	
16	See	U8	3	E1	

Figure 5-27 ASRC and IN DIP SW Wire-Wrap List (CPU: U12)

U13

Part No. and name 74LS244 SP & B DISBUFS

FROM		TO			
pin no.		U no.	pin no.	SIGNAL NAME	REMARKS
1	See	U32	19	@DLD	*See note.
2		U21	11	SØ	
3		U11	3	DISØ	
4		U21	12	S1	
5		U11	4	DIS1	
6		U21	13	S2	
7		U11	5	DIS2	
8		U21	14	S3	
9		U11	6	DIS3	
1Ø				GND	
11		U2	18	B3	
12		U11	11	D3	
13		U2	2Ø	B2	
14		U11	12	D2	
15		U2	22	B1	
16		U11	13	D1	
17		U2	1	BØ	
18		U11	14	DØ	
19		U16	14	@DSBBUS	Display BBUS
2Ø				Vcc	

*Note: Tie to U16--9 if control system is not constructed.

Figure 5-28 SP and B DISBUFS Wire-Wrap List (CPU: U13)

Part No. and name	74LS244			F & ADR DISBUFS	U14
FROM	TO				
pin no.	U no.	pin no.	SIGNAL NAME	REMARKS	
1	U16	13	@DSFBUS	Display FBUS	
2	U5	14	FØ		
3	U14	18	DISØ		
4	U5	13	F1		
5	U14	16	DIS1		
6	U5	12	F2		
7	U14	14	DIS2		
8	U5	11	F3		
9	U14	12	DIS3		
1Ø			GND		
11	U5	6	ADR3		
12	U13	9	DIS3		
13	U5	5	ADR2		
14	U13	7	DIS2		
15	U5	4	ADR1		
16	U13	5	DIS1		
17	U5	3	ADRØ		
18	U13	3	DISØ		
19	U16	12	@DSADR	Display ADR BUS	
2Ø			Vcc		

Figure 5-29 F and ADR DISBUFS Wire-Wrap List (CPU: U14)

Part No. and name		74LS244			FL & IR DISBUFS	U15
FROM		TO				
pin no.		U no.	pin no.	SIGNAL NAME	REMARKS	
1		U16	11	@DSFL	Display flag REG	
2	See	U6	3	EF		
3		U15	18	DISØ		
4	See	U6	4	CF		
5		U15	16	DIS1		
6	See	U6	5	SF		
7		U15	14	DIS2		
8	See	U35	5	MPX	CTRL system signal*	
9		U15	12	DIS3		
1Ø				GND		
11	See	U32	5	MOD	CTRL system signal**	
12		U14	9	DIS3		
13	See	U7	5	IR2		
14		U14	7	DIS2		
15	See	U7	4	IR1		
16		U14	5	DIS1		
17	See	U7	3	IRØ		
18		U14	3	DISØ		
19		U16	1Ø	@DSIR		
2Ø				Vcc		

*Ties to GND only if the control system is not constructed, or bit 3 of displayed flags will always be high. Otherwise, if connected to the conditional address bit (MPX in control system) the MPX bit will be displayed together with the flags that determine its value. This helps in observing the control of conditional branches.

**Ties to GND only if the control system is not constructed, or bit IR3 will always be high. Avoids confusion.

Figure 5-30 FL and IR DISBUFS Wire-Wrap List (CPU: U15)

Part No. and name 74LS138 DSLCT **U16**

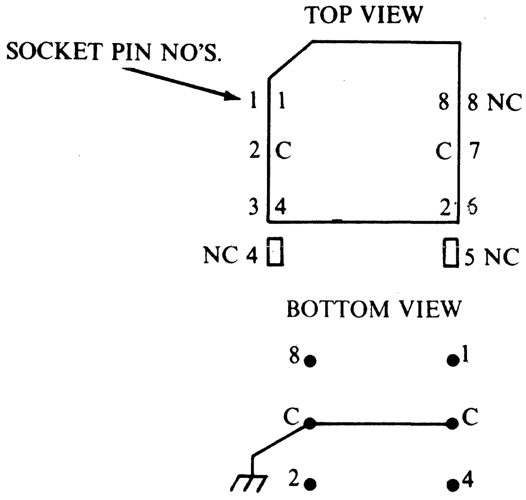
FROM pin no.		TO U no.	pin no.	SIGNAL NAME	REMARKS
1		U23	9	DAØ	Display Select Address bit, DAØ
2		U23	1Ø	DA1	Display Select Address bit, DA1
3		U23	11	DA2	Display Select Address bit, DA2
4		U16	5	Chip enable	Tied low to enable
5		U16	8	Chip enable	Tied low to enable
6		U23	2	Chip enable	Pulled high to enable
7	See	U11	2	@DSOUT	
8				GND	
9		NC		@DSSP	*See note. Important!
1Ø	See	U15	19	@DSIR	
11	See	U15	1	@DSFL	
12	See	U14	19	@DSADR	
13	See	U14	1	@DSFBUS	
14	See	U13	19	@DSBBUS	
15	See	U1Ø	19	@DSDBUS	
16				Vcc	

*Note: Leave as NC when the control system is constructed. May be used to enable display of spare buffer when control system is not constructed by connecting to U13--1.

Figure 5-31 DSLCT Wire-Wrap List (CPU: U16)

DIAL NO.	DOTS INDICATE TERMINALS CONNECTED TO COMMON			
	1	2	4	8
0				
1	•			
2		•		
3	•	•		
4			•	
5	•		•	
6		•	•	
7	•	•	•	
8				•
9	•			•

TRUTH TABLE
TYPE B02–RED SWITCH



U17

Part No. and name 23ØØØ2G (EECO) Switch, BCD

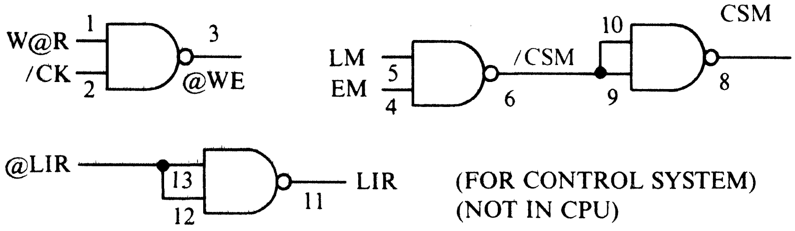
FROM pin no.	TO U no.	pin no.	SIGNAL NAME	REMARKS
1	U16	1	DAØ	
2	U17	7	GND	
3	U16	3	DA2	
4	NC			Not used--locate
5	NC			SW on pin 1 of socket.
6	U16	2	DA1	
7			GND	
8	NC			Not used

Figure 5-32 Switch, BCD Wire-Wrap List (CPU: U17)

Part No. and name	TIL311	HEX DISPLAY, TI	**U18**

FROM	TO			
pin no.	U no.	pin no.	SIGNAL NAME	REMARKS
1	U18	14	Vcc	For LEDs
2	U15	5	DIS1	
3	U15	3	DISØ	
4	NC			Not used
5	U18	7	STB	Used as enabled
6	NC			Not used
7			GND	
8	U18	5	blanking	Inactivated
9	NC			Not used
1Ø	NC			Not used
11	NC			Not used
12	U15	9	DIS3	
13	U15	7	DIS2	
14			Vcc	For logic

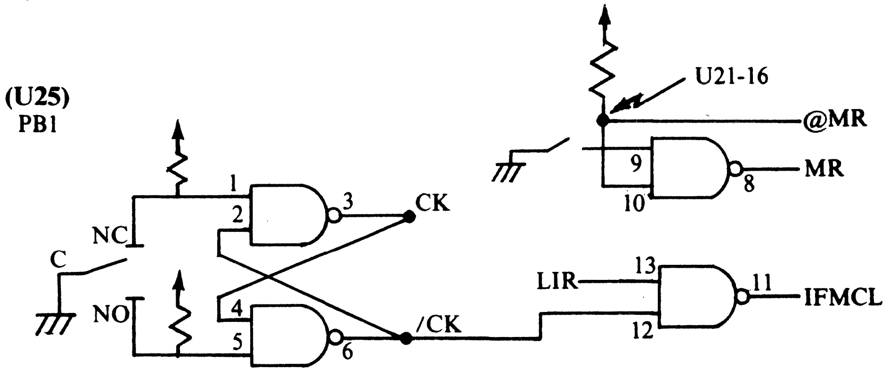
Figure 5-33 Hex Display, TI Wire-Wrap List (CPU: U18)



Part No. and name	74LSØØ	NAND GATES	**U19**

FROM		TO			
pin no.		U no.	pin no.	SIGNAL NAME	REMARKS
1	See	U1	11	W@R	
2		U2Ø	2	/CK	
3	See	U1	12	@WE(R)	
4	See	U8	5	@EM	
5	See	U9	1Ø	@LM	
6		U19	9	/CSM	
7				GND	
8	See	U9	8	CSM	Chip select memory
9		U19	1Ø	/CSM	Inversion of CSM
1Ø	See	U19	9	/CSM	
11		U2Ø	13	LIR	For control system
12		U19	13	@LIR	
13		U22	11	@LIR	
14				Vcc	

Figure 5-34 NAND-Gate Wire-Wrap List (CPU: U19)

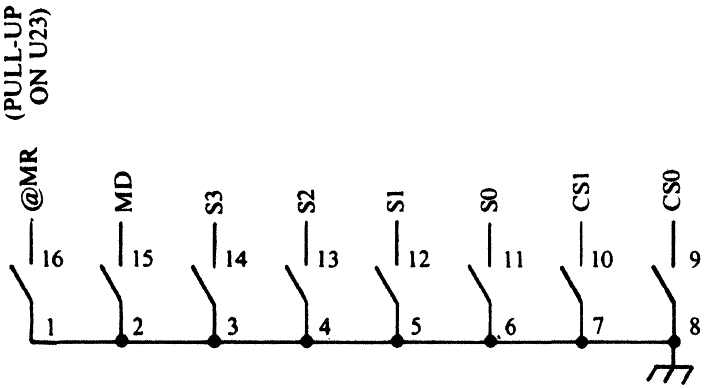


(FOR CONTROL SYSTEM NOT IN CPU)

Part No. and name	74LSØØ	NAND GATES	U20

FROM pin no.		TO U no.	 pin no.	SIGNAL NAME	REMARKS
1		U23	14	PB1NC	Pull-up
2		U2Ø	6	/CK	
3		U2Ø	4	CK	
4	See	U2Ø	3	CK	
5		U23	15	PB1NO	Pull-up
6	See	U2Ø	2	/CK	
7				GND	
8	See	U5	15	MR	
9		U2Ø	1Ø	@MR	
1Ø	See	U2Ø	9	@MR	
11	See	U3Ø	11	@IFMCL	Used only with control system
12		U2Ø	6	/CK	Used only with control system
13	See	U19	11	LIR	Used only with control system
14				Vcc	

Figure 5-35 NAND-Gate Wire-Wrap List (CPU: U20)



NOTE:
SEE U24,
U23 FOR
PULL-UPS

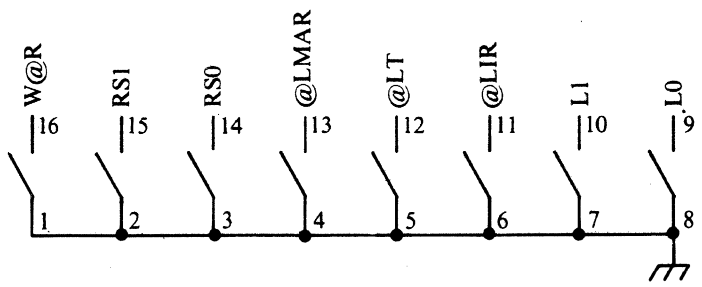
Part No.
and name

CSTR1 DIPSW

U21

FROM		TO			
pin no.		U no.	pin no.	SIGNAL NAME	REMARKS
1		U21	2	L	to GND
2		U21	3	L	to GND
3		U21	4	L	to GND
4		U21	5	L	to GND
5		U21	6	L	to GND
6		U21	7	L	to GND
7		U21	8	L	to GND
8				GND	
9	See	U3	14	CSØ	
1Ø	See	U3	2	CS1	
11	See	U13	2	SØ	
12	See	U13	4	S1	
13	See	U13	6	S2	
14	See	U13	8	S3	
15	See	U2	8	MD	
16		U2Ø	9	@MR	

Figure 5-36 CSTR1 DIPSW Wire-Wrap List (CPU: U21)



NOTE: SEE U24 FOR PULL-UPS.

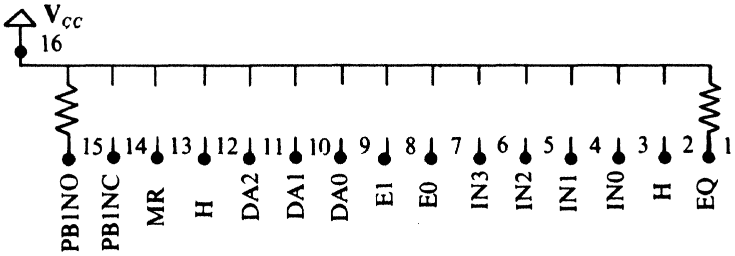
Part No.
and name

CSTRØ DIPSW

U22

FROM		TO			
pin no.		U	pin	SIGNAL NAME	REMARKS
		no.	no.		
1		U22	2	L	to GND
2		U22	3	L	to GND
3		U22	4	L	to GND
4		U22	5	L	to GND
5		U22	6	L	to GND
6		U22	7	L	to GND
7		U22	8	L	to GND
8				GND	
9	See	U8	14	LØ	
1Ø	See	U8	13	L1	
11	See	U7	1Ø	@LIR	
12	See	U4	1Ø	@LT	
13	See	U5	1Ø	@LMAR	
14		U1	14	RSØ	
15		U1	13	RS1	
16		U19	1	W@R	

Figure 5-37 CSTR0 DIPSW Wire-Wrap List (CPU: U22)



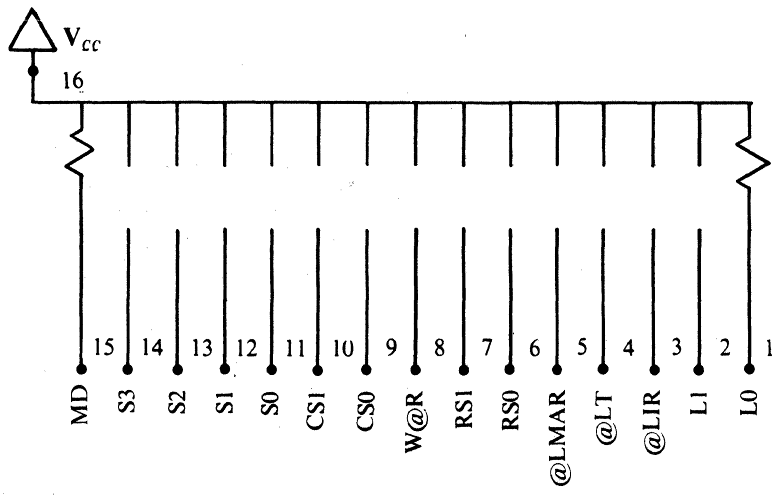
Part No.
and name

4.7 KΩ , 15 resistor DIP

U23

FROM pin no.		TO U no.	 pin no.	SIGNAL NAME	REMARKS
1		U2	14	EQ	
2	See	U16	6	H	Pull-up to en. U16
3		U12	9	INØ	Input switch
4		U12	1Ø	IN1	Input switch
5		U12	11	IN2	Input switch
6		U12	12	IN3	Input switch
7		U12	15	EØ	DBUS source (CSTR2)
8		U12	16	E1	DBUS source (CSTR2)
9	See	U16	1	DAØ	Display address of selector
1Ø	See	U16	2	DA1	Display address of selector
11	See	U16	3	DA2	Display address of selector
12	See	U3	3	H	Pull-up for CI
13		U21	16	@MR	
14	See	U2Ø	1	PB1NC	To push-button clock
15	See	U2Ø	5	PB1NO	To push-button clock
16				Vcc	

Figure 5-38 4.7 KΩ 15-resistor pack (CPU: U23)



Part No. and name 4.7 KΩ , 15 resistor DIP **U24**

FROM	TO			
pin no.	U no.	pin no.	SIGNAL NAME	REMARKS
1	U22	9	LØ	CSTRØ signal simulation
2	U22	1Ø	L1	CSTRØ signal simulation
3	U22	11	@LIR	CSTRØ signal simulation
4	U22	12	@LT	CSTRØ signal simulation
5	U22	13	@LMAR	CSTRØ signal simulation
6	U22	14	RSØ	CSTRØ signal simulation
7	U22	15	RS1	CSTRØ signal simulation
8	U22	16	W@R	CSTRØ signal simulation
9	U21	9	CSØ	CSTR1 signal simulation
1Ø	U21	1Ø	CS1	CSTR1 signal simulation
11	U21	11	SØ	CSTR1 signal simulation
12	U21	12	S1	CSTR1 signal simulation
13	U21	13	S2	CSTR1 signal simulation
14	U21	14	S3	CSTR1 signal simulation
15	U21	15	MD	CSTR1 signal simulation
16			Vcc	

Figure 5-39 4.7 KΩ 15-resistor pack (CPU: U24)

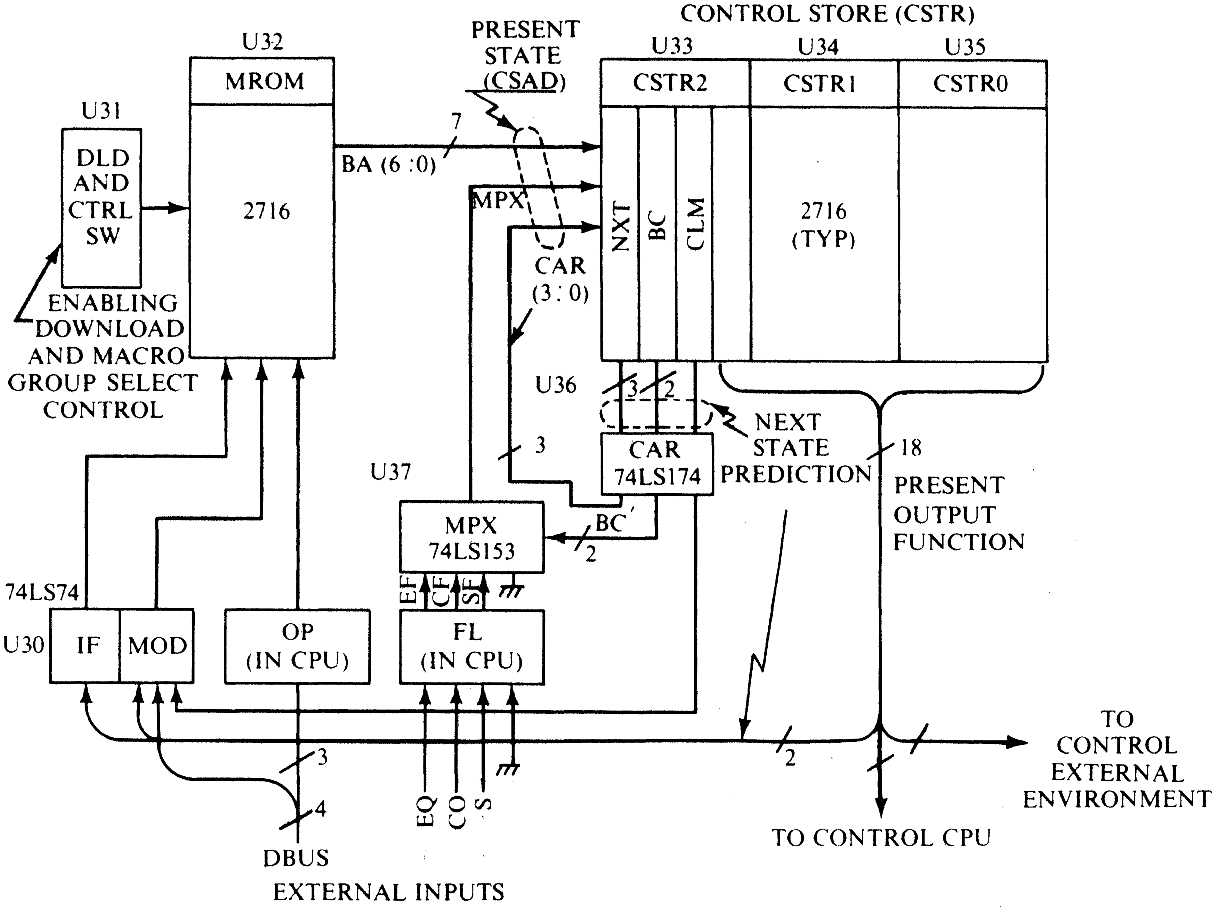


Figure 5-40 STUPIDD V Control-System Block Diagram

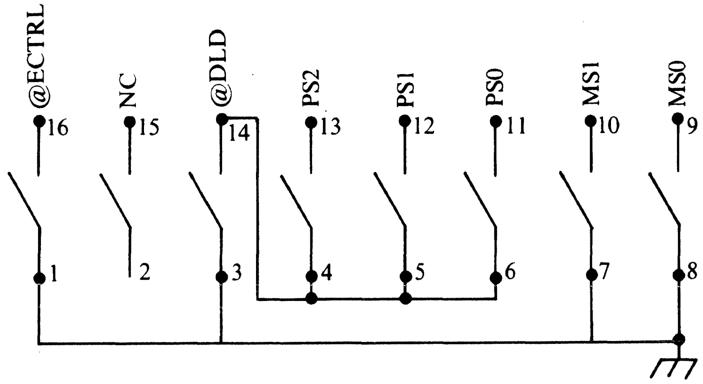
Item No.	Part No.	Description and Source	Quant.
1	74LS74	IF, MOD flip-flops; U3Ø	1
2	74LS241	3-state driver, DLD; U38	1
3	2716	EPROM, U32-35	4
4	74LS174	CAR; U36	1
5	74LS153	MPX; U37	1
6		Socket, wire wrap, 3 high, 24 pin	4
7		Socket, wire wrap, 3 high, 2Ø pin	1
8		Socket, wire wrap, 3 high, 16 pin	3
9		Socket, wire wrap, 3 high, 14 pin	1
1Ø		15-resistor array, 4.7 KΩ	1
11	74LS241	DIP SW, SPST, 8 pos.; U31	1

Note: PC board, etc., are in CPU system.

Figure 5-41 Control-System Parts List

Part No. and name	74LS74			IF & MOD	U30
1	U39	9	H	Pull-up	
2	U34	17	ST		
3	U3Ø	11	@IFMCL		
4	U21	16	@MR		
5	U32	4	IF		
6	NC			Not used	
7			GND		
8	NC			Not used	
9	U32	5	MOD		
1Ø	U3Ø	1	H	Pull-up	
11	See U2Ø	11	@IFMCL		
12	U7	11	D3		
13	U36	2	@CLMD		
14			Vcc		
FROM	TO				
pin no.	U no.	pin no.	SIGNAL NAME	REMARKS	

Figure 5-42 IF and MOD Wire-Wrap List (Control: U30)



1. PULL-UPS ON U39

2. PIN NUMBERS ARE THOSE OF SOCKET.

Part No. and name	SPST DIP SWITCH SINGLE-POLE, SINGLE-THROW			DLD & MACRO SET CONTROL	U31
FROM	TO				
pin no.	U no.	pin no.	SIGNAL NAME	REMARKS	
1	U31	3	L		
2	NC			Not used	
3	U31	7	L		
4	U31	14	@DLD	@DLD also controls	
5	U31	4	@DLD	operation of program	
6	U31	5	@DLD	select switches	
7	U31	8	L		
8			GND		
9	U32	3	MSØ	{Macro set select control	
1Ø	U32	2	MS1	Macro set select control	
11	U32	1	PSØ	{Program select (PS)	
12	U32	23	PS1	during down-load;	
13	U32	22	PS2	otherwise H	
14	U32	19	@DLD	Down-load	
15	NC			Not used	
16	U39	1	@ECTRL		

Figure 5-43 DLD and Macro-Set Control Wire-Wrap List (Control: U31)

Part No. and name   2716   MROM   **U32**

FROM pin no.		TO U no.	pin no.	SIGNAL NAME	REMARKS
1	See	U31	11	PSØ	
2	See	U31	1Ø	MS1	
3	See	U31	9	MSØ	
4	See	U3Ø	5	IF	
5		U15	11	MOD	
6		U7	5	IR2	
7		U7	4	IR1	Carry in
8		U7	3	IRØ	
9		U33	4	BAØ	
1Ø		U33	3	BA1	
11		U33	2	BA2	
12				GND	
13		U33	1	BA3	
14		U33	23	BA4	
15		U33	22	BA5	
16		U33	19	BA6	
17		NC			Not used
18		U33	18	@ECTRL	
19		U13	1	@DLD	
2Ø		U32	18	@ECTRL	
21		U32	24		Vpp; ties to Vcc for operation
22	See	U31	13	PS2	
23	See	U31	12	PS1	
24				Vcc	

Figure 5-44 MROM Wire-Wrap List (Control: U32)

Part No. and name   2716   CSTR2   **U33**

FROM pin no.	TO U no.	pin no.	SIGNAL NAME	REMARKS
1	U34	1	BA3	
2	U34	2	BA2	
3	U34	3	BA1	
4	U34	4	BAØ	
5	U34	5	MPX	
6	U34	6	CAR2	
7	U34	7	CAR1	
8	U34	8	CARØ	
9	U8	2	EØ	
1Ø	U8	3	E1	
11	U36	3	CLM	
12			GND	
13	U36	4	BCØ	
14	U36	6	BC1	
15	U36	11	NØ	
16	U36	13	N1	
17	U36	14	N2	
18	U34	18	@ECTRL	
19	U34	19	BA6	
2Ø	U33	18	@ECTRL	
21	U33	24		Vpp; ties to Vcc for operation
22	U34	22	BA5	
23	U34	23	BA4	
24			Vcc	

Figure 5-45 CSTR2 Wire-Wrap List (Control: U33)

Part No. and name 2716 CSTR1 **U34**

FROM pin no.		TO U no.	TO pin no.	SIGNAL NAME	REMARKS
1		U35	1	BA3	
2		U35	2	BA2	
3		U35	3	BA1	
4		U35	4	BAØ	
5		U35	5	MPX	
6		U35	6	CAR2	
7		U35	7	CAR1	
8		U35	8	CARØ	
9		U21	9	CSØ	
1Ø		U21	1Ø	CS1	
11		U21	11	SØ	
12				GND	
13		U21	12	S1	
14		U21	13	S2	
15		U21	14	S3	
16		U21	15	MD	
17	See	U3Ø	2	ST	
18		U35	18	@ECTRL	
19		U35	19	BA6	
2Ø		U34	18	@ECTRL	
21		U34	24		Vpp; ties to Vcc for operation
22		U35	22	BA5	
23		U35	23	BA4	
24				Vcc	

Figure 5-46 CSTR1 Wire-Wrap List (Control: U34)

Part No. and name 2716 CSTRØ **U35**

FROM pin no.		TO U no.	TO pin no.	SIGNAL NAME	REMARKS
1	See	U34	1	BA3	
2	See	U34	2	BA2	
3	See	U34	3	BA1	
4	See	U34	4	BAØ	
5		U15	8	MPX	
6	See	U34	6	CAR2	
7	See	U34	7	CAR1	
8	See	U34	8	CARØ	
9		U22	9	LØ	
1Ø		U22	1Ø	L1	
11		U22	11	@LIR	
12				GND	
13		U22	12	@LT	
14		U22	13	@LMAR	
15		U22	14	RSØ	
16		U22	15	RS1	
17		U22	16	W@R	
18	See	U34	18	@ECTRL	
19	See	U34	19	BA6	
2Ø		U35	18	@ECTRL	
21		U35	24		Vpp, ties to Vcc for operation
22	See	U34	22	BA5	
23	See	U34	23	BA4	
24				Vcc	

Figure 5-47 CSTR0 Wire-Wrap List (Control: U35)

Part No. and name	74LS174	CAR	**U36**

FROM		TO			
pin no.		U no.	pin no.	SIGNAL NAME	REMARKS
1		U3Ø	4	@MR	
2	See	U3Ø	13	@CLMD	
3	See	U33	11	CLM	
4	See	U33	13	BCØ	
5		U37	14	BCØP	
6	See	U33	14	BC1	
7		U37	2	BC1P	
8				GND	
9		U2Ø	4	CK	
1Ø		U33	8	CARØ	
11	See	U33	15	NØ	
12		U33	7	CAR1	
13	See	U33	16	N1	
14	See	U33	17	N2	
15		U33	6	CAR2	
16				Vcc	

Figure 5-48 CAR Wire-Wrap List (Control: U36)

Part No. and name	74LS153 (½)	MPX	**U37**

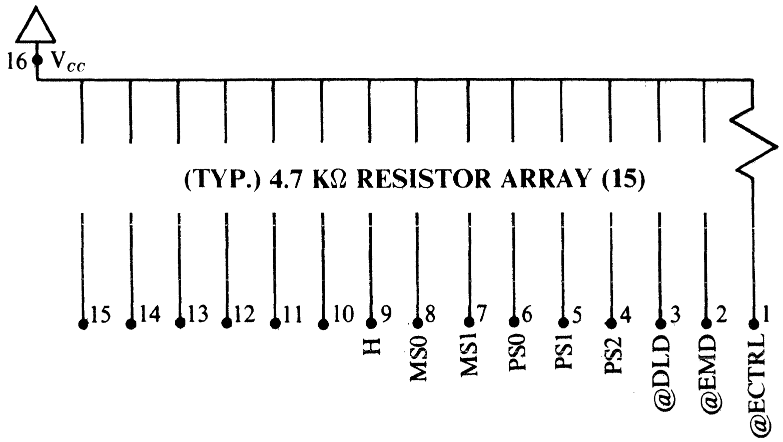
FROM		TO			
pin no.		U no.	pin no.	SIGNAL NAME	REMARKS
1		U37	8	L	Tie to GND
2	See	U36	7	BC1P	Branch control output from CAR
3		U6	3	EF	MPX equal flag
4		U6	5	SF	MPX sign flag
5		U6	4	CF	MPX carry flag
6		U37	8	L	MPX a low (tie to GND)
7		U33	5	MPX	
8				GND	
9		NC			Not used
1Ø		NC			Not used
11		NC			Not used
12		NC			Not used
13		NC			Not used
14	See	U36	5	BCØP	Branch control output from CAR
15		NC			Not used
16				Vcc	

Figure 5-49 MPX Wire-Wrap List (Control: U37)

Part No. and name 74LS241 DLD CTRL **U38**

FROM pin no.	TO U no.	 pin no.	SIGNAL NAME	REMARKS
1	U31	14	@DLD	
2	U36	1Ø	CARØ	
3	NC			Not used
4	U36	12	CAR1	
5	NC			Not used
6	U36	15	CAR2	
7	NC			Not used
8	U37	7	MPX	
9	U39	2	@EMD	
1Ø			GND	
11	U38	1Ø		to GND
12	U5	6	ADR3	
13	NC			Not used
14	U5	5	ADR2	
15	NC			Not used
16	U5	4	ADR1	
17	NC			Not used
18	U5	3	ADRØ	
19	U38	1	@DLD	
2Ø			Vcc	

Figure 5-50 DLD CTRL Wire-Wrap List (Control: U38)



Part No. and name 4.7 KΩ , 15-resistor DIP pack **U39**

FROM pin no.	TO U no.	 pin no.	SIGNAL NAME	REMARKS
1	U32	18	@ECTRL	
2	U5	1	@EMD	
3	U31	14	@DLD	
4	U31	13	PS2	
5	U31	12	PS1	
6	U31	11	PSØ	
7	U31	1Ø	MS1	
8	U31	9	MSØ	
9	See U3Ø	1	H	Pull-up
1Ø	NC			Not used
11	NC			Not used or spare
12	NC			Not used or spare
13	NC			Not used or spare
14	NC			Not used or spare
15	NC			Not used or spare
16			Vcc	

Figure 5-51 4.7 KΩ, 15 Res. DIP Pack Wire-Wrap List (Control: U39)

Microprogramming Formats: The Control Word

System Control Word

The link between the control system and the resulting behavior of both the CPU and its external environment is embodied in the *system control word.* We begin our study of its symbolic notations, formats, and applied methods of usage in this section. This control word is a set of binary signals, produced by the control system, which affects the entire system–not just the CPU. The control system emits this control word at the start of each period, as defined by the system clock. At the start of the current period, the control signals are not likely to be valid. Some time is required for them to stabilize at their correct binary level.

System Timing Characteristics

Portions of the control word influence purely combinational-logic circuitry, *immediately;* the balance control the loading of synchronous memory devices by the clock at the *end* of the current period. For example, if data is to be fed, say, from STUPIDD's T register onto the DBUS to the inputs of a selected sink, it must commence as soon as possible–so that the data, bus-path, and device-control signals will have stabilized before the end of the current period. At the *end* of the period, those synchronous devices so specified by the control word *capture* this information. This captured information becomes available for use during the *next* clock period. In reality, the system clock is the part of the control word that says "Go" to the enabled synchronous memory devices. By convention, we rarely talk about the clock as part of the control word–but it is.

Both the combinational-logic and synchronous device-control signals are emitted by the control system at the start of each new period. Once the data has stabilized on the newly enabled data paths and the synchronous device-control levels have stabilized too, then and only then can the clock signal safely terminate the current period. Some of the stored information is fed back to the control system to influence its future actions. Although the control system is a finite-state machine, it can make state-path choices only within its finite state-set, based on the stored results from the current period and the present external inputs. The required signal-stabilization time is an important factor in limiting the speed of systems. Our many past references to clock signals and the clocking characteristics of synchronous devices are about to be utilized in a system context. First, though, we must look at the symbolic notations and formats used in the control word.

Symbolic Microprogramming Notations

No design language or computer is yet capable of inventing a systems concept–this is a human's creative contribution. Creative expression evolves out of our ability to state desired systems performance in a convenient notation. Computer design and description languages provide orderly means of documenting, reviewing, simulating, and modifying systems concepts. Their utility in the design of instruction-set algorithms in the development of a microprogrammed control store will become evident as we proceed. The notation used here is only slightly different from others in popular use, in that some of their minor shortcomings are avoided. First, the symbol set used is restricted to a subset of the standard 7-level ASCII code that is used by the keyboards of all our CRT terminals and personal computers. Design or description languages often use symbol sets not available on most terminals, processors, and personal computers. This is important to us if we want to automate the implementation of microcode later, using a terminal or personal computer. Second, new symbology for the expression of clock-timing relationships is presented. This aspect is mentioned now; it actually will become useful in a planned subsequent book on the *synthesis* of processor systems. In this text, we are concentrating on the analysis and microprogramming aspects of a given processor, whose behavior has general applicability.

CPDL Notation

We now introduce some of the *control-processor design language* (CPDL) symbols and their meaning, which we currently use in the development of microcode. These are used in a form of *register-transfer notation* (RTN) for specifying operations that are to occur within a line of microcode. For further insight into the design-language approach, the reader is referred to the Duley and Dietmeyer References at the end of this chapter. Table 5-5 summarizes the symbology used in the balance of this text.

Table 5-5
Introductory Set of CPDL Symbols

Symbol	*Description*	*Example*	*Meaning*
/	Negation operator	/A	NOT-A
@	Active-low indicator	@A	A is active (true) when low.
<:	Replacement operator	A <: B	A is to receive the contents B at the end of the current clock period; the contents of B are not altered.
>:	State transfer operator	>: C	We are to go to state C at the end of this clock period.
&	Boolean AND operator	A&B	A AND B
/&	NAND function	A/&B	A NAND B
+	Boolean OR operator	A+B	A OR B
/+	NOR function	A /+ B	A NOR B
?	Exclusive-OR function	A ? B	A XOR B or A ⊕ B
/?	Equivalence function	A /? B	A ⊙ B
pl	Arithmetic addition	A pl B	A plus B (See ALU data sheets.)
mi	Arithmetic subtraction	A mi B	A minus B

ELECTRICAL-LEVEL MICROCODING FORMS

The electrical-level coding format for specifying the control word is presented in Figure 5-52. This blank form is used several times, and the reader will want to make sufficient copies of it for the exercises that follow. The leftmost column will contain the CPDL symbolic notation that specifies what is to be accomplished by this line of microcode. The notation presented in Table 5-5 is elaborated on as we proceed. As a convenience, the darkened dots at the top of

CSTR ADDRESS:							ASSIGNED OP-CODE:																		
LINE NO.	MACRO: ADR. CPDL DESCRIPTION:	NEXT ADR.			BRCH. CTRL.		MOD	DBUS SOURCE FIELD		STATE	ALU FIELD							SINK CONTROL FIELD							
											ALU FUNC.					CS		R ARRAY			LOAD ENABLES				
		N2	N1	N0	BC1	BC0	CLM	E1	E0		MD	S3	S2	S1	S0	CS1	CS0	W@R	RS1	RS0	LMAR	LT	LIR	L1	L0
0																									
1																									
2																									
3																									
4																									
5																									
6																									
7																									
8	0																								
9	1																								
A	2																								
B	3																								
C	4																								
D	5																								
E	6																								
F	7																								
CPDL NOTATIONS		CSTR2								CSTR1								CSTR0							

'igure 5-52 Microprogram Electrical-Level Coding Form (Blank) some columns indicate that the associated control signals are active low. We now examine the meaning and function of the signal terms found in the control word's fields. These are the terms we must specify in creating microcode.

Microcoding Fields and Signals Terms

At the bottom of Figure 5-52 appear the legends CSTR0 (*control store 0*), CSTR1, and CSTR2. Each of these is a byte of information produced by the control system. In this beginning phase, we can simulate most of these signals with the set of DIP switches on the construction-project board. The object is to manage each IC's control pins as the system would. The function of these IC pins was described earlier in this chapter. At the top of Figure 5-52, the control word is broken up into meaningful convenient fields. Starting at the left, switch CSTR2 contains the *next* address (NXT), *branch control* (BC), and the operand addressing mode *modification* (MOD) bit fields. These three fields relate purely to the control-system operation and will be analyzed in Chapter 6. In addition, CSTR2 contains the DBUS source field. The two signals from this field drive half of the DEC0 IC (U8), to select the master of the DBUS, as discussed previously.

CSTR1 contains the major *state* (ST) and ALU fields. The ALU field is broken down into the signal subfields that control the ALU. These are MD, for arithmetic/logic *mode* selection; the four S lines (S3 .. S0), which select the current function; and the carry-select lines, CS1 and CS0. These last two are displayed as a subset of the total ALU field, the CS field, since the carry-in, CI, of the ALU also controls the precise arithmetic-mode function performed. The control signal details for the individual IC's have already been presented. The ST field issues the signal that contributes to the transitions between the IF and EX major states. This field too is purely a part of the control system. In the earlier discussion of the IR, it was pointed out that it is valid *only* when @LIR, which is one of the signals of CSTR0, is active. LIR stands for *load instruction register.* When @LIR is true, the signal ST determines the next major state, which must be either IF, when the ST line is high, or EX, when it is low. LIR and ST are used, simultaneously, only at the *very end* of an algorithm—where it is necessary that we coordinate a major state change. Together, they control transitions between the IF and EX major states of the cycle of computation.

Horizontal versus Vertical Microprogramming

CSTR0 encompasses all of the register *array* as well as the discrete register *sink*-control fields. It is the source of the load-enable signals that control the clocking of the synchronous information-storage devices of the system. One very often wants to load more than one of the discrete registers simultaneously. When this is the case, *horizontal* microprogramming methods (one signal for each control function) for the control signals are used. Note, however, that this requires a larger (wider) control word. Use of short fields and decoders is termed *vertical* microgrammed design. These two popular terms, which have already been discussed, represent the two extremes in governing the size of a control word or of one of its fields.

Two signals in the sink-control field, L1 and L0, illustrate a vertically microprogrammed subfield. They drive half of the 74LS139 decoder (U8). The clock signal is used in a special way in this application. Note that the input to pin 15, the chip-enable input for this half of the IC U8, is driven by the system clock. This means that all the outputs are automatically disabled when the clock is high, and only the addressed output of this half of U8 can be active in the last half of the period, when the clock is low. This neatly avoids noise problems associated with state transitions during the first half of the period, by masking them out. Important: since the outputs are clock-qualified, they may also be directly used as write strobes to the devices that this half of U8 serves. The DEC1 outputs control the loading of the discrete registers and memory writes as shown in Table 5-6.

Table 5-6
Vertical Sink-Control Switch Settings

L1	*L0*	*Signal Produced (Register enabled)*
L	L	@LM (Load Memory)
L	H	@LF (Load Flags)
H	L	@LO (Load Output)
H	H	Not Connected (No register enabled)

In Table 5-6, @LM is a signal authorizing a memory write. The function of the memory-address register, MAR, is to *select* the specific location we desire to communicate with (for both reads and writes). When @LM is active, memory is written into at the location specified in MAR. Note too that, by using HH as the selection-input condition to DEC1, no device is enabled to be loaded. The net result is that three load-enables have been compressed into two control-system signals, thus permitting a smaller control word to be used. The number of control signals that can be encoded into a vertical field is a power of 2 function of the field size. Considerable control-word-size compression can be obtained this way, but at the expense of using subsequent decoders. In this application, there is no need to make the loading of these registers mutually exclusive. There was a need, however, to reduce the size of the control word–trading off flexibility for hardware simplicity in the process.

Register Array Microprogram Control

The register-array subfield of the sink-control signals manages the loading and reading of the 4 × 4 generalized register array, R (U1). W@R is the signal that determines whether a read or write is to take place. This array is normally in the read mode, unless a write is specifically desired. So long as we do not unintentially press the clock button, a temporary write switch setting has no effect. The other two bits, RS1 and RS0, of the R field determine which particular register is to be addressed within the array. Register usage varies, but, as noted earlier, R3 is reserved for PC purposes.

Microprogram Functional Effects

What do these control-word signals and fields accomplish? The simple correct answer is that they control functions. Some generalizations on the nature of the control fields and their signals help to clarify their functional use in the system. First, we can consider some as belonging to a group that controls devices such as the ALU. This is straightforward control. Second, we can categorize some of these signals as members of a group that determine access to a system bus. The number of source control fields increases directly with the number of buses the system has. Any one source field controlling access to a single bus benefits from some scheme of mutual exclusivity, such as vertical microprogramming. This guarantees that only one source at a time is master of the bus. Last, we have the sink-field category, which emits the signals that specify which registers currently receive information. Sink fields tend to benefit from horizontal organization, which allows any number of registers to be loaded simultaneously. Vertical organization may be used in sink fields to shorten the control word, often at the probable expense of more clock cycles to obtain a given result. Of course, where different registers *must* be loaded at the same time, then separate sink signals are required.

In summary, then, it is useful to think of control-word signals as members of one of the following groups: direct device control, bus-access control, or register-sink control. These categorizations are useful in forming a total system functional perspective. In reality, a control is a control. The goal is to understand their system uses to control behavior at some specific point in time.

ALGORITHMS (MACROS)

Macro Overall CPDL Description

Refer once more to Figure 5-52, the electrical-level microcoding sheet that displays the control-word format. Notice the leftmost column, headed by the title MACRO. This is where we enter the name of the macro and, later, its starting address in the control store. Every total operation that STUPIDD performs is referred to as an *algorithm* or a *macro* operation. These terms are synonomous in the current context. Typical macros are the instruction fetch cycle, IF, and the execution of addition, ADD. Beneath the *macro name* is a space reserved for the form of register transfer notation, called CPDL, that is used in this text. The description of the overall *function* to be performed by the macro is placed here, where feasible. This does not tell us *how* the machine did the task, just what net result was obtained by the macro.

Microoperations

This system permits a maximum of sixteen line numbers per algorithm (hex 0 through F). Each separate line is referred to as a *micro* operation, from which the term *microprogramming* is derived. A micro operation is, very simply, the set of control signals that we wish to create during any one clock period–and it corresponds to a single line of this coding sheet. A *macro,* then, is a *collection of micros* that perform some useful algorithm. The *operational capabilities* of a system are expressed by the total set of macros it possesses. These macros establish what it can do for us. For example, the instruction set of a microprogrammed computer consists of its set of macros.

Compatible Sets of Operations

We are about to examine the specification of macros as consisting of *sets of compatible microoperations.* Each line of microcode commences with a CPDL notation or a description of the overall purpose of the line–why it is there. More than one operation may be *simultaneously* specified in a line of microcode. This parallelism of operations is beneficial, since it increases throughput. The microprogrammer, however, has the responsibility of seeing that these operations do not conflict, that is, that they are compatible. Therefore, we must constantly check to see that the architecture, devices, or buses can support each operation and that there are no bus conflicts. In essence, this requires that we really get to know the systems we work with.

Restrictions

Since we have not as yet studied the control-system operation, the following restrictions apply for the present:

> All algorithms start at line 0.
>
> All algorithms progress from line to line in sequence, until complete.

These restrictions will be lifted later. For the present, let us examine how the CPU and its external environment react to our switch-simulated sequence of control words of the IF algorithm. Each step or line of microcode of this sequence is consummated with a clock pulse.

IF ALGORITHM

Electrical-Level Coding Entries

The algorithm that performs the fetch of an instruction (IF) is detailed in Figure 5-53. (Refer as needed to our previous discussions of the cycle of computation in Chapter 1, its major states of IF and EX, and the tasks that are performed during IF.) Figure 5-53 displays the sequence of control signals that accomplish these tasks. That portion of the control word that relates only to control-system operation is ignored for the present. These are the leftmost six columns of the control word in the figure. The coding entries may be either an H (for electrical high), an L (for electrical low), or an X (don't care condition). As noted, the CPDL descriptions of each line's functions appear to the left of each line. These constitute a brief description of the *set* of compatible operations to be performed on this line. The coding table itself is the complete description of exactly how this is carried out. Each line of microcode becomes the CPU system's current orders from the control system at discrete points in time.

The CPDL symbols used here are those introduced in Table 5-5. For example, *B* <: *A* means that the contents of A are to be transferred into B, leaving A unchanged. The symbol >: is the state-transfer operator. The notation >: IF means that the system is to transfer to the IF state. Following the conventions used in the TTL data catalogs for the ALU, we also use *pl* to stand for addition and *mi* for subtraction. The data sheets on the ALU use + to represent the logical OR function as we do in our CPDL notations. Let us now analyze each line of microcode for the instruction-fetch algorithm or macro.

Line 0 establishes the control conditions necessary to execute the compatible set of operations (CSOP) expressed in CPDL as:

MAR, T <: PC

That is, MAR and T are to receive the contents of the program counter. Remember that register 3 of R, the register array, was to be used as STUPIDD's program counter.

Data Path and Architecture Visualizations of Microoperations

Two important questions must be satisfactorily answered to determine whether any CPDL CSOP can, in fact, be successfully implemented. These are

1. Does the chosen architecture contain the data paths necessary to carry out the operations without conflict?

CSTR ADDRESS:					ASSIGNED OP-CODE:																				
LINE NO.	MACRO: IF	NEXT ADR.			BRCH. CTRL.		MOD	DBUS SOURCE FIELD		IF	ALU FIELD							SINK CONTROL FIELD							
	PDL DESCRIPTION: IR <: M{PC}, PC <: PC pl 1, >:EX										ALU FUNC.					CS		R ARRAY			LOAD ENABLES				
		N2	N1	N0	BC1	BC0	CLM ●	E1	E0	ST	MD	S3	S2	S1	S0	CS1	CS0	W@R	RS1	RS0	LMAR ●	LT ●	LIR ●	L1	L0
0	MAR,T <: PC							X	X	X	H	H	L	H	L	X	X	L	H	H	L	L	H	H	H
1	PC <: T pl 1							L	L	X	L	L	L	L	L	L	L	H	H	H	H	H	H	H	H
2	IR <: M{MAR}, >:EX							L	H	L	X	X	X	X	X	X	X	L	X	X	H	H	L	H	H
3																									
4																									
5																									
6																									
7																									
8																									
9																									
A																									
B																									
C																									
D																									
E																									
F																									
CPDL NOTATIONS		CSTR2								CSTR1								CSTR0							

Figure 5-53 IF Macro: Electrical-Level Form

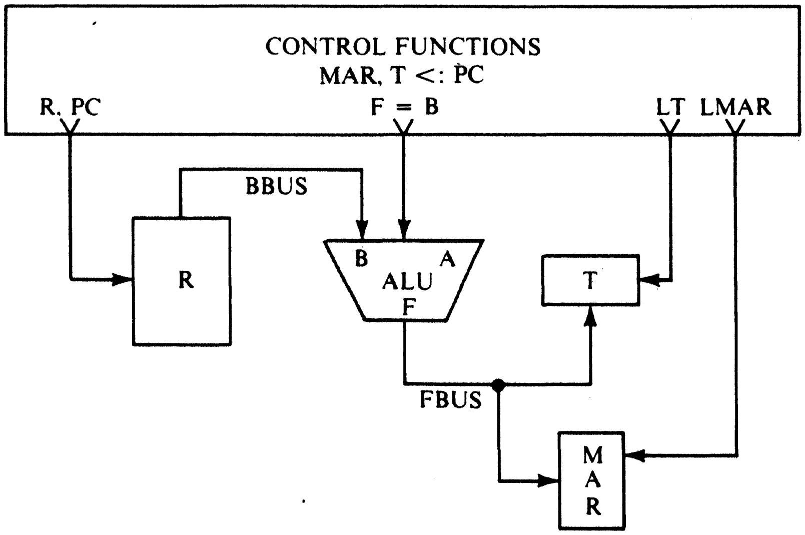


Figure 5-54
IF Algorithm Data Paths, Line 0

2. Does the control word issue all the requisite control signals to effect these operations?

This means, for each operation, can you:

a. Visualize the existence of required, nonconflicting data paths in a given architecture and
b. Show that the existing control word is capable of evoking the required functions, sourcing and sinking, that will accomplish the desired actions?

These two important questions encompass the major thought-provoking aspects of the practice of microprogramming–assuming that one understands the hardware itself.

For the beginner, the resolution of the above questions is aided by the drawing of a subset of the total architecture. Figure 5-54 does this for line 0 of the IF algorithm. It shows that information can flow from R through the ALU to the inputs of both T and MAR. Does the specified control word support this? Going back to Figure 5-53, notice the following:

1. The R array field, a subset of the sink-control field, has placed R in the read mode and selected R3 (PC) to appear at its outputs. This information is also present at the ALU's B inputs. The R array field controls the read/write of the 74LS670 general register array (U1). This is a memory IC used internally by the CPU, sometimes referred to as a *scratch pad* memory.
2. The bits of the ALU field have been specified to select the logic-function mode, and the specific function to be performed is F = B, the *transfer* B to the F port operation. This tranfers information on the BBUS to the FBUS. Since carries do not enter into logic operations, the CS subfield entries are designated as don't cares. *Important*–We are using active-high data conventions for all ALU operations in STUPIDD, as may be seen from the TTL manual.
3. As a result of the foregoing, the contents of the PC are available at the inputs to both MAR and T on the FBUS.

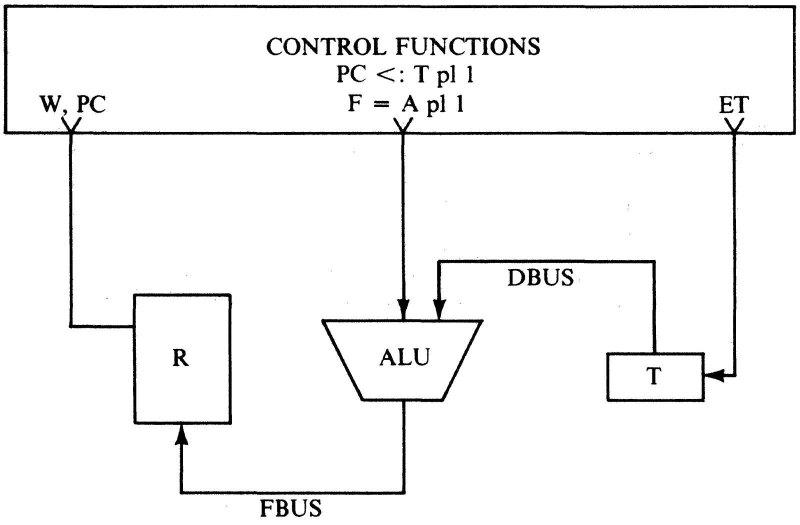


Figure 5-55
IF Algorithm Data Paths, Line 1

4. The ST field is a don't care until the end of the algorithm, when a major state change is required. This is clarified at line 2 of IF.
5. The DBUS is not involved in this subarchitectural visualization. Therefore the DBUS source-field entries are don't cares. In practice, one may wish to specify this field so as to prevent arbitrary data appearing on the bus. In fact, an HH specification can be used to keep all the sources managed by the control system off the bus.
6. The sink field contains clock-enable controls for the registers and memories of the system. The dots at the top of the columns of this field designate those enables that are active low. LMAR, LT, and LM are so designated. Note which registers have been clock-enabled. When the system clock goes high, signifying the end of the current period, MAR and T will now have been synchronously loaded with the data on the FBUS. The other registers simply ignore the clock and remain stable. Realize that, when new information is entered, the *outputs* of these load-enabled registers cannot be considered as stable for'some finite time into the *next* clock period.
7. Realize, too, why we specified the loading of T as well as of MAR on this line of microcode. It is the first step in the incrementation of the PC. On the next line (line 1) the PC will receive its present image (stored in T), incremented by one in passing through the ALU.

Again, we wish to emphasize that the skills-building facet of the preceding consists of learning how to visualize the existence of data paths and coordinating control signals. These are the two fundamental abilities required to microprogram an architecture. Figures 5-55 and 5-56 show the subarchitectures actively participating in lines 1 and 2 of the IF control-word coding. Do they make sense to you?

In line 2, the IF algorithm is about to be terminated at the end of the clock period, because LIR is active. This control line also causes a flip-flop in the control system to be loaded with the logic value designated in the ST field. Since the ST field is specified as a logic low at this point, not only is this algorithm about to be terminated, but the control system will proceed to the EX major state in the next period. In EX, the instruction code loaded into IR at the end of IF informs the

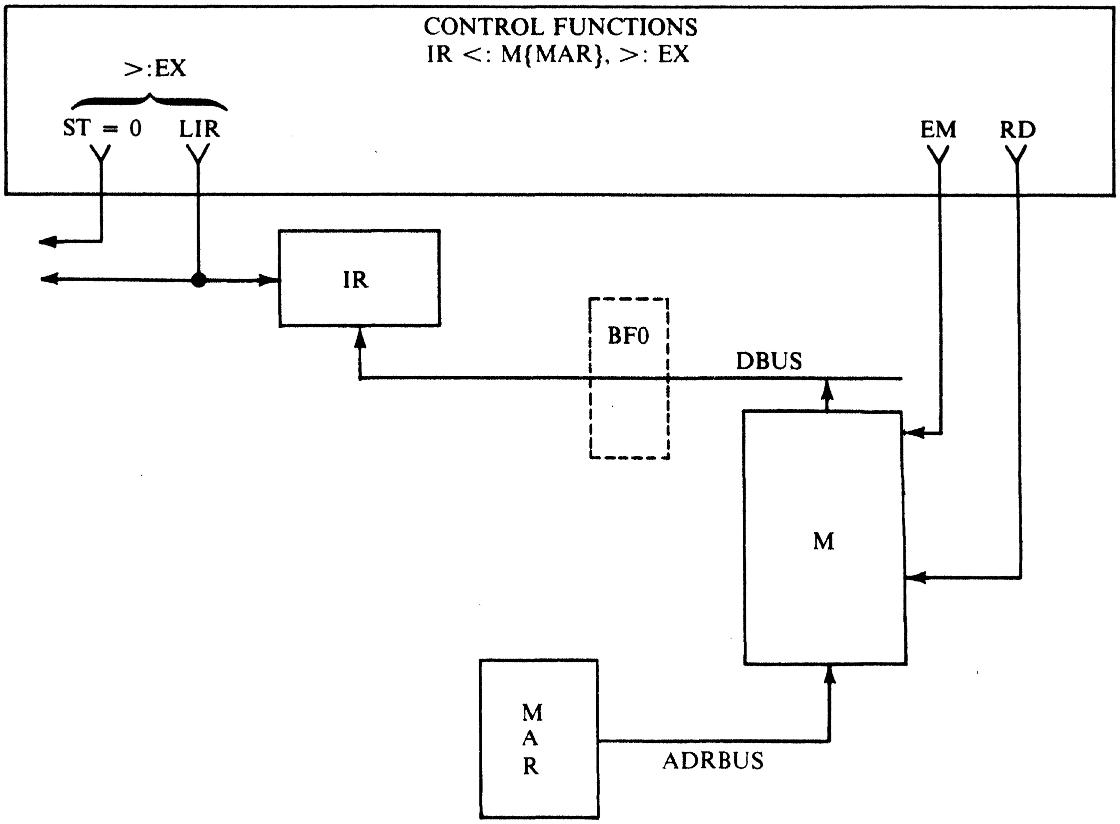


Figure 5-56
IF Algorithm Data Paths, Line 2

control system *which macro* is to be executed. The study of the control system will clarify this further. What is apparent now is that STUPIDD goes from IF to EX and back each time IR is loaded. The ST field specifies the next major state at these times. For the IF macro, ST must always lead to the EX major state. At the end of any of the many possible different EX algorithms, IR is again loaded, with an undefined value in this system at the end of EX, but ST must now specify that the IF major state is about to commence.

Symbolic Microcoding Form

The electrical-level control-word coding format used so far is a necessary one when we need to specify each bit for programming a ROM. For conceptual design, it is cumbersome to use. A symbolic control-word coding sheet is presented in Figure 5-57. This sheet utilizes more recognizable mnemonic symbols as entries in the fields. The possible entries for each field are shown in Figure 5-58. The ALU field does not show all possible entries. Only some sample entries are shown, because this ALU performs many operations–a goodly number of which are useless. New entries to this field will be introduced as we proceed, using the 74181 ALU TTL data sheets as a guide.

Symbolic Code of IF

Figure 5-59 presents the IF algorithm in this simpler symbolic format. Notice that R should always be in the read mode, unless a write operation is specifically called for. In this case, the RX symbol is employed. This precaution prevents the accidental specification of unwanted writes. Working with both microcoding formats is necessary in practice, and the exercises include the use of both. The simpler mnemonic coding form can be made to produce all the L's and H's by processing it through a microcode assembler program. The output of this assembler could be a table showing the L's and H's, as we have already done or, more to the point, it could be information in the form required to program the control-system PROM's. The task of creating this assembler is an interesting project for one seriously interested in the art of microprogramming. In Chapter 6, we shall discuss in detail the programming of the control system's EPROM's.

In the next section, we illustrate the use of this symbolic coding form in the development of the microcode for the execution of some sample instructions for STUPIDD. This also serves to introduce the factors to be considered in the EX phase of the cycle of computation.

Figure 5-57 Microprogram Symbolic Coding Form (Blank)

LINE NO.	MACRO:	NEXT ADDRESS	BRCH CTRL	CLR MOD	DBUS SOURCE FIELD	STATE	ALU FIELD		SINK CONTROL FIELD	
	CPDL DESCRIPTION:			●			ALU FUNC.	CS	R ARRAY	LOAD ENABLES
0										
1										
2										
3										
4										
5										
6										
7										
8_0										
9_1										
A_2										
B_3										
C_4										
D_5										
E_6										
F_7										
PDL NOTATIONS		CSTR2				CSTR1			CSTR0	

CSTR ADDRESS: ASSIGNED OP-CODE:

LINE NO.	MACRO: PDL DESCRIPTION:	NEXT ADDRESS	BRCH CTRL	CLR MOD	DBUS SOURCE FIELD	STATE	ALU FIELD		SINK CONTROL FIELD	
							ALU FUNC.	CS	R ARRAY	LOAD ENABLES
0		0	SF	L	T	IF	A	X	R0	MAR
1		1	CF	H	M	EX	B	CO	W0	T
2	* A pl 1 ⇒ INCR	2	EF	X	I	X	A pl 1	L	R1	M
3	+ ⇒ LOGICAL OR	3	L		0		A + B	CF	W1	OUT
4	A mi B ⇒ SUB	4					A mi B	H	R2	FL
5	A pl B ⇒ ADD	5					ETC.		W2	IR
6	A mi 1 ⇒ DECR	6							RPC	0
7		7					NOTE: F = IS IMPLIED ABOVE. CPDL LOGIC AND ARITHMETIC OPERATORS ARE USED IN THIS COLUMN. REFER TO 74181 ALU FUNCTION TABLES FOR A COMPLETE LIST OF ALU OPERATIONS.		WPC	
8									RX	
9	*Note: USE pl FOR ADDITION, mi FOR SUBTRACTION. THE NOTATIONS ADD, SUB, INCR, AND DECR ARE ALSO USED IN THE TEXT.									
A										
B										
C										
D										
E										
F										
PDL NOTATIONS		CSTR2				CSTR1			CSTR0	

Figure 5-58 Mnemonic Symbols Used in Microprogramming

CSTR ADDRESS:						ASSIGNED OP-CODE:				
LINE NO.	MACRO:	NEXT ADDRESS	BRCH CTRL	CLR MOD	DBUS SOURCE FIELD	STATE	ALU FIELD		SINK CONTROL FIELD	
	PDL DESCRIPTION: IR <: M{PC}, PC <: PC pl 1, >: EX						ALU FUNC.	CS	R ARRAY	LOAD ENABLES
0	MAR, T <: PC				X	X	B	X	RPC	MAR, T
1	PC <: T pl 1				T	X	A pl 1	L	WPC	O
2	IR <: M{MAR}, >: EX, LIR				M	EX	X	X	RX	M
3										
4										
5										
6										
7										
8										
9										
A										
B										
C										
D										
E										
F										
PDL NOTATIONS		CSTR2				CSTR1			CSTR0	

Figure 5-59 IF Macro: Symbolic Form

MICROCODING THE EX MAJOR STATE

Early in this text, the major fields of an instruction word were discussed. It was noted that, in a processor that possesses a small memory-word size, as is typical of many microprocessors, the fields of the complete instruction word are distributed over several memory words. We are careful to make the distinction between a "word" in memory (really the addressable unit of memory) and the full instruction "word," which can occupy several memory "words" or addressable units. The OP-code and MOD-bits are characteristically part of the first memory word. We shall define this to be true of STUPIDD—all OP and MOD bits are contained in the first memory word of its instructions. The *OP* field specifies the operation that is to be performed. The *MOD* field specifies the addressing mode to be used for fetching the operand(s). This one-memory-word format for MOD and OP would not be practical in a commercial 4-bit processor. It allows for the specification of only eight OP codes. At least two 4-bit memory words would be required to construct an adequate instruction set.

The succeeding memory word(s) of an instruction word, if they exist at all, contain either a *value* (v) or an *operand address* (OA). If STUPIDD contained a larger memory space and larger MAR, say 256 locations, two 4-bit memory words would be required to uniquely specify any one location within this range. For these reasons, a typical 8-bit microprocessor has a maximum instruction format of three words with an address-field maximum length of sixteen bits. What does all this have to do with the execution of an instruction? The answer is found in our need to guarantee the proper functioning of the program counter, while developing microprograms for the EX phase of operation.

ADJUSTING THE PC

The program counter must always be adjusted to point to the address of the *next* instruction *before* the IF major state is reentered. Examining some of STUPIDD's instruction-word formats, displayed in Figure 5-60, note that they are either one or two memory words in length. We will also demonstrate the creation of a three-memory-word instruction format later in this section. Let us look at these simpler examples, first.

Take the case of any two-word instruction. At the start of instruction fetch, the program counter (originally) pointed to the *address* of the OP code and MOD bits, which are contained in the first word of the instruction. This address was given to MAR, and then the PC was increased by 1 (through the intermediate use of T). The first instruction field (MOD and OP) is then fetched and placed in IR, and the EX phase is initiated. If the instruction is indeed a single-word instruction, the PC register is in fact pointing to the location of the start of the next instruction. On the other hand, if the OP code just fetched is a member of one of STUPIDD's two-word-long instructions, then the contents of the PC now include either the *address* of a value in main memory or the *address* of the *operand address* in main memory. This is the state of affairs immediately after the instruction fetch, at the start of the EX phase. The PC is, in reality, an address pointer to main memory. One must be careful to ascertain exactly what it points to.

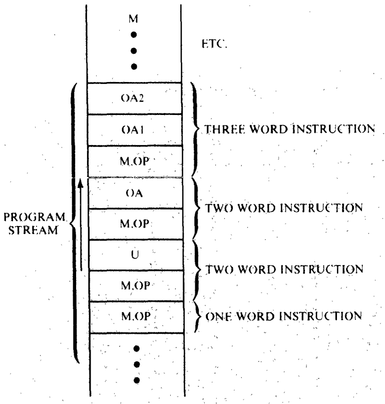


Figure 5-60
Instruction Word Formats in Program Stream

This can be a value, an operand address, or (as we later explore in indirect addressing) another pointer to an operand address.

Survey of Addition Methods

The first thing the creator of an EX macro must ask is whether the PC must be adjusted to point to the next OP code. Failure to microcode this properly will cause the system to malfunction. It is the sort of responsibility that provides interesting job opportunities. Having recognized keeping track of and possibly having to adjust the PC as one great imperative in the microprogramming of any EX algorithm, we must now also recognize a second great imperative: the clear definition of any instruction. There are a variety of ways in which an operation may be performed. We must be certain what a particular instruction actually does, in detail. To illustrate this, let us first explore some macros that perform addition. We will finally select one to include in STUPIDD's final set of macros. These are given in Table 5-7.

Table 5-7
CPDL Descriptions of Possible Add Instructions

Name of Macro	*CPDL Description*
ADR (add to r)	R{OP} <: M{OA} pl R{OP}
	or
	R{OP} <: T pl R{OP}
	or
	R(i) <: R(j) pl R(i)
ADT (add to T)	T <: R{OP} pl T
	or
	T <: M{OA} pl T
ADM (add to M)	M{OA} <: R{OP} pl M{OA}
	or
	M{OA} <: T pl M{OA}

Notes: The macro names above are temporary. Not all can be implemented on STUPIDD. {OP} means "as specified in the OP code field of the instruction word." {OA} means "as specified in the Operand Address field of the instruction word."

This tabulation of some possible candidates for future system macros that perform addition contains several implications that need to be explored. To start, the number of OP bits allocated in a design establishes the maximum number of instructions that may be implemented. The number of MOD bits affects the number of ways in which the *effective address* (EA) of an operand may be determined. Obviously, we must select only one of these candidates. Further, some (as in the last case) may be awkward to implement because of the constraints of STUPIDD's architecture.

The CPDL descriptions of the ADR type of macro imply that something is to be added to a register's contents. The alternate choices disclose that the source of the operand can be memory, T, or another register. One or all of these options could be used in creating STUPIDD's instruction set. Oversophistication of an instruction set to include all possibilities is costly. Unwarranted variability can also make the software difficult to interpret. Suppose that we want to add to any register we choose. One solution is to have the OP code bits of the instruction specify both the source of the operand and the destination register of the result. This is the way the PDP-11 does it. Of course, this machine has sixteen bits to play with and employs a clever encoding scheme of pointers to make this work.

INSTRUCTION SET SPECIFICATION PROBLEMS

In STUPIDD's 4-bit instruction word, this selection of a register in R would use up half the instruction bits available. The problem is that the remaining bits can specify fewer instructions. For small-word machines, there are always more desired instructions possible than can be crammed into a given size of instruction word. Some desirable addressing modes are often sacrificed in the process, particularly among the 8-bit microprocessors. Instruction set optimization and the provision of flexible addressing modes soon runs into the word-size limitations of the small microprocessors. The 2650 8-bit microprocessor sacrificed one bit of address space to obtain a 9-bit instruction word. The excellent instruction set of the 2650 microprocessor, as compared to others of its era, was limited to an address space of 32K bytes, as contrasted with the 64K bytes of address space the rest possessed. This was a high price to pay, indeed, even though its instruction set was far superior.

One solution is to adopt a restrictive and pragmatic compromise that implements an attractive–if not ideal–instruction set. Out of all the possibilities in Table 5-7, let us decide to use R0 as the implied destination of all ADD-type instructions. This situation treats R0 as a special accumulator. It does not require that any OP or MOD bits be used to specify the destination of the result. Investigating our options even further, we find that the bit patterns for the first field of the instruction word shown in Table 5-8 are possible.

Table 5-8
Bit Formats for ADR Type of Instruction

IR Bit Pattern	*CPDL Description*
MOD OP	
0 100	R0 <: M{OA} pl R0
0 101	R0 <: R0 pl R1
0 110	R0 <: R0 pl R2
0 111	R0 <: R0 pl T

If we were to adopt these conventions for STUPIDD, we would see in the first example in Table 5-8 that we can add any memory word specified in the operand address, or OA, portion of the instruction to R0 when the two rightmost bits of the instruction are 00. This means that, when the bit pattern, 0100, appears in the IR, the next memory word is not an OP but must be the OA of the instruction. This coded pattern, when in IR, informs the control system what it is being requested to do during the EX phase. In this case, you, the macro designer, should plan on adjusting the PC in the macro you are creating to direct the control system to perform the correct steps.

Note that the left two bits of the pattern (01XX) specify the ADR operation. If the right two bits are not 00, then Table 5-8 shows that the instruction is completely specified in one word. Inspection of the other examples of the table discloses that they are neither *memory reference* nor *immediate mode* instructions. In these cases, the PC will automatically point to the next OP after the instruction fetch is completed, and so it needs no further adjusting. The reason is that all source and destination fields must be implied in the definition of these single-memory-word types of instructions. That is, the bits of the first instruction-word field contains the code that the macro writer interprets as meaning the implied source and sink. The control system (later) only carries out the steps we provide it with in the macros we develop.

The incrementation of the PC by one during IF is as much as could done toward adjusting the PC in a generalized IF macro, short of designing a computer that has the ability to anticipate the length of an instruction word its control system has not yet seen. It is the microprogrammer's responsibility to make any further adjustments to PC, if necessary.

SELECTION OF ADD MACRO

Commercial 4-bit processors use more than one memory word to specify the complete instruction, so do not panic. STUPIDD is only a learning tool. Yet with it we can demonstrate the inner workings of most instruction types, even though it is not a large machine. For now, we have succeeded in introducing some new concepts: a single-memory-word instruction-word format, a two-memory-word or longer instruction-word format, and the fact that the PC may require adjusting during the EX phase of operation. These new concepts resurface as we proceed. Let us finally select, for microcode implementation, the following two-word instruction as an example of only one way we can microcode the operation of addition:

$$\text{ADD: R0} <: \text{R0 pl M}\{\text{OA}\}$$

The macro for addition, above, is the one we finally select for implementation. The second memory word of the complete instruction word specifies the address of the operand. The preceding discussion leading to this final choice reveals that the selection of an appropriate set of instructions and addressing modes–to be encoded into a fixed number of bits of an instruction word–is not a trivial task.

As noted, it is important to be very clear about the precise meaning of any instruction; these meanings are usually spelled out in symbolic *Register Transfer Notation* (RTN) form in the programming cards or in the literature provided with commercial processors. Only the final symbolic results are presented in these RTN descriptions, often without any clue as to how the microcode achieved it for a given architecture. In the above CPDL description of the specific instruction ADD that we are now studying, R0 is to receive the sum of its own contents and the contents of the memory location (specified in the OA field). Let us adopt this definition for now, as the precise statement of how STUPIDD performs addition. We now give it the official name of ADD. If we do not like this way of adding, all we have to do is create another macro for addition.

Experience has shown the need to focus attention on the process of questioning the accuracy of one's own interpretation of any new instruction. Obviously, it helps to be familiar with the nature and structure of the hardware system. If the days of the purely hardware or purely software types are not numbered, this age of microcomputers has at least restricted the use of their talents. There are many types of microcomputers in service, and we must work closely with them—particularly with the many programmable peripheral devices they now contain. We need to understand the hardware, nowadays, to create the software. The reference literature offered by microprocessor manufacturers usually presents a form of symbolic RTN to describe the net effect of each instruction in its set. The CPDL notation is used here as a form of RTN. Gaining some intuitive feel for how hardware functions in general is the intent of STUPIDD's design. In more complex systems, these RTN definitions of what an instruction actually achieves are the programmer's best aid to understanding what a machine actually does in executing an instruction, particularly if we can visualize the internal hardware organization of the system.

The symbolic coding of the addition algorithm, or macro, is presented in Figure 5-61. The chosen symbolic name of this algorithm was ADD. It specifies that the contents of the memory location specified in the OA field are to be added to R0. This differs from the generalities of Table 5-7, where we were first exploring the range of possibilities. The symbolic CPDL notation of the net result produced by this instruction is displayed under the name of the macro as its overall function. It is just this sort of concise summary of the instruction that one must carefully interpret and rely on when using a computer.

SYMBOLIC MICROCODE FOR ADD

The details of how the net results are obtained and the CPDL notation of each line of our ADD instruction reside in each line of the completed macro coding sheet of Figure 5-61. An interpretation of these lines follows:

> Line 0 loads both MAR and T with the contents of the PC. This means that MAR now contains the *address* of the *operand address.* Since this is a two-word instruction, recall that the PC was left pointing to the OA after IF. The memory maps sketched on the coding sheets can be used as a visual aid in this process.
>
> In line 1, the PC receives T pl 1, that is, it is incremented to advance it toward the next OP code. In this example, the PC now holds the address of the next OP field when this is done. This completes the adjustment of the PC required before IF begins again.

Note the similarity to the first two lines of the IF macro. We have thus properly adjusted the PC for the next IF cycle and are currently holding the *address* of the *operand address* in MAR. Carefully think these distinctions out now, or suffer later:

> Line 2 commences the execution proper of the instruction. Here we fetch the operand address and place it into MAR, so as to be able to fetch the operand itself later. The beginner's mistake is to confuse the operand address (or pointer) with the actual operand.
>
> In specifying line 3, the constraints of this architecture dictate that T receive the sum of the contents of R0 and M, as specified

LINE NO.	MACRO: ADD	NEXT ADDRESS	BRCH CTRL	CLR MOD	DBUS SOURCE FIELD	STATE	ALU FIELD		SINK CONTROL FIELD	
	PDL DESCRIPTION: R0 <: R0 pl M{OA}, FL <: S, C, EQ						ALU FUNC.	CS	R ARRAY	LOAD ENABLES
0	MAR,T <: PC			●	0	X	B	X	RPC	MAR, T
1	PC <: T pl 1				T	X	A pl 1	L	WPC	0
2	MAR <: M{MAR}				M	X	A	X	RX	MAR
3	T <: R0 pl M{MAR}, FL				M	X	A pl B	H	R0	T, FL
4	R0 <: T, >: IF				T	IF	A	X	W0	IR
5										
6										
7										
8										
9										
A										
B										
C										
D										
E										
F										
PDL NOTATIONS		CSTR2				CSTR1			CSTR0	

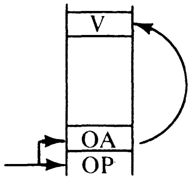


Figure 5-61 ADD Instruction Microcode

> by the OA, which now resides in MAR. Specifically, the constraint is that we cannot simultaneously read and write the same register in the array, R. This necessitates the intermediate use of T. Now we clearly see how a register may be used to hold intermediate results temporarily. Not only was the actual addition performed in line 3, but the flags are also loaded by the use of the mnemonic FL in the sink field. The loading of the flags means that information about the addition just performed–whether it generated a carry, its sign, or whatever–is available to a following instruction (usually, the next one) for decision-making via conditional-jump types of instructions. We shall learn more about what controls the state-paths of both software and hardware later. For now, note the recording of the flags at the operation, which is in essence the heart of the algorithm.
>
> Finally, in line 4, R0 receives the result of the addition from T, and the system is told to fetch the next instruction, thereby ending this macro. The major state transfer is initiated by the clock, since both LIR and IF are specified (at the bit level) by the semantics of this line.

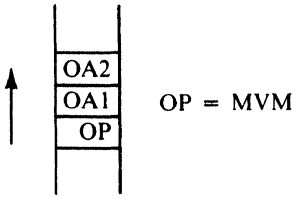
There is some mystery here, in that IR is loaded with an undefined value on line 4. As we shall soon see better, the answer is that this particular control system ignores IR during the IF phase. That is, IR is consulted by (and guides) the control system only in the EX phase. Since the IF sequence commences in the *next* period, the use of the LIR control signal now to also enable the loading of the IF flip-flop in the control system produces no ill effects. This eliminates the need for an extra control line, which is the reason for taking advantage of this possibility.

Now we have seen the complete basic cycle of operation of a processor: IF to EX and back again.

SYSTEM OPERATION SUMMARY OF MICROCODING

The preceding paragraphs may seem short after the lengthy introductory treatment, but that's all there is to it. Having seen one macro of each major phase of operation, we have basically been exposed to the nature of all of them, though some general comments are appropriate. Single-word instructions enter the EX phase with the PC already pointing at the next OP code–no adjustment of the PC is necessary. These types of macros do not utilize the microcode of lines 0 and 1 of the ADD algorithm to adjust the PC. They directly attack the execution of the algorithm, starting on line 0. Commercial 4-bit microprocessors typically possess an address space that is three memory words (12 bits) long. The address space of 8-bit microprocessors is generally two memory words (16 bits) long, while 16-bit systems now offer address spaces of 20 or more bits. Adjusting the PC for these classes of machines can be more complex but, via STUPIDD, we have gained insight into the methodologies and procedures that render these systems comprehensible. Since systems vary widely, it is how we think about them that is important, rather than how any single unit functions–up to the point we choose one to work with.

Figure 5-62
MVM Instruction Format



1. THE CPDL FORMAT IS M{OA2} <: M{OA1}.
2. THE MOD BIT IS IGNORED AND PRESUMED LOW.

CREATING NEW MACROS

MVM–A 2-Address Macro

The task before us now is to use this background creatively to invent and demonstrate our own instruction set for STUPIDD. For example, can this machine support a three-word instruction? Of course it can, if we want to design it. Let us invent a memory-to-memory data-movement instruction, which we call MVM. The first step is to visualize the format of the proposed instruction as it would appear in memory. This is presented in Figure 5-62.

The overall CPDL instruction description in Figure 5-62 specifies that the memory location, as specified in the OA2 instruction-word field, is to receive the contents of the memory location specified by the OA1 field of the instruction word. Further, a "good" instruction does not affect the general registers in the register array R when they are not involved in the instruction. We shall not use any register in R for MVM. As noted, we presume the MOD bit to be an L, for now. This is discussed, with address modification, in Chapter 6.

The next step is to visualize both the microlines of code required and the subarchitectures involved at each line. The goal of this is to find the shortest sequence of microcode that the architecture can support that will do the job. Figure 5-63 presents the completed macro for the MVM instruction on the symbolic coding form. The restriction that we apply now–and explain later–is that this macro must be confined to the first eight lines of the coding sheet. There is a hardware reason for this, as we shall see.

The first two lines of code are easy, for we have learned that the PC must be adjusted at least once. These are

0) MAR, T <: PC

1) PC <: T pl 1

This all looks familiar. MAR is currently holding the *address* of OA1, and PC now holds the *address* of OA2. Now comes the dilemma, for the PC needs to be adjusted still further. Its current contents must be used to obtain OA2. If we fetch OA1 and place it into MAR, then *temporarily* store the source data in T, we can surmount the problem. The next lines of code therefore read:

2) MAR <: M{MAR}

T may now receive the source data:

3) T <: M{MAR}

The flags could be loaded in the above line, if one wanted to collect information on the nature of the data being moved. Some machines do this. Since T is occupied with holding the piece of data to be moved, we cannot proceed in the previous manner of the PC incrementation. It can be included, however, in several steps:

CSOP Microcode	*Comments*
4) MAR <: PC	(MAR receives ADDRESS of OA2)
5) MAR <: M{MAR}	(MAR receives OA2)
6) T <: PC, M{MAR} <: T	(data to dest., the PC to T)
7) PC <: T pl 1, >: IF, LIR	(advance the PC, End)

These steps are a bit more involved, but they accomplish the task.

Line 6 is of special interest. Note how the architecture was used to perform a compatible set of operations. The T register and memory were simultaneously enabled for loading of fresh information. The current contents of T are safely stored into M, due to T's true edge-triggered behavior, before T's output changes. At the same time, T's inputs were enabled to accept new information during the T_s .. T_h interval, as discussed in chapters 2 and 3.

The example above was sophisticated. We should draw partial architectures graphing the flow of data, as in the previous section, to be sure it is understood. This can easily be done by making a few copies of the CPU architecture. The MVM instruction format is also an example of a *two-address* instruction format. A *source* and *destination* address are part of the instruction-word format. This type of instruction-word formatting is employed in the PDP-11 processors and others patterned after it, such as the 68000 microprocessors. These machines differ, however, in how they actually perform source and destination addressing. In them, encoded fields of the instruction word point to CPU registers through which all external addressing is accomplished. We have illustrated, though, that STUPIDD can demonstrate two-address machine principles.

In microprogrammed systems, all processor instructions are macros residing in the control-system store. We need to examine a few more to investigate the general spectrum of a processor's procedures, but this must come after the study of the operation of the control system begins. The last items deserving of mention at this point are the external environment and the instructions related to it. These are macros that cause the control system to emit signals at appropriate lines of microcode that affect the world beyond the CPU. This has already been demonstrated in the reading and writing of memory. STUPIDD presents too simplistic a picture of I/O operations, and this is an area deserving more complete coverage. Concocting simple input and output algorithms with the current architecture is included in the problem set at the end of this chapter.

LINE NO.	MACRO: MVM PDL DESCRIPTION: M{OA2} <: M{0A1} *(NOTES)	NEXT ADDRESS	BRCH CTRL	CLR MOD	DBUS SOURCE FIELD	STATE	ALU FIELD		SINK CONTROL FIELD	
							ALU FUNC.	CS	R ARRAY	LOAD ENABLES
0	T, MAR <: PC (MAR <: A0A1)				0	X	B	X	RPC	MAR, T
1	PC <: T pl 1 (PC <: A0A2)				T	X	A pl 1	L	WPC	0
2	MAR <: M{MAR} (MAR <: 0A1)				M	X	A	X	RX	MAR
3	T <: M{MAR} (T <: SOURCE DATA)				M	X	A	X	RX	T
4	MAR <: PC (MAR <: A0A2)				0	X	B	X	RPC	MAR
5	MAR <: M{MAR} (MAR <: 0A2)				M	X	A	X	RX	MAR
6	T <: PC, M{MAR}<: T				T	X	B	X	RPC	T
7	PC <: T pl 1, >: IF, LIR				T	IF	A pl 1	L	WPC	IR
8										
9	*NOTE: A0A(n) STANDS FOR									
A	ADDRESS OF OPERAND									
B	ADDRESS n									
C										
D										
E										
F										
	PDL NOTATIONS	CSTR2				CSTR1			CSTR0	

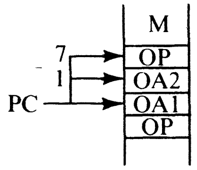


Figure 5-63 MVM Instruction Microcode

PROBLEMS

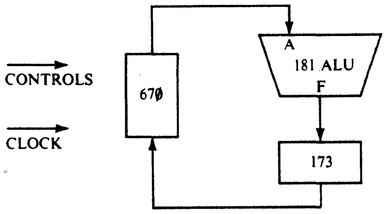


Figure 5-64
Closed Data Loop for Problem 2

Notes: Where microcoding is required in the following problems, first develop the microcode, then demonstrate the microcode's actions, using the control-system simulation switches.

Sketch the data-flow paths for each line of microcode, until you learn to visualize readily the data-flow path possibilities. This is especially helpful when dealing with compatible sets of operations in a line of microcode.

1. Obtain the functional descriptions for all the pins of two microprocessor IC's, say, the 8080 and the 68000. Organize the pin names into three groups: data bus, address-bus lines, and control-bus interface signals. For the logic lines of the control bus group, further categorize these pin names into functional subgroupings. For example, some pins are devoted to directional control of data transfers and others to bus access control or I/0 vs. memory operations, etc. The object is to form your own perspective of the minimal subset of the types of control functions performed that will encompass what the control bus signals do.
2. Obtain the timing diagrams for the 74LS670 register array and the timing parameters for the 74LS181 ALU and 74LS173 4-bit register. The ALU is to perform the A pl 1 function. For the closed circuit shown in Figure 5-64, using these elements, draw the combined timing relationships of the circuit that:
   a. Show that the propagation delays of the circuit permit the 173 to be loaded with its own modified output while the 670 is in the transparent state.
   b. Establish the maximum speed of operation for the Read, Modify, Write of a register within the 670. The clocks received by the 670 and 173 must be nonoverlapping.
3. Obtain the data sheets and timing diagrams for any of the 1 × 64 K-bit dynamic memory IC's used in a microcomputer.
   a. Discuss the meaning and timing relationships of the RAS and CAS signals.
   b. Explain what must be done to refresh this IC.
4. Obtain the equivalent logic diagram of the 74LS194 shift register from the TTL data catalog. Redraw the gating and flip-flop for one of its typical memory cells. Making four copies of this circuit will help in the following tasks. With reference to this logic diagram, explain its operation for:
   a. Parallel loading
   b. Shift right
   c. Shift left,
   d. Do nothing
5. Use the electrical-level microcoding sheet for this problem. Develop the microcode for the instruction that will shift left R0 while filling the vacated LSB position with a 1. Call it SHL1. The carry-out is stored in the FL register.

6. Use the symbolic coding form for the following problem. Invent an algorithm for STUPIDD that *compares* the contents of R0 and the contents of memory specified in the OA field of the instruction. Only the flags are to be set in its operation. Name it CMP. If you are not sure of what a compare-type instruction does, research this operation in the instruction set of the PDP-11 minicomputer or any of the microprocessors.
7. Use the electrical-level (H & L) coding form for this problem. The instruction to be microcoded is DEC, R2. The PDL net result description is

   R2 <: R2 mi 1, LFL

8. Use the symbolic coding form for this problem. Microcode the instruction XOR, M. The PDL description is

   R0 <: R0 ⊕ M{OA}

9. Develop the microcode for the instruction rotate left through carry (RLC) using the electrical-level (H or L) coding form. The instruction operates as follows, all events occurring at the same time:

   CF <: R0[3]
   R0[i] <: R0[i − 1], i = 3,2,1
   R0[0] <: CF

10. Develop and demonstrate a short program that adds two values in memory and outputs the sum to the OUT register.
11. Write the symbolic microcode for the instruction that POPs data off the stack. Do the same to create the instruction that PUSHes data onto the stack. Use R0 as the implied register through which all stack-data transactions occur.
12. Create a macro for the add with carry (ADC) instruction that is consistent with the add instruction in the text. Discuss its use in multiple-precision addition. Write a program that uses these instructions to add two 8-bit quantities.
13. Create a macro for the move immediate (MVI, R2) instruction. In this instruction, a data value is embedded in the program stream. That is, the memory word following the MOD, OP field is a data value. The CPDL description is

   R2 <: M{PC} pl 1

   where the contents of the PC referred to above are the address of the first memory word of the instruction.

Chapter 6
The Control System

Control System Overview

A block diagram of the control system of STUPIDD is presented in Figure 6-1. A few compromises have been made to keep this figure simple yet adequate for the purpose of providing insight into systems operation. These compromises are noted in the following overview of the system.

Control Store EPROM's

The control store (CSTR) is at the heart of the system. It consists of three 2716 *Erasable Programmable Read Only Memories* (EPROM's). These devices are 2K × 8-bit read only memories whose contents may be erased with the aid of special ultraviolet lamps. Any other memory device would serve as well, but, since these can be erased and reprogrammed, they are popular for development work. After a design is frozen, EPROM's may be replaced with the permanent, lower cost, production type of ROM's. Both erasure and programming of this device require special equipment almost universally available in industry and schools, but not necessarily in the home. Therefore, it may be difficult for an individual to program the control system, without ready access to the resources needed. Computer clubs and advanced home hobbyists often possess this gear. Economical EPROM programmers are available for many personal computers. In fact, the Computer Science Laboratory at California State Polytechnic University, Pomona, uses an IBM PC, an Advanced Microsystems PROM programmer, and UVP's EPROM eraser for this phase of the work. The construction of the control system therefore is strongly recommended. As an interim measure, we can always use the simulation switches to make the CPU respond to the control system's directives, once we understand how it operates.

What the CSTR produces is a 24-bit control word, most of which we can simulate with the switches of our primitive CPU, as detailed in the previous sections. Should we construct the control store, its output—the control word—simply replaces the actions of these switches on the CPU, which either must be removed or put in the disabled (open) position while we are using CSTR. Thus, we can demonstrate complete CPU system coordination either automatically or by manually setting switches. Explanation and understanding of the control-system workings, though, are essential to our progress. If this is followed by a simulation of CSTR's sequence of outputs by using the switches to control the CPU, then we will have demonstrated a knowledge of what it achieves, and how it does this.

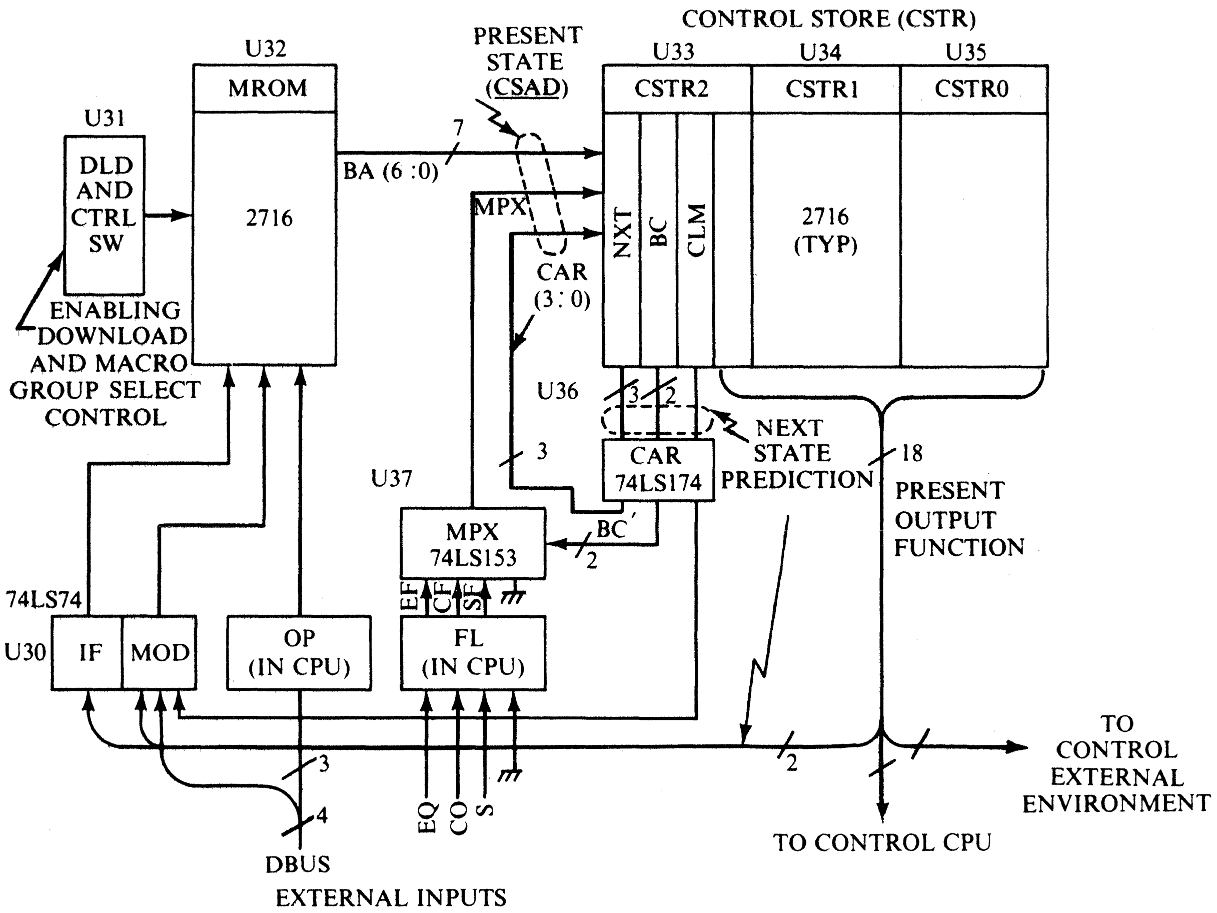


Figure 6-1
STUPIDD V Control-System Block Diagram

STATE-TABLE ASPECTS OF CONTROL STORE

Present-State Identification

Figure 6-1 is organized so as to once again emphasize the sequential machine aspects of the control system. It is an extension of the basic microprogrammable von Neumann architecture introduced in Chapter 1. The control store is a memory that contains the elements of a state table for a sequential machine. These are—as a function of the present state and external input sets—the next state prediction and the present output function. These sets of entries are the contents of the CSTR memory. *The present state of the system is the set of control store address (CSAD) lines* addressing the EPROM's of the control store. This must be so, since whatever the machine is to do emanates from CSTR. These CSAD inputs are the only specifiers of where we are in CSTR, or the indicators of the present state. This address is affected by the external inputs, which are signals from the CPU, plus the MOD and IF flip-flops. The net result of this addressing procedure is to bring out of the CSTR memory a group of bits that we call a *line of microcode.* The first six of these plus the ST field and LIR are *next state* predictors. Thus we have shown the presence of the elements of the 5-tuple that describe a sequential machine, namely <I, Q, Z, *n*, *p*> of Chapter 4.

Next-State Prediction

We postponed an examination of the leftmost six columns of the activity-level microcoding forms, Figure 5-52, until now. The first three columns, which make up the NEXT address field of the coding sheets, are devoted to unconditional selection of the lower three bits of the *next state prediction.* This prediction is presented to the inputs of the current address register (CAR). After each clock pulse has loaded CAR, these three inputs become a part of the *current* state, which is the first three bits of the control store address (CSAD)–when the signals settle down in the next period. The total CSAD address also consists of MPX and BA, as detailed below. Clearly, the NEXT field of CSTR predicts a portion of the next state. It is important to realize that these three bits select one of eight possible lines of microcode. This is why we restricted the MVM macro of the previous section to eight lines. Yet, a single macro may be made to possess up to 16 lines of microcode, as we shall now discuss.

Branch Control

The two columns following NEXT are used for the branch control (BC) field. Their use can lead to the production of the signal that controls access to the other eight lines of microcode and is also used in the execution of conditional operations. These conditional operations are operations wherein the line of microcode selected for execution depends on the logical condition of an external input selected by the multiplexer driven by the BC field. These two lines are also loaded into a portion of CAR where–in the next period–they address a 4-line-to-1-line multiplexer, a 74LS153 called MPX.

Pipeline Operations

The function of this device is to select the source of the fourth bit of the CSAD,the current address of CSTR. This selected bit may come from one of several external sources, as controlled by the BC field. As we can see, though, we must anticipate and specify its use in the clock period *before* it is to take effect. This present specification of a future event in microcode is sometimes referred to as *pipelining* an architecture. Where it is used, we must learn to think ahead and generate *now* the proper signals that will take effect in the *next* or some future clock period.

Flag Selection

STUPIDD was limited in size to keep the chip count low. Previous 8-bit microprocessor versions of the STUPIDD projects required in excess of fifty IC's. We have compromised here by feeding an L and only three flags to the inputs of MPX. Thus, the BC field of CSTR can control the binary value of the *fourth* bit of CSAD (the present state) with only these four sources to MPX. It performs this by selecting either an absolute value of L or by permitting its value to be dictated by the current value of the one of the three flags from FL we choose to utilize. This permits both absolute and conditional control of this bit. It would be nice if we could also select an H or some additional flags or could have more absolute control bits to provide greater flexibility.

MPX Upper/Lower Half Selection

Note carefully that the current BC field from CSTR affects the MPX in the next clock period. Again, one has to think ahead in developing the microcode, which is usual in a so-called *pipelined* machine. This is how the first four bits of the next state are predicted, or formed, by the present state of CSTR. The 4-bit range of this portion of CSAD limits the maximum size of each individual macro to sixteen lines. Use of the BC field can lead to the conditional selection of CSTR addresses in either the lower or upper *half* of the block of sixteen lines of microcode allocated to each macro. The NEXT field leads to the specific selection of one of the eight lines *within* the half of the macro-block space determined by the BC field.

Block Addresses

All the other CSAD address lines are devoted to the selection of the *block address* (BA) of the current macro that is to be executed. These BA lines come from the *mapping ROM,* MROM, which we shall discuss shortly. These BA values are held constant during the execution of a macro and change only when we want to perform another macro. Therefore, the BA changes whenever the system goes from the IF to the EX major state, or back. Some of the first four lines of CSAD typically change with each clock pulse, to select the current line of microcode to be performed within a macro. The high-order bits of CSAD–that is, the BA–only change when *another macro* is in the process of being selected. The concurrent specification of an ST value and LIR initiates this change. Again, the BA remains constant during a macro's execution. Larger commercial systems operate on related principles. The major difference is that, in utilizing both absolute and conditional control of next state predictions, these systems are designed to make both more efficient and more flexible use of CSTR memory space, through the use of more bits than our small system contains.

Clearing of MOD Bit

Last of these previously unexplained six bits is the *clear MOD* bit, CLM. This bit is also loaded into CAR, where it can clear the MOD flip-flop after the address modification macro has been performed. To employ it effectively, one must look ahead to anticipate its use, as in the previous cases. None of these six signals affects the current state, just the next one after the current clock period has run its course, when they appear at CAR's outputs. Let us emphasize again that their use must be anticipated one clock pulse before they are to take effect–due to CAR's single-period pipeline delay.

Present Output Signal Feedback

The balance of the control word emitted by CSTR is sent to the CPU and its external environment, except for the signals that affect the loading of the IF and MOD bit flip-flops, LIR and ST. These signals could have been implemented in other ways, such as also incorporating the IF and MOD bits into CSTR and CAR, but at the expense of requiring a larger system. They are, in reality, a part of the state machine whose entire state table could be held in CSTR. In particular, IF is displayed apart from CSTR, both for simplicity and to emphasize the two major states of the cycle of computation. The output of IF drives the *most significant bit* (MSB) of the address lines of the EPROM labeled MROM. In this implementation, as the design shows, IF is loaded only when LIR is asserted. Its next state is predicted by the present value of the ST field of the microcode. Therefore, the portion of the total next state prediction emanating from CSTR is determined by NXT, BC, CLM, LIR, and ST. The next state of a sequential circuit of the Mealy or Moore type is also a function of external inputs, as we shall now discuss for the flags, MOD and OP inputs, next.

MOD Flip-Flop and IR

The flip-flop labeled MOD is a part of the IR. It gives us the capability to simulate the use of MOD–bits in instruction words that specify operand address modification. Instruction register contents generally consist of two fields, the MOD and OP fields. The MOD field specifies the type of operand-addressing mode to be used during the execution of an instruction; the OP field specifies what operation is to be performed on the operand after it has been fetched. The MOD flip-flop is set at the end of the IF macro by instructions in which address modification is requested. Thus, the execution of the events that occur when either *indexed* or *indirect* addressing is specified can be displayed on this system. The MOD bit flip-flop output is fed to an input line of the EPROM named MROM, where it becomes the *second* most significant bit of the address space of MROM used in the normal operation of STUPIDD. The order in which MROM is addressed by IF and MOD is important, as we shall soon see.

The rest of IR, which is the OP field in this system, drives the *low-order* address bits of MROM. It contains the code that specifies the current instruction to be performed during the EX phase of the cycle of computation. Both this OP register and the MOD bit are valid only during the EX phase of operation. The IF flip-flop specifies whether the system is in the EX or IF phase. We can see that the inputs to IR are truly external inputs, having come from the external memory, M, where the program stream resides. In point of origin, OP and MOD were generated outside the control system hardware. Globally speaking, a programmer actually created them.

Mapping ROM

MROM Space Allocation for a Macro Set

MROM, another 2716 EPROM, serves as a mapping read only memory. Mapping ROM's are frequently applied to the design of processor control systems, though not necessarily in the simplistic manner used here. Their function is straightforward. It is to map the input fields (such as our IF, MOD, and OP fields) into a block address (BA) for accessing a macro in the CSTR. It is important to note that the output of MROM selects a block address for CSTR, within which block the control system is currently operating. The NXT and BC fields from CSTR that are currently stored in CAR, on the other hand, select the specific line of microcode within this block that is currently active. MROM therefore maps its inputs into block addresses that appear at its outputs. It is emphasized that *each* block address selects a different macro or algorithm for STUPIDD to perform. Only five of MROM's input address lines are used in demonstrating the operation of a set of macros. Two other address lines are switch-selectable to increase to four the number of macro sets that can be demonstrated. In addition, there are three address bits of MROM that are used to download one of eight possible demonstration programs stored in CSTR, for ease of operation. These refinements are discussed in the next section.

The manner of MROM's use in STUPIDD is wasteful of memory space. The system was kept simple. It does, however, have its elegant aspects. MROM can be made to serve as a priority encoder, while selecting block addresses. This is a concept that needs to be fully understood to use this system. Figure 6-2 illustrates how the memory address space may be allocated in *one* of the four possible sets of macros that can reside in MROM. Several sets of macros may exist and be demonstrated, but only one set at a time is selected for use with a demonstration program. The concepts of the mapping operation, then, can be explained by examining the mapping of a typical set. This pattern is the same for all other sets of macros.

MROM Priority Coded Operation

Note, in Figure 6-2, that IF is the most significant bit of the input address lines for one set of macros. If IF is high, then the location selected must be somewhere in the *upper half* of MROM's macro set address space for this set, regardless of the current value of the other less significant address lines. If the programmer of the EPROM chooses to write the same value into *all* locations of this upper half of the address space used by this set of macros, then IF's being high always causes a *constant block address* to be emitted, regardless of the other inputs. Thus, we program IF to have priority over *all* the other inputs of MROM. The block address emitted from MROM when IF is high is the starting address of the block location of the IF macro. This is sent to the high-order address lines of CSTR. Recall again that CAR and BC are selecting a specific current line of microcode *within* that block. Therefore, making the output of the IF flip-flop the MSB input to MROM for a set of macros gives it *priority* over all other inputs—when MROM is programmed in the manner described above.

At the end of the IF macro, the IF flip-flop is set low by our concurrent use of the *state* and LIR signals. Now all that the other inputs can do is select a location in the lower half of MROM's address space. When the MOD bit is high, the location selected must fall into the *second quarter* of MROM's macro set address space. It is to be hoped that the programmer of MROM has written a single unique value into all these locations, too. This then produces the block address for the operand address-modification algorithm that the system must perform, before actually obtaining the

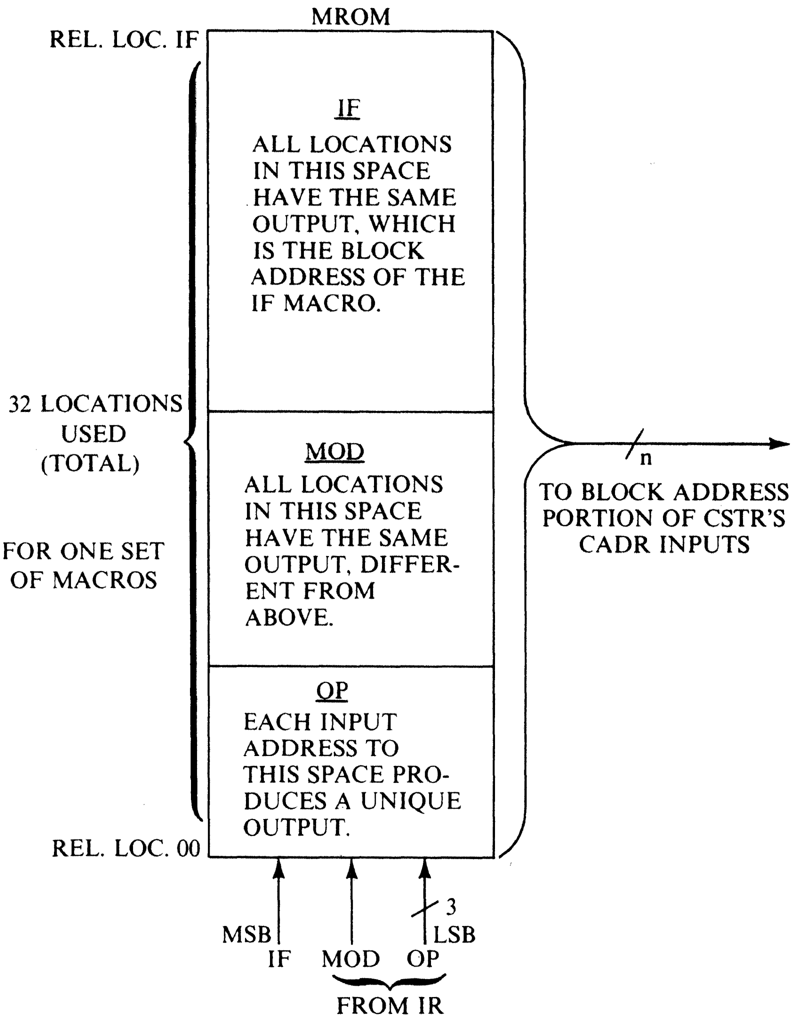

MROM
REL. LOC. IF
IF
ALL LOCATIONS IN THIS SPACE HAVE THE SAME OUTPUT, WHICH IS THE BLOCK ADDRESS OF THE IF MACRO.
32 LOCATIONS USED (TOTAL)
FOR ONE SET OF MACROS
MOD
ALL LOCATIONS IN THIS SPACE HAVE THE SAME OUTPUT, DIFFERENT FROM ABOVE.
OP
EACH INPUT ADDRESS TO THIS SPACE PRODUCES A UNIQUE OUTPUT.
REL. LOC. 00
n
TO BLOCK ADDRESS PORTION OF CSTR'S CADR INPUTS
3
MSB
LSB
IF
MOD
OP
FROM IR

Figure 6-2
Priority-Encoded MROM-Space Allocation (for One Set of Macros)

operand and executing the operation specified upon it in OP. At the end of the address-modification procedure, the MOD bit is set low. With both IF and MOD low, the remaining inputs are capable only of selecting a location in the *lowest quarter* of MROM's macro set address space. Where more than one MOD bit is present, this priority encoding scheme may be employed again and again, as long as MROM has a sufficiently large address space.

When both the IF and MOD bits are low, only then is the OP field permitted to have its say. Now the programmer of the EPROM resorts to a new tactic. Each OP selects a location in MROM into which separate and usually unique block addresses are programmed. We are no longer priority encoding as well as mapping—we are performing straightforward mapping. That is how the system functions, regarding the selection of the BA's within each set of macros of the system. CAR dictates, within these macro block-space bounds, precisely where the current control word is to come from. CAR, then, selects the lines of microcode of each macro that we originally created on the coding-form sheets.

The addresses shown in Figure 6-2 are relative to a base address, as controlled by the macro-select switches provided in the control system. There are four sets of different macros possible, each one containing eight OP codes and one address-modification macro. The user can now invent and demonstrate a wide variety of instructions with the construction project.

This wasteful but effective use of MROM's address space is economically justified only for educational purposes. Actual implementations of CSTR addressing use larger CAR's, typically with expanded control over the high- and low-order bits. The low-order bit is sometimes used to multiplex in an external value for control of looping, while the high-order bit(s) is employed in branch or jump operations within CSTR.

IF, MOD Flip-Flops

Both IF and MOD are implemented using a 74LS74 dual flip-flop. When LIR is active, that is, when the instruction register is being loaded, both these flip-flops are also loaded. To repeat, the use of LIR to enable the loading of IF is convenient, as it eliminates the need for a separate control signal. Also recall that, when LIR is active at the end of the EX phase, the loading of IR has no effect on the IF macro, because of IF's priority over all other lower order MROM-input bits, as discussed above. The IF flip-flop's data-input bit comes directly from CSTR as the ST signal. When this is high and LIR is active, the next state will be the start of the IF phase. At the end of IF, ST must be low to advance to the EX phase. The last line to be executed (not necessarily the highest-numbered line) in every macro *must* cause a major state transition to whatever is the opposite phase, IF or EX. The only exception is the address-modification type of macro, which employs the MOD flip-flop.

The MOD flip-flop is part of what we call the IR register. When the IR is loaded at the end of IF, the logic value of D3 of the data bus is entered into MOD. This is the MSB of our instruction word and is used to signify to the control system that operand-address modification is to be performed, as in indexed or indirect addressing of operands. Within the address-modification algorithm—on the last line—MOD is specified to be cleared via the direct clear input at the *start* of the next period. This is accomplished by specifying an L in the CLR MOD field. Again, the one-period delay is due to the fact that this value must appear at the output of CAR at the start of the next period. Within the address-modification macro, excepting the last line, the CLR MOD field must always be high. It must also be specified as H at the end of IF. Anyplace else, this bit's value has no effect on system performance. Why is this so?

The enabling of the control system and the procedure for downloading stored programs are presented, along with the constuction details, in the next section. This completes the control-system overview. In a future section, we look at the details of macros that include these control-system fields. The study of these instruction macros should clear up the remaining questions about the operation of the control system and its relationship to microcoding. Two topics relevant to control-system design—control sequencers and pipelining—are beyond STUPIDD's scope. Nonetheless, this coverage is reasonably comprehensive for the addition of only eight more IC's to those of the CPU. Build it, and enjoy debugging it. The construction details, in the form of the parts list, associated wire lists, and assorted suggestions, were presented in Chapter 5, along with those of the CPU.

THE CONTROL-SYSTEM DEVICE DETAILS AND MEMORY MAPS

This section contains the memory maps and logic schematics of the individual IC's used in the construction of the control system. The fundamental system operational features of the IC's in the control system are also introduced. This is the same type of interface presentation seen in our previous discussion of the CPU portion of STUPIDD in Chapter 5. For convenience, the parts list and wire lists both for the control system and for the CPU are all grouped together at the end of Chapter 5 (see Figures 5-15 to 5-51). Construction and debugging guides are explained there, too. If any of the new IC's introduced here are not familiar, the reader should study their data sheets. Because the 2716 EPROM is likely to be a new acquaintance, not found in a TTL data catalog, its data sheets are presented later in this chapter, in Figure 6-11. For the utmost ease and optimal use of the construction project, we want to program and erase these EPROM's. We shall discuss the programming environment later. All other IC's are found in the TTL data catalog. Since the reader should by now have acquired the skills for independently studying an IC data sheet, we shall now turn to a discussion of the control-system design details that are of interest at this point.

The control system has been designed to enable the user to operate the system automatically. Manipulating the individual switches that are provided for manual operation of the CPU can soon grow tedious. The automatic features include both the downloading of demonstration programs and their subsequent operation. This means that everything required for demonstrating both the development of macros and their subsequent use in demonstration programs can be conveniently preprogrammed into the EPROM's of MROM and the CSTR. Operation and the confirmation of correctness are now reduced to the manual operation of the clock push button. If one chooses to drive the clock input with a commercial TTL-compatible clock-signal generator, STUPIDD's performance may be tracked on an oscilloscope. Operation speeds in excess of 1 MHz have been observed. The construction of the control system makes the use and demonstration of STUPIDD very convenient.

ENABLING OF CONTROL STORE

The convenience features for the operation of the control system are represented in the DIP-switch connections shown in Figure 6-3. These switches are located in the socket for U31; the corresponding wire list and associated schematic sheet, in Chapter 5, should be consulted, too. At the very top of Figure 6-3 is the switch that generates the level of the logic line @ECTRL–*enable control.* If this switch is open (H), all of the EPROM's in MROM and CSTR are not selected and their outputs are tri-stated–they are therefore inoperable. Under this condition, one may freely use the control-store simulation switches, which are part of the CPU, to operate STUPIDD. *All these manual CPU-control switches should be opened or removed before closing @ECTRL.* If these switches are removed, the input port (IN) may be operated by using a four-position DIP switch in the U12 socket, properly located. This removal precaution is necessary to prevent conflicts between the CPU's simulation switches and the outputs of the EPROM's. With two devices attempting to drive a line simultaneously to conflicting logic

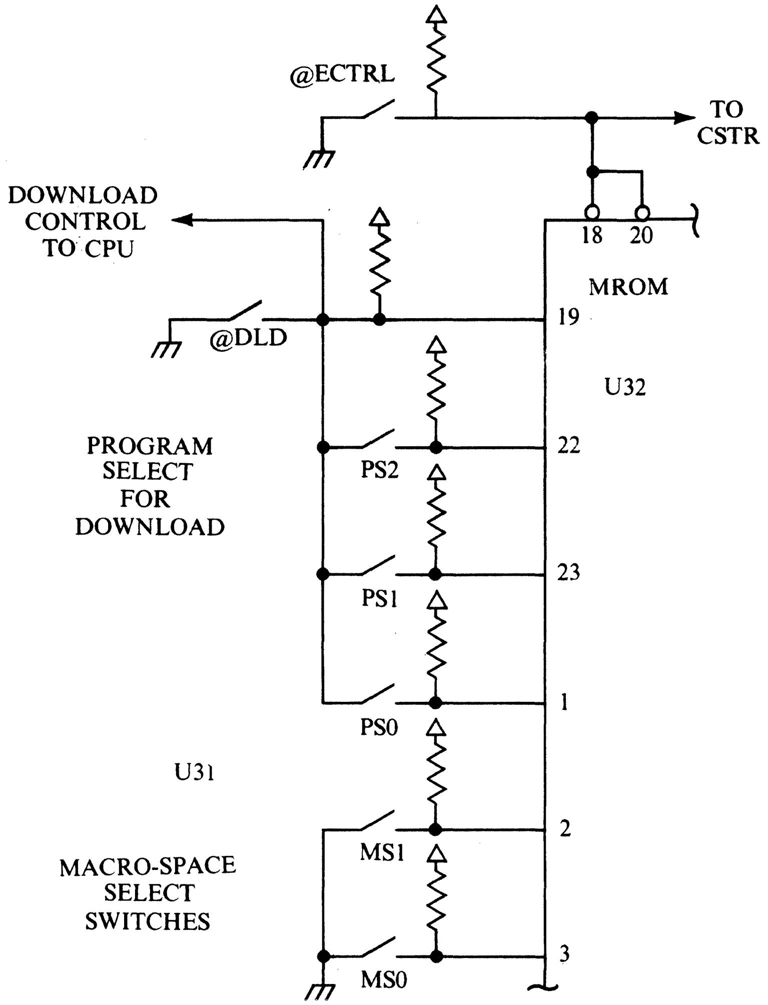


Figure 6-3
Download and Macro-Space Selection Switches

levels, a resulting voltage that is neither a logic 1 nor a logic 0 may exist. Logic ½s do not compute. When @ECTRL is active (L), the control system takes over. With the switches removed, the master-reset signal may be applied with the use of a short jumper wire at the position the MR switch occupies.

DOWNLOADING OPERATION

The next switch below this is @DLD, *download control.* A partial schematic of how this affects both system operation and program download is presented in Figure 6-4. When active, this logic line disables MAR (U5) and DEC0 (U8) in the CPU. This causes all CPU interfaces to both the DBUS and the ABUS to be tri-stated, to facilitate the following DMA-type of operations. This is achieved with the help of the 74LS241 buffer, U38. One half of this buffer (see the logic schematics of both Figure 6-4 and Figure 6-13) is active-high enabled; the four drivers of the other half are active-low enabled. When @DLD is high, the half of the buffer that is enabled activates a buffer gate whose input is tied low. The output of this gate is therefore low, when enabled. This is the logic line @EMD that enables both MAR and half of DEC0. Should @DLD be low, this part of U38 is disabled, and @EMD is pulled high

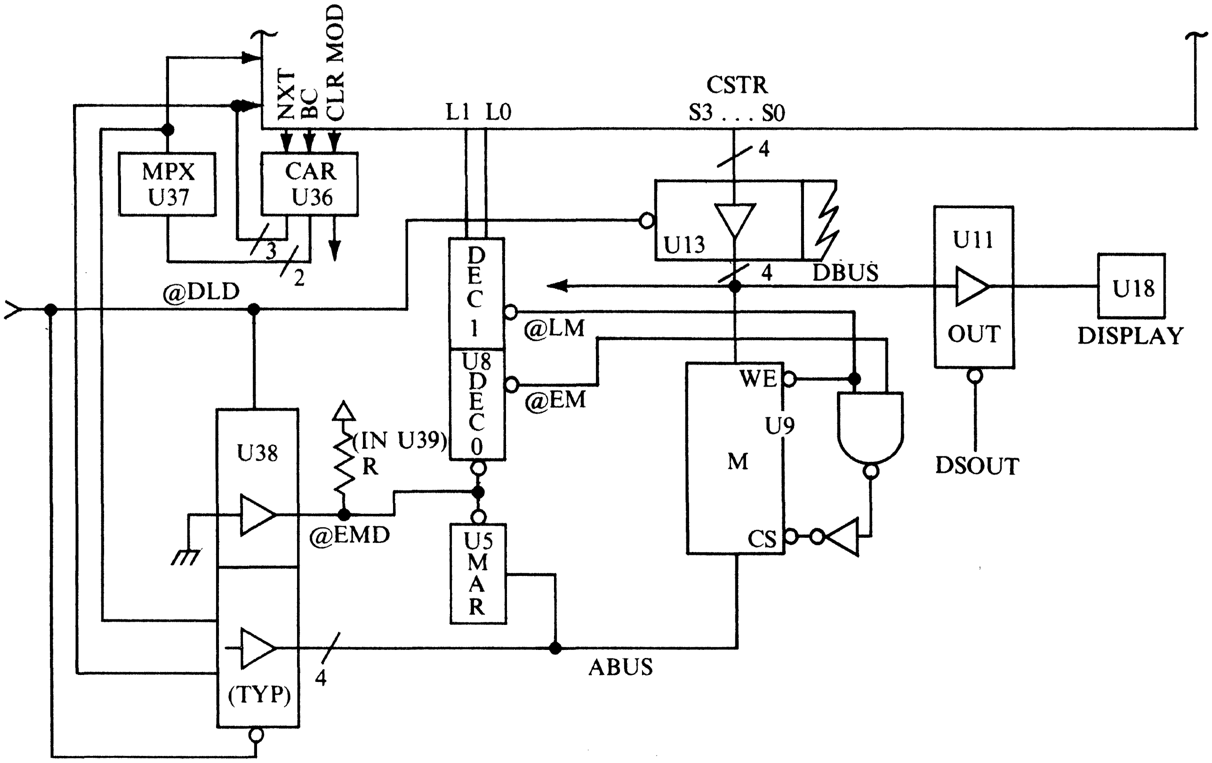


Figure 6-4
Download Control: Partial Schematic

by a pull-up resistor in U39. The active-low half of buffer U38, also controlled by @DLD, enables the four CSAD adress bits of CAR's (U36) and MPX's (U37) outputs to be placed onto the ABUS. Thus, memory is being directly addressed by something other than the CPU's MAR register. The data that accompanies these addresses comes from the S3 . . S0 field of CSTR, during downloading.

The four S3 . . S0 signals out of CSTR are presented to half of the 74LS244 buffer (U13) in the CPU. This buffer is also directly controlled by @DLD, and it becomes the master of the DBUS when activated. The data now being presented to primary memory comes not from the CPU but from an external DMA type of device. Since DEC0 is disabled during a download, no conflicts for bus access can occur. The S3 . . S0 field of CSTR, which will be programmed to contain the execution program's steps and data for future demonstration of its operation, may now be downloaded into memory in a coordinated manner.

Under the conditions above, both the main memory address and the line address within a CSTR macro block are the same. Both come from CAR's outputs. For each clock pulse, these addresses are controllable through use of the NEXT and BC fields of CSTR. Thus entries you place into CSTR's S3 . . S0 portion of the ALU field for a demonstration program may be systematically written into RAM memory from CSTR, if we also remember to specify a memory write at the same time. The administration of a clock pulse accomplishes this. The NEXT and BC field entries in CSTR, feeding the inputs of CAR, determine the next address for memory as well as CSTR. Due to the one-period delay of CAR, these take effect just after the application of the clock pulse.

In summary, so far we can program a macro block in CSTR so that it contains an operational demonstration program for future downloading and execution. These downloading operations are analogous to what actually occurs during DMA operations on a microprocessor system. If the downloaded program loops back into itself, sustained continuous operation is observable. The fields of CSTR to be programmed in this process are NEXT, BC, the S3 .. S0 field of the ALU, and the memory-loading signal LM. In addition to this, we are required to initialize the PC and the carry flag in the downloading procedure steps to be described later.

PROGRAM SELECTION

The switches in socket U31 are also used to control the locations of the block addresses that MROM emits. The three switches–PS0 .. PS2–below @DLD, as shown in Figure 6-3, are used for *program selection.* Eight demonstration programs may be placed in CSTR. On the other hand, the bottom two switches–MS1 and MS0–permit four separate *sets* of macros to be programmed into CSTR. MROM still only emits BA's. They are the BA's of all macros plus stored programs. Let us investigate how this occurs. The reasoning is similar to that for the priority-encoding use of the IF signal, previously described. @DLD, when low, controls downloading. Since it is physically the most significant bit of MROM, the eight stored demonstration program BA's actually reside in the lower half of MROM. This half of MROM is subdivided into eight blocks by the three PS switches. Utilizing the priority-encoding ability of MROM, introduced in the previous section, the remaining seven address inputs to MROM have no effect if all of the 128 locations they serve within a program select block contain the same BA. The BA's emitted by MROM will be programmed to be different for each setting of the PS0 .. PS2 switches.

MROM MEMORY MAP

The final resulting memory-use mapping of MROM is presented in Figure 6-5. When @DLD is inactive (high), this figure shows that the BA's of the four switch-selectable sets of macros all reside at the very top of MROM in groups of 32 bytes each. Figure 6-3 also discloses that, in the operate mode, where @DLD is high, the PS0, PS1, and PS2 signals addressing MROM are also pulled high, regardless of their DIP-switch settings–because of how these connections are wired. With the upper four address lines clamped high, we are now in the upper eighth of MROM. This space is further subdivided into groups of 32 bytes by the MS1 and MS0 macro-set select switches. At this point, the addressing behavior of MROM is as illustrated in Figure 6-2 and its related text, except that we can program into MROM the BA's of up to four separate sets of macros. The BA's emitted for each macro set therefore come from the upper part of MROM.

Block Addresses

There is a new possible variation on how one may use the 32-byte macro-set BA space in MROM, as indicated in Figure 6-5. If we do not wish to include an address-modification algorithm within any one set of macros and we choose to use the MOD bit as the most significant bit of the instruction, we can use the lower 16 bytes of the macro-

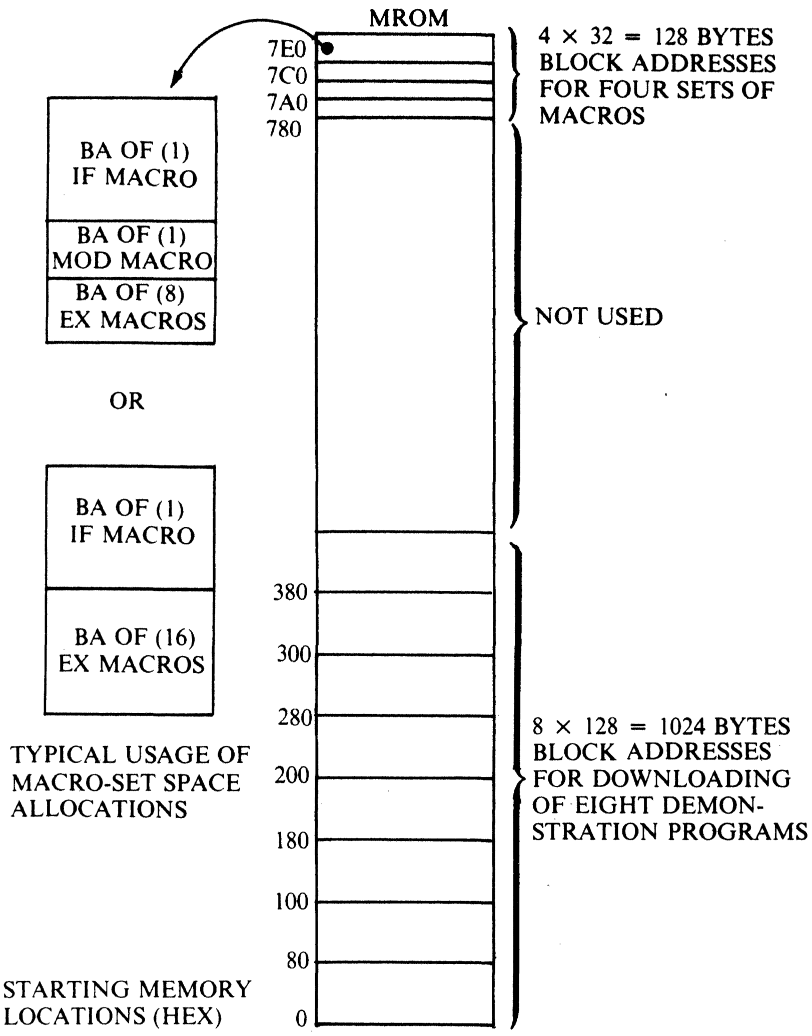

MROM
7E0
7C0
7A0
780
4 × 32 = 128 BYTES BLOCK ADDRESSES FOR FOUR SETS OF MACROS
BA OF (1) IF MACRO
BA OF (1) MOD MACRO
BA OF (8) EX MACROS
OR
BA OF (1) IF MACRO
BA OF (16) EX MACROS
NOT USED
380
300
280
200
180
100
80
0
TYPICAL USAGE OF MACRO-SET SPACE ALLOCATIONS
8 × 128 = 1024 BYTES BLOCK ADDRESSES FOR DOWNLOADING OF EIGHT DEMON-STRATION PROGRAMS
STARTING MEMORY LOCATIONS (HEX)

Figure 6-5
MROM-Use Map

set space to create BA's for 16 EX state macros. To do this, however, the CLR MOD bit in CSTR must be used carefully. Can you explain how it should be used? All sets of macro mappings in MROM must include a priority-encoded BA for the IF macro, but the BA assigned may be the same for all four. This is so because there need be only one IF procedure, shared by all sets of macros.

Block Address Assignment

The BA's emitted by MROM form the *most significant bits* (MSB's) of CSTR's address (present state). Figures 6-6 and 6-7 provide assistance in keeping track of the BA assignments in CSTR. Figure 6-6 contains the suggested BA assignments for the IF, X (indexed addressing), and I (indirect addressing) macros. As noted in the figures, the high-order hexadecimal digits displayed in the *HEX Starting Address* column are also the BA that MROM must be programmed to emit when a macro is assigned to a given address. Figure 6-7 is a generalized assignment worksheet for those who like to do it their way. Since there is space in CSTR for 128 macros and a maximum of only about 64 BA's may be emitted from MROM, there is room to spare in CSTR.

Figure 6-6a
CSTR Macro-Assignment
Map Form (Numbered)

	HEX STARTING ADDRESS	MACRO MNEMONIC	MACRO DESCRIPTION
MACRO SET 0	00	IF	INSTRUCTION FETCH
	10		
	20		
	30		
	40		
	50		
	60		
	70		
	80		
	90	X	INDEXED ADDRESS MODIFICATION
MACRO SET 1	A0		
	B0		
	C0		
	D0		
	E0		
	F0		
	100		
	110		
	120	I	INDIRECT ADDRESS MODIFICATION
	130		
	140		
	150		
	160		
	170		
	180		
	190		
	1A0		
	1B0		
	1C0		
	1D0		
	1E0		
	1F0		
	200		
	210		
	220		
	230		
	240		
	250		
	260		
	270		

NOTE: HIGH ORDER HEX DIGITS ARE BA FROM MROM.

Figure 6-6b
CSTR Macro-Assignment
Map Form (Unnumbered)

HEX STARTING ADDRESS	MACRO MNEMONIC	MACRO DESCRIPTION

NOTE: HIGH ORDER HEX DIGITS ARE BA FROM MROM.

CSTR ADDRESS			ASSIGNED OP-CODE							
LINE NO.	MACRO: DLD PDL DESCRIPTION: DOWNLOAD MACRO FIELD DESCRIPTIONS	NEXT ADDRESS	BRCH CTRL	CLR MOD	DBUS SOURCE FIELD	STATE	ALU FIELD		SINK CONTROL FIELD	
							ALU FUNC.	CS	R ARRAY	LOAD ENABLES
0	PC <:1, LFL	1	L	L	X	X	A pl 1	L	WPC	FL
1		2					↑	X	R0	M
2		3								
3		4								
4		5					YOUR PROGRAM			
5		6					AND DATA			
6		7	↓				IN THE S3 . . S0 FIELD			
7	SET TO MPX IN CF ON THIS LINE.	8	CF							
8		9								
9		A								
A		B								
B		C								
C		D								
D		E								
E		F	↓							
F	GO TO LINE 0.	0	L	↓	↓	↓	↓	↓	↓	↓
PDL NOTATIONS		CSTR2				CSTR1			CSTR0	

Figure 6-7 Microprogram Symbolic Coding Forms for Download Programs

NOTE: X = DON'T CARE.

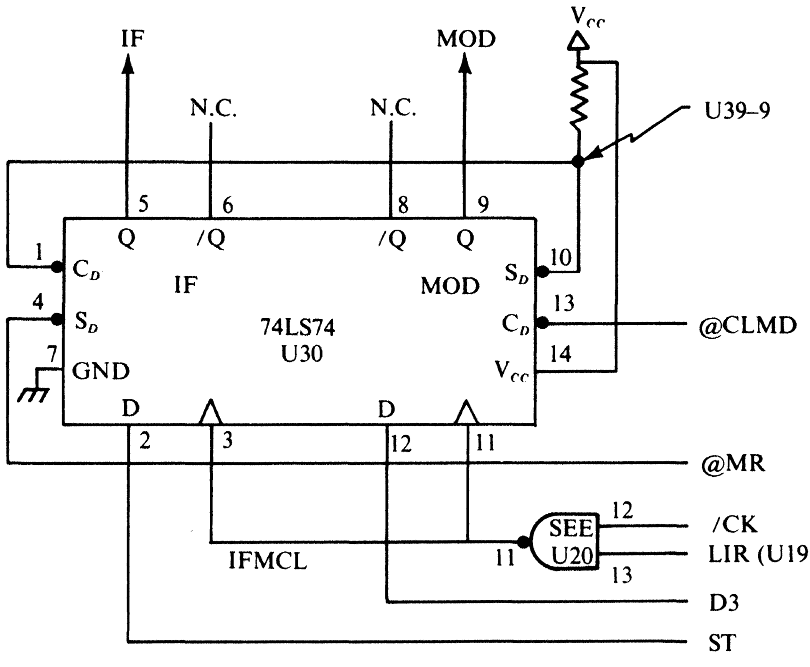


Figure 6-8
IF and MOD Bit Logic

Downloading Procedures

DLD Microcoding Format

Let us now briefly discuss the facilities for downloading stored demonstration programs. The microcoding format for creating downloadable programs is presented in Figure 6-8.

DLD Setting the Carry Flag

Line 0 is special. On this line, we must first initialize the PC and the carry flag. If the ALU is programmed, on line 0, to perform an A pl 1 function, the S3 . . S0 field will contain all low entries.

DLD PC Initialization

During download, this very same field is also presented to the A port of the ALU by the DMA-type of action going on over the DBUS. Since the ALU's control lines cause the value on the DBUS to be incremented by one and appear on the FBUS, the PC may be set to location 1 at the end of this period. This initializes the PC to point to the first demonstration program step, which must appear in location 1 of memory. At the same time, on line 0, we specified the loading of the flags. Since 0 plus 1 does not create a carry—and the carries are active low when active-high data conventions are used—then it follows that this will set the carry flag (CF in the FL register).

This preestablished high residing in the CF flip-flop becomes handy when we want to use lines 8 through F of the macro-block space in a demonstration program. If the program we are creating goes beyond line 7, then line 7 must specify the MPXing in of the CF. In the next period, the MPX line of CSAD is then high. This places the CSTR address into the upper half of the macro-block space. While we wish to remain in this region, the BC field must continue to multiplex in the CF flag.

Last Microline of Download

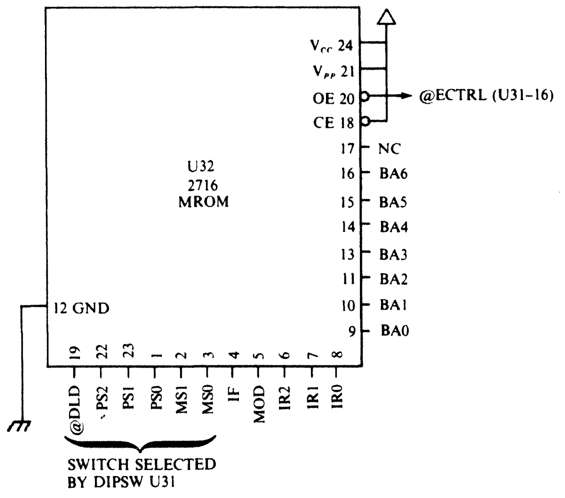
The last line of a program need not be line F. Whatever one chooses to make the last line should cause the *next address* field to specify a jump to line 0. If the last line happens to be in the upper half of the allowed space, an L in the BC field'reinforces this by specifying a return to the lower half of the macro space. This rule of operation causes the download procedure to cycle through a closed loop within the utilized block space. Issuing sixteen or more clock pulses downloads any demonstration program and then simply commences harmlessly repeating the process, if one does not stop precisely at the end boundary.

Table 6-1 lists the steps of the procedure for downloading and executing programs. The precaution repeated here is that @ECTRL, the switch that enables the control system, should not be closed unless *all* the CSTR-simulation switches in the CPU section are either *open* or *removed.* If, from the very start, we choose to construct STUPIDD only for use in the automatic mode of operation, then there is absolutely no need to implement the simulation switches at all. This is recommended, for the control-system-simulation switches provided in the CPU are only a convenience for those building just the CPU and for those without access to an EPROM programmer. If one already has the switches, they may be removed for safety during automatic operation. In this case, as already noted, the input switches (IN) may be used if the original eight-position DIP switch is replaced with a properly located four-position one. The master reset may be applied with a single DIP switch or a jumper wire.

Table 6-1
Download and Program Execution Procedure

1. Start with the power *off* and @ECTRL *open* on U31.
2. *Open* all of the CSTR simulation switches in the CPU section of STUPIDD. These are found in sockets U12, U21, and U22. The set of four input switches on U12 may be ignored now, but they can be used within a program.
3. Set the display-select rotary switch to position 7, to display the contents of the DBUS. The DBUS, during download, carries the contents of the S3 . . S0 field of the ALU function. The display presents the Hex equivalent of the program steps and data (except for the first line displayed, when this field is zero).
4. Turn the power *on*.
5. Cycle the Master Reset (MR) switch on U21. This clears registers MAR, T, IF, FL, IR, and CAR. Clearing CAR also automatically clears the MOD bit. The MR switch is normally *open* during system operation.
6. *Close* @ECTRL on U31. The control system is now activated and in charge of the CPU.
7. *Close* @DLD on U31. The system is now in the Download mode.
8. *Press* the clock button at least 16 times while observing the display for verification of the contents being loaded into memory.
9. *Open* @DLD on U31. The system is now in the Operate mode.
10. Each subsequent clock pulse administered causes the execution of a line of microcode in whichever macro is currently being used by the demonstration program. This assumes that the macros invoked by the program already exist in CSTR.

Figure 6-9
Mapping ROM Logic Diagram



CONTROL STORE IC DETAILS

The balance of the details in this section consist of the logic diagrams of the individual IC's. The associated parts and wire lists appear in the construction details in Chapter 5. The logic diagrams of the switch and pull-up resistor packages are shown on their respective wire-list sheets. Since we have worked with data catalogs for IC's, the following is a short summary of the manner of application and use of the IC interfaces in the control system.

IF AND MOD (U30)

Figure 6-8 portrays the interface of the 74LS74 dual D-type flip-flops used to implement IF and MOD. In the case of IF, the *set direct* (SD) line is tied to @MR, the master reset. After the reset push button is depressed, the system will be in the IF major state. The *clear direct* (CD) line is held high by the pull-up resistor in U39, for noise suppression. The ST signal from CSTR is the data input of the IF flip-flop, controlling the major state of the machine during normal operation. Notice that the clocking of both IF and MOD is qualified by both LIR and /CK in a NAND gate of U20. The CD line for MOD is tied to the @CLMD signal produced by CAR, U36. This is somewhat different from other applications of IC's discussed thus far, in that both synchronous data loading and asynchronous direct clears of this IC occur during normal operation. The microprogram in CSTR anticipates the clearing of MOD one bit period before it is to take effect. CAR produces a low as we enter the next period, immediately clearing the MOD bit near the start of the new period. The MOD flip-flop is loaded synchronously at the end of each macro. The data input for this is the D3 line of the DBUS. At the end of the IF phase of operation, the DBUS carries the first word of an instruction extracted from memory by the PC via MAR. This information contains the MOD and OP bits of the full instruction word. As noted previously, the loading of the instruction word at the end of the EX macros does not affect the system's conduct.

MROM (U32)

The MROM, like the three IC's of CSTR, consists of 2716 EPROM's. The data sheets for this IC are presented in Figure 6-10. The data intel®

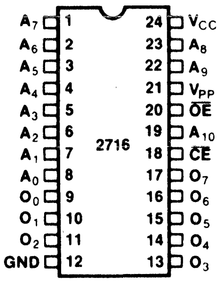
2716
16K (2K x 8) UV ERASABLE PROM

- **Fast Access Time**
  - **— 2716-1: 350 ns Max.**
  - **— 2716-2: 390 ns Max.**
  - **— 2716: 450 ns Max.**
  - **— 2716-5: 490 ns Max.**
  - **— 2716-6: 650 ns Max.**
- **Single +5V Power Supply**
- **Low Power Dissipation**
  - **— Active Power: 525 mW Max.**
  - **— Standby Power: 132 mW Max.**
- **Pin Compatible to Intel 2732A EPROM**
- **Simple Programming Requirements**
  - **— Single Location Programming**
  - **— Programs with One 50 ms Pulse**
- **Inputs and Outputs TTL Compatible During Read and Program**
- **Completely Static**

The Intel 2716 is a 16,384-bit ultraviolet erasable and electrically programmable read-only memory (EPROM). The 2716 operates from a single 5-volt power supply, has a static standby mode, and features fast single-address programming. It makes designing with EPROMs fast, easy and economical.

The 2716, with its single 5-volt supply and with an access time up to 350 ns, is ideal for use with high-performance +5V microprocessors such as Intel's 8085 and 8086. Selected 2716-5s and 2716-6s are also available for slower speed applications. The 2716 also has a static standby mode which reduces power consumption without increasing access time. The maximum active power dissipation is 525 mW while the maximum standby power dissipation is only 132 mW, a 75% savings.

The 2716 uses a simple and fast method for programming—a single TTL-level pulse. There is no need for high voltage pulsing because all programming controls are handled by TTL signals. Programming of any location at any time—either individually, sequentially or at random is possible with the 2716's single-address programming. Total programming time for all 16,384 bits is only 100 seconds.



PIN NAMES

A_0-A_{10}	ADDRESSES
$\overline{CE}$	CHIP ENABLE
$\overline{OE}$	OUTPUT ENABLE
O_0-O_7	OUTPUTS

Figure 1. Pin Configuration

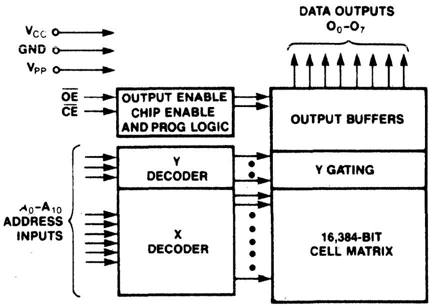
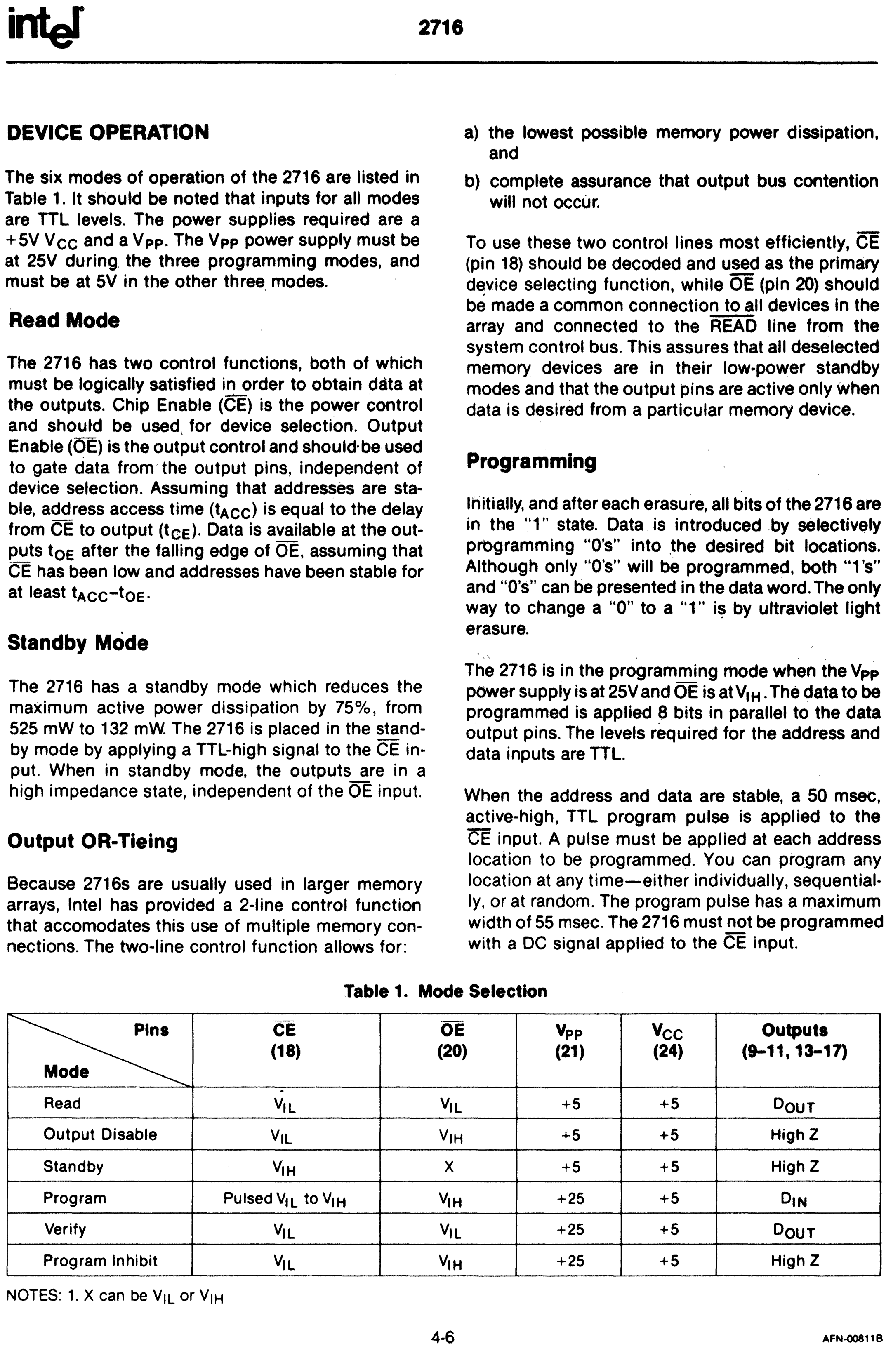


Figure 2. Block Diagram

NOVEMBER 1982

Figure 6-10a 2716 (2K × 8) UV-Erasable PROM Data Sheet *(Reprinted by permission of Intel Corporation © 1982, Intel Corporation.)*



2716

DEVICE OPERATION

The six modes of operation of the 2716 are listed in Table 1. It should be noted that inputs for all modes are TTL levels. The power supplies required are a +5V V_{CC} and a V_{PP}. The V_{PP} power supply must be at 25V during the three programming modes, and must be at 5V in the other three modes.

Read Mode

The 2716 has two control functions, both of which must be logically satisfied in order to obtain data at the outputs. Chip Enable ($\overline{CE}$) is the power control and should be used for device selection. Output Enable ($\overline{OE}$) is the output control and should be used to gate data from the output pins, independent of device selection. Assuming that addresses are stable, address access time (t_{ACC}) is equal to the delay from $\overline{CE}$ to output (t_{CE}). Data is available at the outputs t_{OE} after the falling edge of $\overline{OE}$, assuming that $\overline{CE}$ has been low and addresses have been stable for at least t_{ACC}–t_{OE}.

Standby Mode

The 2716 has a standby mode which reduces the maximum active power dissipation by 75%, from 525 mW to 132 mW. The 2716 is placed in the standby mode by applying a TTL-high signal to the $\overline{CE}$ input. When in standby mode, the outputs are in a high impedance state, independent of the $\overline{OE}$ input.

Output OR-Tieing

Because 2716s are usually used in larger memory arrays, Intel has provided a 2-line control function that accomodates this use of multiple memory connections. The two-line control function allows for:

a) the lowest possible memory power dissipation, and

b) complete assurance that output bus contention will not occur.

To use these two control lines most efficiently, $\overline{CE}$ (pin 18) should be decoded and used as the primary device selecting function, while $\overline{OE}$ (pin 20) should be made a common connection to all devices in the array and connected to the $\overline{READ}$ line from the system control bus. This assures that all deselected memory devices are in their low-power standby modes and that the output pins are active only when data is desired from a particular memory device.

Programming

Initially, and after each erasure, all bits of the 2716 are in the "1" state. Data is introduced by selectively programming "0's" into the desired bit locations. Although only "0's" will be programmed, both "1's" and "0's" can be presented in the data word. The only way to change a "0" to a "1" is by ultraviolet light erasure.

The 2716 is in the programming mode when the V_{PP} power supply is at 25V and $\overline{OE}$ is at V_{IH}. The data to be programmed is applied 8 bits in parallel to the data output pins. The levels required for the address and data inputs are TTL.

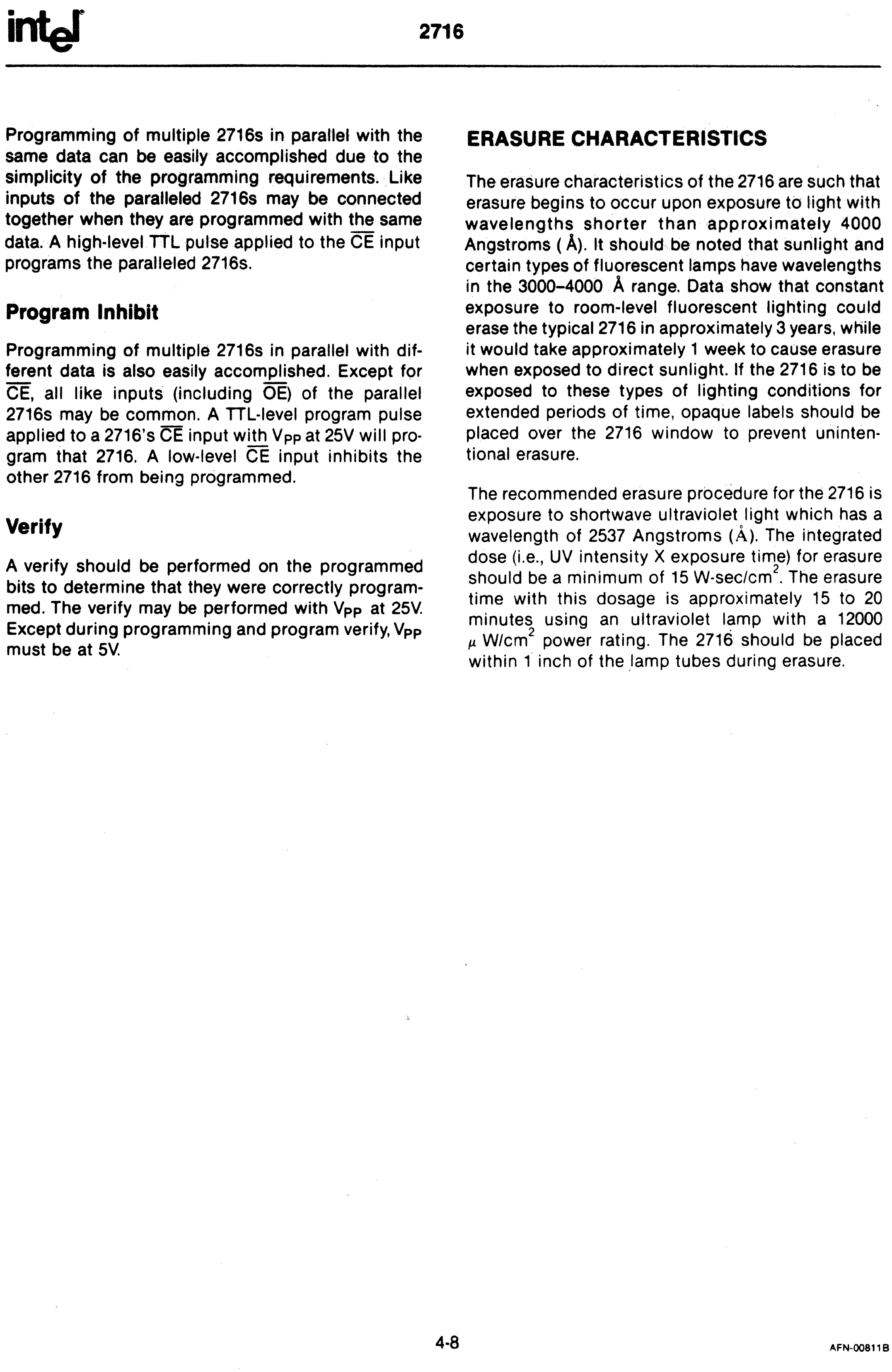
When the address and data are stable, a 50 msec, active-high, TTL program pulse is applied to the $\overline{CE}$ input. A pulse must be applied at each address location to be programmed. You can program any location at any time—either individually, sequentially, or at random. The program pulse has a maximum width of 55 msec. The 2716 must not be programmed with a DC signal applied to the $\overline{CE}$ input.

Table 1. Mode Selection

Pins Mode	$\overline{CE}$ (18)	$\overline{OE}$ (20)	V_{PP} (21)	V_{CC} (24)	Outputs (9–11, 13–17)
Read	V_{IL}	V_{IL}	+5	+5	D_{OUT}
Output Disable	V_{IL}	V_{IH}	+5	+5	High Z
Standby	V_{IH}	X	+5	+5	High Z
Program	Pulsed V_{IL} to V_{IH}	V_{IH}	+25	+5	D_{IN}
Verify	V_{IL}	V_{IL}	+25	+5	D_{OUT}
Program Inhibit	V_{IL}	V_{IH}	+25	+5	High Z

NOTES: 1. X can be V_{IL} or V_{IH}

Figure 6-10b 2716 (2K × 8) UV-Erasable PROM Data Sheet *(Reprinted by permission of Intel Corporation © 1982, Intel Corporation.)*



Programming of multiple 2716s in parallel with the same data can be easily accomplished due to the simplicity of the programming requirements. Like inputs of the paralleled 2716s may be connected together when they are programmed with the same data. A high-level TTL pulse applied to the $\overline{CE}$ input programs the paralleled 2716s.

Program Inhibit

Programming of multiple 2716s in parallel with different data is also easily accomplished. Except for $\overline{CE}$, all like inputs (including $\overline{OE}$) of the parallel 2716s may be common. A TTL-level program pulse applied to a 2716's $\overline{CE}$ input with V_{PP} at 25V will program that 2716. A low-level $\overline{CE}$ input inhibits the other 2716 from being programmed.

Verify

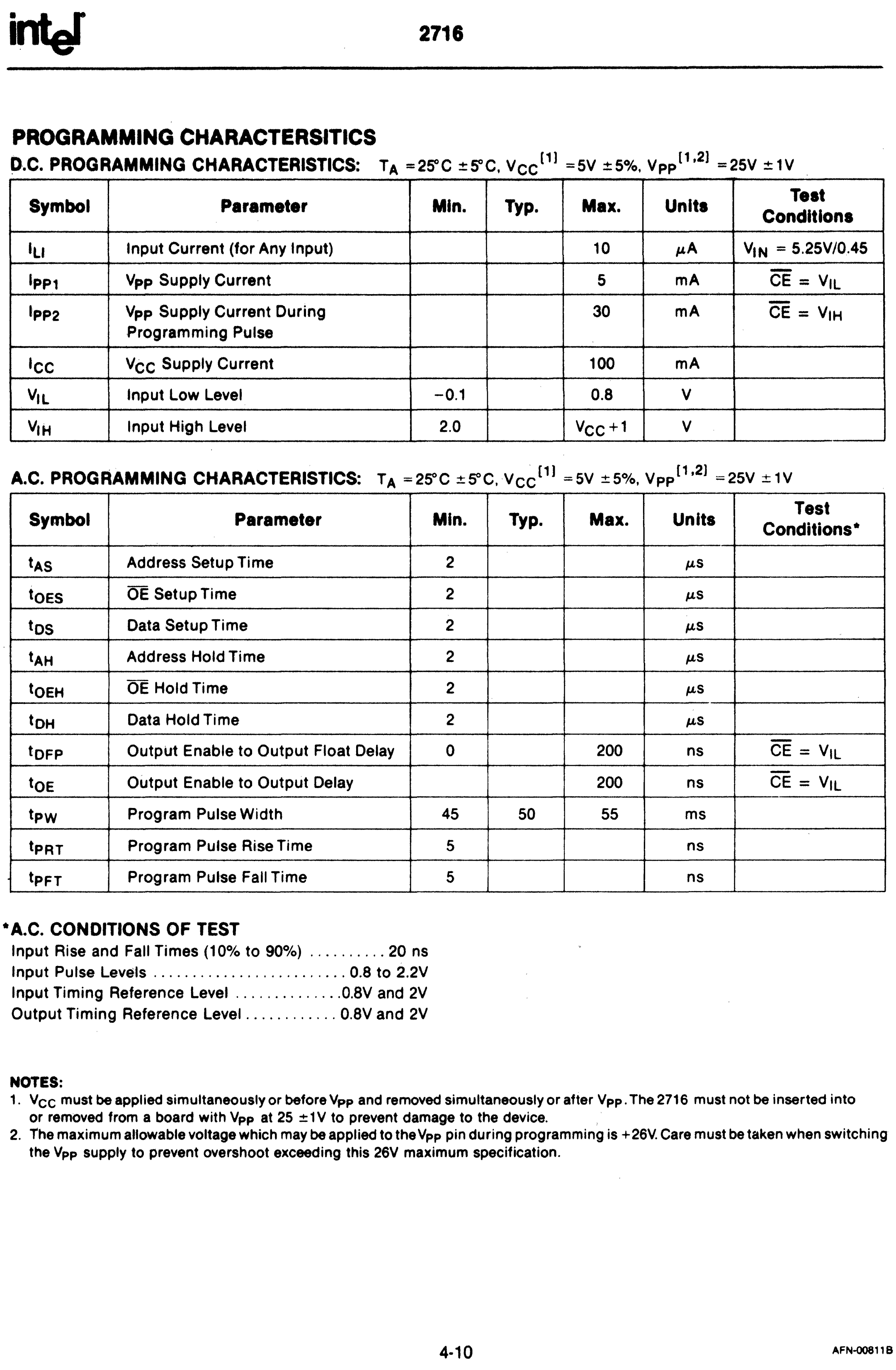
A verify should be performed on the programmed bits to determine that they were correctly programmed. The verify may be performed with V_{PP} at 25V. Except during programming and program verify, V_{PP} must be at 5V.

ERASURE CHARACTERISTICS

The erasure characteristics of the 2716 are such that erasure begins to occur upon exposure to light with wavelengths shorter than approximately 4000 Angstroms (Å). It should be noted that sunlight and certain types of fluorescent lamps have wavelengths in the 3000–4000 Å range. Data show that constant exposure to room-level fluorescent lighting could erase the typical 2716 in approximately 3 years, while it would take approximately 1 week to cause erasure when exposed to direct sunlight. If the 2716 is to be exposed to these types of lighting conditions for extended periods of time, opaque labels should be placed over the 2716 window to prevent unintentional erasure.

The recommended erasure procedure for the 2716 is exposure to shortwave ultraviolet light which has a wavelength of 2537 Angstroms (Å). The integrated dose (i.e., UV intensity X exposure time) for erasure should be a minimum of 15 W-sec/cm^2. The erasure time with this dosage is approximately 15 to 20 minutes using an ultraviolet lamp with a 12000 μ W/cm^2 power rating. The 2716 should be placed within 1 inch of the lamp tubes during erasure.

Figure 6-10c 2716 (2K × 8) UV-Erasable PROM Data Sheet *(Reprinted by permission of Intel Corporation © 1982, Intel Corporation.)*



2716

PROGRAMMING CHARACTERSITICS

D.C. PROGRAMMING CHARACTERISTICS: $T_A = 25°C \pm 5°C$, V_{CC}[1] $= 5V \pm 5\%$, V_{PP}[1,2] $= 25V \pm 1V$

Symbol	Parameter	Min.	Typ.	Max.	Units	Test Conditions
I_{LI}	Input Current (for Any Input)			10	μA	V_{IN} = 5.25V/0.45
I_{PP1}	V_{PP} Supply Current			5	mA	$\overline{CE} = V_{IL}$
I_{PP2}	V_{PP} Supply Current During Programming Pulse			30	mA	$\overline{CE} = V_{IH}$
I_{CC}	V_{CC} Supply Current			100	mA	
V_{IL}	Input Low Level	−0.1		0.8	V	
V_{IH}	Input High Level	2.0		$V_{CC}+1$	V	

A.C. PROGRAMMING CHARACTERISTICS: $T_A = 25°C \pm 5°C$, V_{CC}[1] $= 5V \pm 5\%$, V_{PP}[1,2] $= 25V \pm 1V$

Symbol	Parameter	Min.	Typ.	Max.	Units	Test Conditions*
t_{AS}	Address Setup Time	2			μs	
t_{OES}	$\overline{OE}$ Setup Time	2			μs	
t_{DS}	Data Setup Time	2			μs	
t_{AH}	Address Hold Time	2			μs	
t_{OEH}	$\overline{OE}$ Hold Time	2			μs	
t_{DH}	Data Hold Time	2			μs	
t_{DFP}	Output Enable to Output Float Delay	0		200	ns	$\overline{CE} = V_{IL}$
t_{OE}	Output Enable to Output Delay			200	ns	$\overline{CE} = V_{IL}$
t_{PW}	Program Pulse Width	45	50	55	ms	
t_{PRT}	Program Pulse Rise Time	5			ns	
t_{PFT}	Program Pulse Fall Time	5			ns	

***A.C. CONDITIONS OF TEST**

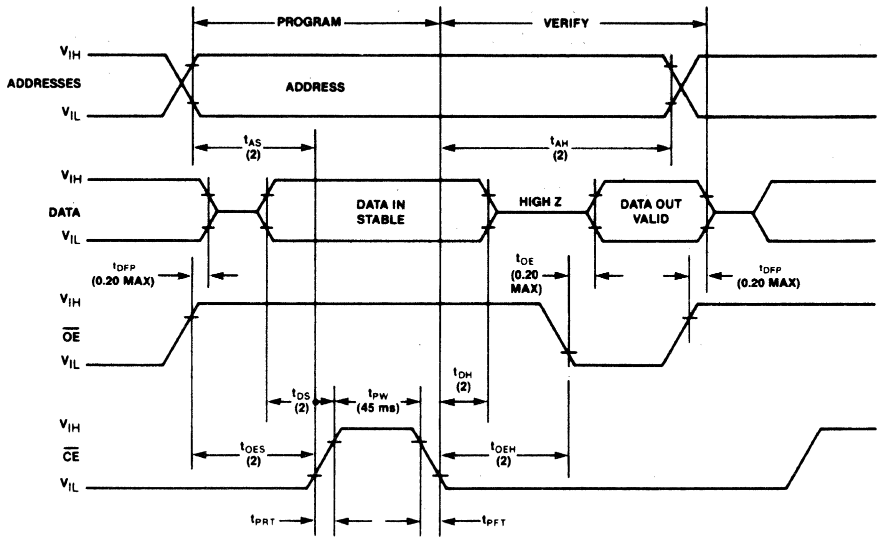
Input Rise and Fall Times (10% to 90%) .... 20 ns
Input Pulse Levels .... 0.8 to 2.2V
Input Timing Reference Level ....0.8V and 2V
Output Timing Reference Level .... 0.8V and 2V

NOTES:

1. V_{CC} must be applied simultaneously or before V_{PP} and removed simultaneously or after V_{PP}. The 2716 must not be inserted into or removed from a board with V_{PP} at 25 ±1V to prevent damage to the device.
2. The maximum allowable voltage which may be applied to the V_{PP} pin during programming is +26V. Care must be taken when switching the V_{PP} supply to prevent overshoot exceeding this 26V maximum specification.

Figure 6-10d 2716 (2K × 8) UV-Erasable PROM Data Sheet *(Reprinted by permission of Intel Corporation © 1982, Intel Corporation.)*

PROGRAMMING WAVEFORMS



NOTE:
1. ALL TIMES SHOWN IN PARENTHESIS ARE MINIMUM TIMES AND ARE μ SEC UNLESS OTHERWISE NOTED.
2. t_{OE} AND t_{DFP} ARE CHARACTERISTICS OF THE DEVICE BUT MUST BE ACCOMMODATED BY THE PROGRAMMER.

Figure 6-10e 2716 (2K × 8) UV-Erasable PROM Data Sheet *(Reprinted by permission of Intel Corporation © 1982, Intel Corporation.)*

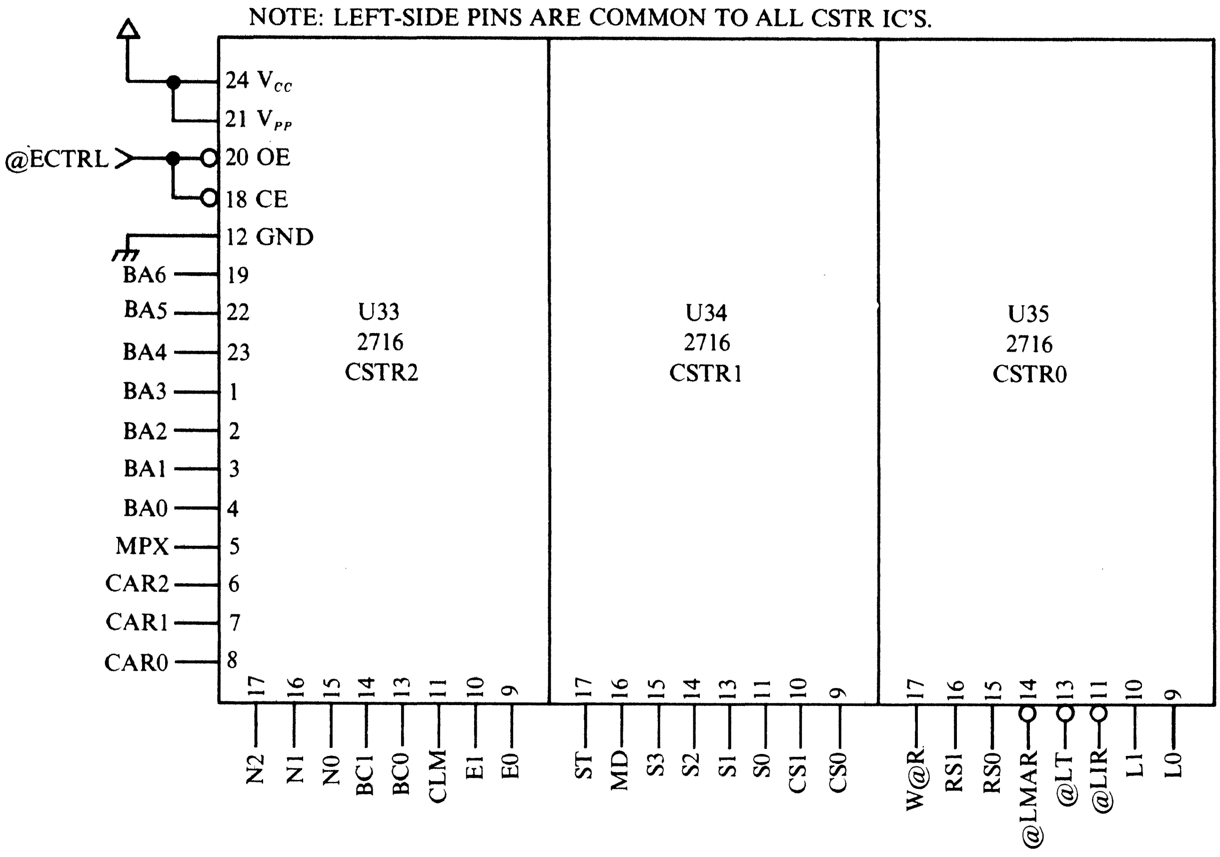


Figure 6-11
Control-Store Logic Diagram

sheets contain valuable information on the IC, its erasure, and its programming aspects. MROM's logic interface is presented in Figure 6-9. The tri-state interface control (OE) and the *chip enable* (CE) control of each IC are connected to the @ECTRL line. @ECTRL is a manual switch setting provided for operational control of the normal operation and program downloading modes of usage. The nature of the inputs has been discussed at length in this chapter. The outputs of MROM are the BA inputs to CSTR, also discussed.

CSTR 2 . . 0 (U33 THROUGH U35)

The address inputs to the IC's of CSTR are all shared in common. The hierarchical nature of the addressing is shown in Figure 6-11. CAR0 . . CAR2, which originate in CSTR2's own NEXT field, are fed back to all the EPROM's of CSTR with a one-bit delay, via CAR. These are the line-selection bits for the half of the macro-block space that is currently selected. The MPX input determines which half of the block space we are operating in. It is specified in the BC field of CSTR2 and fed back via the one-bit delay of CAR, acting on the address inputs of the multiplexer, MPX. Unlike the BA lines, these lines can change with each clock pulse. Finally, the specific macro we are executing is selected by the BA lines from MROM. These BA lines remain stable for the duration of a macro, changing only when a new macro is selected. Except for CLM, ST, and LIR, which feed back to the control system for the uses previously discussed, the rest of CSTR's outputs directly control the CPU or its

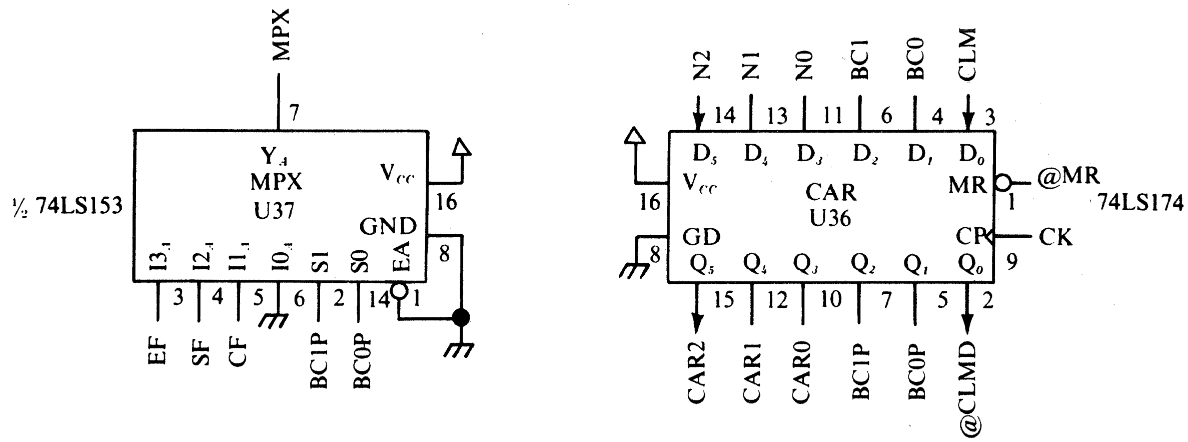


Figure 6-12
MPX and CAR Logic Diagrams

external environment. These lines are also the ones available in the simulation switches for CPU control.

MPX AND CAR (U37 AND U36)

The MPX and CAR interfaces are presented in Figure 6-12. MPX (half of a 74LS153) is a one-of-four selector addressed by the BC bits emitted by CAR. It can select one of the flag register's three utilized outputs or an absolute low. Its output controls the half of the macro-block space that the microprogram is operating in. CAR is a 74LS174 6-bit D-type edge-triggered register used in a straightforward manner. We have already discussed the functional aspects of its interface.

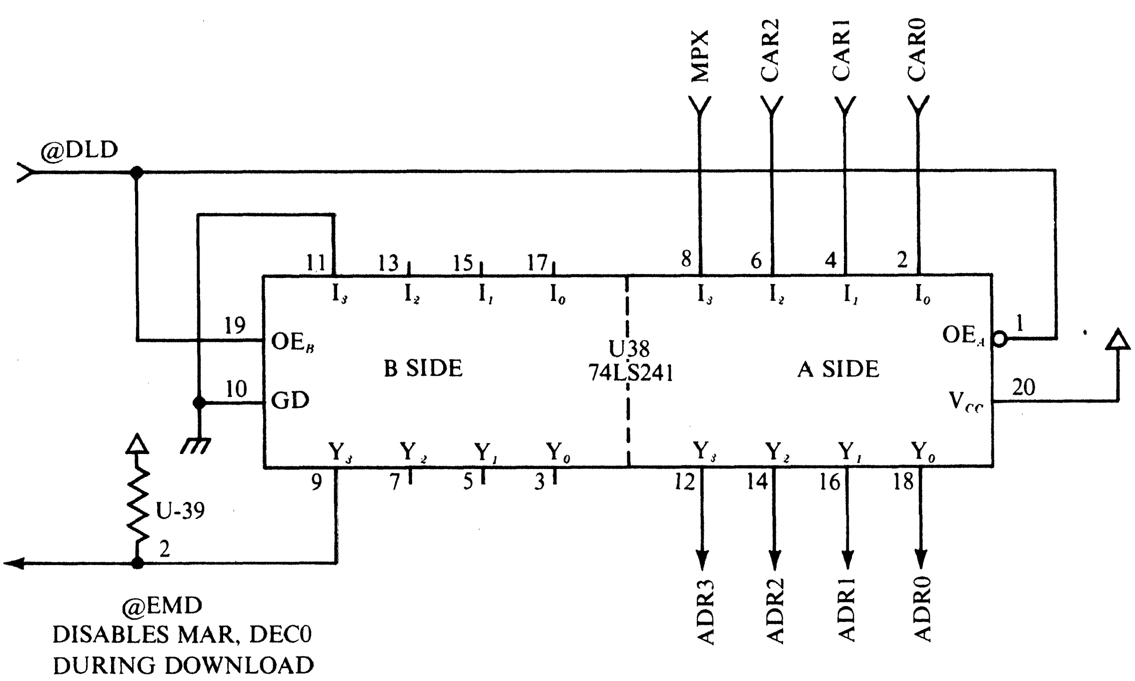
DOWNLOAD (DLD) TRI-STATE CONTROL (U38)

The operational aspects of the DLD interface were fully described in this section. (See Figure 6-4 and the associated discussion.) The tri-state control interface is centered around the 74LS241 (U38) IC. The A side portrayed in Figure 6-13 becomes the master of the address bus during downloading. The B side produces the @EMD signal that enables MAR and half of DEC0 during normal operation and disables both of them during downloading. The analogy between this procedure and the effects of a DMA operation has been commented on. In essence, this consists of the CPU's relinquishing control of memory to another entity—which is basically what happens here during downloading.

MICROCODING AND THE CONTROL SYSTEM

THE FIRST SIX CSTR SIGNALS

To understand the portion of the microcode that applies to the control system, let us return to the IF macro. Figure 6-14 presents the IF



DISABLES MAR, DECO DURING DOWNLOAD

ADDRESSES MEMORY IN DOWNLOAD STATE

Figure 6-13
Download Tri-State Control Logic Diagram

macro with the first three fields supplied on the symbolic coding form (this contains the first six signals of the activity-level coding forms).

Next Address Field

The first field, *Next Address*, is to be programmed to specify the first three bits of CSAD–that is, the first three address lines of CSTR–in the next period. CAR, a six-bit register, accepts these bits at the end of each clock period and presents them at its outputs at the start of the new period. The NEXT field need not proceed in the orderly way shown, except as a help in following the code.

NEXT Field Terminating Convention

On line 0, the next address is specified as 1; on line 1, it is specified as 2. In general, this is an easy way to proceed, except for the last line to be executed in any macro. *Here a 0 must be specified.* This ensures that the system starts the EX macro selected on line 0. Similarly, all other EX macros, except those that use the MOD bit, contain a 0 entry in the NEXT field on the very last line to be executed. *This is a convention we adopt to guarantee that all the macros start on the same line.* That is, they must all have the same entry point, which is derived from the last executable line of the preceding macro.

Conditional Branching in CSTR

Other than the last line, the NEXT entries could have been any value between one and seven. In practice, to use memory space efficiently, the lines of microcode often are not in sequential order. Thus, the order is not important. What is important is the effects of the sequence of algorithm's logical controls (the fields of the present output function) produced by CSTR, contained in each line of code. This accounts for address bits CSAD0 .. CSAD2, but what about CSAD3? This is controlled by the BC field. Specifying a zero now means that CSAD3

CSTR ADDRESS			ASSIGNED OP-CODE							
LINE NO.	MACRO: IF	NEXT ADDRESS	BRCH CTRL	CLR MOD	DBUS SOURCE FIELD	STATE	ALU FIELD		SINK CONTROL FIELD	
	PDL DESCRIPTION: IR <: M{PC}, PC <: PC pl 1, >: EX						ALU FUNC.	CS	R ARRAY	LOAD ENABLES
0	MAR, T <: PC	1	0	X	X	X	B	X	RPC	MAR, T
1	PC <: T pl 1	2	0	H	T	X	A pl 1	L	WPC	0
2	IR <: M{MAR}, >: EX	0	0	H	M	EX	X	X	RX	IR
3										
4										
5										
6										
7										
8										
9										
A										
B										
C										
D										
E										
F										
PDL NOTATIONS		CSTR2				CSTR1			CSTR0	

Figure 6-14 Microprogram Symbolic Coding Form: IF Macro with Control System Fields

will be low in the next period. When we specify one of the flags (SF, CF, or EF) in the BC field, the two-bit output of BC causes the multiplexer, MPX, to present that flag's current value as the value of CSAD3 in the following clock period, due to the one-period delay imposed by CAR. The selected value may be either low or high, depending on the value of the flag at the time it was stored in the flag register. Thus, we have the ability to perform *conditional branches* both within a macro and within demonstration programs. What about *absolute* branches?

Absolute Branching in CSTR

Absolute branches may be performed as well. If a flag has a known value that is useful in controlling the future value of CSAD3, then we simply multiplex it in when it is needed. An absolute low is always directly available to us. If we need an H flag, we can create one. For example, an arithmetic operation with active-high data that produces no carry will set the carry flag high. The technique, which is always available to us, is first to establish a known flag value, then to use it for branch control later on. If we wish to use the equal flag to establish an H, all we have to do is produce all highs at the F port, load the flag register, and later multiplex in this value to gain access to the upper half of the macro-block space. We will now be on the line specified in NEXT. In this way one can, in principle as well as in practice, access all sixteen lines of microcode allocated to each macro. Marketable systems are more efficient and flexible in their usage of this macro space, but our purpose is to demonstrate the principles under which systems operate. The activity-level codes for the selection of the flags by MPX are presented in Table 6-2.

Table 6-2
Flag-Selection Codes for MPX

BC Field Entry		*FLAG Selected*
BC1	*BC0*	*CSAD3 Source*
L	L	Low level
L	H	CF (Carry Flag)
H	L	SF (Sign Flag)
H	H	EF (Equal Flag)

Terminating BC Field Convention

We must specify the branch control field to be a zero, as we do for *next*, at the end of all macros. This ensures that, as we enter each new macro, bit 4 of CSAD will always be low. This convention maintains system consistency among the macros. Recall that, for STUPIDD, the entry point of all macros is line 0 of the lower half of the macro-block space.

Clear MOD Field

The CLR MOD field resets the MOD flip-flop via CAR. MOD is loaded at the end of the IF phase, because it is a part of the instruction register. This means that *the last two lines of the IF macro must contain an H in the CLR MOD field*, to avoid clearing MOD before address modification has been performed. It may not be clear why the last two executable lines of the IF macro contain an H in this field, instead of only the last one alone. This precaution has to do with the mixing of asynchronous direct-reset signals, the synchronous loading of data, and possible propagation delays, extending the direct-reset signal into the start of the next period. In short, we want to guarantee that there will be no low-going noise pulses on MOD's direct-clear input at the start of EX. If CAR is loaded one time frame earlier, it will be flicker-free at the direct-clear input. A useful exercise is to draw the timing diagram of the signals involved in these transactions.

CLR MOD Conventions

No such precaution is necessary at the end of the EX phase, because, when we enter IF, the IF bit is high and maps us into the upper half of MROM, regardless of the current value of the MOD bit or of any of the OP bits. The CLR MOD field must be specified as a low on the last line of any of the address-modification algorithms, such as indexing or indirect addressing. Once cleared, the MOD flip-flop is not set in any way that can affect operation, except at the end of IF. This means that, during the EX phase, the CLR MOD bit could have either an H or L value, without influencing operation. Under these conditions, an X (don't-care) entry may be used in the CLR MOD field during EX. Recapitulating these rules, CLR MOD must be high for the last two lines of the IF macro and for all lines of any effective address (EA) calculation macro—*except* the last one to be executed. On the last line of these operand address modification types of macros, CLR MOD will be specified as an L, to ensure the termination of this macro immediately as we *enter* the next period.

This, then, explains the nature of the first six signals of CSTR2, which are associated with the operation of the control system. Let us now examine some new macros for possible inclusion in an instruction set, whose operation we can demonstrate with STUPIDD.

MICROCODING EXAMPLES

EXAMPLE 6-1: DEC, R2

The first example of a completely specified microcoding sheet is a simple macro, DECrement, R2. Figure 6-15 presents the electrical-level microcode for this macro. One of the first things to do in creating a macro is to determine how many memory words it requires and to then decide whether the PC should be incremented one or more times in such a way that it will point to the start of the next instruction *before* IF is reentered. An operation of this DEC, R2 type is generally specified within one memory word on 8-bit and larger machines. Let us assume that this is the case now and—since this instruction does not reference memory—that the next word in memory is an OP, which is

CSTR ADDRESS		ASSIGNED OP-CODE																							
LINE NO.	MACRO: DEC, R2	NEXT ADR.			BRCH. CTRL.		MOD	DBUS SOURCE FIELD		IF	ALU FIELD							SINK CONTROL FIELD							
	PDL DESCRIPTION: R2 <: R2 mi 1, LFL										ALU FUNC.					CS		R ARRAY			LOAD ENABLES				
		N2	N1	N0	BC1	BC0	CLM●	E1	E0	ST	MD	S3	S2	S1	S0	CS1	CS0	W@R	RS1	RS0	LMAR●	LT●	LIR●	L1	L0
0	T <: R2	L	L	H	L	L	X	X	X	X	H	H	L	H	L	X	X	L	H	L	H	L	H	H	H
1	R2 <: T mi 1, LFL, >: IF	L	L	L	L	L	X	L	L	H	L	H	H	H	H	H	H	H	H	L	H	H	L	L	H
2																									
3																									
4																									
5																									
6																									
7																									
8																									
9																									
A																									
B																									
C																									
D																									
E																									
F																									
CPDL NOTATIONS		CSTR2								CSTR1								CSTR0							

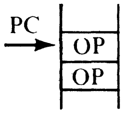


Figure 6-15 DEC R2 Macro: Complete Electrical Form

the start of the next instruction. The coding forms presented here often contain sketches of the mapping of the instruction stream in main memory, as a visual aid. Recall that the PC pointed to the location of the current instruction in memory at the start of IF. It was incremented once during IF, and it currently (in the EX phase of this particular macro) points to the next instruction to be fetched. Therefore we conclude that no further adjustment of the PC is needed to fetch a valid OP at the end of this macro.

The execution is straightforward. On line 0, we transfer the contents of R2 into T. The reasons for this should be clearly understood, since they are dictated by the nature and shortcomings of the IC's used in the system. The ALU does not support a decrement operation on its B port. This is not true of all ALU's. Further, the register array cannot read and write into the same register in one clock period. We must often learn about and live with the limitations of components and architecture. The NEXT line entry is to be 1, CSAD3 shall be a low, and the CLR MOD bit is specified as a don't care. The balance of line 0 is easily interpreted from our past discussions. Line 1 causes R2 to receive the decremented contents of T (the original contents of R2), load the flags, and initiate the next IF cycle. On line 2, the NEXT field is accordingly a 0, BC is an L, and CLR MOD is an X. The ST signal is an H, signifying a return to the IF major state–provided that LIR is active, which it is.

EXAMPLE 6-2: XOR, M

An example of a two-memory-word instruction is presented in the symbolic coding of exclusive OR (XOR, M), Figure 6-16. The CPDL summation of the net result of this macro is

R0 <: R0 ? M{OA}

This is a memory-reference instruction and, as a result, occupies two words of memory. On entering this macro, the contents of the PC point to the operand address (OA) field of the instruction word (see the memory map in Figure 6-16). This field contains the address–not the value–of the operand. On line 0, two goals are specified. By loading T with the contents of the PC, we start the incrementation (adjustment) process for the PC. By loading MAR with the same value at this time, too, we address memory, so as to be able to obtain the operand address. Note that, just after the IF phase, the value in the PC points to the location in memory following the OP field. The PC, in this case, contains the address of the operand address. Think this over. The PC serves only as a pointer into main memory. It is critical to identify clearly exactly what is pointed to at each step of operation by PC.

Line 1 specifies that the PC is to store its incremented value. Since it now points to the OP field of the next instruction, we will be ready for the new IF cycle when execution of the balance of this current macro is completed. On line 2 of the macro, MAR will receive the contents of memory (the address of the operand) that MAR is currently addressing. On line 3, T is to receive the XOR of R0 and the contents of memory pointed to by MAR: the actual operand, at last. It takes careful consideration of the meaning of an instruction to avoid confusing operands and their addresses. The flags could also have been loaded on line 3 of this macro, if we had wished to be able to control

CSTR ADDRESS			ASSIGNED OP-CODE							
LINE NO.	MACRO: XOR, M PDL DESCRIPTION: R0 <: R0 ? M{OA}	NEXT ADDRESS	BRCH CTRL	CLR MOD	DBUS SOURCE FIELD	STATE	ALU FIELD		SINK CONTROL FIELD	
							ALU FUNC.	CS	R ARRAY	LOAD ENABLES
0	MAR, T <: PC	1	0	X	0	X	B	X	RPC	MAR, T
1	PC <: T pl 1	2	0	X	T	X	A pl 1	L	WPC	0
2	MAR <: M{MAR}	3	0	X	M	X	A	X	RX	MAR
3	T <: R0 ? M{MAR}	4	0	X	M	X	?	X	RR0	T
4	R0 <: T, >: IF	0	0	X	T	IF	A	X	WR0	IR
5										
6										
7										
8										
9										
A										
B										
C										
D										
E										
F										
PDL NOTATIONS		CSTR2				CSTR1			CSTR0	

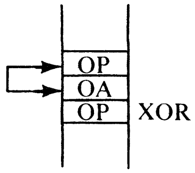


Figure 6-16 Microprogram Symbolic Coding Form: XOR, M—A Memory-Reference Macro

CSTR ADDRESS		ASSIGNED OP-CODE								
LINE NO.	MACRO: BMEQ, R0 PDL DESCRIPTION: BR TO 0A2 IF M{0A1} = R0	NEXT ADDRESS	BRCH CTRL	CLR MOD	DBUS SOURCE FIELD	STATE	ALU FIELD		SINK CONTROL FIELD	
							ALU FUNC.	CS	R ARRAY	LOAD ENABLES
0	T, MAR <: PC	1	0	H	0	X	B	X	RPC	T, MAR
1	PC <: T pl 1	2	0	H	T	X	A pl 1	H	WPC	0
2	MAR <: M[MAR]	3	0	H	M	X	A	X	RX	MAR
3	M[MAR] min R0, LFL	4	0	H	M	X	A min B	H	RR0	FL
4	T, MAR <: PC	5	EQ	H	0	X	B	X	RPC	T, MAR
5	PC <: T pl 1, >: IF	0	0	H	T	IF	A pl 1	L	WPC	IR
6										
7										
8										
9										
A										
B										
C										
D	PC <: M[MAR], >: IF	0	0	H	M	IF	A	X	WPC	IR
E										
F										
PDL NOTATIONS		CSTR2				CSTR1			CSTR0	

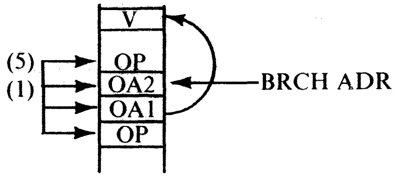


Figure 6-17 Microprogram Symbolic Coding Form: BMEQ, R0 Microcode

the flow of the software with this instruction in a subsequent operation. Line 4 stores the result in R0 and initiates a return to the IF phase of the cycle of computation.

EXAMPLE 6-3: BMEQ, R0

The next macro illustrates the use of the BC field, as well as showing again that it is feasible to create three-memory-word instructions. Figure 6-17 presents the symbolic microcode for the macro BMEQ, R0. As noted in the figure and its memory map, this is a three-memory-word instruction. The first word contains the OP code, the second the operand address of the memory location to be used in the comparison (OA1), and the third word (OA2) is devoted to providing the branch address in the event that equality is established. The explanations of lines 0, 1, and 2 are the same as in the previous macro. In this macro, however, note that we still have to adjust the PC at the end of these lines, to be able to proceed correctly on to the next IF cycle. We will pick up the sequence on line 3.

Conditional Branching

On line 3, the ALU is placed in the subtract mode, so that we can use its equal output for the comparison. Refer to the 74181 data sheets if you have any questions on this point. The *results* of the subtraction are *not* stored–only the flags are loaded on this line. This is customary practice for the compare-type instructions. We only want to learn about the nature of the operand (OA1) in relation to the contents of R0. Further, we do this in a way that permits the control of the flow of the software by loading the flags. Having loaded the flags on line 3, we may now consider how to complete the rest of our tasks for this macro, using the flags. In line 4, we start the second incrementation of the PC, as well as the addressing of OA2, the branch address, in case we need it. On this line, we also specify, in the BC field, that the pipeline register CAR is to receive the *equal flag* (EF) selection code. In the *next* period, line 5, MPX will present the stored value of the equal flag to CSTR as the current value of CSAD3. In this next period, the machine will be *either* on line 5 (EF = L) *or* on line D (EF = H), *depending on the previously stored value of the flag* now selected by MPX.

The equal flag is always active high, regardless of the data's activity level. Therefore line 5 represents the case in which the equality-detection condition failed, and we do not branch. As a result, we merely specify the completion of the incrementation of the PC and initiate a return to IF on this line. Line D, however, represents the case in which equality was detected, and so we wish to generate a branch. Instead of completing the incrementation of the PC, which is no longer necessary if we are on this line, we simply reload the PC with the branch address contained in the memory location now addressed by MAR and end this macro by initiating a return to IF. Note that the BC field had to anticipate the use of the flags by one clock period in line 4, due to the pipeline-delay effect of CAR.

CSTR ADDRESS				ASSIGNED OP-CODE						
LINE NO.	MACRO: BRX PDL DESCRIPTION: R1 :< R1 mi 1, IF (R1 = 0) THEN PC :< 0A ELSE >: IF	NEXT ADDRESS	BRCH CTRL	CLR MOD	DBUS SOURCE FIELD	STATE	ALU FIELD		SINK CONTROL FIELD	
							ALU FUNC.	CS	R ARRAY	LOAD ENABLES
0	T, MAR :< PC	1	0	X	0	X	B	X	RPC	T, MAR
1	PC :< T pl 1	2	0	X	T	X	A pl 1	L	WPC	0
2	T :< R1	3	0	X	0	X	B	X		T
3	T, R1 :< T mi 1, LFL	4	EF	X	T	X	A mi 1	H	WR1	T, FL
4	PC :< M[MAR], >:IF	0	0	X	M	IF	A	X	WPC	IR
5										
6										
7										
8										
9										
A										
B										
C	>: IF (NO BRANCH)	0	0	X	0	IF	X	X	RX	IR
D										
E										
F										
PDL NOTATIONS		CSTR2				CSTR1			CSTR0	

BRCH ADR

OF
OA
BRX

Figure 6-18 Microprogram Symbolic Coding Form: BRX

EXAMPLE 6-4: BRX

Another example of a macro that contains a conditional operation is the BRX macro of Figure 6-18. It is of particular interest, in that it reveals the true nature of how the equal logic functions within the 74181, as well as the underlying principles of the autodecrementation (or incrementation) and branch types of instructions.

Index Register Usage

First, there is no explicit zero flag in STUPIDD. Second, many machines contain an instruction macro for an index register that is incremented or decremented each time this macro is invoked. An instruction such as this is useful in searching a table a predetermined number of times, for example. The initial value previously placed in the index register is related to the number of times we wish to repeat the loop. In this example, the number of times, *n*, we wish to go through a loop has already been placed into R1, by the execution of a prior instruction. We use R1 as an example of index register operation.

Loop Termination

After each pass through the loop, this register is to be decremented and the flags are to be loaded. *If* we have not performed this *n* times, *then* we branch to the address specified in the OA field of the instruction, *else* we fall out of the loop by proceeding on to the next instruction of a program's sequence.

A = B Functioning

Consult the equivalent-logic diagram found in the ALU data sheets for the following explanation of equality-detection operation. What this operation really detects is the presence of all highs at the F port. The A = B output comes from an AND device, whose output is high only when all inputs are high. This electrical behavior implies that, any time all of the F outputs of the ALU are high, since they are the inputs to the AND device, then the A = B outputs are also high. Not only does this explain how equality is detected, it also provides us with a means of detecting zero. If the contents of a register are decremented until the result at the output of the ALU is -1 (all ones), then the A = B output is high. Loading the flags as we decrement the contents of R1 in passing it through the ALU, we may subsequently employ conditional branching on the equal flag result. By these means, we can specify repeated branching within an instruction macro to control how many times a loop is traversed. On line 3 of this macro, while we combinationally decrement, we also enable the loading of the flags. On this line, too, the MPX is to select EF (in the next period). As a result, we specify the branch on line 4 to go through the loop again. The other possible action, on line C, simply lets us fall out of the loop and proceed to fetch the next OP.

Flag Activity Levels

A flag may not always be active high. For example, if one were to use the carry flag for controlling branches when using active-high data, the line on which the condition CF = true occurs when CF = L. This is the opposite of the case above. Some care, then, must be exercised in determining which of the two possible lines of conditional microcode represents the true condition, since CSAD3 depends on the current value of a flag. The machine executes only one of these lines in any particular execution of the macro. Thus we see that the choice of the

CSTR ADDRESS	ASSIGNED OP-CODE

LINE NO.	MACRO: CALL OA PDL DESCRIPTION: SP <: SP mi 1, M{SP} <: PC PC <: M{OA}	NEXT ADDRESS	BRCH CTRL	CLR MOD	DBUS SOURCE FIELD	STATE	ALU FIELD		SINK CONTROL FIELD	
							ALU FUNC.	CS	R ARRAY	LOAD ENABLES
0	T <: R2	1	0	H	0	X	B	X	RR2	
1	MAR, R2 < T mi 1	2	0	H	T	X	A mi 1	H	WR2	MAR
2	T <: PC	3	0	H	0	X	B	X	RPC	T
3	T <: T pl 1	4	0	H	T	X	A pl 1	L	RX	T
4	M[MAR] <: T, MAR <: T mi 1	5	0	H	T	X	A mi 1	H	RX	MAR, M
5	PC <: M[MAR], >: IF	0	0	H	M	IF	A	X	WPC	IR
6										
7										
8										
9										
A										
B										
C										
D										
E										
F										
PDL NOTATIONS		CSTR2				CSTR1			CSTR0	

OP	
OA	SUBROUTINE ADDRESS
OP	CALL

Figure 6-19 Microprogram Symbolic Coding Form: Call, OA

line of microcode on which a branch is to occur depends both on the activity level of the selected flag and on its subsequent logical usage. Note, too, that the first three address bits of CSAD are the same for both lines when branch controlling flag selection is in effect. The fourth bit alone depends on the value of the flag. This helps establish where the two lines appear on the microcoding forms. When the BC field is specified as an L, this fourth address bit of CSTR is always low in the next time period.

EXAMPLE 6-5: CALL, OA

Yet another useful illustration of the workings of microcode is the CALL, OA instruction. In Chapter 3, we discussed stack-pointer operation in relation to calls and returns. The mechanics of implementing them requires that a register in R be designated as the *stack pointer* (SP). We shall use R2 to illustrate SP action. Adjusting the PC so as to point to the next instruction's OP field on reentering IF can often be a subtle process. The object is to use as few lines of microcode as possible. The CPDL description of the net result of the CALL instruction is

SP <: SP mi 1

M{SP} <: PC

PC <: M{OA}

The CPDL notation above discloses the net result of the action, not necessarily the sequence of actual steps in which it is performed. Figure 6-19 presents the symbolic microcode of one way to implement the call.

Lines 0 and 1 are devoted to decrementing SP and placing this value in MAR. MAR now points to the location on the stack where the old PC will be preserved for the eventual return. In line 2 of the microcode, we temporarily store the contents of the PC in T. In line 3, we increment T. This value represents the value of PC to be preserved for the eventual return, where PC must fetch the OP field of the next instruction on entering IF again. Line 4 may appear a bit tricky. Two compatible actions occur simultaneously. In one, we *decrement* the value of T by passing it through the ALU while it is in the decrement mode—to prepare to store this decremented value in MAR. This action will result in MAR's addressing the memory location that is the OA (the address of the called subroutine) in the next period. In the other simultaneous compatible operation, since the current value of T (which now contains the return address) is on the DBUS, we simply store it away at the location in memory dictated by the current value of MAR. During the period of line 4, MAR contains the decremented value of the SP, which specifies where the old value of the PC is to be saved. In the next period, as a result of these actions, MAR will contain the address of the operand (the branch address). This operand holds the new value of PC. Finally, on line 5, the PC receives the address of the called subroutine, and a transfer to IF is initiated. This is not the only five-line solution to this problem. Can you invent another?

In the next IF, the instruction received is the first instruction of the subroutine. In practice, several other tasks may be performed in a system during a call as well. On larger machines, the contents of the flag register are often saved by the Call macro. These flags are sometimes referred to as the *program status word* (PSW). STUPIDD's architecture does not support the saving of flags, but not to do so would be a fatal design flaw in the real world. Thus we have illustrated the fundamental operation of the call and some of the devices used to make its microcode compact, which translates into speed of operation.

EXAMPLE 6-6: X (INDEXED ADDRESSING)

For our last example of microcode, we shall illustrate the microcoding of the Indexed Addressing macro. Figure 6-20 illustrates the operand-address modification algorithm to be performed during indexed addressing. In discussing address modification, we need to introduce the term *effective address*, or EA, as we shall call it. Up to this point, the EA of the operand for memory reference instructions has been the one(s) specified or implied (as for MVI) in the instruction word. If the MSB of instruction is high, the MOD flip-flop is set when STUPIDD enters its EX phase of operation. Let us use R1 as the system index register for purposes of illustrating the operational aspects of indexed operand address modification. *Indexed operand addressing* is defined as the calculation of the EA of an operand, performed by summing the contents of the OA value supplied in the instruction word and the contents of the designated index register (R1 in our example). This sum becomes the EA and is to be placed into MAR to fetch the desired target operand, as illustrated in Figure 6-20.

Priority Encoding of Operand Address Modification

The stratagem for selecting the BA of this macro relies on the use of the priority-encoding potential of MROM's input address lines. If MOD has been set high *and* the IF bit is low, the MROM address must be in the second quarter of its current macro-set address space. If we use the MOD bit to illustrate indexed addressing, the block address now emitted by MROM will be the block address of this macro, provided that this quarter of the macro-set address space is entirely filled with a single BA value. As long as MOD is high, the system will ignore the lower ordered bits of the actual OP code to be performed. STUPIDD is now in the EX phase of operation, but it is performing the prioritized effective-address calculation of indexed addressing macro—not the execution of the actual instruction fetched during IF. That is, not until the MOD bit is cleared within the Indexed Addressing macro will the steps after the calculation of the operand's address of the fetched instruction begin.

Figure 6-21 presents the microcode for the indexed addressing effective-address calculation. On line 0, MAR and T receive the PC. MAR is now pointing to the OA literally expressed in the instruction, not to the desired effective operand address EA. In line 1, the PC is adjusted so as to point to the next OP, in anticipation of the next IF. On line 2, MAR receives the results of the summation that forms the

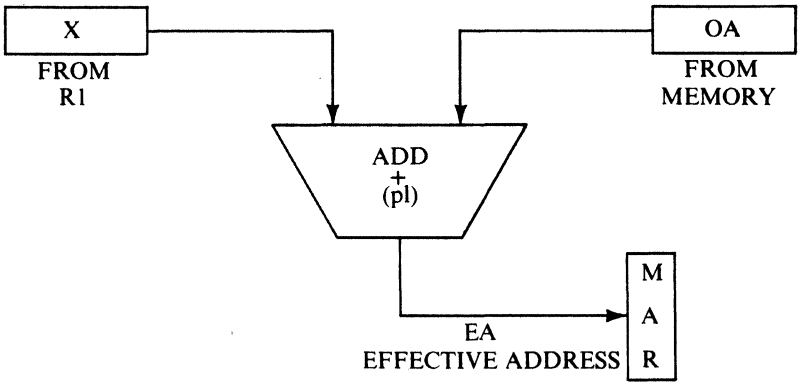


Figure 6-20
Operand-Address Modification by Indexing

EA. Up to this point, the CLR MOD field had to be specified as an H. It must now be specied as an L. This value will be available at the output of CAR at the *start* of the next period. Once MOD has been cleared, the block address emitted by MROM will be determined solely by the value of the current OP field in IR. The only question still to be resolved is what is the proper value to place in the *next* address field on line 2?

Operand Address Modification Fall-through

To find a simple answer to the question above, we must adopt a consistent system-wide convention for all EA calculations. The indexed addressing algorithm must be considered in relation to the instruction whose operand address is to be modified. Let us use the ADD macro of Figure 5-61 to illustrate a simple method of systematically reconciling the termination of indexing with the completion of the instruction to be performed. For instructional purposes, we may deviate from this rule elsewhere. Nevertheless, all EA address-modification algorithms we create are assumed to coordinate with the ADD instruction, unless otherwise specified. Note that, on line 2 of the ADD macro, MAR is being set up to receive the EA of the nonindexed instruction. On line 2 of the Index macro, the same event is about to occur. In both cases, MAR will hold the EA in the next period. Therefore, the NEXT address field of line 2 of the Index macro should contain the value 3. In this manner, when MOD is cleared, the system falls through from the Index macro to the instruction-execution line of code *at the point where the EA in MAR will be used to complete the instruction* in the customary manner. Also note that the PC has already been properly adjusted to point to the next OP in the Indexed Addressing macro. By its very nature, it had to have at least one operand address.

Note that, in the above, the system does not return to IF until the end of the actual instruction execution. The priority-encoding features of MROM are employed to establish a macro for the calculation of the modified EA, and, when MOD is cleared, we fall through to the line of the instruction where the EA is used. A production system is more complex and flexible, but, in essence, it performs the same type of tasks. If STUPIDD, for example, had a larger block space for a macro, then we could afford to perform customized address modification within each particular macro that is controlled by the OP field of each instruction. The resulting advantage is that, rather than adopt a rigid

CSTR ADDRESS			ASSIGNED OP-CODE							
LINE NO.	MACRO: INDEX (R1) PDL DESCRIPTION:	NEXT ADDRESS	BRCH CTRL	CLR MOD	DBUS SOURCE FIELD	STATE	ALU FIELD		SINK CONTROL FIELD	
							ALU FUNC.	CS	R ARRAY	LOAD ENABLES
0	T, MAR <: PC	1	0	H	0	X	B	X	RPC	T, MAR
1	PC <: T pl	2	0	H	T	X	ADD	L	WPC	0
2	MAR <: M[MAR] pl R1, CLM	3	0	L		X	ADD	H	RR1	MAR
3										
4										
5										
6										
7										
8										
9										
A										
B										
C										
D										
E										
F										
PDL NOTATIONS		CSTR2				CSTR1			CSTR0	

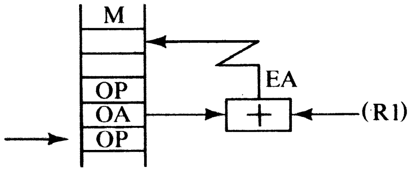


Figure 6-21 Indexed-Addressing Macro

fall-through convention, we can customize the EA calculation to suit each individual instruction. In practice, there will be some variations in EA-calculation fall-through, but not too many. In these cases, subroutine calls located within CSTR save space and neatly solve the problem.

The creation of the important macro for indirect addressing is left as an exercise. In this EA-calculation macro, the OA field of the instruction word contains the *address of the address of the operand.* In solving this problem, one should first draw the memory map indicating what the instruction word and its subsequent pointers are pointing to.

This completes our analysis of the organization, functioning, and microprogramming of the typical microprogrammed von Neumann type of computer/processor architecture. This type includes the vast majority of those one encounters in ordinary practice. These last, brief comments may appear anticlimactic, after all the preparation up to this point of the book. Practice is what counts now. With these examples as a guide, we can try our own hand at creating instructions in microcode for STUPIDD. For starters, take the example of indirect addressing. A number of other interesting macro-generation problems appear in the problem set for this chapter, including one for multiplication.

PROM Programming the Control System

The use of EPROM programming in the development of instruction sets and demonstration programs for STUPIDD is a great convenience. Since the 2716 EPROM's employed for MROM and CSTR are erasable and reusable, mistakes are readily corrected and innovative experimentation is facilitated. After the EPROM's are programmed, they are reinserted in their respective sockets, then demonstration programs are downloaded into memory (using the procedure presented in Table 6-2) and placed into operation by simply pressing the clock button at the rate we wish to proceed. Between clock pulses, we can examine the state of the machine, using the LED display or a voltmeter to probe the voltage levels at the pins of various IC's. As noted, manual operation of the CPU control switches is a good introductory tool, but it soon grows old. The programming of the EPROM's is a more highly developed approach to experimentation. The environment for PROM programming provides readers with useful experience, in that they may expect to encounter it in industrial development of firmware or of programs residing in some form of permanent memory. PROM programming is economical and is within the reach of both hobbyists and schools.

Most personal computers are capable of supporting PROM programming. The three hardware elements required are the personal computer of choice, a PROM programmer that interfaces to it, and a UV EPROM-erasing lamp. In selecting these components, make sure that they come with sufficient software support. A reasonable amount of software development and adaptation readily creates the desired type of programming environment. The author uses a PROM programmer produced by Advanced Microcomputer Systems, Ft. Lauderdale, Florida. Figure 6-22 illustrates the adaptability of these peripheral devices. One of the two cards is inserted into a slot of the IBM PC or XT personal computer; the other adapts the devices to sys-

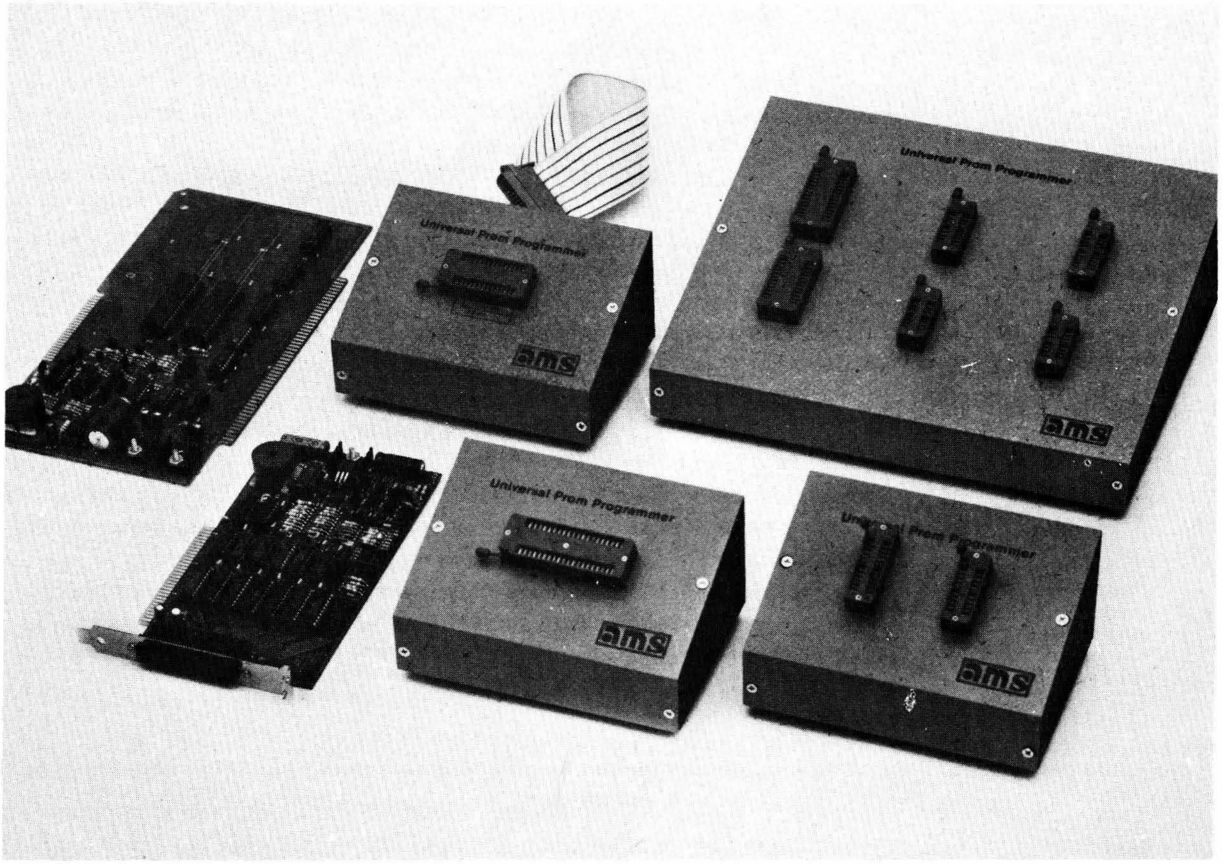


Figure 6-22
The PROM Programmer
(Courtesy of Advanced Microcomputer Systems, Inc., Ft. Lauderdale, Florida)

tems using the S100 bus. Once the card is inserted into an expansion slot, the supplied cabling is connected to the selected external PROM-programming box. The EPROM programmer selected for use was installed in an IBM XT. The PROM-programmer hardware came with the essential core of software and with a PROM programming manual. This elementary body of software can be readily incorporated into a user-friendly firmware-development environment, which we shall describe shortly.

An example of a UV erasing device is illustrated in Figure 6-23. This particular one is produced by UVP, Inc., San Gabriel, California. It contains a sliding drawer into which EPROM's are placed. The drawer is closed, the erasing action is initiated, and in less than 15 minutes the erased EPROM's are available for reprogramming. A note of caution: These erasers have built-in safety features. *Never* try to defeat them, under any circumstances. Staring into the ultraviolet can damage your eyes.

This presents an overview of the basic equipment required to establish our own PROM-programming environment. As noted, many personal computer enthusiasts already possess such equipment. A visit or two to a personal-computer club or user's group is often the best way to obtain specific assistance with your particular machine. Knowledgeable people at a personal-computer store may be able to assist you, too. The advertisements in the serious computer enthusiasts' magazines, such as *Dr. Dobb's Journal,* and in the more technical magazines devoted to the various personal computers are an excellent source of information on systems and software development, as well as on hardware configuration.

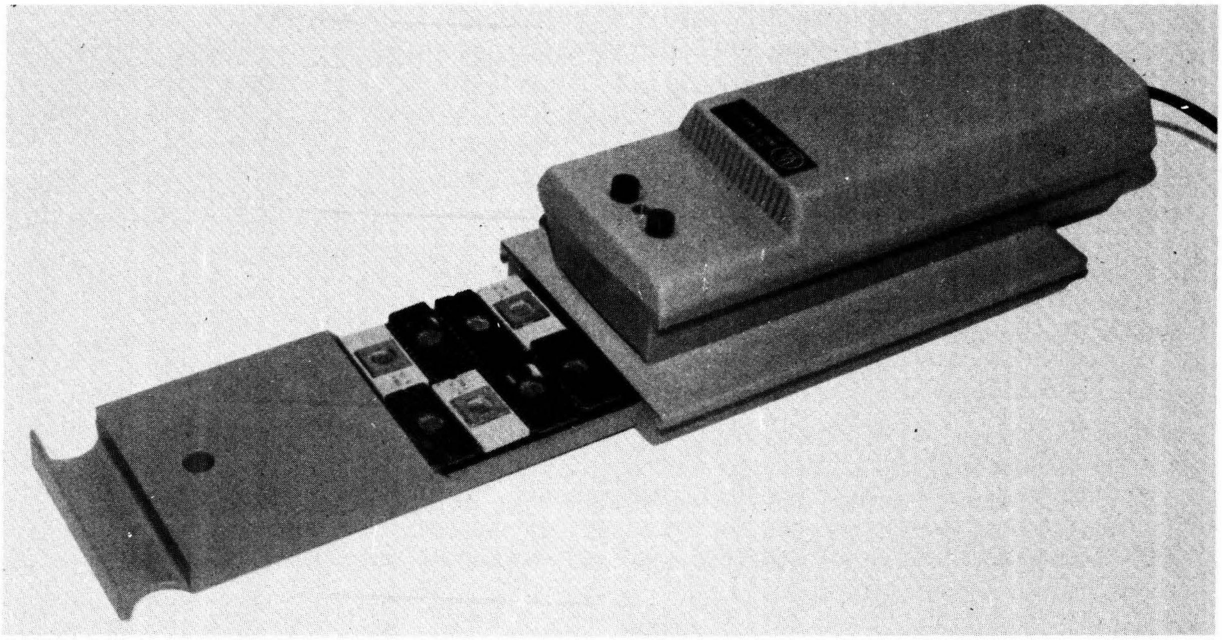


Figure 6-23
The PROM UV Eraser
(Courtesy UVP, Inc., San Gabriel, Calif. Memorase® is a registered trademark of UVP, Inc.)

The IBM XT personal computer permits instant operation and provides the facilities for the maintenance of a large data base of programs, utilities, and instructional aids in the PROM programming environment. These are available for student use on the XT's 10-megabyte hard disk. Users can keep their personal work on their own floppy disks, while taking advantage of the software PROM-programming environment available to them on the hard disk. Figure 6-24 shows the laboratory setup in the Computer Science Department's laboratory at the California State Polytechnic University, Pomona. This illustration shows a student team at work, with the attached EPROM programmer and a UV erasing device at their disposal. Let us now examine the type of development environment that will provide a user-friendly PROM-programming environment. This will be presented in as general a way as possible, since the users of software often wish to have it programmed in a different language from the one used by the creator; furthermore, the operating systems often are incompatible.

First, let us introduce a demonstration program for STUPIDD. This establishes the nature of the macros in one of the sets of macros required to run this program. The simple program selected is presented in Table 6-3, in Assembly-language format.

Table 6-3
Demonstration Program 0 (PROG0)

Line	*Label*	*Op Code*	*Operands*
1	START	MVI	R0,03H
2	LOOP	OUT	R0
3		DCBZ	R0,REPT
4		BRU	LOOP
5	REPT	OUT	R0
6		BRU	START
7		END	START

In this program we first move a hexadecimal value, arbitrarily chosen as 3, into R0. Next, the contents of R0 are output to the LED Hexadecimal

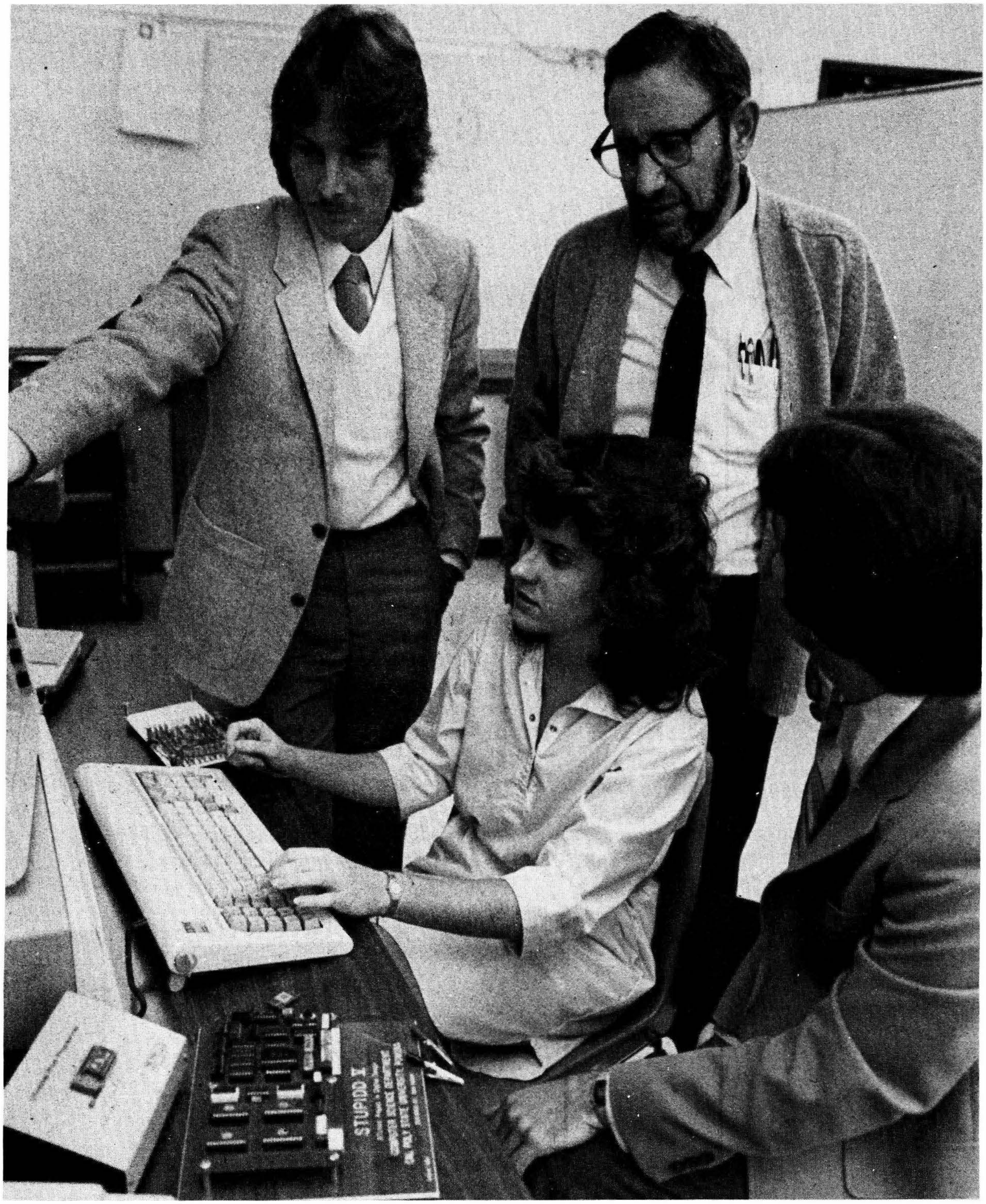


Figure 6-24
PROM Programming Session in Progress
(Photograph courtesy of the author.)

display. Line 3 means that we decrement R0 and then, if the contents of R0 are zero after the decrementation, we branch to symbolic location REPT. This test will fail more often than it will succeed. If it fails, the program loops back to symbolic location LOOP and outputs the new (decremented) value, then proceeds on in the loop. At symbolic location REPT (line 5), we output the value of R0, i.e., zero, to the display and then, on line 6, branch right back to start the process all over again.

Thus, the machine has been put into tight inner and outer loops for continuous operation and observance of clock-by-clock execution of the instructions and program we develop for it. In this process, the need to create some new instructions for the machine has arisen. A conditional OP code (DCBZ), outputs to the display, and branch instructions are used. The object is to invent all of these as we proceed, while describing a development environment that includes PROM programming (or *PROM burning,* in the vernacular). It may seem strange at first that a desired body of software has given rise to the requirements of an instruction set. While this body of software is not significant in the larger scheme of things, the priorities make good sense. Software goals should guide instruction-set as well as hardware-design activities.

Since a single set of macros includes more than the above instructions, we specify Macro Set 0 in Table 6-4.

Table 6-4
Macros Selected for Macro Set 0

Macro		*Op Code*	*CSTR Address (Hex)*
IF		–	*0H*
INM	*adr*	*000*	*10H*
MVI	*R0,val*	*001*	*20H*
BRU	*adr*	*010*	*30H*
OUT	*R0*	*011*	*40H*
DCBZ	*adr*	*100*	*50H*
LDR0	*adr*	*101*	*60H*
STR0	*adr*	*110*	*70H*
ADD	*adr*	*111*	*80H*
X		*1XXX*	*90H*

We have rounded out the macros of Macro Set 0 to include loading and storing R0, addition, input to memory (INM) from the switches, and the indexed-addressing macro (X). The microcoding details of each are presented shortly. First, let us look at the microcoded form sheet of Demonstration Program 0 (PROG0), in Figure 6-25.

Figure 6-25 displays the pertinent symbolic entries for the demonstration program to be invoked when we download PROG0. (The ALU field contains the actual binary values.) As previously noted, line 0 initializes the PC at the value 1 and is not a part of the program that will be demonstrated. The actual program steps start on line 1, with the MVI instruction. The succeeding ALU-field entries contain OP codes, addresses, or values in accordance with the purposes of the program. Some points are worth comment. Line A is the last line of the actual program. Here, the NEXT address field contains a 0 entry, which causes the *download procedure to loop back to line zero* at this point, so that we do not have to laboriously count the number of times we press the clock button during download (assuming we press it more than 11 times). All entries below line A are don't-cares.

Also observe that the ALU field of line A contains the value 0001. This value is the branch address of the BRU instruction on the line above. This guarantees that the program will repeatedly loop back to symbolic location START and run continuously, to make it easy to observe the details of its operation. On line 7 of the form, the CF flag is invoked in the BC field, to enable the second half of the macro space to

CSTR ADDRESS: 270H			ASSIGNED OP-CODE: –							
LINE NO.	MACRO: PROG0 ADR. CPDL DESCRIPTION: DOWN-LOAD PROG 0	NEXT ADDRESS	BRCH CTRL	CLR MOD	DBUS SOURCE FIELD	STATE	ALU FIELD		SINK CONTROL FIELD	
							ALU FUNC.	CS	R ARRAY	LOAD ENABLES
0	PC<1, LFL *(INIT. PC)*	1	L	L	X	X	0000 (A pl 1)	L	WPC	F L
1	MVI R0, U *(OP-CODE)*	2	L	L	X	X	0001	X	R0	M
2	VALUE *(0F H)*	3	L	L	X	X	0011	X	R0	M
3	OUT R0	4	L	L	X	X	0011	X	R0	M
4	DCBZ R0, ADR *(OP-CODE)*	5	L	L	X	X	0100	X	R0	M
5	ADR *(08H)*	6	L	L	X	X	1000	X	R0	M
6	BRU ADR *(OP-CODE)*	7	L	L	X	X	0010	X	R0	M
7	ADR *(03H)*	8	CF	L	X	X	0011	X	R0	M
8	OUT R0 *(OP-CODE)*	9	CF	L	X	X	0011	X	R0	M
9	BRU ADR *(OP-CODE)*	A	CF	L	X	X	0010	X	R0	M
A	ADR *(01H)*	0	L	L	X	X	0001	X	R0	M
B		X	X	X	X	X	X	X	X	X
C		X	X	X	X	X	X	X	X	X
D		X	X	X	X	X	X	X	X	X
E		X	X	X	X	X	X	X	X	X
F		X	X	X	X	X	X	X	X	X
CPDL NOTATIONS		CSTR2				CSTR1			CSTR0	

Figure 6-25
Microprogram Symbolic Coding Form: Demonstration Program 0 (PROG 0)

	HEX STARTING ADDRESS	MACRO MNEMONIC	MACRO DESCRIPTION
MACRO SET 0	00	IF	INSTRUCTION FETCH
	10	INPM	M[OA]<:*(SW)*
	20	MVI	R0<:M[OA]
	30	BRU	PC<:*(OA)*
	40	OUT	DISP<:*(SW)*
	50	DCBZ	R0<:R0 mi 1, IF R0=0 THEN PC:<(OA)
	60	LDR0	R0<:M{OA}
	70	STR0	M{OA}<:(R0)
	80	ADD	R0<:M{OA} pl (R0)
	90	X	INDEXED ADDRESS MODIFICATION
MACRO SET 1	A0		
	B0		
	C0		
	D0		
	E0		
	F0		
	100		
	110		
	120	I	INDIRECT ADDRESS MODIFICATION
	130		
	140		
	150		
	160		
	170		
	230		
	240		
	250		
	260		
	270	PROG0	LOOPING PROG USING DCBZ

NOTE: HIGH ORDER HEX DIGITS ARE BA FROM MROM.

Figure 6-26
Macro Assignment Map:
Macro Set 0 and PROG0

be used. Corresponding to this, line A contains an L in this field, to cause the program to return to line 1 (of the first half of the macro space) during operation. Finally, the flags were loaded on line 0, to set the CF high for the uses just described. On all other relevant lines, memory (M) is to be loaded during the download process. This takes the program in CSTR's ALU field and installs it into memory for subsequent operation.

Figure 6-26 summarizes much of the preceding by presenting the CSTR starting-address assignments for all the macros of Macro Set 0 (MACS0), as well as for that of download PROG0. It will be helpful when we wish to determine the values to be burned into MROM and to identify the locations in CSTR where the macros are to be placed. Figures 6-27 through 6-36 present the complete electrical coding forms for each macro of Macro Set 0. For ease of interpretation, entries on these forms are in terms of the truth-value terms of 1 and 0. Active-high logic conventions are specified for their translation into the electrical levels of H and L. Industrial PROM programming often provides these options, wherein the user specifies whether the truth values of the coding sheets are to apply to active-high or active-low programming of a PROM. These assignment and coding sheets shall now be analyzed to determine the actual contents of the locations of MROM, CSTR2, CSTR1, and CSTR0 required for successful operation of STUPIDD.

The next step is to establish the contents of MROM. Keep in mind that we are dealing with only one macro set (MACS0) and one download program (PROG0) at this time. The system can hold four sets of macros and eight downloadable programs based on the use of one of the macro sets. The 2716 EPROM's will contain all high values after erasure. Don't-care entries on the coding forms and all unused locations will remain high after the programming of the PROM's. Table 6-5 presents the locations of MROM that are to be programmed for our current purposes, together with their contents. Hexadecimal values are used for naming locations and their contents. The use of Hex notation is customary practice.

Table 6-5
MROM Mappings for PROG0, Macro Set 0

Hexadecimal Address	*Hexadecimal Value*	*Remarks*
0H–7F	27	BA of PROG0
80H–77F	FF	*Not Used
780	01	BA of INM
781	02	BA of MVI
782	03	BA of BRU
783	04	BA of OUT
784	05	BA of DCBC
785	06	BA of LDR0
786	07	BA of STR0
787	08	BA of ADD
788–78F	09	BA of Index Adr.
790–78F	00	BA of IF
7A0–7FF	FF	*Not Used

**Some of the unused locations may be used for other macro and program-set mappings in MROM. See Figure 6-5.*

With reference to Figure 6-5, which contains the use map of MROM, note that the first 128 locations of MROM are to contain the BA of PROG0. Accordingly, the Hex value 27H is specified in Table 6-5 for these locations. This program's contents actually start at location 270H in CSTR. The first hexadecimal digit of this program specification and of all the macros defines the maximum space in CSTR that each may occupy. Therefore all block addresses consist of all the higher order Hex digits to the left of this first one.

The BA's of the macro sets reside in upper MROM locations. (See Figure 6-5.) MACS0 BA's start at location 780H. The locations and nature of their contents have been defined in accordance with the use map of Figure 6-2. This usage makes space to include only eight OP codes in the instruction set. If the user desires more instructions and is willing to sacrifice operand-address modification in the process, then, as suggested in Figure 6-5, the number of macros in a set may be expanded to sixteen. The observant reader will notice that MACS0 must be modified to use the indexed-addressing mode. Why is this so? A good first user project is to alter MACS0 so that the index macro can also be incorporated into a demonstration program. All macro sets need use only one IF macro in CSTR. The BA of the IF macro specified in MROM locations 7A0H . . 7BFH of MACS0 is therefore 00H, and it will be the same for all sets.

Table 6-6
Hexadecimal Contents of CSTR for Macro Set 0 and Download Program 0

	Hexadecimal Contents of CSTR .			
Hex Address	*CSTR2*	*CSTR1*	*CSTR0*	*Remarks*
000	27	EB	67	IF
001	44	80	FF	
002	05	7F	7B	
003–00F	FF	FF	FF	*
010	27	EB	67	INM
011	44	80	FF	
012	06	FF	78	
013–01F	FF	FF	FF	*
020	27	EB	67	MVI
021	44	80	FF	
022	05	FF	9B	
023–02F	FF	FF	FF	*
030	27	EB	6F	BRU
031	05	FF	FB	
032–03F	FF	FF	FF	*
040	27	EB	17	OUT
041	04	FF	7A	
042–04F	FF	FF	FF	*
050	27	EB	67	DCBZ
051	44	80	FF	
052	67	EB	17	
053	9C	BF	9D	
054	07	FF	7B	
055–05B	FF	FF	FF	*
05C	05	FF	FB	
05D–05F	FF	FF	FF	*
060	27	EB	67	LDR0
061	44	80	FF	
062	65	FF	6F	
063	05	FF	9B	

Once we have decided where to place control-system values into CSTR, it is time to examine their contents. The contents of CSTR are presented in Table 6-6. The entries were taken from the individual coding forms of Figures 6-27 through 6-36, translated from the original binary into hexadecimal values. Each CSTR location used by a macro is divided into three fields: CSTR2, CSTR1, and CSTR0. These are straightforward conversions of the fields of the coding sheets from which they were taken. Of greater significance now is the columnar organization of the values for CSTR2, CSTR1, and CSTR0. Each column contains the information to be programmed into the corresponding PROM. This means that we have reordered the information of the macro-coding sheets to obtain the string of Hex values to be burned into each EPROM in CSTR.

The software support provided with the AMS prom programmer takes a file of the proper form, already residing in memory, and uses it to program the PROM. Each column of Table 6-6 may be used to create separate files for the EPROM's of CSTR. The process may then be run on the IBM PC or XT to program these PROM's. Let us say that

Table 6-6 (cont.)

	Hexadecimal Contents of CSTR			
Hex Address	*CSTR2*	*CSTR1*	*CSTR0*	*Remarks*
064–06F	FF	FF	FF	*
070	27	EB	67	STR0
071	44	80	FF	
072	65	FF	6F	
073	87	EB	17	
074	04	FF	78	
075–07F	FF	FF	FF	*
080	27	EB	67	ADD
081	44	80	FF	
082	65	FF	6F	
083	85	A7	15	
084	04	FF	9B	
085–08F	FF	FF	FF	*
090	27	EB	67	INDEX
091	44	80	FF	
092	61	A7	2F	
093–26F	FF	FF	FF	*
270	23	80	FD	PROG1
271	43	C7	1C	Loops Using DCBZ
272	63	FF	1C	
273	83	CF	1C	
274	A3	D3	1C	
275	C3	E3	1C	
276	E3	CB	1C	
277	0B	CF	1C	
278	2B	CF	1C	
279	4B	CB	1C	
27A	03	C7	1C	
27B–27F	FF	FF	FF	*

**Note: The asterisk denotes locations and ranges of locations that are not used. An erased PROM location will remain high (FF) if not programmed.*

this has been done and that PROG0 is to be demonstrated. After programming, the PROM's are reinserted into their respective sockets in STUPIDD. The display select may be set to display the contents of the OUT register, to check gross overall program performance. The program is downloaded and run. The user may encounter a few surprises in the demonstration phase of PROG0—or any other demonstration program. The first value displayed in OUT has nothing to do with the program. It is the random value that happens to be present in the OUT register after downloading. The number of clock pulses to be manually administered is the next surprise. The operation actually consumes more pulses than one would expect, since we must cycle through the IF phase again each time the EX phase of an instruction is completed. We must understand and track every step of the total system's operation if it is to be adequately explained to an observer.

The students shown in Figure 6-24 are currently developing a more sophisticated approach, to improve the PROM-programming environment. This consists of a user-friendly menu-driven microcode Assembler based on the computer-aided application of the easy-to-use symbolic coding forms. Such a project is an excellent exercise for the more software-oriented reader who wants to apply software expertise in working close to the hardware.

This approach establishes the environment in which a user may select, from the initial menu, the choice of programming the entire control store in one sitting. After this menu option is selected, the symbolic coding sheet form of a macro is displayed on the screen. The user may now enter the easy-to-use symbolic form into the fields of each line of microcode. When the macro is complete and edited to the user's satisfaction, the user assigns a CSTR address to it, before going on to another macro or terminating the session. The final task that can take place under user control is automatically translating this macro into the Hex programming form and placing the Hex values into their respective string files, at their proper locations, for the future programming of the EPROM's of CSTR. If a complete string file for a PROM has already been created, the user may now choose to program it, by menu selection. This choice simply invokes the software that came with the PROM programmer in the first place.

At this point we are in total control of the processor system: We can conveniently create its instruction sets and incorporate them into demonstration programs. We now realize that we can be better than this or any other processor we may encounter in the future, for we have touched the soul of our own machine.

CSTR ADDRESS: 0H		ASSIGNED OP-CODE: –																							
LINE NO.	MACRO: IF CPDL DESCRIPTION: INSTRUCTION FETCH	NEXT ADDRESS			BRCH CTRL		MOD	DBUS SOURCE FIELD		STATE	ALU FIELD							SINK CONTROL FIELD							
											ALU FUNC.					CS		R ARRAY			LOAD ENABLES				
		N2	N1	N0	BC1	BC0	●CLM	E1	E0		MD	S3	S2	S1	S0	CS1	CS0	W@R	RS1	RS0	●LMAR	●LT	●LIR	L1	L0
0	MAR, T <: PC	0	0	1	0	0	1	1	1	1	1	1	0	1	0	1	1	0	1	1	0	0	1	1	1
1	PC <: T pl 1	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
2	IR <: M{MAR}, >: EX	0	0	0	0	0	1	0	1	0	1	1	1	1	1	1	1	0	1	1	1	1	0	1	1
3																									
4																									
5																									
6																									
7																									
8	0																								
9	1																								
A	2																								
B	3																								
C	4																								
D	5																								
E	6																								
F	7																								
CPDL NOTATIONS		CSTR2									CSTR1							CSTR0							

Figure 6-27 Microprogram Electrical Level Coding Form: IF Macro

CSTR ADDRESS: 10H

ASSIGNED OP CODE: 0

LINE NO.	MACRO: INM	NEXT ADDRESS			BRCH CTRL		MOD	DBUS SOURCE FIELD		STATE	ALU FIELD							SINK CONTROL FIELD							
	CPDL DESCRIPTION: M{OA}<:IN										ALU FUNC.					CS		R ARRAY			LOAD ENABLES				
		N2	N1	N0	BC1	BC0	CLM●	E1	E0		MD	S3	S2	S1	S0	CS1	CS0	W@R	RS1	RS0	LMAR●	LT●	LIR●	L1	L0
0	MAR,T<:PC	0	0	1	0	0	1	1	1	1	1	1	0	1	0	1	1	0	1	1	0	0	1	1	1
1	PC<:T pl 1	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
2	M{MAR}<:IN,>:IF	0	0	0	0	0	1	1	0	1	1	1	1	1	1	1	1	0	1	1	1	1	0	0	0
3																									
4																									
5																									
6																									
7																									
8																									
9																									
A																									
B																									
C																									
D																									
E																									
F																									
CPDL NOTATIONS		CSTR2								CSTR1								CSTR0							

Figure 6-28 Microprogram Electrical Level Coding Form: INM Macro

CSTR ADDRESS: 20H									ASSIGNED OP-CODE: 0001																
LINE NO.	MACRO: MVI	NEXT ADDRESS			BRCH CTRL		MOD	DBUS SOURCE FIELD		STATE	ALU FIELD							SINK CONTROL FIELD							
	RO<: M{PC pl 1,}										ALU FUNC.					CS		R ARRAY			LOAD ENABLES				
		N2	N1	N0	BC1	BC0	CLR	E1	E0	STATE	MD	S3	S2	S1	S0	CS1	CS0	W@R	RS1	RS0	LMAR	LT	LIR	L1	L0
0	MAR, T <: PC	0	0	1	0	0	1	1	1	1	1	1	0	1	0	1	1	0	1	1	0	0	1	1	1
1	PC <: T pl 1	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
2	RO <: M{MAR}	0	0	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	0	0	1	1	0	1	1
3																									
4																									
5																									
6																									
7																									
8																									
9																									
A																									
B																									
C																									
D																									
E																									
F																									
CPDL NOTATIONS		CSTR2								CSTR1								CSTR0							

Figure 6-29 Microprogram Electrical Level Coding Form: MVI Macro

CSTR ADDRESS: 30H	ASSIGNED OP-CODE: 0010

LINE NO.	MACRO: BRU	NEXT ADDRESS			BRCH CTRL		MOD	DBUS SOURCE FIELD		STATE	ALU FIELD							SINK CONTROL FIELD								
											ALU FUNC.					CS		R ARRAY			LOAD ENABLES				
	CPDL DESCRIPTION: PC <: M{PC pl 1}	N2	N1	N0	BC1	BC0	CLM	E1	E0		MD	S3	S2	S1	S0	CS1	CS0	W@R	RS1	RS0	LMAR	LT	LIR	L1	L0
0	MAR <: PC	0	0	1	0	0	1	1	1	1	1	1	0	1	0	1	1	0	1	1	0	1	1	1	1
1	PC <: M{MAR}	0	0	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
2																									
3																									
4																									
5																									
6																									
7																									
8	0																								
9	1																								
A	2																								
B	3																								
C	4																								
D	5																								
E	6																								
F	7																								
CPDL NOTATIONS		CSTR2								CSTR1								CSTR0							

Figure 6-30 Microprogram Symbolic Coding Form: BRU Macro

CSTR ADDRESS: 40H								ASSIGNED OP-CODE: 0011																	
LINE NO.	MACRO: OUT	NEXT ADDRESS			BRCH CTRL		MOD	DBUS SOURCE FIELD		STATE	ALU FIELD							SINK CONTROL FIELD							
											ALU FUNC.					CS		R ARRAY			LOAD ENABLES				
	CPDL DESCRIPTION OUT <: R0	N2	N1	N0	BC1	BC0	CLM	E1	E0		MD	S3	S2	S1	S0	CS1	CS0	W@R	RS1	RS0	LMAR	LT	LIR	L1	L0
0	T<:R0	0	0	1	0	0	1	1	1	1	1	1	0	1	0	1	1	0	0	0	1	0	1	1	0
1	OUT<:T,>:IF	0	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1	0	1	1	1	1	0	1	0
2																									
3																									
4																									
5																									
6																									
7																									
8	0																								
9	1																								
A	2																								
B	3																								
C	4																								
D	5																								
E	6																								
F	7																								
CPDL NOTATIONS		CSTR2								CSTR1								CSTR0							

Figure 6-31 Microprogram Electrical-Level Coding Form: OUT Macro

CSTR ADDRESS: 50H	ASSIGNED OP-CODE: 0100

LINE NO.	MACRO: DCBZ	NEXT ADDRESS			BRCH CTRL		MOD	DBUS SOURCE FIELD		STATE	ALU FIELD							SINK CONTROL FIELD							
	CPDL DESCRIPTION; R0 <: R0 mi 1, LFL, If R0 = 0 Then PC <: 0A, Else >: IF										ALU FUNC.					CS		R ARRAY			LOAD ENABLES				
		N2	N1	N0	BC1	BC0	CLR	E1	E0		MD	S3	S2	S1	S0	CS1	CS0	W@R	RS1	RS0	LMAR	LT	LIR	L1	L0
0	MAR, T <: PC	0	0	1	0	0	1	1	1	1	1	1	0	1	0	1	1	0	1	1	0	0	1	1	1
1	PC <: T pl 1	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
2	T <: R0	0	1	1	0	0	1	1	1	1	1	1	0	1	0	1	1	0	0	0	1	0	1	1	1
3	RO <: T mi 1, LFL	1	0	0	1	1	1	0	0	1	0	1	1	1	1	1	1	1	0	0	1	1	1	0	1
4	>: IF	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	0	1	1
5																									
6																									
7																									
8	0																								
9	1																								
A	2																								
B	3																								
C	4 PC <: M {MAR}, >: IF	0	0	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
D	5																								
E	6																								
F	7																								
CPDL NOTATIONS		CSTR2									CSTR1								CSTR0						

Figure 6-32 Microprogram Electrical-Level Coding Form: DCBZ Macro

CSTR ADDRESS: 60H

ASSIGNED OP-CODE: 0101

LINE NO	MACRO: LDR0	NEXT ADDRESS			BRCH CTRL		MOD	DBUS SOURCE FIELD		STATE	ALU FIELD							SINK CONTROL FIELD							
	CPDL DESCRIPTION: R0 <: M{OA}										ALU FUNC.					CS		R ARRAY			LOAD ENABLES				
		N2	N1	N0	BC1	BC0	●CLM	E1	E0		MD	S3	S2	S1	S0	CS1	CS0	W@R	RS1	RS0	●LMAR	●LT	●LIR	L1	L0
0	T, MAR <: PC	0	0	1	0	0	1	1	1	1	1	1	0	1	0	1	1	0	1	1	0	0	1	1	1
1	PC <: T pl 1	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
2	MAR <: M{MAR}	0	1	1	0	0	1	0	1	1	1	1	1	1	1	1	1	0	1	1	0	1	1	1	1
3	R0 <: M{MAR}, >: IF	0	0	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	0	0	1	1	0	1	1
4																									
5																									
6																									
7																									
8	0																								
9	1																								
A	2																								
B	3																								
C	4																								
D	5																								
E	6																								
F	7																								
CPDL NOTATIONS		CSTR2								CSTR1								CSTR0							

Figure 6-33 Microprogram Electrical-Level Coding Form: LDR0 Macro

CSTR ADDRESS: 70		ASSIGNED OP-CODE: 0110																							
LINE NO.	MACRO: STR0	NEXT ADDRESS			BRCH CTRL		MOD	DBUS SOURCE FIELD		STATE	ALU FIELD							SINK CONTROL FIELD							
	CPDL DESCRIPTION: M{OA} <: R0										ALU FUNC.					CS		R ARRAY			LOAD ENABLES				
		N2	N1	N0	BC1	BC0	CLM	E1	E0		MD	S3	S2	S1	S0	CS1	CS0	W@R	RS1	RS0	LMAR	LT	LIR	L1	L0
0	T, MAR <: PC	0	0	1	0	0	1	1	1	1	1	1	0	1	0	1	1	0	1	1	0	0	1	1	1
1	PC <: T pl 1	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
2	MAR <: M{MAR}	0	1	1	0	0	1	0	1	1	1	1	1	1	1	1	1	0	1	1	0	1	1	1	1
3	T <: R0	1	0	0	0	0	1	1	1	1	1	1	0	1	0	1	1	0	0	0	1	0	1	1	1
4	M{MAR} <: T, >: IF	0	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1	0	1	1	1	1	0	0	0
5																									
6																									
7																									
8																									
9																									
A																									
B																									
C																									
D																									
E																									
F																									
CPDL NOTATIONS		CSTR2								CSTR1								CSTR0							

Figure 6-34 Microprogram Electrical-Level Coding Form: STR0 Macro

CSTR ADDRESS: 80H								ASSIGNED OP-CODE: 0111																	
LINE NO.	MACRO: ADD	NEXT ADDRESS			BRCH CTRL		MOD	DBUS SOURCE FIELD		STATE	ALU FIELD							SINK CONTROL FIELD							
	CPDL DESCRIPTION: R0 pl M{OA}										ALU FUNC.					CS		R ARRAY			LOAD ENABLES				
		N2	N1	N0	BC1	BC0	CLM	E1	E0		MD	S3	S2	S1	S0	CS1	CS0	W@R	RS1	RS0	LMAR	LT	LIR	L1	L0
0	T, MAR <: PC	0	0	1	0	0	1	1	1	1	1	1	0	1	0	1	1	0	1	1	0	0	1	1	1
1	PC <: T pl 1	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
2	MAR <: M{MAR}	0	1	1	0	0	1	0	1	1	1	1	1	1	1	1	1	0	1	1	0	1	1	1	1
3	T <: R0 pl M{MAR}, LFL	1	0	0	0	0	1	0	1	1	0	1	0	0	1	1	1	0	0	0	1	0	1	0	1
4	R0 <: T, >: IF	0	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1	1	0	0	1	1	0	1.	1
5																									
6																									
7																									
8	0																								
9	1																								
A	2																								
B	3																								
C	4																								
D	5																								
E	6																								
F	7																								
CPDL NOTATIONS		CSTR2								CSTR1								CSTR0							

Figure 6-35 Microprogram Electrical-Level Coding Form: ADD Macro

CSTR ADDRESS: 90H

ASSIGNED OP-CODE: 1XXX

LINE NO	MACRO: INDEX ADR.	NEXT ADDRESS			BRCH CTRL		MOD	DBUS SOURCE FIELD			ALU FIELD							SINK CONTROL FIELD							
											ALU FUNC.					CS		R ARRAY			LOAD ENABLES				
	CPDL DESCRIPTION: EA <: M{OA} pl R1	N2	N1	N0	BC1	BC0	CLM	E1	E0	STATE	MD	S3	S2	S1	S0	CS1	CS0	W@R	RS1	RS0	LMAR	LT	LIR	L1	L0
0	MAR, T <: PC	0	0	1	0	0	1	1	1	1	1	1	0	1	0	1	1	0	1	1	0	0	1	1	1
1	PC <: T pl 1	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
2	MAR <: M{MAR} pl R1, CLM	0	1	1	0	0	0	0	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1	1
3																									
4																									
5																									
6																									
7																									
8	0																								
9	1																								
A	2																								
B	3																								
C	4																								
D	5																								
E	6																								
F	7																								
CPDL NOTATIONS		CSTR2								CSTR1								CSTR0							

Figure 6-36 Microprogram Electrical-Level Coding: Index ADR Macro

BIBLIOGRAPHY

Build an Am2900 Microcomputer. Vol. I–IX. Sunnyvale, California: Advanced Micro Devices, Inc., 1978.

Dietmeyer, Donald L. *Logic Design of Digital Systems.* Boston: Allyn and Bacon, 1978.

Husson, Samir S. *Microprogramming Principles and Practices.* Englewood Cliffs, New Jersey: Prentice-Hall, 1970.

Microprogram Design with the 2900 Family. Sunnyvale, California: Advanced Micro Devices, Inc., 1978.

Wilkes, M.V. "The Best Way to Design an Automatic Calculating Machine." Manchester University Computer Inaugural Conference, July 1951.

Wilkes, M.V., and Stringer, J.B. "Microprogramming and the Design of the Control Circuits in an Electronic Digital Computer." Proc. Cambridge Phil. Soc., Pt. 2, Vol. 49, pp. 230–38, April 1953.

Yaohan Chu. *Computer Organization and Microprogramming.* Englewood Cliffs, New Jersey: Prentice-Hall, 1972.

PROBLEMS

Note: First microcode the following problems on the appropriate coding form, then demonstrate their operation on the construction project.

1. Use the electrical-level (H or L) coding form for this problem. Microcode the instruction CALL, OA. For this instruction, use R2 as the system stack pointer (SP). The CPDL description of the results is as follows:

   SP <: SP mi 1
   M{SP} <: PC
   PC <: M{OA}

   Hint: Carefully consider the order in which one must adjust the PC and SP in STUPIDD's architecture. Recall that the PC must point to the next OP upon execution of a return.
2. Use the symbolic coding form for the following problem. This is an address-modification type of algorithm. The MOD bit on STUPIDD is to be used for indexing with automatic preincrementation of the index register, R2. That is, R2 is to be incremented *before* the effective address is formed. Name this macro $+X$. As described in the text, fall-through shall be in relation to the ADD macro.
3. Develop the microcode for the instruction *rotate left through carry* (RLC), using the electrical-level (H or L) coding form. The instruction operates as follows:

   CF <: R0[3]
   R0[i] <: R0[i mi 1], i=3,2,1

R0[0] <: CF

4. Use the symbolic coding form for this problem. Develop the microcode for the instruction ReTurn from Subroutine (RTS). Register R2 is to be used as a stack pointer (SP). The instruction operates as follows:

   PC <: M{SP}

   SP <: SP pl 1

5. Use the symbolic coding form for this problem. Develop the microcode for the instruction PUSH R0. Register R2 is to serve as the stack pointer (SP). The instruction operates as follows:

   SP <: SP mi 1

   M{SP} <: R0

6. Use the symbolic coding form for this problem. Develop the microcode for the instruction *branch if memory equal to R0* (BMEQ, R0). This is a three-word instruction. The first memory word is the OP code, BMEQ R0. The next memory word specifies the address of the operand (OA1) that is to be compared with the contents of R0. The third memory word of the instruction, which we can call OA2, contains the branch address to be used if M{OA1} = R0. Just be careful to be sure that the PC is pointing to one of the two possible next OP codes when the IF major state is reached.
7. Write the symbolic microcode for STUPIDD for the instruction that POPs data from the stack. Use R0 as the implied register through which all stack data communications occur.
8. Create a macro, using the symbolic coding form, that compares the contents of R0 with the contents of memory specified in the OA1 field. IF (M{OA1} ≥ R0), *then* jump to the location specified in OA2, *else* fetch the next instruction in sequence.
9. Write the symbolic microcode for the instruction that increments R0 and branches to the operand address if a carry has occurred. Call it INBC.
10. Explain why the contents of STUPIDD's instruction register have no effect on the instruction fetch during the IF phase of operation.
11. Develop a macro for clearing the carry flag (CF) and the sign flag (SF). Do the same for the setting of these flags. Discuss differences between the activity levels of these flags and the macro's effects on the algorithms. Given STUPIDD's architecture, is it possible to affect one flag and not the other? Discuss, commenting on desirable changes in architecture.
12. If the MAR, the T, and the IR registers were to be loaded only one at a time, instead of in parallel, as the microcoding forms now permit, how could you reduce the size of STUPIDD's control word by one bit? Do not use any integrated circuits that are not used in the existing design.
13. Develop new methods for performing both conditional and unconditional branches between the upper and lower halves of

the macro-block space. Consider all four types of cases for each type of branch.

14. This problem is a thought-provoking challenge. Unsigned multiplication may be performed by successive addition. The results occupy two registers R1 and R2. Assume that the multiplier has already been placed in R0. The multiplier is decremented and tested for the nonzero condition. If it is not zero, then the multiplicand is added to the accumulator registers. The location of the multiplicand is specified in the OA field of the instruction word. The initial contents of R1 and R2 are unknown. Microcode this algorithm, using the symbolic coding sheet. *Hint:* First solve Problem 13.

15. What would you do to prevent unimplemented OP codes from affecting normal, useful operation of a processor system?

16. Create a macro for the indirect addressing of operands. Call it I. First draw the memory map of the chain of pointers, starting with the OA field of the instruction. Define what is contained in the locations that are pointed to. Submit your memory map with the solution. Remember that the PC also is a pointer into memory. Assume that the instruction we fall through to after the EA calculation is the ADD macro, shown in Figure 5-13.

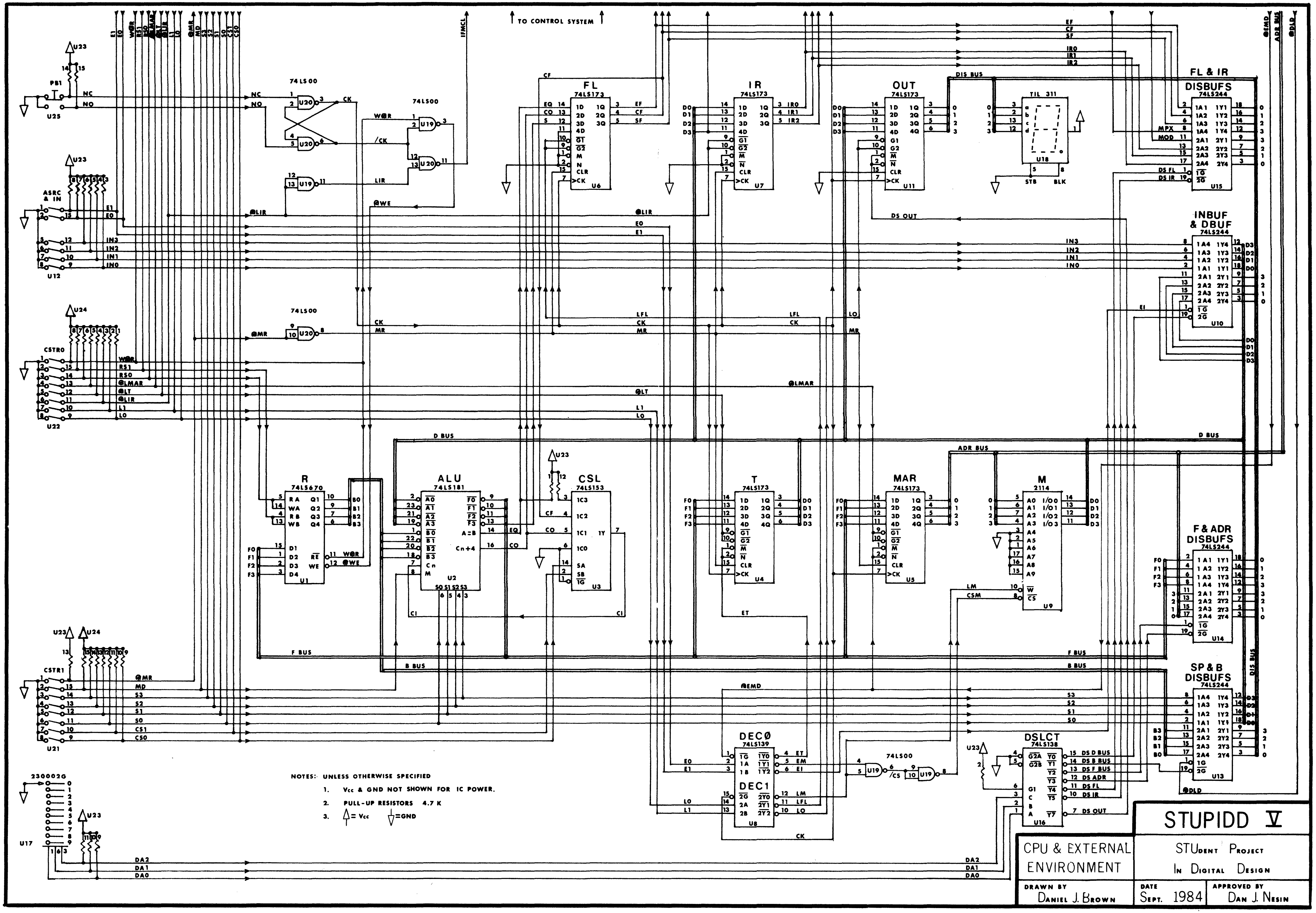
17. Create a macro for the *branch relative* (BRR) instruction. This is a two-word instruction, where the first word is the OP field and the second is the *signed* relative displacement to the location we are to branch to. That is, forward as well as backward branches can occur—within the range of ±7 locations from the current location. The PDP-11 and other computers perform relative branches. Their hardware and assembly-language manuals provide good reference material on the operation of this type of instruction.

18. Develop an algorithm for unsigned integer division. The numerator is to be an eight-bit quantity already residing in two registers of the array R. The denominator's location in primary memory is specified by the OA field of the instruction word. Specify how you plan to handle the quotient and the remainder, define the algorithm, and give all other assumptions you plan to use.

19. Create a macro, on the electrical-level coding format, for inputting data from STUPIDD's input switches, that uses a handshake. That is, the MSB of the in switches is to be used as an indication that data is ready when this line is high. The other three switches are data. The algorithm is to loop until data is ready. When the data is ready, it is to be placed into the T register with the MSB changed to a low (the ready signal is stripped out).

20. Use the symbolic coding form for this problem. Write the macro for the instruction that decrements R1 and jumps to the address specified in the OA field of the instruction *if* the results are negative. Call this one DJR1.

READY REFERENCE OF KEY TOPICS



NOTES: UNLESS OTHERWISE SPECIFIED

1. Vcc & GND NOT SHOWN FOR IC POWER.
2. PULL-UP RESISTORS 4.7 K
3. ⇧= Vcc ⇩=GND

STUPIDD V

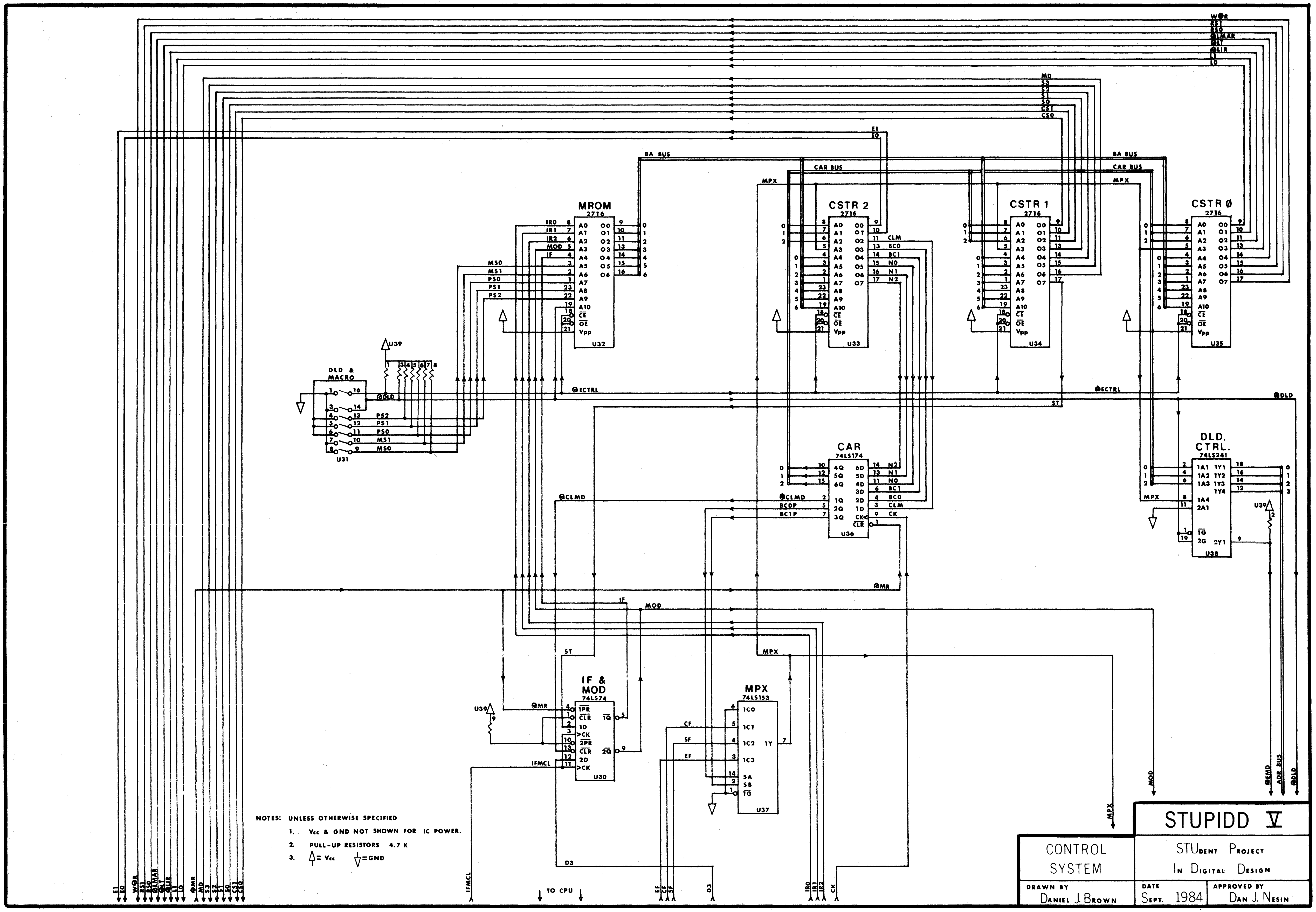
CPU & EXTERNAL ENVIRONMENT

STUdent Project In Digital Design

DRAWN BY Daniel J. Brown

DATE Sept. 1984

APPROVED BY Dan J. Nesin



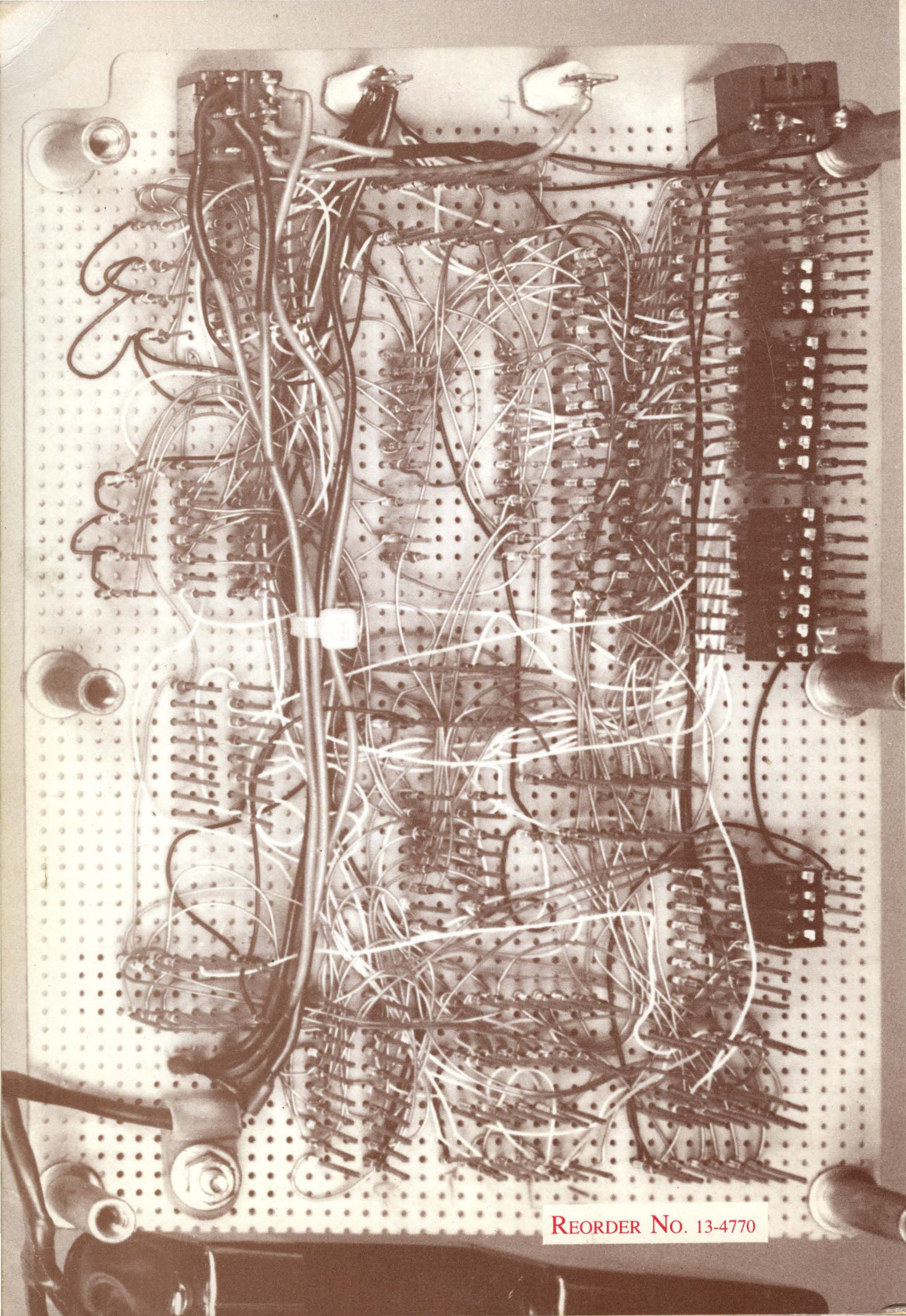
NOTES: UNLESS OTHERWISE SPECIFIED

1. Vcc & GND NOT SHOWN FOR IC POWER.

2. PULL-UP RESISTORS 4.7 K

3. △= Vcc ▽=GND

CONTROL SYSTEM	STUPIDD V STUDENT PROJECT IN DIGITAL DESIGN	
DRAWN BY DANIEL J. BROWN	DATE SEPT. 1984	APPROVED BY DAN J. NESIN



REORDER NO. 13-4770