

INTERNATIONAL EDITION

COMPUTER SYSTEM ARCHITECTURE

THIRD EDITION



M. Morris Mano

Preface

This book deals with computer architecture as well as computer organization and design. Computer architecture is concerned with the structure and behavior of the various functional modules of the computer and how they interact to provide the processing needs of the user. Computer organization is concerned with the way the hardware components are connected together to form a computer system. Computer design is concerned with the development of the hardware for the computer taking into consideration a given set of specifications.

The book provides the basic knowledge necessary to understand the hardware operation of digital computers and covers the three subjects associated with computer hardware. Chapters 1 through 4 present the various digital components used in the organization and design of digital computers. Chapters 5 through 7 show the detailed steps that a designer must go through in order to design an elementary basic computer. Chapters 8 through 10 deal with the organization and architecture of the central processing unit. Chapters 11 and 12 present the organization and architecture of input-output and memory. Chapter 13 introduces the concept of multiprocessing. The plan of the book is to present the simpler material first and introduce the more advanced subjects later. Thus, the first seven chapters cover material needed for the basic understanding of computer organization, design, and programming of a simple digital computer. The last six chapters present the organization and architecture of the separate functional units of the digital computer with an emphasis on more advanced topics.

The material in the third edition is organized in the same manner as in the second edition and many of the features remain the same. The third edition, however, offers several improvements over the second edition. All chapters except two (6 and 10) have been completely revised to bring the material up to date and to clarify the presentation. Two new chapters were added: chapter 9 on pipeline and vector processing, and chapter 13 on multiprocessors. Two sections deal with the reduced instruction set computer (RISC). Chapter 5 has been revised completely to simplify and clarify the design of the basic computer. New problems have been formulated for eleven of the thirteen chapters.

The physical organization of a particular computer including its registers,

the data flow, the microoperations, and control functions can be described symbolically by means of a hardware description language. In this book we develop a simple register transfer language and use it to specify various computer operations in a concise and precise manner. The relation of the register transfer language to the hardware organization and design of digital computers is fully explained.

The book does not assume prior knowledge of computer hardware and the material can be understood without the need of prerequisites. However, some experience in assembly language programming with a microcomputer will make the material easier to understand. Chapters 1 through 3 can be skipped if the reader is familiar with digital logic design.

The following is a brief description of the subjects that are covered in each chapter with an emphasis on the revisions that were made in the third edition.

Chapter 1 introduces the fundamental knowledge needed for the design of digital systems constructed with individual gates and flip-flops. It covers Boolean algebra, combinational circuits, and sequential circuits. This provides the necessary background for understanding the digital circuits to be presented.

Chapter 2 explains in detail the logical operation of the most common standard digital components. It includes decoders, multiplexers, registers, counters, and memories. These digital components are used as building blocks for the design of larger units in the chapters that follow.

Chapter 3 shows how the various data types found in digital computers are represented in binary form in computer registers. Emphasis is on the representation of numbers employed in arithmetic operations, and on the binary coding of symbols used in data processing.

Chapter 4 introduces a register transfer language and shows how it is used to express microoperations in symbolic form. Symbols are defined for arithmetic, logic, and shift microoperations. A composite arithmetic logic shift unit is developed to show the hardware design of the most common microoperations.

Chapter 5 presents the organization and design of a basic digital computer. Although the computer is simple compared to commercial computers, it nevertheless encompasses enough functional capabilities to demonstrate the power of a stored program general purpose device. Register transfer language is used to describe the internal operation of the computer and to specify the requirements for its design. The basic computer uses the same set of instructions as in the second edition but its hardware organization and design has been completely revised. By going through the detailed steps of the design presented in this chapter, the student will be able to understand the inner workings of digital computers.

Chapter 6 utilizes the twenty five instructions of the basic computer to illustrate techniques used in assembly language programming. Programming examples are presented for a number of data processing tasks. The relationship

between binary programs and symbolic code is explained by examples. The basic operations of an assembler are presented to show the translation from symbolic code to an equivalent binary program.

Chapter 7 introduces the concept of microprogramming. A specific microprogrammed control unit is developed to show by example how to write microcode for a typical set of instructions. The design of the control unit is carried-out in detail including the hardware for the microprogram sequencer.

Chapter 8 deals with the central processing unit (CPU). An execution unit with common buses and an arithmetic logic unit is developed to show the general register organization of a typical CPU. The operation of a memory stack is explained and some of its applications are demonstrated. Various instruction formats are illustrated together with a variety of addressing modes. The most common instructions found in computers are enumerated with an explanation of their function. The last section introduces the reduced instruction set computer (RISC) concept and discusses its characteristics and advantages.

Chapter 9 on pipeline and vector processing is a new chapter in the third edition. (The material on arithmetic operations from the second edition has been moved to Chapter 10.) The concept of pipelining is explained and the way it can speed-up processing is illustrated with several examples. Both arithmetic and instruction pipeline is considered. It is shown how RISC processors can achieve single-cycle instruction execution by using an efficient instruction pipeline together with the delayed load and delayed branch techniques. Vector processing is introduced and examples are shown of floating-point operations using pipeline procedures.

Chapter 10 presents arithmetic algorithms for addition, subtraction, multiplication, and division and shows the procedures for implementing them with digital hardware. Procedures are developed for signed-magnitude and signed-2's complement fixed-point numbers, for floating-point binary numbers, and for binary coded decimal (BCD) numbers. The algorithms are presented by means of flowcharts that use the register transfer language to specify the sequence of microoperations and control decisions required for their implementation.

Chapter 11 discusses the techniques that computers use to communicate with input and output devices. Interface units are presented to show the way that the processor interacts with external peripherals. The procedure for asynchronous transfer of either parallel or serial data is explained. Four modes of transfer are discussed: programmed I/O, interrupt initiated transfer, direct memory access, and the use of input-output processors. Specific examples illustrate procedures for serial data transmission.

Chapter 12 introduces the concept of memory hierarchy, composed of cache memory, main memory, and auxiliary memory such as magnetic disks. The organization and operation of associative memories is explained in detail. The concept of memory management is introduced through the presentation of the hardware requirements for a cache memory and a virtual memory system.

Chapter 13 presents the basic characteristics of multiprocessors. Various interconnection structures are presented. The need for interprocessor arbitration, communication, and synchronization is discussed. The cache coherence problem is explained together with some possible solutions.

Every chapter includes a set of problems and a list of references. Some of the problems serve as exercises for the material covered in the chapter. Others are of a more advanced nature and are intended to provide practice in solving problems associated with computer hardware architecture and design. A solutions manual is available for the instructor from the publisher.

The book is suitable for a course in computer hardware systems in an electrical engineering, computer engineering, or computer science department. Parts of the book can be used in a variety of ways: as a first course in computer hardware by covering Chapters 1 through 7; as a course in computer organization and design with previous knowledge of digital logic design by reviewing Chapter 4 and then covering chapters 5 through 13; as a course in computer organization and architecture that covers the five functional units of digital computers including control (Chapter 7), processing unit (Chapters 8 and 9), arithmetic operations (Chapter 10), input-output (Chapter 11), and memory (Chapter 12). The book is also suitable for self-study by engineers and scientists who need to acquire the basic knowledge of computer hardware architecture.

Acknowledgments

My thanks goes to those who reviewed the text: particularly Professor Thomas L. Casavant of the University of Iowa; Professor Murray R. Berkowitz of George Mason University; Professor Cem Ersoy of Brooklyn Polytechnic University; Professor Upkar Varshney of the University of Missouri, Kansas City; Professor Karan Watson of Texas A&M University, and Professor Scott F. Midkiff of the Virginia Polytechnic Institute.

M. Morris Mano

Contents

Preface

xv

CHAPTER ONE

Digital Logic Circuits

1

1-1	Digital Computers	1
1-2	Logic Gates	4
1-3	Boolean Algebra	7
	Complement of a Function	10
1-4	Map Simplification	11
	Product-of-Sums Simplification	14
	Don't-Care Conditions	16
1-5	Combinational Circuits	18
	Half-Adder	19
	Full-Adder	20
1-6	Flip-Flops	22
	SR Flip-Flop	22
	D Flip-Flop	23
	JK Flip-Flop	24
	T Flip-Flop	24
	Edge-Triggered Flip-Flops	25
	Excitation Tables	27
1-7	Sequential Circuits	28
	Flip-Flop Input Equations	28
	State Table	30
	State Diagram	31
	Design Example	32
	Design Procedure	36
	Problems	
	References	

C H A P T E R T W O

Digital Components 41

2-1	Integrated Circuits	41
2-2	Decoders	43
	NAND Gate Decoder	45
	Decoder Expansion	46
	Encoders	47
2-3	Multiplexers	48
2-4	Registers	50
	Register with Parallel Load	51
2-5	Shift Registers	53
	Bidirectional Shift Register with Parallel Load	53
2-6	Binary Counters	56
	Binary Counter with Parallel Load	58
2-7	Memory Unit	58
	Random-Access Memory	60
	Read-Only Memory	61
	Types of ROMs	62
	Problems	63
	References	65

C H A P T E R T H R E E

Data Representation 67

3-1	Data Types	67
	Number Systems	68
	Octal and Hexadecimal Numbers	69
	Decimal Representation	72
	Alphanumeric Representation	73
3-2	Complements	74
	($r-1$)'s Complement	75
	(r 's) Complement	75
	Subtraction of Unsigned Numbers	76
3-3	Fixed-Point Representation	77
	Integer Representation	78
	Arithmetic Addition	79
	Arithmetic Subtraction	80
	Overflow	80
	Decimal Fixed-Point Representation	81

3-4	Floating-Point Representation	83
3-5	Other Binary Codes	84
	Gray Code	84
	Other Decimal Codes	85
	Other Alphanumeric Codes	86
3-6	Error Detection Codes	87
	Problems	89
	References	91

CHAPTER FOUR

	Register Transfer and Microoperations	93
4-1	Register Transfer Language	93
4-2	Register Transfer	95
4-3	Bus and Memory Transfers	97
	Three-State Bus Buffers	100
	Memory Transfer	101
4-4	Arithmetic Microoperations	102
	Binary Adder	103
	Binary Adder-Subtractor	104
	Binary Incrementer	105
	Arithmetic Circuit	106
4-5	Logic Microoperations	108
	List of Logic Microoperations	109
	Hardware Implementation	111
	Some Applications	111
4-6	Shift Microoperations	114
	Hardware Implementation	115
4-7	Arithmetic Logic Shift Unit	116
	Problems	119
	References	122

CHAPTER FIVE

	Basic Computer Organization and Design	123
5-1	Instruction Codes	123
	Stored Program Organization	125
	Indirect Address	126

5-2	Computer Registers	127
	Common Bus System	129
5-3	Computer Instructions	132
	Instruction Set Completeness	134
5-4	Timing and Control	135
5-5	Instruction Cycle	139
	Fetch and Decode	139
	Determine the Type of Instruction	141
	Register-Reference Instructions	143
5-6	Memory-Reference Instructions	145
	AND to AC	145
	ADD to AC	146
	LDA: Load to AC	146
	STA: Store AC	147
	BUN: Branch Unconditionally	147
	BSA: Branch and Save Return Address	147
	ISZ: Increment and Skip If Zero	149
	Control Flowchart	149
5-7	Input-Output and Interrupt	150
	Input-Output Configuration	151
	Input-Output Instructions	152
	Program Interrupt	153
	Interrupt Cycle	156
5-8	Complete Computer Description	157
5-9	Design of Basic Computer	157
	Control Logic Gates	160
	Control of Registers and Memory	160
	Control of Single Flip-Flops	162
	Control of Common Bus	162
5-10	Design of Accumulator Logic	164
	Control of AC Register	165
	Adder and Logic Circuit	166
	Problems	167
	References	171

CHAPTER SIX

	Programming the Basic Computer	173
6-1	Introduction	173
6-2	Machine Language	174

6-3	Assembly Language	179
	<i>Rules of the Language</i> 179	
	<i>An Example</i> 181	
	<i>Translation to Binary</i> 182	
6-4	The Assembler	183
	<i>Representation of Symbolic Program</i> in Memory 184	
	<i>First Pass</i> 185	
	<i>Second Pass</i> 187	
6-5	Program Loops	190
6-6	Programming Arithmetic and Logic Operations	192
	<i>Multiplication Program</i> 193	
	<i>Double-Precision Addition</i> 196	
	<i>Logic Operations</i> 197	
	<i>Shift Operations</i> 197	
6-7	Subroutines	198
	<i>Subroutines Parameters and Data Linkage</i> 200	
6-8	Input-Output Programming	203
	<i>Character Manipulation</i> 204	
	<i>Program Interrupt</i> 205	
	Problems	208
	References	211

CHAPTER SEVEN

	Microprogrammed Control	213
7-1	Control Memory	213
7-2	Address Sequencing	216
	<i>Conditional Branching</i> 217	
	<i>Mapping of Instruction</i> 219	
	<i>Subroutines</i> 220	
7-3	Microprogram Example	220
	<i>Computer Configuration</i> 220	
	<i>Microinstruction Format</i> 222	
	<i>Symbolic Microinstructions</i> 225	
	<i>The Fetch Routine</i> 226	
	<i>Symbolic Microprogram</i> 227	
	<i>Binary Microprogram</i> 229	

7-4	Design of Control Unit	231
	Microprogram Sequencer	232
	Problems	235
	References	238

CHAPTER EIGHT

	Central Processing Unit	241
8-1	Introduction	241
8-2	General Register Organization	242
	Control Word	244
	Examples of Microoperations	246
8-3	Stack Organization	247
	Register Stack	247
	Memory Stack	249
	Reverse Polish Notation	251
	Evaluation of Arithmetic Expressions	253
8-4	Instruction Formats	255
	Three-Address Instructions	258
	Two-Address Instructions	258
	One-Address Instructions	259
	Zero-Address Instructions	259
	RISC Instructions	259
8-5	Addressing Modes	260
	Numerical Example	264
8-6	Data Transfer and Manipulation	266
	Data Transfer Instructions	267
	Data Manipulation Instructions	268
	Arithmetic Instructions	269
	Logical and Bit Manipulation Instructions	270
	Shift Instructions	271
8-7	Program Control	273
	Status Bit Conditions	274
	Conditional Branch Instructions	275
	Subroutine Call and Return	278
	Program Interrupt	279
	Types of Interrupts	281
8-8	Reduced Instruction Set Computer (RISC)	282
	CISC Characteristics	283
	RISC Characteristics	284

Overlapped Register Windows	285	
Berkeley RISC I	288	
Problems		291
References		297

CHAPTER NINE

Pipeline and Vector Processing		299
9-1 Parallel Processing		299
9-2 Pipelining		302
General Considerations	304	
9-3 Arithmetic Pipeline		307
9-4 Instruction Pipeline		310
Example: Four-Segment Instruction Pipeline	311	
Data Dependency	313	
Handling of Branch Instructions	314	
9-5 RISC Pipeline		315
Example: Three-Segment Instruction Pipeline	316	
Delayed Load	317	
Delayed Branch	318	
9-6 Vector Processing		319
Vector Operations	321	
Matrix Multiplication	322	
Memory Interleaving	324	
Supercomputers	325	
9-7 Array Processors		326
Attached Array Processor	326	
SIMD Array Processor	327	
Problems		329
References		330

CHAPTER TEN

Computer Arithmetic		333
10-1 Introduction		333
10-2 Addition and Subtraction		334
Addition and Subtraction with Signed-Magnitude Data	335	

	Hardware Implementation	336	
	Hardware Algorithm	337	
	Addition and Subtraction with Signed-2's Complement Data	338	
10-3	Multiplication Algorithms		340
	Hardware Implementation for Signed-Magnitude Data	341	
	Hardware Algorithm	342	
	Booth Multiplication Algorithm	343	
	Array Multiplier	346	
10-4	Division Algorithms		348
	Hardware Implementation for Signed-Magnitude Data	349	
	Divide Overflow	351	
	Hardware Algorithm	352	
	Other Algorithms	353	
10-5	Floating-Point Arithmetic Operations		354
	Basic Considerations	354	
	Register Configuration	357	
	Addition and Subtraction	358	
	Multiplication	360	
	Division	362	
10-6	Decimal Arithmetic Unit		363
	BCD Adder	365	
	BCD Subtraction	368	
10-7	Decimal Arithmetic Operations		369
	Addition and Subtraction	371	
	Multiplication	371	
	Division	374	
	Floating-Point Operations	376	
	Problems		376
	References		380

CHAPTER ELEVEN

Input-Output Organization

11-1	Peripheral Devices		381
	ASCII Alphanumeric Characters	383	
11-2	Input-Output Interface		385
	I/O Bus and Interface Modules	386	
	I/O versus Memory Bus	387	

	<i>Isolated versus Memory-Mapped I/O</i>	388	
	<i>Example of I/O Interface</i>	389	
11-3	Asynchronous Data Transfer		391
	Strobe Control	391	
	Handshaking	393	
	Asynchronous Serial Transfer	396	
	Asynchronous Communication Interface	398	
	First-In, First-Out Buffer	400	
11-4	Modes of Transfer		402
	Example of Programmed I/O	403	
	Interrupt-Initiated I/O	406	
	Software Considerations	406	
11-5	Priority Interrupt		407
	Daisy-Chaining Priority	408	
	Parallel Priority Interrupt	409	
	Priority Encoder	411	
	Interrupt Cycle	412	
	Software Routines	413	
	Initial and Final Operations	414	
11-6	Direct Memory Access (DMA)		415
	DMA Controller	416	
	DMA Transfer	418	
11-7	Input-Output Processor (IOP)		420
	CPU-IOP Communication	422	
	IBM 370 I/O Channel	423	
	Intel 8089 IOP	427	
11-8	Serial Communication		429
	Character-Oriented Protocol	432	
	Transmission Example	433	
	Data Transparency	436	
	Bit-Oriented Protocol	437	
	Problems		439
	References		442
C H A P T E R T W E L V E			
	Memory Organization		445
12-1	Memory Hierarchy		445
12-2	Main Memory		448
	RAM and ROM Chips	449	

	Memory Address Map	450	
	Memory Connection to CPU	452	
12-3	Auxiliary Memory		452
	Magnetic Disks	454	
	Magnetic Tape	455	
12-4	Associative Memory		456
	Hardware Organization	457	
	Match Logic	459	
	Read Operation	460	
	Write Operation	461	
12-5	Cache Memory		462
	Associative Mapping	464	
	Direct Mapping	465	
	Set-Associative Mapping	467	
	Writing into Cache	468	
	Cache Initialization	469	
12-6	Virtual Memory		469
	Address Space and Memory Space	470	
	Address Mapping Using Pages	472	
	Associative Memory Page Table	474	
	Page Replacement	475	
12-7	Memory Management Hardware		476
	Segmented-Page Mapping	477	
	Numerical Example	479	
	Memory Protection	482	
	Problems		483
	References		486

CHAPTER THIRTEEN

	Multiprocessors		489
13-1	Characteristics of Multiprocessors		489
13-2	Interconnection Structures		491
	Time-Shared Common Bus	491	
	Multipoint Memory	493	
	Crossbar Switch	494	
	Multistage Switching Network	496	
	Hypercube Interconnection	498	
13-3	Interprocessor Arbitration		500
	System Bus	500	

	<i>Serial Arbitration Procedure</i>	502	
	<i>Parallel Arbitration Logic</i>	503	
	<i>Dynamic Arbitration Algorithms</i>	505	
13-4	Interprocessor Communication and Synchronization		506
	<i>Interprocessor Synchronization</i>	507	
	<i>Mutual Exclusion with a Semaphore</i>	508	
13-5	Cache Coherence		509
	<i>Conditions for Incoherence</i>	509	
	<i>Solutions to the Cache Coherence Problem</i>	510	
	Problems		512
	References		514
	 Index		 515

Digital Logic Circuits

IN THIS CHAPTER

- 1-1 Digital Computers
- 1-2 Logic Gates
- 1-3 Boolean Algebra
- 1-4 Map Simplification
- 1-5 Combinational Circuits
- 1-6 Flip-Flops
- 1-7 Sequential Circuits

1-1 Digital Computers

The digital computer is a digital system that performs various computational tasks. The word *digital* implies that the information in the computer is represented by variables that take a limited number of discrete values. These values are processed internally by components that can maintain a limited number of discrete states. The decimal digits 0, 1, 2, . . . , 9, for example, provide 10 discrete values. The first electronic digital computers, developed in the late 1940s, were used primarily for numerical computations. In this case the discrete elements are the digits. From this application the term *digital computer* has emerged. In practice, digital computers function more reliably if only two states are used. Because of the physical restriction of components, and because human logic tends to be binary (i.e., true-or-false, yes-or-no statements), digital components that are constrained to take discrete values are further constrained to take only two values and are said to be *binary*.

Digital computers use the binary number system, which has two digits: 0 and 1. A binary digit is called a *bit*. Information is represented in digital computers in groups of bits. By using various coding techniques, groups of bits can be made to represent not only binary numbers but also other discrete

symbols, such as decimal digits or letters of the alphabet. By judicious use of binary arrangements and by using various coding techniques, the groups of bits are used to develop complete sets of instructions for performing various types of computations.

In contrast to the common decimal numbers that employ the base 10 system, binary numbers use a base 2 system with two digits: 0 and 1. The decimal equivalent of a binary number can be found by expanding it into a power series with a base of 2. For example, the binary number 1001011 represents a quantity that can be converted to a decimal number by multiplying each bit by the base 2 raised to an integer power as follows:

$$1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 75$$

The seven bits 1001011 represent a binary number whose decimal equivalent is 75. However, this same group of seven bits represents the letter K when used in conjunction with a binary code for the letters of the alphabet. It may also represent a control code for specifying some decision logic in a particular digital computer. In other words, groups of bits in a digital computer are used to represent many different things. This is similar to the concept that the same letters of an alphabet are used to construct different languages, such as English and French.

A computer system is sometimes subdivided into two functional entities: hardware and software. The hardware of the computer consists of all the electronic components and electromechanical devices that comprise the physical entity of the device. Computer software consists of the instructions and data that the computer manipulates to perform various data-processing tasks. A sequence of instructions for the computer is called a *program*. The data that are manipulated by the program constitute the *data base*.

A computer system is composed of its hardware and the system software available for its use. The system software of a computer consists of a collection of programs whose purpose is to make more effective use of the computer. The programs included in a systems software package are referred to as the *operating system*. They are distinguished from application programs written by the user for the purpose of solving particular problems. For example, a high-level language program written by a user to solve particular data-processing needs is an application program, but the compiler that translates the high-level language program to machine language is a system program. The customer who buys a computer system would need, in addition to the hardware, any available software needed for effective operation of the computer. The system software is an indispensable part of a total computer system. Its function is to compensate for the differences that exist between user needs and the capability of the hardware.

The hardware of the computer is usually divided into three major parts, as shown in Fig. 1-1. The central processing unit (CPU) contains an arithmetic

program

computer hardware

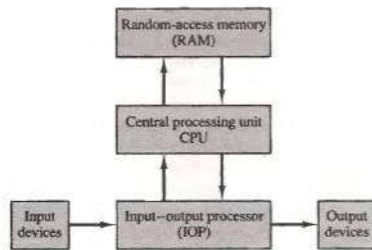


Figure 1-1 Block diagram of a digital computer.

and logic unit for manipulating data, a number of registers for storing data, and control circuits for fetching and executing instructions. The memory of a computer contains storage for instructions and data. It is called a random-access memory (RAM) because the CPU can access any location in memory at random and retrieve the binary information within a fixed interval of time. The input and output processor (IOP) contains electronic circuits for communicating and controlling the transfer of information between the computer and the outside world. The input and output devices connected to the computer include keyboards, printers, terminals, magnetic disk drives, and other communication devices.

This book provides the basic knowledge necessary to understand the hardware operations of a computer system. The subject is sometimes considered from three different points of view, depending on the interest of the investigator. When dealing with computer hardware it is customary to distinguish between what is referred to as computer organization, computer design, and computer architecture.

Computer organization is concerned with the way the hardware components operate and the way they are connected together to form the computer system. The various components are assumed to be in place and the task is to investigate the organizational structure to verify that the computer parts operate as intended.

Computer design is concerned with the hardware design of the computer. Once the computer specifications are formulated, it is the task of the designer to develop hardware for the system. Computer design is concerned with the determination of what hardware should be used and how the parts should be connected. This aspect of computer hardware is sometimes referred to as *computer implementation*.

Computer architecture is concerned with the structure and behavior of the computer as seen by the user. It includes the information formats, the instruc-

*computer
organization*

computer design

*computer
architecture*

tion set, and techniques for addressing memory. The architectural design of a computer system is concerned with the specifications of the various functional modules, such as processors and memories, and structuring them together into a computer system.

The book deals with all three subjects associated with computer hardware. In Chapters 1 through 4 we present the various digital components used in the organization and design of computer systems. Chapters 5 through 7 cover the steps that a designer must go through to design and program an elementary digital computer. Chapters 8 and 9 deal with the architecture of the central processing unit. In Chapters 11 and 12 we present the organization and architecture of the input-output processor and the memory unit.

1-2 Logic Gates

Binary information is represented in digital computers by physical quantities called *signals*. Electrical signals such as voltages exist throughout the computer in either one of two recognizable states. The two states represent a binary variable that can be equal to 1 or 0. For example, a particular digital computer may employ a signal of 3 volts to represent binary 1 and 0.5 volt to represent binary 0. The input terminals of digital circuits accept binary signals of 3 and 0.5 volts and the circuits respond at the output terminals with signals of 3 and 0.5 volts to represent binary input and output corresponding to 1 and 0, respectively.

Binary logic deals with binary variables and with operations that assume a logical meaning. It is used to describe, in algebraic or tabular form, the manipulation and processing of binary information. The manipulation of binary information is done by logic circuits called *gates*. Gates are blocks of hardware that produce signals of binary 1 or 0 when input logic requirements are satisfied. A variety of logic gates are commonly used in digital computer systems. Each gate has a distinct graphic symbol and its operation can be described by means of an algebraic expression. The input-output relationship of the binary variables for each gate can be represented in tabular form by a *truth table*.

The names, graphic symbols, algebraic functions, and truth tables of eight logic gates are listed in Fig. 1-2. Each gate has one or two binary input variables designated by *A* and *B* and one binary output variable designated by *x*. The AND gate produces the AND logic function: that is, the output is 1 if input *A* and input *B* are both equal to 1; otherwise, the output is 0. These conditions are also specified in the truth table for the AND gate. The table shows that output *x* is 1 only when both input *A* and input *B* are 1. The algebraic operation symbol of the AND function is the same as the multiplication symbol of ordinary arithmetic. We can either use a dot between the variables or

gates

AND









Name	Graphic symbol	Algebraic function	Truth table															
AND		$x = A \cdot B$ or $x = AB$	<table><tr><th>A</th><th>B</th><th>x</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	x	0	0	0	0	1	0	1	0	0	1	1	1
A	B	x																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR		$x = A + B$	<table><tr><th>A</th><th>B</th><th>x</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	x	0	0	0	0	1	1	1	0	1	1	1	1
A	B	x																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
Inverter		$x = A'$	<table><tr><th>A</th><th>x</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	x	0	1	1	0									
A	x																	
0	1																	
1	0																	
Buffer		$x = A$	<table><tr><th>A</th><th>x</th></tr><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td></tr></table>	A	x	0	0	1	1									
A	x																	
0	0																	
1	1																	
NAND		$x = (AB)'$	<table><tr><th>A</th><th>B</th><th>x</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	x	0	0	1	0	1	1	1	0	1	1	1	0
A	B	x																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR		$x = (A + B)'$	<table><tr><th>A</th><th>B</th><th>x</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	x	0	0	1	0	1	0	1	0	0	1	1	0
A	B	x																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
Exclusive-OR (XOR)		$x = A \oplus B$ or $x = A'B + AB'$	<table><tr><th>A</th><th>B</th><th>x</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	x	0	0	0	0	1	1	1	0	1	1	1	0
A	B	x																
0	0	0																
0	1	1																
1	0	1																
1	1	0																
Exclusive-NOR or equivalence		$x = (A \oplus B)'$ or $x = A'B' + AB$	<table><tr><th>A</th><th>B</th><th>x</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	x	0	0	1	0	1	0	1	0	0	1	1	1
A	B	x																
0	0	1																
0	1	0																
1	0	0																
1	1	1																

Figure 1-2 Digital logic gates.

concatenate the variables without an operation symbol between them. AND gates may have more than two inputs, and by definition, the output is 1 if and only if all inputs are 1.

OR

The OR gate produces the inclusive-OR function; that is, the output is 1 if input *A* or input *B* or both inputs are 1; otherwise, the output is 0. The algebraic symbol of the OR function is +, similar to arithmetic addition. OR gates may have more than two inputs, and by definition, the output is 1 if any input is 1.

inverter

The inverter circuit inverts the logic sense of a binary signal. It produces the NOT, or complement, function. The algebraic symbol used for the logic complement is either a prime or a bar over the variable symbol. In this book we use a prime for the logic complement of a binary variable, while a bar over the letter is reserved for designating a complement microoperation as defined in Chap. 4.

The small circle in the output of the graphic symbol of an inverter designates a logic complement. A triangle symbol by itself designates a buffer circuit. A buffer does not produce any particular logic function since the binary value of the output is the same as the binary value of the input. This circuit is used merely for power amplification. For example, a buffer that uses 3 volts for binary 1 will produce an output of 3 volts when its input is 3 volts. However, the amount of electrical power needed at the input of the buffer is much less than the power produced at the output of the buffer. The main purpose of the buffer is to drive other gates that require a large amount of power.

NAND

The NAND function is the complement of the AND function, as indicated by the graphic symbol, which consists of an AND graphic symbol followed by a small circle. The designation NAND is derived from the abbreviation of NOT-AND. The NOR gate is the complement of the OR gate and uses an OR graphic symbol followed by a small circle. Both NAND and NOR gates may have more than two inputs, and the output is always the complement of the AND or OR function, respectively.

NOR

exclusive-OR

The exclusive-OR gate has a graphic symbol similar to the OR gate except for the additional curved line on the input side. The output of this gate is 1 if any input is 1 but excludes the combination when both inputs are 1. The exclusive-OR function has its own algebraic symbol or can be expressed in terms of AND, OR, and complement operations as shown in Fig. 1-2. The exclusive-NOR is the complement of the exclusive-OR, as indicated by the small circle in the graphic symbol. The output of this gate is 1 only if both inputs are equal to 1 or both inputs are equal to 0. A more fitting name for the exclusive-OR operation would be an odd function; that is, its output is 1 if an odd number of inputs are 1. Thus in a three-input exclusive-OR (odd) function, the output is 1 if only one input is 1 or if all three inputs are 1. The exclusive-OR and exclusive-NOR gates are commonly available with two inputs, and only seldom are they found with three or more inputs.

1-3 Boolean Algebra

Boolean algebra is an algebra that deals with binary variables and logic operations. The variables are designated by letters such as A , B , x , and y . The three basic logic operations are AND, OR, and complement. A Boolean function can be expressed algebraically with binary variables, the logic operation symbols, parentheses, and equal sign. For a given value of the variables, the Boolean function can be either 1 or 0. Consider, for example, the Boolean function

$$F = x + y'z$$

The function F is equal to 1 if x is 1 or if both y' and z are equal to 1; F is equal to 0 otherwise. But saying that $y' = 1$ is equivalent to saying that $y = 0$ since y' is the complement of y . Therefore, we may say that F is equal to 1 if $x = 1$ or if $yz = 01$. The relationship between a function and its binary variables can be represented in a truth table. To represent a function in a truth table we need a list of the 2^n combinations of the n binary variables. As shown in Fig. 1-3(a), there are eight possible distinct combinations for assigning bits to the three variables x , y , and z . The function F is equal to 1 for those combinations where $x = 1$ or $yz = 01$; it is equal to 0 for all other combinations.

A Boolean function can be transformed from an algebraic expression into a logic diagram composed of AND, OR, and inverter gates. The logic diagram for F is shown in Fig. 1-3(b). There is an inverter for input y to generate its complement y' . There is an AND gate for the term $y'z$, and an OR gate is used to combine the two terms. In a logic diagram, the variables of the function are taken to be the inputs of the circuit, and the variable symbol of the function is taken as the output of the circuit.

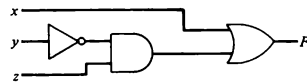
The purpose of Boolean algebra is to facilitate the analysis and design of digital circuits. It provides a convenient tool to:

1. Express in algebraic form a truth table relationship between binary variables.

Figure 1-3 Truth table and logic diagram for $F = x + y'z$.

x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

(a) Truth table



(b) Logic diagram

- 2. Express in algebraic form the input-output relationship of logic diagrams.
- 3. Find simpler circuits for the same function.

Boolean expression

A Boolean function specified by a truth table can be expressed algebraically in many different ways. By manipulating a Boolean expression according to Boolean algebra rules, one may obtain a simpler expression that will require fewer gates. To see how this is done, we must first study the manipulative capabilities of Boolean algebra.

Table 1-1 lists the most basic identities of Boolean algebra. All the identities in the table can be proven by means of truth tables. The first eight identities show the basic relationship between a single variable and itself, or in conjunction with the binary constants 1 and 0. The next five identities (9 through 13) are similar to ordinary algebra. Identity 14 does not apply in ordinary algebra but is very useful in manipulating Boolean expressions. Identities 15 and 16 are called DeMorgan's theorems and are discussed below. The last identity states that if a variable is complemented twice, one obtains the original value of the variable.

TABLE 1-1 Basic Identities of Boolean Algebra

(1) $x + 0 = x$	(2) $x \cdot 0 = 0$
(3) $x + 1 = 1$	(4) $x \cdot 1 = x$
(5) $x + x = x$	(6) $x \cdot x = x$
(7) $x + x' = 1$	(8) $x \cdot x' = 0$
(9) $x + y = y + x$	(10) $xy = yx$
(11) $x + (y + z) = (x + y) + z$	(12) $x(yz) = (xy)z$
(13) $x(y + z) = xy + xz$	(14) $x + yx = (x + y)(x + z)$
(15) $(x + y)' = x'y'$	(16) $(xy)' = x' + y'$
(17) $(x')' = x$	

The identities listed in the table apply to single variables or to Boolean functions expressed in terms of binary variables. For example, consider the following Boolean algebra expression:

$$AB' + C'D + AB' + C'D$$

By letting $x = AB' + C'D$ the expression can be written as $x + x$. From identity 5 in Table 1-1 we find that $x + x = x$. Thus the expression can be reduced to only two terms:

$$AB' + C'D + A'B + C'D = AB' + C'D$$

DeMorgan's theorem

DeMorgan's theorem is very important in dealing with NOR and NAND gates. It states that a NOR gate that performs the $(x + y)'$ function is equivalent

to the function $x'y'$. Similarly, a NAND function can be expressed by either $(xy)'$ or $(x' + y')$. For this reason the NOR and NAND gates have two distinct graphic symbols, as shown in Figs. 1-4 and 1-5. Instead of representing a NOR gate with an OR graphic symbol followed by a circle, we can represent it by an AND graphic symbol preceded by circles in all inputs. The invert-AND symbol for the NOR gate follows from DeMorgan's theorem and from the convention that small circles denote complementation. Similarly, the NAND gate has two distinct symbols, as shown in Fig. 1-5.

To see how Boolean algebra manipulation is used to simplify digital circuits, consider the logic diagram of Fig. 1-6(a). The output of the circuit can be expressed algebraically as follows:

$$F = ABC + ABC' + A'C$$

Each term corresponds to one AND gate, and the OR gate forms the logical sum of the three terms. Two inverters are needed to complement A' and C' . The expression can be simplified using Boolean algebra.

$$\begin{aligned} F &= ABC + ABC' + A'C = AB(C + C') + A'C \\ &= AB + A'C \end{aligned}$$

Note that $(C + C') = 1$ by identity 7 and $AB \cdot 1 = AB$ by identity 4 in Table 1-1.

The logic diagram of the simplified expression is drawn in Fig. 1-6(b). It requires only four gates rather than the six gates used in the circuit of Fig. 1-6(a). The two circuits are equivalent and produce the same truth table relationship between inputs A , B , C and output F .

Figure 1-4 Two graphic symbols for NOR gate.

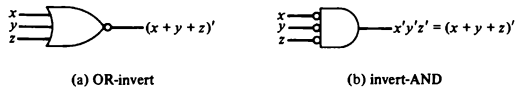
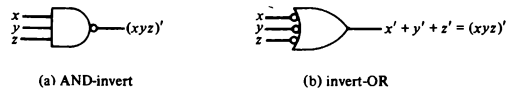


Figure 1-5 Two graphic symbols for NAND gate.



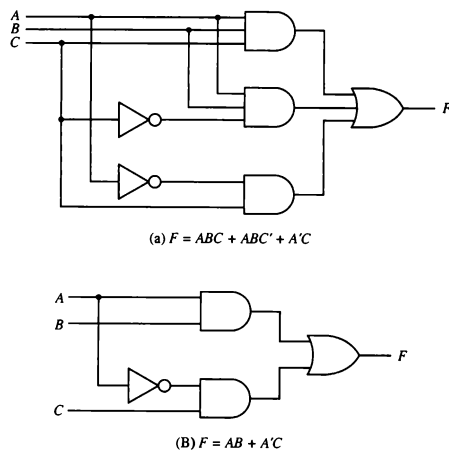


Figure 1-6 Two logic diagrams for the same Boolean function.

Complement of a Function

The complement of a function F when expressed in a truth table is obtained by interchanging 1's and 0's in the values of F in the truth table. When the function is expressed in algebraic form, the complement of the function can be derived by means of DeMorgan's theorem. The general form of DeMorgan's theorem can be expressed as follows:

$$(x_1 + x_2 + x_3 + \cdots + x_n)' = x_1' x_2' x_3' \cdots x_n'$$

$$(x_1 x_2 x_3 \cdots x_n)' = x_1' + x_2' + x_3' + \cdots + x_n'$$

From the general DeMorgan's theorem we can derive a simple procedure for obtaining the complement of an algebraic expression. This is done by changing all OR operations to AND operations and all AND operations to OR operations and then complementing each individual letter variable. As an example, consider the following expression and its complement:

$$F = AB + C'D' + B'D$$

$$F' = (A' + B')(C + D)(B + D')$$

The complement expression is obtained by interchanging AND and OR operations and complementing each individual variable. Note that the complement of C' is C .

1-4 Map Simplification

The complexity of the logic diagram that implements a Boolean function is related directly to the complexity of the algebraic expression from which the function is implemented. The truth table representation of a function is unique, but the function can appear in many different forms when expressed algebraically. The expression may be simplified using the basic relations of Boolean algebra. However, this procedure is sometimes difficult because it lacks specific rules for predicting each succeeding step in the manipulative process. The map method provides a simple, straightforward procedure for simplifying Boolean expressions. This method may be regarded as a pictorial arrangement of the truth table which allows an easy interpretation for choosing the minimum number of terms needed to express the function algebraically. The map method is also known as the Karnaugh map or K-map.

Each combination of the variables in a truth table is called a minterm. For example, the truth table of Fig. 1-3 contains eight minterms. When expressed in a truth table a function of n variables will have 2^n minterms, equivalent to the 2^n binary numbers obtained from n bits. A Boolean function is equal to 1 for some minterms and to 0 for others. The information contained in a truth table may be expressed in compact form by listing the decimal equivalent of those minterms that produce a 1 for the function. For example, the truth table of Fig. 1-3 can be expressed as follows:

$$F(x, y, z) = \sum (1, 4, 5, 6, 7)$$

The letters in parentheses list the binary variables in the order that they appear in the truth table. The symbol \sum stands for the sum of the minterms that follow in parentheses. The minterms that produce 1 for the function are listed in their decimal equivalent. The minterms missing from the list are the ones that produce 0 for the function.

The map is a diagram made up of squares, with each square representing one minterm. The squares corresponding to minterms that produce 1 for the function are marked by a 1 and the others are marked by a 0 or are left empty. By recognizing various patterns and combining squares marked by 1's in the map, it is possible to derive alternative algebraic expressions for the function, from which the most convenient may be selected.

The maps for functions of two, three, and four variables are shown in Fig. 1-7. The number of squares in a map of n variables is 2^n . The 2^n minterms are listed by an equivalent decimal number for easy reference. The minterm

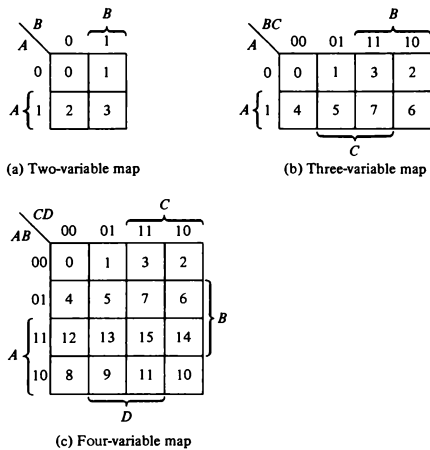


Figure 1-7 Maps for two-, three-, and four-variable functions.

numbers are assigned in an orderly arrangement such that adjacent squares represent minterms that differ by only one variable. The variable names are listed across both sides of the diagonal line in the corner of the map. The 0's and 1's marked along each row and each column designate the value of the variables. Each variable under brackets contains half of the squares in the map where that variable appears unprimed. The variable appears with a prime (complemented) in the remaining half of the squares.

The minterm represented by a square is determined from the binary assignments of the variables along the left and top edges in the map. For example, minterm 5 in the three-variable map is 101 in binary, which may be obtained from the 1 in the second row concatenated with the 01 of the second column. This minterm represents a value for the binary variables A , B , and C , with A and C being unprimed and B being primed (i.e., $AB'C$). On the other hand, minterm 5 in the four-variable map represents a minterm for four variables. The binary number contains the four bits 0101, and the corresponding term it represents is $A'BC'D$.

Minterms of adjacent squares in the map are identical except for one variable, which appears complemented in one square and uncomplemented in the adjacent square. According to this definition of adjacency, the squares at the extreme ends of the same horizontal row are also to be considered

adjacent squares

adjacent. The same applies to the top and bottom squares of a column. As a result, the four corner squares of a map must also be considered to be adjacent.

A Boolean function represented by a truth table is plotted into the map by inserting 1's in those squares where the function is 1. The squares containing 1's are combined in groups of adjacent squares. These groups must contain a number of squares that is an integral power of 2. Groups of combined adjacent squares may share one or more squares with one or more groups. Each group of squares represents an algebraic term, and the OR of those terms gives the simplified algebraic expression for the function. The following examples show the use of the map for simplifying Boolean functions.

In the first example we will simplify the Boolean function

$$F(A, B, C) = \sum (3, 4, 6, 7)$$

The three-variable map for this function is shown in Fig. 1-8. There are four squares marked with 1's, one for each minterm that produces 1 for the function. These squares belong to minterms 3, 4, 6, and 7 and are recognized from Fig. 1-7(b). Two adjacent squares are combined in the third column. This column belongs to both B and C and produces the term BC . The remaining two squares with 1's in the two corners of the second row are adjacent and belong to row A and the two columns of C' , so they produce the term AC' . The simplified algebraic expression for the function is the OR of the two terms:

$$F = BC + AC'$$

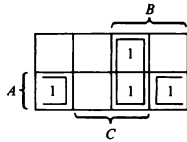
The second example simplifies the following Boolean function:

$$F(A, B, C) = \sum (0, 2, 4, 5, 6)$$

The five minterms are marked with 1's in the corresponding squares of the three-variable map shown in Fig. 1-9. The four squares in the first and fourth columns are adjacent and represent the term C' . The remaining square marked with a 1 belongs to minterm 5 and can be combined with the square of minterm 4 to produce the term AB' . The simplified function is

$$F = C' + AB'$$

Figure 1-8 Map for $F(A, B, C) = \sum (3, 4, 6, 7)$.



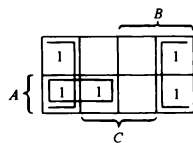


Figure 1-9 Map for $F(A, B, C) = \Sigma(0, 2, 4, 5, 6)$.

The third example needs a four-variable map.

$$F(A, B, C, D) = \Sigma(0, 1, 2, 6, 8, 9, 10)$$

The area in the map covered by this four-variable function consists of the squares marked with 1's in Fig. 1-10. The function contains 1's in the four corners that, when taken as a group, give the term $B'D'$. This is possible because these four squares are adjacent when the map is considered with top and bottom or left and right edges touching. The two 1's on the left of the top row are combined with the two 1's on the left of the bottom row to give the term $B'C'$. The remaining 1 in the square of minterm 6 is combined with minterm 2 to give the term $A'CD'$. The simplified function is

$$F = B'D' + B'C' + A'CD'$$

Product-of-Sums Simplification

The Boolean expressions derived from the maps in the preceding examples were expressed in sum-of-products form. The product terms are AND terms and the sum denotes the ORing of these terms. It is sometimes convenient to obtain the algebraic expression for the function in a product-of-sums form. The

Figure 1-10 Map for $F(A, B, C, D) = \Sigma(0, 1, 2, 6, 8, 9, 10)$.

