Computer, Network, Software, and Hardware Engineering with Applications

IEEE Press
445 Hoes Lane
Piscataway, NJ 08854

Technical Reviewers

Michael R. Lyu
The Chinese University of Hong Kong

Daniel Zulaica
Naval Postgraduate School

Computer, Network, Software, and Hardware Engineering with Applications

Norman F. Schneidewind

IEEE PRESS

A John Wiley & Sons, Inc., Publication

Library of Congress Cataloging-in-Publication Data:

Schneidewind, Norman.
Computer, network, software, and hardware engineering with applications / Norman Schneidewind.
p. cm.
Includes index.
ISBN 978-1-118-03745-4 (cloth)
1. Computer engineering. 2. Computer networks. 3. Software engineering. I. Title.
TK7885.S2564 2012
005.1–dc23

2011033591

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Contents

Preface

There are many books on computers, networks, and software engineering but none that integrate the three with *applications*. Integration is important because, increasingly, software dominates the performance, reliability, maintainability, and availability of complex computer and systems. Books on software engineering typically portray software as if it exists in a vacuum with no relationship to the wider system. This is wrong because a system is more than software. It is comprised of people, organizations, processes, hardware, and software. All of these components must be considered in an integrative fashion when designing systems. On the other hand, books on computers and networks do not demonstrate a deep understanding of the intricacies of developing software. In this book you will learn, for example, how to *quantitatively* analyze the performance, reliability, maintainability, and availability of computers, networks, and software in relation to the *total system*. Furthermore, you will learn how to evaluate and mitigate the risk of deploying integrated systems. You will learn how to apply many models dealing with the optimization of systems. Numerous quantitative examples are provided to help you understand and interpret model results.

The following topics are covered:

- application of quantitative models to solving computer, network, and software engineering problems
- mathematical and statistical models of reliability, maintainability, and availability
- statistical process and product control
- fault tree analysis
- risk management
- software metrics
- resource allocation and assignment
- software reliability models and tools
- computer security
- optimal network routing

Solutions to problems that consider only a single facet of a problem are doomed to be suboptimal. Because of its breadth, this book provides a new perspective for computer, network, and software engineers to consider the big picture in order to develop optimal solutions.

This book can be used as a text, handbook, and reference by advanced undergraduates and first-year graduate students in academia as well as by computer, network, and software engineer practitioners in the worldwide industry.

Norman F. Schneidewind
Professor Emeritus of Information Sciences
Department of Information Sciences
and the Software Engineering Group
Naval Postgraduate School

About the Author

Dr. Norman F. Schneidewind is Professor Emeritus of Information Sciences in the Department of Information Sciences and the Software Engineering Group at the Naval Postgraduate School. He is now doing research and publishing articles and books in software reliability engineering with his consulting company Computer Research. Dr. Schneidewind is a Fellow of the Institute of Electrical and Electronics Engineers (IEEE), elected in 1992 for "contributions to software measurement models in reliability and metrics, and for leadership in advancing the field of software maintenance." In 2001, he received the IEEE "Reliability Engineer of the Year" award from the IEEE Reliability Society. In 2011, he received the "Outstanding Engineer" award from the IEEE Santa Clara Valley Section. In 1993 and 1999, he received awards for Outstanding Research Achievement by the Naval Postgraduate School. Dr. Schneidewind was selected for an IEEE-USA Congressional Fellowship in 2005 and worked with the Committee on Homeland Security and Government Affairs, United States Senate, focusing on homeland security and cyber security (see photo below).

In July 2011, Dr. Schneidewind was named the Outstanding Engineer of Santa Clara Valley by the IEEE Chapter of Santa Clara Valley. In addition, he has been named Outstanding Engineer of the San Francisco Bay Area. Furthermore, he has been named Outstanding Engineer of Region 6 of the IEEE.

IEEE-USA's four Government Fellows began their Fellowships in January 2005: Randall Brouwer (with Rep. Dana Rohrabacher); Gordon Day (with Sen. Jay Rockefeller); Norman Schneidewind (on the Senate Homeland Security Committee); and Nick Zayed (with the State Department Office of Science and Technology Cooperation).

Shown at the Jefferson Memorial in Washington, D.C., are, from left to right, IEEE-USA Government Fellows Norman Schneidewind, Nick Zayed, Randall Brouwer, and Gordon Day.

In March 2006, he received the IEEE Computer Society Outstanding Contribution Award "for outstanding technical and leadership contributions as the Chair of the Working Group revising IEEE Standard 982.1," signed by Debra Cooper, President of the IEEE.

He is the developer of the Schneidewind software reliability model that is used by the National Aeronautics and Space Administration (NASA) to assist in the prediction of software reliability of the Space Shuttle by the Naval Surface Warfare Center for Tomahawk cruise missile launch and Trident software reliability prediction, and by the Marine Corps Tactical Systems Support Activity for distributed system software reliability assessment and prediction. This model is one of the models recommended by the IEEE/AIAA Recommended Practice for Software Reliability. In addition, the model is implemented in the Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS) software reliability modeling tool.

Dr. Schneidewind has been interviewed by several organizations regarding his work in software reliability, including the following: a *New York Times* article, which was published on February 7, 2003, about the Space Shuttle software development process in conjunction with the Columbia tragedy and by the Associated Press about the same subject; National Public Radio, Montgomery, Alabama on April 1, 2002; and by *The Bent*, Tau Beta Pi's (all engineering society) magazine, about his professional accomplishments on November 4. 2002. This article was part of a series about prominent Tau Beta Pi members.

He is a member of the IEEE-USA Committee on Communications and Information Technology Policy (CCIP). The objective of the CCIP is to influence the communication and information technology policies of the executive and legislative branches of federal and state governments. His primary contribution is developing policies and models to defeat cyber security attacks. He has also contributed to IEEE-USA Committee on Communications Policy in the area of personal identification privacy and security.

Part One

Computer Engineering

Chapter 1

Digital Logic and Microprocessor Design

This chapter focuses on the fundamentals of digital logic and design, with numerous examples from both computer hardware design and "everyday life" events to demonstrate that digital logic is not confined to designing computers. My objective is to equip the engineer or student with sufficient knowledge of design principles to be able to design a digital computer. In addition, I integrate the important role that software plays in modern computer systems with the hardware design principles. Numerous design examples and solved problems are provided to support learning objectives.

MICROPROCESSOR DESIGN

Functions

Using its arithmetic logic unit(ALU), a microprocessor can perform mathematical and logic operations like addition, subtraction, multiplication, division, and comparison. Modern microprocessors contain complete floating-point processors that can perform extremely sophisticated operations on large variable-length numbers. In addition, a microprocessor can perform the following functions:

Move data from one memory location to another.

Make decisions and jump to a new set of computer program instructions based on those decisions.

Use an RD (read) and WR (write) line to tell the memory whether it wants to read from or write to the addressed location.

Use a clock line to transmit clock pulses (CPs) to sequence the microprocessor. For example, when numbers are added by the microprocessor, which you

Computer, Network, Software, and Hardware Engineering with Applications, First Edition. Norman F. Schneidewind.

will see later, addition takes place bit by bit, and the clock triggers each binary bit addition to ultimately form a decimal result.

Uses a reset line to reset the program counter to zero and restart execution.

Components

Microprocessor components are the building blocks of modern computers. These components are the following:

- ***ALU.*** Consists of accumulators, registers, and control unit.
  - The ALU executes instructions and manipulates data.
  - An 8-bit ALU can add, subtract, multiply, and divide two 8-bit numbers, while a 32-bit ALU can manipulate 8-bit, 16-bit, and 32-bit numbers.
  - An 8-bit ALU would have to execute four instructions to add two 32-bit numbers (four add instructions, each of which adds 8-bit numbers), whereas a 32-bit ALU can do it in one instruction.
- ***Accumulator.*** Holds data and instructions for processing by the ALU.
- ***Register.*** Temporary storage of instructions and data.
  - ***Program Counter* (PC).** Contains the address of next instruction to be executed
  - ***Instruction Register* (IR).** Holds address of current instruction being executed
  - ***General Registers.*** Holds operator (e.g., code for add instruction), operands (e.g., numbers to be added), and data while an instruction is executed
- ***Stack.*** Temporary storage of instructions and data, usually on a last in, first out (LIFO) basis. Also called push-down stack.
- ***Control Unit.*** Fetches and decodes instructions, generates signals for the ALU to execute instructions
- ***Busses***
  - ***Address Bus.*** Path over which addresses flow for directing memory and input/output (I/O) data transfers. An address bus may be 8, 16, or 32 bits wide that sends an address to memory or I/O for accessing memory or I/O.
  - ***Data Bus.*** Transfers data. A data bus may be 8, 16, or 32 bits wide that can send data to memory or I/O and receive data from memory or I/O. The number of address bus lines determine the amount of addressable memory (n lines = 2^n addressable words).
  - ***Control Bus.*** Communicates control and status information.
- ***Chip.*** A chip is also called an integrated circuit. Generally it is a small, thin piece of silicon onto which the transistors making up the microprocessor have been etched. A chip might be as large as an inch on a side and can contain tens of millions of transistors. Simpler processors might consist of a few

thousand transistors etched onto a chip just a few millimeters square. Microns are the width of the smallest wire on the chip. For comparison, a human hair is 100 μm thick. As the feature size on the chip goes down, the number of transistors rises.

Characteristics

Microprocessor characteristics govern the speed and functionality of computer operations. Important characteristics include the following presented in the succeeding paragraphs.

Smaller microprocessors can be combined into a larger one (four 4-bit microprocessors combined into one 16-bit microprocessor).

A crystal-controlled clock sequences the operations of a microprocessor (e.g., the sequence of computer program instruction execution) by generating CPs. Clock speed is specified in cycles per second, where 1 MHz is equal to 1 million cycles per second. Clock speed is the maximum speed of the chip.

Instructions require one or more clock cycles to execute the following, depending on its complexity: fetch instruction from memory, decode the operation code, fetch operands from memory, execute the instruction, and store the result in memory. In addition to clock speed, an important performance metric is the number of floating-point operations per second or flops.

***Complex instruction set computing* (CISC).** A single instruction can perform several operations. This design simplifies programming because, for example, a single instruction can fetch instruction from memory, decode the operation code, fetch operands from memory, execute the instruction, and store the result in memory. However, the downside is the relatively slow speed of the computer [RAF05].

***Reduced instruction set computing* (RISC).** Several operations are required to execute a single instruction. This design provides high speed, for example, well suited to real-time applications that must meet deadlines, but at the expense of relatively complex programming.

Performance

One measure of the computing power of a microprocessor is its clock speed, measured in millions of cycles per second (MHz). It usually takes from one to seven cycles of a microprocessor's internal clock to fully process an instruction. The faster the internal clock, the more instructions can be processed per unit of time. For the microprocessors in laptop and desktop computers, clock speeds are usually greater than 100 MHz. The fastest microprocessors can run at a speed of 2 GHz. From a user standpoint, the most important performance metric is program execution time, defined as [HAR07]:

Program execution time = (Number of instructions in program)
∗(Clock cycles per instruction)∗(Time per clock cycle).

Another measure of performance is the number of instructions that can be processed per second, referred to as MIPS, for million instructions per second. The MIPS rating of a microprocessor depends on both the clock speed and the number of instructions that can be executed per clock cycle. Simple microprocessors can execute a maximum of one instruction per clock cycle. Advanced microprocessors can execute up to six or eight instructions per clock cycle. The relationship between clock speed and MIPS is not straightforward, however, because some instructions may take more than one clock cycle to execute, depending on the program. The product of clock speed and the number of instructions that can be executed per cycle may be greater than MIPS. The maximum clock speed is a function of the manufacturing process and delays within the chip. MIPS is proportional to the clock speed and inversely proportional to the number of clock cycles per instruction.

Another indication of microprocessor speed is the word length, as measured by the number of bits of information that can be transferred simultaneously. Long words allow the microprocessor to handle data and perform complex tasks more efficiently. The number of bits per word has been steadily increasing with the growth of circuit technology. Thus 4-, 8-, 16-, 32-, and 64-bit microprocessors are now common. Some personal computers use 32-bit microprocessors. More powerful computers use 64-bit microprocessors. The 4-, 8-, or 16-bit devices are usually employed in simple embedded applications, such as microwave ovens, electric shavers, and televisions. Figure 1.1 shows the microprocessor architecture.

Pipeline Systems

An important aid to performance is the pipeline system. The purpose of a pipeline system is to reduce delay caused by the computer processor having to wait for instructions to complete. With a pipeline design, the processor begins the execution of the next instruction while the current instruction is executing. Thus, various phases of instruction execution are overlapped. The concept is to keep the pipeline full, with as many execution sequences as possible. For example, due to overlapped instruction execution, each instruction overlaps during $(n - 1)$ clock cycles, and each of $m = 4$ instructions requires one clock cycle, yielding $(n - 1) + m = 7$ clock cycles, total, as shown in Figure 1.2.

Problem: How is the *increase in speed*, obtained by a pipelined system over a conventional system, computed?

Answer: Using Figure 1.2 as an example, the increase is computed as follows:

The number of clock cycles required in conventional system is $mn = 4 * 4 = 16$ in the example of Figure 1.2. Thus, the decrease in number of clock cycles for a pipelined system is:

$$mn - ((n-1) + m) = 16 - 7 = 9,$$

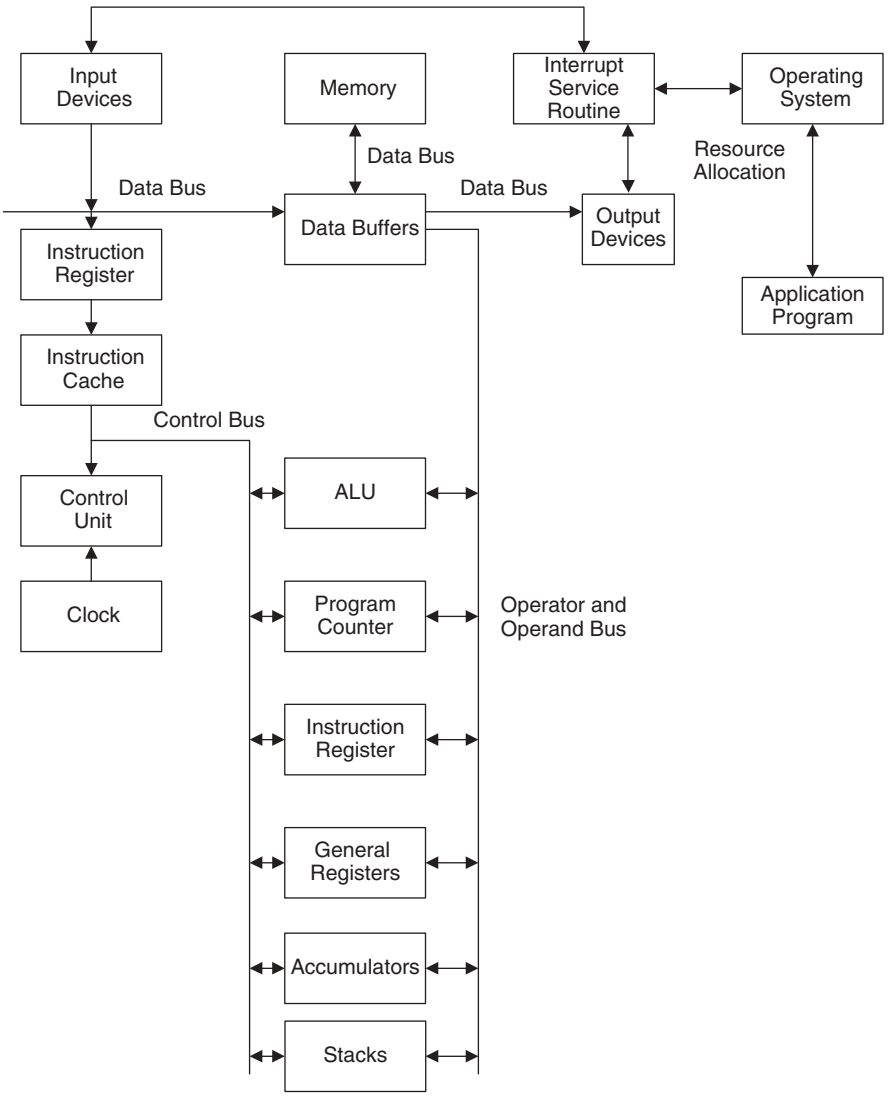

Figure 1.1 Microprocessor architecture.

and the *increase in speed* (number of clock cycles required in conventional system/ number of clock cycles required in a pipelined system) is:

$$(mn)/((n-1)+m) = n/(((n-1)/m)+1) = 16/7 = 2.286.$$

If m is large, the increase in speed approaches n clock cycles per instruction—maximum speed increase.

The pipeline *throughput* is defined as the *number of instructions*, m, per *total clock cycle time* required to process m instructions:

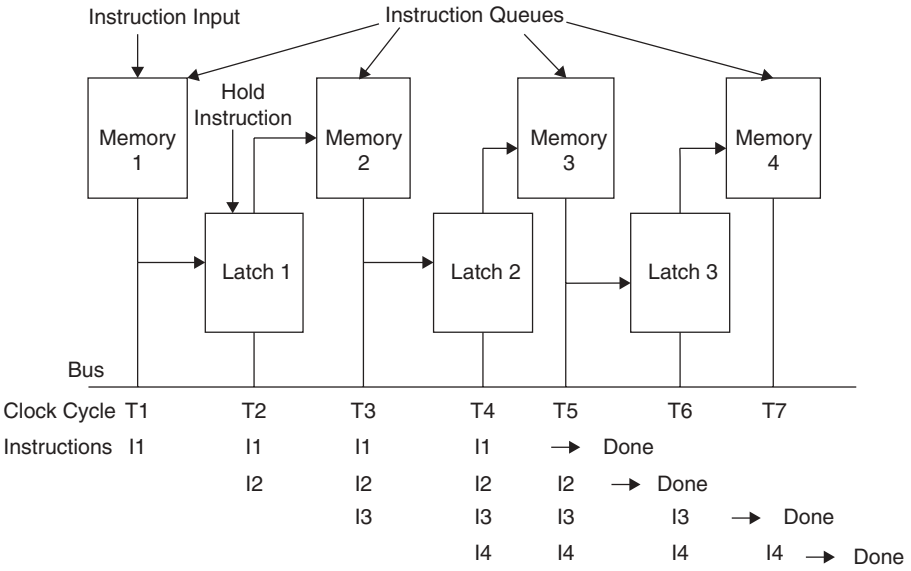

Figure 1.2 Pipelined system. n, clock cycle per instruction; m, instructions, each requiring one clock cycle; (n − 1) + m = 7 clock cycles (each instruction overlaps for [n − 1] clock cycles).

$$\frac{\text{m instructions}}{\text{Number of clock cycles per instruction} * \text{Time per clock cycle}} = \frac{\text{m}}{\text{m} + (\text{n} - 1)\text{T}},$$

where T is clock cycle time per instruction.

Problem: Compute the throughput of the pipeline microprocessor in Figure 1.2.

Answer: For a clock speed of 10 Mhz (10^7 clock cycles per second), T = $1/10^7$ seconds, the throughput is:

$$\text{m} / ((\text{m} + \text{n} - 1)\text{T}) = 4 / ((7)(1/107)) = (4)(107) / 7 = 5.71 \text{ MIPS}.$$

Pipeline efficiency is computed as: speed increase/maximum speed increase (n = 4 clock cycles per instruction) = 2.286/4 = 0.5715.

Pipeline System Delay

When a pipeline instruction is unable to complete on the scheduled clock cycle, then

- Finish the earlier instructions on schedule and
- Delay the later instructions
- This is called stalling the pipeline

*Structural hazard*s are pipeline hardware delays.

Example: Memory does not respond to a request as fast as it is expected.

Data hazards arise when data are not ready in a pipeline at the time they are needed.

Example: An instruction needs data in a register that a previous instruction is still modifying.

Control hazards arise when the central processing unit (CPU) needs to manage a pipeline but instead must increment the program counter.

Example: Nonpipelined conditional branch instruction jumps to a pipelined instruction.

Problem: Delay in a pipelined operation is illustrated in this problem that compares the clock cycle delay for nonjump instructions with that of jump instructions.

If a jump instruction is executed in the pipelined CPU in Figure 1.2, what is the clock cycle delay?

Answer: Since the target of the jump instruction (another instruction) cannot be decoded (i.e., program counter updated) until the jump instruction is executed, there is a delay of three clock cycles.

Problem: What cam be done in a pipeline system to maintain performance when a *structural hazard* occurs?

Answer: More resources can be employed, if available, or the pipeline can be stalled (i.e., no instructions executed until needed hardware is available).

Problem: Is the microprocessor architecture in Figure 1.1 a pipeline computer?

Answer: No, it is not because only one instruction can be executed at a time.

Problem: What determines the clock cycle frequency of a pipeline system?

Answer: The clock cycle frequency of a *pipeline system* is governed by the *pipeline* with the slowest processing time. For example, whichever pipeline queue in Figure 1.2 experiences the slowest processing determines clock cycle frequency.

Operating System

The operating system contains the software necessary to manage the resources of a computer system. An example is a signal called an interrupt that is used to indicate to the microprocessor that an I/O device needs attention (i.e., data input or data output) or that there is an error condition (e.g., attempted divide by zero). The interrupt service routine is shown in Figure 1.1. In addition to managing resources, the operating system is responsible for allocating resources, for example, allocating memory to the application program, as depicted in Figure 1.1.

Memory

Because computer performance depends on the characteristics of memory systems in addition to the microprocessor architecture, it is important to consider the former

[HAR07]. Two important types of memory systems are main memory (random access memory, RAM) and secondary memory (hard disk, USB flash). Main memory can be divided between a relatively slow RAM for program and data access and a fast cache memory for accessing recently used instructions and data. In addition, secondary memory can be classified as virtual, meaning that pages on a hard disk can be mapped to main memory locations under the control of a memory management unit. A microprocessor may be equipped with special hardware, called direct memory access (DMA), which allows I/O devices to communicate directly with memory rather than using intermediate devices (such as data buffers in Fig. 1.1).

RAM

RAM contains bytes of information that the microprocessor can read or write, depending on whether the RD or WR line is activated. One problem with RAM chips is that they are volatile; the RAM contents are lost once the power goes off. That is why the microprocessor needs read-only memory (ROM).

ROM

All microprocessors contain ROM. A ROM chip is programmed with a permanent collection of preset bytes. The address bus tells the ROM chip which byte to read and place on the data bus. The RD line signal causes the ROM chip to transfer the selected byte to the data bus. On a personal computer, the program in the ROM is called the BIOS (basic input/output system). When the microprocessor starts, it begins executing instructions it finds in the BIOS. The BIOS instructions test the hardware, and then control is transferred to the hard disk to fetch the boot sector. The boot sector is another small program that the BIOS stores in RAM after reading it from the disk. The microprocessor then begins executing the boot sector's instructions from RAM. The boot sector program will tell the microprocessor to fetch more instructions from the hard disk into RAM, which the microprocessor then executes, and so on. This is how the microprocessor loads and executes the entire operating system.

Read/Write (R/W) Control Line

This single wire is driven by the microprocessor to control memory functions. If the R/W control line is asserted as a logical 1 (i.e., true), then the microprocessor performs a read operation. If it is asserted as a logic 0 (i.e., false), then the microprocessor performs a write operation. The relationship between logic level and voltage level can vary, depending on the implementation. For example, a logical 0 corresponds to a voltage of 0 V, and a logical 1 corresponds to a voltage of 5 V. Figure 1.3 is a block diagram of the microprocessor and memory, showing the R/W control line.

Address Bus

These wires are controlled by the microprocessor to select a particular location in memory for reading or writing. The microprocessor in Figure 1.3 uses a memory chip that has 15 address wires.

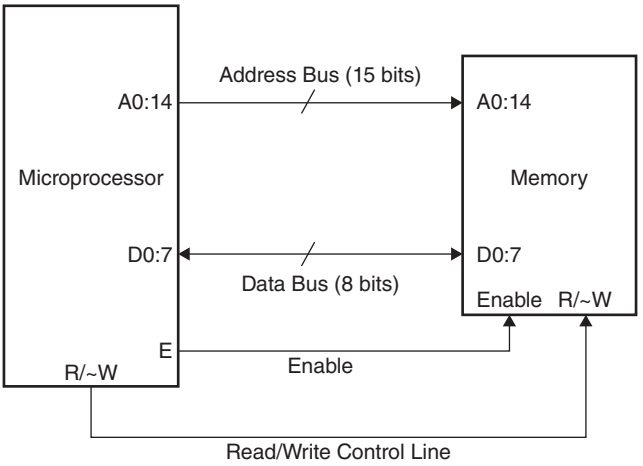

Figure 1.3 Diagram of microprocessor and memory.

Problem: How many locations can be addressed in Figure 1.3?

Answer: Since each wire has two states (it can be a digital 1 or a 0), $2^{15} = 32{,}768$ locations are possible. Thus, the system is said to have 32K of memory (1K = 1024 bytes).

Data Bus

These wires are used to pass data between the microprocessor and the memory. When data are written to the memory, the microprocessor drives the data bus; when data are read from the memory, memory drives the bus. In the example, in Figure 1.3, there are eight data wires (or bits). These wires can transfer one of 2^8 or 256 different binary values per transfer. The data size of 8 bits is commonly referred to as a byte. A data size of 4 bits is frequently referred to as a nibble.

Memory Enable Control Line

This wire, called the Enable line, connects to the enable circuitry of the memory in Figure 1.3. When the memory is enabled, it performs either a read or write operation as determined by the status of the R/W line.

Memory System Performance

Memory system performance is computed by considering hit and miss rates and the order of accessing memory components: cache memory, main memory, and hard disk. These rates are related to whether the instructions or data that are required by a program are available, first, in the cache memory, or second, in the main memory. If the instructions or data are in the cache, the access is scored as a cache hit; otherwise, the access is scored as a cache miss. Similarly, if the instructions or data

are not in the cache but are in main memory, the access is scored as a main memory hit; otherwise, the access is scored as a main memory miss because the instructions or data are only available on the hard disk [HAR07]. Thus, hit and miss rates are computed as follows:

$$\text{Cache hit rate (CHR)} = \frac{\text{Number of cache hits}}{\text{Total number of memory accesses}},$$

$$\text{Cache miss rate (CMR)} = \frac{\text{Number of cache misses}}{\text{Total number of memory accesses}},$$

$$\text{Main memory hit rate (MMHR)} = \frac{\text{Number of main memory hits}}{\text{Total number of memory accesses}},$$

$$\text{Main memory miss rate (MMMR)} = \frac{\text{Number of main memory misses}}{\text{Total number of memory accesses}},$$

$$\text{Number of hard disk accesses (HAD)} = \text{Total number of memory accesses} - (\text{Number of cache memory hits} + \text{Number of main memory hits} + \text{Number of main memory misses}).$$

Note that when there is a cache memory miss, the main memory access is attempted. Thus, it is not necessary to count cache memory misses in the foregoing computation:

$$\text{Hard disk access rate (HDAR)} = \text{HAD} / \text{Total number of memory accesses}.$$

Problem: For example, consider the following case:

4000 total number of memory accesses

1200 cache accesses are hits and 800 are misses

Of the 800 cache misses that require access to the main memory, 200 are hits and 600 are misses

Compute CHR, CMR, MMHR, MMMR, HAD, and HDAR.

Answer: CHR = 1200/4000 = 30%

CMR = 800/4000 = 20%

MMHR = 200/4000 = 5%

MMMR = 600/4000 = 1%

HAD = 4000 − (1200 + 200 + 600) = 2000

HDAR = 1200/4000 = 50%

Another memory performance metric is average access time (AAT), which is computed as follows:

$$\text{AAT} = \text{CHR} * (\text{cache access time}) + \text{MMHR} * (\text{main memory access time}) + \text{HDAR} * (\text{hard disk access time}).$$

Problem: For the following typical access times: cache = 2 ns, main memory = 60 ns, and hard disk = 35 ms, and using the above hit and miss access rates, compute the AAT.

Answer: AAT = (0.30)(2) + (0.04)(60) + (0.50)(35 $*$ 10^6) ns = 20.50 $*$ 10^6 ns (of course, hard disk access time dominates).

Multiplexing Data and Address Signals

On the Motorola 68HC11 microprocessor, in Figure 1.4, the 8-bit address/data bus takes turns acting as an address bus and a data bus. When a memory location is accessed (for reading or writing), the bus first acts as an address bus, transmitting the 8 lower-order bits of the address. Then the bus functions as a data bus, either transmitting a data byte (for a write cycle) or receiving a data byte (for a read cycle). This kind of split-personality bus is referred to as a multiplexed address and data bus. The support needed by the memory is provided by an 8-bit latch (a device that can store an address), using a multiplexed address/data bus. This chip (HC373) performs the function of latching the lower 8 address bits, when combined with the upper 7 address bits from the microprocessor, will provide the full 15-bit address for reading or writing data.

Figure 1.4 shows how the latch is wired. The upper 7 address bits run directly from the microprocessor to the memory. The lower 8 *address bits* are multiplexed with 8 *data bits*. When an *address* appears on the wires AD: 07, the latch connects the address bits of the microprocessor to the memory. On the other hand, when *data* appears on the wires AD0:7, the latch connects the data bits of the microprocessor to the memory. An additional signal, the address strobe (AS) output of the

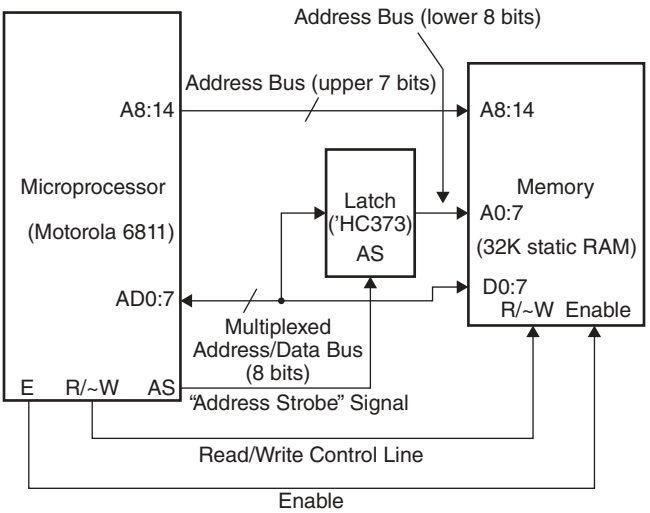

Figure 1.4 Block diagram of microprocessor and memory with latch.

microprocessor, tells the latch when to obtain the address bits from the address/data bus. When the full 15-bit address is available to the memory (upper 7 bits direct from the microprocessor (wires A8: 14) and lower 8 bits from the latch (wires AD: 07), the read or write access can occur. Because the address/data bus is also wired directly to the memory, data can flow in either direction between the memory and the microprocessor. The entire process is managed by the microprocessor. The Enable (E) clock, the R/W line, and the AS line perform in tight synchronization to make sure these operations happen in the correct sequence and within the timing capacities of the microprocessor hardware.

Memory Mapping the RAM

Memory mapping refers to allocating blocks of memory to different functions, such as the operating system and the application program. If a microprocessor has 15 address bits, it has 32,728 (32K bytes) of addressable locations that can be mapped. This address space would be used by the 32K memory chip in Figure 1.5. The technique used to map the memory is fairly simple. Whenever the microprocessor's A15 (the highest order address bit) is logic 1, the high-order address bit is selected. The other 15 address bits (A0 through A14) determine the address within that 32K block. If A15 is logic 0, the 32K block is not selected.

A NAND gate (actually a portion of a programmable logic device called a PAL) is used to enable the memory when A15 and the E Clock equal 1 in Figure 1.5. (See the "Digital Logic" section below for the explanation of NAND and other gates).

The E Clock controls the timing of the chip enable line. Some memory chips use an active low (sometimes called "active false") signal to enable inputs, meaning that they are enabled when the enable input is 0. The method for denoting an input that is active low (i.e., 0) is shown in Figure 1.5, where the chip enable input connects to a circle; this circle indicates an active low input. Also, the name for the signal, CE, is prefixed with a ~ symbol.

Interrupt Handling

The microprocessor has a bank of interrupt vectors, as shown in Figure 1.5, which are hardware-defined locations in the memory address space where the microproces-

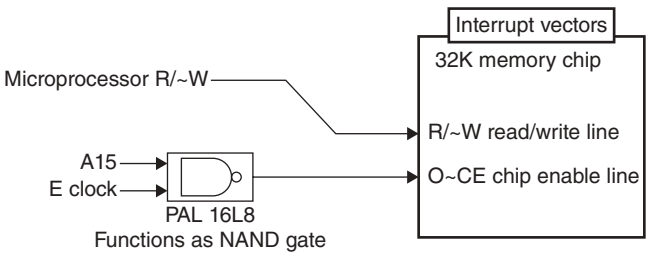

Figure 1.5 Enabling the memory.

Table 1.1 NOT Truth Table

Input	Output
A	$\overline{A}$
0	1
1	0

sor expects to find pointers to interrupt handling routines, for processing input and output data, arithmetic overflow, and so on. Also, when the microprocessor is reset, it finds the reset vector to determine where it should begin running a program. These vectors are located in the address space of the memory.

DIGITAL LOGIC

The fundamental logic operations of a microprocessor are performed by the following circuits. The results of those operations are represented in truth tables, where the binary value 0 is considered "low" (e.g., low voltage) and the binary value 1 is considered "high" (e.g., high voltage). While digital logic is used in the design of microprocessors, "everyday" examples are provided to show that the logic operations are not restricted to microprocessors.

NOT: represented in Table 1.1 and implemented with an inverter in Figure 1.6.

Application: The application is to complement the input A, producing the output $\overline{A}$.

Microprocessor example: the binary bit input was caused by an arithmetic overflow condition, so it is ignored and *not* used in the computation.

Everyday example: if we are to leave on an automobile trip, where A = 1 represents leaving at 1000, $\overline{A} = 0$ represents all times *not* equal to 1000.

OR: represented in Table 1.2 and implemented with OR gate in Figure 1.6.

Application: The application is to produce a 1 output if *any* or *both* of the inputs are 1.

Microprocessor example: the inputs are binary bits from memory stick or hard disk, so the microprocessor can accept *either* or *both* to perform a computation, depending on the current computer program instruction.

Everyday example: if A = 1 represents the decision to purchase a house and B = 1 represents the decision to purchase an automobile, Z = 1 represents the decision to purchase a house *or* an automobile *or* both.

AND: represented n Table 1.3 and implemented with an AND gate in Figure 1.6.

Application: The application is to produce a 1 output if *all* inputs are 1.

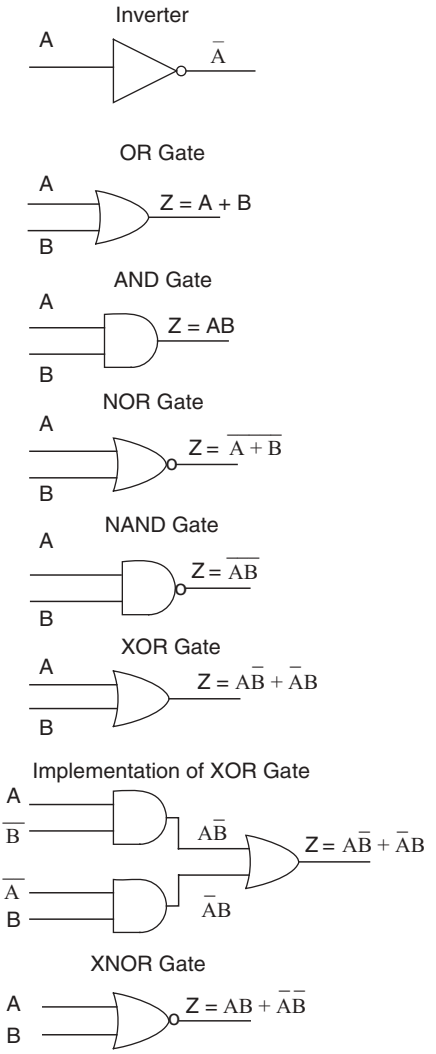

Figure 1.6 Logic operations.

Table 1.2 OR Truth Table

Input	Input	Output
A	B	Z = A + B
0	0	0
0	1	1
1	0	1
1	1	1

Table 1.3 AND Truth Table

Input	Input	Output
A	B	Z = AB
0	0	0
0	1	0
1	0	0
1	1	1

Table 1.4 NOR Truth Table

Input	Input	Output
A	B	$Z = \overline{A} + \overline{B}$
0	0	1
0	1	0
1	0	0
1	1	0

Table 1.5 NAND Truth Table

Input	Input	Output
A	B	$Z = \overline{AB}$
0	0	1
0	1	1
1	0	1
1	1	0

Microprocessor example: the microprocessor uses a signal Z = 1 to tell it that an interrupt has occurred on input line A *and* signifying that data input occurs on B, which the microprocessor will transfer to its memory.

Everyday example: if A = 1 represents a gas station and B = 1 represents a restaurant, we would stop our automobile at location Z, if Z has *both* a gas station *and* a restaurant.

NOR: represented in Table 1.4 and implemented with NOR gate in Figure 1.6.

Application: The application is to produce a 1 output if all inputs are 0.

Microprocessor example: the microprocessor Z = 1 output is recognized as interrupt code AB = 00.

Everyday example: if A = 0 represents the decision to *not* purchase a home and B = 0 represents the decision *not* to purchase an automobile, then Z = 1 represents the decision to *neither* purchase a home *nor* purchase an automobile.

NAND: represented in Table 1.5 and implemented with NAND gate in Figure 1.6.

Application: The application is to produce a 1 output if all inputs are *not* 1.

Microprocessor example: the microprocessor program produces the complement of the product of binary bits. This would be the case, for example, when Z = 1 signals that 0s occur on *either or both* of two input channels.

Everyday example: if A = 1 represents a gas station and B = 1 represents a restaurant, we would stop our automobile at location Z, if Z has only a gas station, or has only a restaurant, or has neither (i.e., rest stop).

Exclusive OR (XOR): represented in Table 1.6 and implemented with EXCLUSIVE OR gate in Figure 1.6. The figure also shows how the gate can be implemented, using AND and OR gates.

Application: The application is to produce a 1 output if *any* of the inputs is 1, but *not all* inputs are 1, and *not all* inputs are 0.

Microprocessor example: the main microprocessor receives a signal Z = 1 from the output of the I/O microprocessor that a binary bit A = 1 from a memory stick *or* B = 1 from a hard disk, and is ready for input, but these inputs are *not concurrent*.

Everyday example: if A = 1 represents the decision to purchase a house and B = 1 represents the decision to purchase an automobile, Z = 1 represents the decision to purchase a house *or* an automobile, but *not both at the same time*.

Exclusive NOR (XNOR): represented in Table 1.7 and implemented with XNOR gate in Figure 1.6. The *NOR* gate is the negation of the *XOR* gate from Table 1.6, as indicated in Table 1.7.

Table 1.6 EXCLUSIVE OR Truth Table

Input	Input	Output
A	B	$Z = A\overline{B} + \overline{A}B$
0	0	0
0	1	1
1	0	1
1	1	0

Table 1.7 EXCLUSIVE NOR (XNOR) Truth Table

Input	Input	Output
A	B	$Z = A\overline{B} + \overline{A}B = (A\overline{B})(\overline{A}B) = (\overline{A} + B)(A + \overline{B}) = \overline{A}A + AB + \overline{A}\overline{B} + B\overline{B} = AB + \overline{A}\overline{B}$
0	0	1
0	1	0
1	0	0
1	1	1

Application: The application is to produce a 1 output if all inputs are 0 *or* all inputs are 1.

Microprocessor example: Two hard drives are identified as A = 0 and A = 1; two flash memories are identified as B = 0, and B = 1. The microprocessor is programmed to input data from a hard drive and a flash memory *concurrently*. Therefore, it reads A = 0 *and* B = 0 *or* A = 1 *and* B = 1.

Everyday example: if A = 1 represents a gas station and B = 1 represents a restaurant, we would stop our automobile at location Z, if Z has *neither* a gas station *nor* a restaurant (i.e., rest stop) *or* has *both* a gas station and restaurant (i.e., get gas and eat).

De Morgan's theorem [GRE80] is used to simplify complex logic equations and the resultant digital logic. The theorem is used to simplify relatively simple expressions, as contrasted with Karnaugh maps (K-maps), described in the next section. The application of this theorem is shown in the following example:

$$\text{Theorem: } \overline{A+B} = \overline{A}\,\overline{B} \text{ and } \overline{AB} = \overline{A} + \overline{B}.$$

Suppose it is required to simplify $F = \overline{((\overline{AB})(\overline{AB}))}$.

Applying the theorem:

$$\overline{AB} = \overline{A} + \overline{B}, (\overline{AB})(\overline{AB}) = (\overline{A} + \overline{B})(\overline{A} + \overline{B})$$
$$= \overline{A}\ \overline{A} + \overline{A}\ \overline{B} + \overline{A}\ \overline{B} + \overline{B}\ \overline{B} = \overline{A} + \overline{A}\ \overline{B} + \overline{B} = \overline{A} + (\overline{A} + 1)\overline{B} = \overline{A} + \overline{B}$$
$$F = \overline{(\overline{A} + \overline{B})}(\overline{A} + \overline{B} = (\overline{\overline{A} + \overline{B}}) + (\overline{\overline{A} + \overline{B}}) = AB + AB = B.$$

Then, use Table 1.8 to demonstrate the equivalence between $\overline{((\overline{AB})(\overline{AB}))}$ and AB.

K-MAPS

A K-map in Table 1.9 is used to minimize a complex Boolean expression [RAF05]. Each square of a K-map represents a minterm (i.e., product terms). The process proceeds by listing the binary equivalents of the terms A and BC on the axes of Table 1.9, ordering them so that there is only a 1-bit difference between adjacent cells. Then, the minimum number of cells is enclosed. Next, minterms are identified

Table 1.8 Truth Table to Demonstrate Equivalence between F and AB

A	B	$\overline{AB}$	$\overline{AB}\,\overline{AB}$	$F = \overline{((\overline{AB})(\overline{AB}))}$	AB
0	0	1	1	0	0
0	1	1	1	0	0
1	0	1	1	0	0
1	1	0	0	1	1

according to terms that are common to all cells in the enclosure. Last, the product terms are summed. Notice what a clever method this is. Minimization is achieved by noting the combination of terms that yields the minimum difference!

Example: Simplify $F = \overline{A}\ \overline{B}\ \overline{C} + A\ \overline{B}\ \overline{C} + \overline{A}\ \overline{B}\ C + A\ \overline{B}\ C$.

Table 1.9 K-Map for $F = \overline{A}\overline{B}\overline{C} + A\overline{B}\overline{C} + \overline{A}\overline{B}C + A\overline{B}C$

		$\overline{B}\,\overline{C}$	$\overline{B}C$	BC	$B\overline{C}$
		00	01	11	10
$\overline{A}$	0	1	1		
A	1	1	1		

In minterm form, $F = \overline{A}\ \overline{B}\ \overline{C} + A\ \overline{B}\ \overline{C} + \overline{A}\ \overline{B}\ C + A\ \overline{B}\ C = \overline{B}$

In the K-map, $\overline{B}$ is common to the enclosed minterms. Therefore, $F = \overline{B}$. Table 1.10 demonstrates this result. The considerable reduction from the original function would result in significant savings in circuitry to implement the function.

Prime Implicant

A prime implicant is the *product term* obtained by enclosing the *maximum* number of adjacent cells in a K-map. For example, in the K-map of Table 1.9, $F = \overline{B}$ is a prime implicant. The prime implicant is only useful for providing a name for the maximum enclosure in a K-map.

Quine–McCluskey Method

This method is an alternative to the K-map for minimizing a Boolean function. The method is illustrated in Table 1.11 by minimizing the function $F = \overline{A}\ \overline{B}\ \overline{C} + A\ \overline{B}\ \overline{C} + \overline{A}\ \overline{B}\ C + A\ \overline{B}\ C$, where these minterms are placed in Table

Table 1.10 F Function Truth Table

A	B	C	$F = \overline{A}\ \overline{B}\ \overline{C} + A\ \overline{B}\ \overline{C} + \overline{A}\ \overline{B}\ C + A\ \overline{B}\ C$	$\overline{F} = \overline{B}$
0	0	0	1	1
0	0	1	1	1
0	1	0	0	0
0	1	1	0	0
1	0	0	1	1
1	0	1	1	1
1	1	0	0	0
1	1	1	0	0

Table 1.11 Quine–McCluskey Method for $F = \overline{A}\ \overline{B}\ \overline{C} + A\ \overline{B}\ \overline{C} + \overline{A}\ \overline{B}\ C + A\ \overline{B}\ C = \overline{B}$

Minterm		ABC	Difference of 1 Minterms		Difference of 1 Minterms	Minterms	Prime implicant
0	$\overline{A}\ \overline{B}\ \overline{C}$	000	0,1	00-	0,1,4,5	-0-	$\overline{B}$
1	$\overline{A}\ \overline{B}\ C$	001					
4	$A\ \overline{B}\ \overline{C}$	100	4,5	10-			
5	$A\ \overline{B}\ C$	101					

Table 1.12 One-Bit Adder Truth Table

A	B	Q	CO
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

1.11. This method is used to represent a difference of 1 between two adjacent minterms, such as $\overline{A}\ \overline{B}\ \overline{C}$ and $\overline{A}\ \overline{B}\ C$, yielding $\overline{A}\ \overline{B}\text{-} = 00\text{-}$. The symbol - is placed where there is a difference in minterm bit values, such as between 00- and 10- in Table 1.11, yielding -0-. This process continues until the four minterms 0, 1, 4, and 5 show a difference of 1 (00- compared with 10-), yielding prime implicant $\overline{B}$(-0-). The same result is obtained as was the case using the K-map in Table 1.9. Of the two methods, the K-map is easier to apply.

COMBINATIONAL CIRCUITS

These are circuits that use logic gates to produce outputs at any time that are only dependent on the *current* values of the inputs, meaning that it is not necessary to use a CP to trigger outputs [HAR07]. A typical combinational circuit is the adder.

One-Bit Adder with Carry Out

A and B are added, producing Q output and CO (carry out). Q and CO are implemented according to the truth table shown in Table 1.12.

Two-bit Adder with Carry In and CO

What if you want to add two 8-bit bytes? This becomes slightly harder. In this case, you need to create a full binary adder. The difference between a full adder and the

Table 1.13 Two-Bit Adder Truth Table

					Q = 1	CO = 1
CI	A	B	Q	CO	Minterms	Minterms
0	0	0	0	0		
0	0	1	1	0	$\overline{CI}\ \overline{A}\ B$	
0	1	0	1	0	$\overline{CI}\ A\ \overline{B}$	
0	1	1	0	1		$\overline{CI}\ A\ B$
1	0	0	1	0	$CI\ \overline{A}\ \overline{B}$	
1	0	1	0	1		$CI\ \overline{A}\ B$
1	1	0	0	1		$CI\ A\ \overline{B}$
1	1	1	1	1	$CI\ A\ B$	$CI\ A\ B$

Q Product Terms: $\overline{CI}\ \overline{A}\ B + \overline{CI}\ A\ \overline{B} + CI\ \overline{A}\ \overline{B} + CIAB$

$Q = \overline{CI}\ (\overline{A}\ B + A\ \overline{B}) + CI\ (\overline{A\ B} + AB)$

CO Product Terms: $\overline{CI}A\ B + CI\ \overline{A}\ B + CI\ A\ \overline{B} + CI\ A\ B = AB\ (\overline{CI} + CI) + CI(\overline{A}\ B + A\ \overline{B})$

$CO = AB + CI\ (\overline{A}\ B + A\ \overline{B})$

Table 1.14 K-Map for $Q = \overline{CI}\ \overline{A}\ B + \overline{CI}\ A\ \overline{B} + CI\ \overline{A}\ \overline{B} + CIAB = \overline{CI}(\overline{A}\ B + A\ \overline{B}) + CI(\overline{A}\ \overline{B} + AB)$

	AB			
CI	00	01	11	10
0		1		1
0	1		1	

$CI\overline{A}\ \overline{B}$ $\quad \overline{CI}\overline{A}\ B$ $\quad CIAB$ $\quad \overline{CI}A\ \overline{B}$

1-bit adder is that a full adder accepts A and B inputs plus a carry-in (CI) input. Once you have a full adder, you can string eight of them together to create a byte-wide adder and cascade the carry bit from one adder to the next. The truth table for a full adder is slightly more complicated than the previous truth table because now there are 3 input bits.

A combinational circuit minterm is represented by a product in a row of the truth table as shown in Table 1.13, corresponding to a 1 in the Q or CO output columns; for example, the fourth row for CO and the second row for Q in Table 1.13 [GIB80]. The values of Q and CO product terms are obtained by ORing the products in each row of Table 1.13 where Q = 1 or CO = 1, and then summing these terms, followed by simplifying the expressions, as demonstrated in Table 1.13. Further simplification *may* be possible by using a K-map.

As can be seen in Table 1.14, the adder output Q cannot be simplified by using a K-map because there are no adjacent cells. However, simplification is achieved

Table 1.15 K-Map for Carry Out $(CO) = \overline{C}IAB + C\overline{I}AB + CIA\overline{B} + CIAB = AB + CI(\overline{A}B + A\overline{B})$

	AB			
CI	00	01	11	10
0			1	
1			1	1

CI $\overline{A}$ B $\quad$ $\overline{C}IAB$ $\quad$ CIAB $\quad$ AB $\quad$ CIA $\overline{B}$

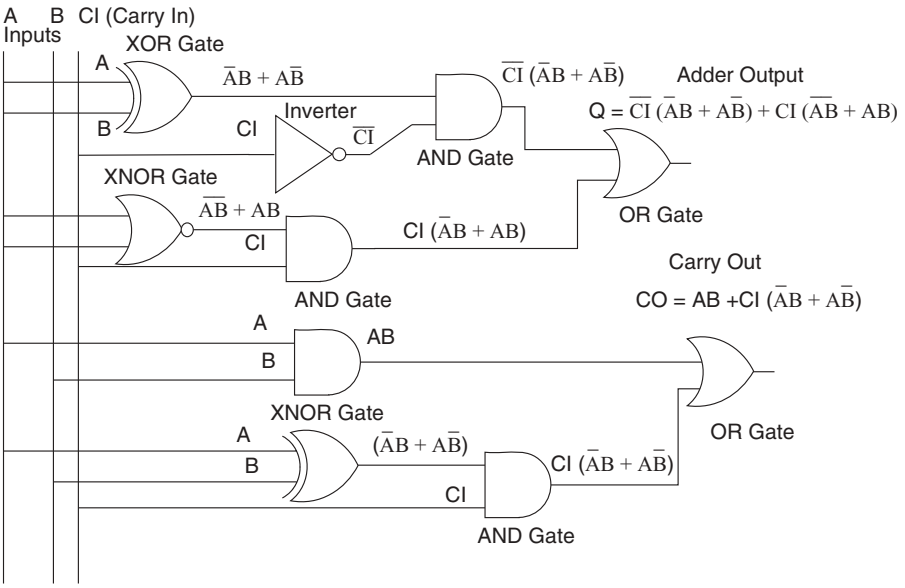

Figure 1.7 Adder circuit.

for CO, as shown in Table 1.15, producing $CO = AB + CI(\overline{A}B + A\overline{B})$. The relevant minterm cells in Table 1.15 that comprise the minimized function are outlined in red. Minterm logic is called *sum of products*. The full adder logic that corresponds to the minterms in Table 1.13 is shown in Figure 1.7, showing the adder output Q and the CO.

MULTIPLE OUTPUT COMBINATIONAL CIRCUITS

Combinational circuits can have multiple outputs [RAF05]. Each output is expressed as a function of the inputs, as shown in Table 1.16, where the inputs are binary-coded decimal (BCD) bits W, X, Y, and Z, corresponding to the decimal digits 0,…, 9. A

Table 1.16 Truth Table for Binary-Coded Decimal (BCD) Converter

Decimal digit	BCD input bits				Computer display segment output bits						
	W	X	Y	Z	a	b	c	d	e	f	g
0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	0
1	0	0	0	1	0	**1**	**1**	0	0	0	0
2	0	0	1	0	**1**	**1**	0	**1**	**1**	0	**1**
3	0	0	1	1	**1**	**1**	**1**	**1**	0	0	**1**
4	0	1	0	0	0	**1**	**1**	0	0	**1**	**1**
5	0	1	0	1	**1**	0	**1**	**1**	0	**1**	**1**
6	0	1	1	0	0	0	**1**	**1**	**1**	**1**	**1**
7	0	1	1	1	**1**	**1**	0	**1**	0	0	0
8	1	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**
9	1	0	0	1	**1**	**1**	**1**	0	0	**1**	**1**

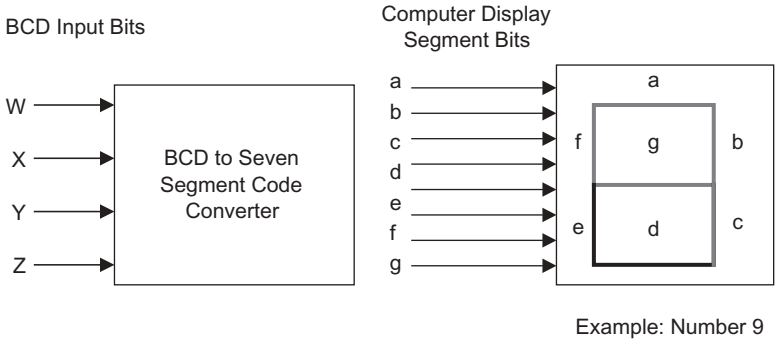

Figure 1.8 BCD to seven-segment code converter.

binary coded decimal converter is an example shown in Figure 1.8, showing how the number 9 can be displayed. The outputs are computer display segment bits a,…, g that represent the 1s necessary to generate the display decimal numbers. The code converter transforms the BCD numbers 0000,…, 1001 to display segments. The converter does not represent decimal numbers greater than 9. The K-maps use "don't cares" = Xs in order to simplify the logic; the "don't cares" should not be confused with the BCD bit = X in Table 1.16. The "don't cares" are used to advantage in forming minterms, as, for example, in Tables 1.17–1.23.

In order to generate the K-maps, place a 1 in the K-map cells corresponding to the 1s that appear in Table 1.16. For example, for *segment a* in Table 1.17, a 1 is recorded in the cell WXYZ = 0000, corresponding to the **1** (bolded) in the *segment a* column in Table 1.16.

The K-maps will lead to simplifying the equations for the seven-segment computer display (Fig. 1.8). The equations will then be used to design the digital logic circuit in Figures 1.9 and 1.10.