Digital Computer Electronics

An Introduction to Microcomputers

Second Edition

Albert Paul Malvino, Ph.D.

Gregg Division
McGraw-Hill Book Company

New York Atlanta Dallas St. Louis San Francisco
Auckland Bogotá Guatemala Hamburg Johannesburg
Lisbon London Madrid Mexico Montreal New Delhi
Panama Paris San Juan São Paulo Singapore Sydney
Tokyo Toronto

ALSO BY ALBERT P. MALVINO

Electronic Principles

Experiments for Electronic Principles (with G. Johnson)

Transistor Circuit Approximations

Experiments for Transistor Circuit Approximations

Resistive and Reactive Circuits

Electronic Instrumentation Fundamentals

Digital Principles and Applications (with D. Leach)

Sponsoring Editor: Paul Berk
Editing Supervisors: Tim Perrin and Larry Goldberg
Design and Art Supervisors: Nancy Axelrod and Meri Shardin
Production Supervisor: Priscilla Taguer

Text Designer: Ampersand Studio
Cover Designer: Ampersand Studio
Cover Illustrator: Jon Weiman
Technical Studio: Fine Line, Inc.

Library of Congress Cataloging in Publication Data

Malvino, Albert Paul.
Digital computer electronics.

Includes index.
1. Electronic digital computers.
2. Microcomputers. 3. INTEL 8085 (Computer)
I. Title.
TK7888.3.M337 1982 621.3819'58 82-8952
ISBN 0-07-039901-8 AACR2

Digital Computer Electronics: An Introduction to Microcomputers, Second Edition

1 2 3 4 5 6 7 8 9 0 SEMSEM 8 9 8 7 6 5 4 3 2

ISBN 0-07-039901-8

To my wife, Joanna, who encourages me to write.
And to my daughters, Joanna, Antonia, Lucinda,
Patricia, and Miriam, who keep me young.

CONTENTS

PREFACE

Textbooks on microprocessors and microcomputers are very often hard to understand. Sometimes it seems as if something important had been left out of the discussion; this book is my attempt at putting everything back in.

The early chapters of *Digital Computer Electronics*, Second Edition, cover digital theory and devices. In later chapters this information is applied to microprocessors, and finally, you will learn about the construction and operation of microcomputer systems. The only prerequisite to using this textbook is an understanding of diodes and transistors.

I have featured the 8085 microprocessor (an enhanced version of the 8080) because this 8-bit device is an ideal subject of study for a fundamental microcomputer textbook. Once you understand the 8085, you pass a major hurdle and things begin to make sense in the microcomputer world.

To help you master the 8085, you will first study an educational computer called SAP (simple-as-possible). This computer has three generations: SAP-1, SAP-2, and SAP-3. SAP-1 is a bare-bones computer built with TTL chips. You will see every wire, every signal, and every circuit used in this elementary computer. This will reinforce your grasp of digital electronics and prepare you to understand the more advanced computer concepts in SAP-2 and SAP-3. Many of the operational details of the 8080 and 8085 microprocessors are covered in SAP-2 and SAP-3. After studying these you will have learned almost the entire 8080/8085 instruction set.

The later chapters discuss advanced microcomputer topics such as handshaking, interrupts, memory shadows, and D/A and A/D conversion. When finished with this book, you will have a deep and solid understanding of microcomputer basics. With that kind of foundation you will find it relatively easy to branch out to systems that use other 8-bit, as well as 16-bit, microprocessors.

A correlated laboratory manual, *Experiments for Digital Computer Electronics* by Michael A. Miller, is available for use with this textbook. Early experiments cover the basics of digital electronics: gates, adders, flip-flops, and more. Later experiments are about program counters, instruction decoders, and accumulators. In the final experiments you assemble and program a SAP-1 computer.

During the preparation of this textbook, many people made valuable suggestions. I want to thank Charles Counts of Intel Corporation, Michael A. Miller of the DeVry Institute of Technology, William H. Murray of Broome Community College, Richard Raines of Shasta College, Michael Slater of Logical Services Incorporated, and the staff of the Sylvania Technical School.

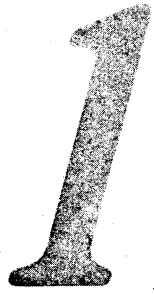
Albert Paul Malvino

A man of true science uses but few hard words,
and those only when none other will answer his purpose;
whereas the smatterer in science thinks that
by mouthing hard words he understands hard things.

Herman Melville

Digital Computer Electronics

Number Systems and Codes



Modern computers don't work with decimal numbers. Instead, they process *binary numbers*, groups of 0s and 1s. Why binary numbers? Because electronic devices are most reliable when designed for two-state (binary) operation. This chapter discusses binary numbers and other concepts needed to understand computer operation.

1-1 DECIMAL ODOMETER

René Descartes (1596–1650) said that the way to learn a new subject is to go from the known to the unknown, from the simple to the complex. Let's try it.

The Known

Everyone has seen an odometer (miles indicator) in action. When a car is new, its odometer starts with

00000

After 1 mile the reading becomes

00001

Successive miles produce 00002, 00003, and so on, up to

00009

A familiar thing happens at the end of the tenth mile. When the units wheel turns from 9 back to 0, a tab on this wheel forces the tens wheel to advance by 1. This is why the numbers change to

00010

Reset-and-Carry

The units wheel has reset to 0 and sent a carry to the tens wheel. Let's call this familiar action *reset-and-carry*.

The other wheels also reset and carry. After 999 miles the odometer shows

00999

What does the next mile do? The units wheel resets and carries, the tens wheel resets and carries, the hundreds wheel resets and carries, and the thousands wheel advances by 1, to get

01000

Digits and Strings

The numbers on each odometer wheel are called *digits*. The decimal number system uses ten digits, 0 through 9. In a decimal odometer, each time the units wheel runs out of digits, it resets to 0 and sends a carry to the tens wheel. When the tens wheel runs out of digits, it resets to 0 and sends a carry to the hundreds wheel. And so on with the remaining wheels.

One more point. A *string* is a group of characters (either letters or digits) written one after another. For instance, 734 is a string of 7, 3, and 4. Similarly, 2C8A is a string of 2, C, 8, and A.

1-2 BINARY ODOMETER

Binary means two. The binary number system uses only two digits, 0 and 1. All other digits (2 through 9) are thrown away. In other words, binary numbers are strings of 0s and 1s.

An Unusual Odometer

Visualize an odometer whose wheels have only two digits, 0 and 1. When each wheel turns, it displays 0, then 1, then

back to 0, and the cycle repeats. Because each wheel has only two digits, we call this device a *binary odometer*.

In a car a binary odometer starts with

0000 (zero)

After 1 mile, it indicates

0001 (one)

The next mile forces the units wheel to reset and carry; so the numbers change to

0010 (two)

The third mile results in

0011 (three)

What happens after 4 miles? The units wheel resets and carries, the second wheel resets and carries, and the third wheel advances by 1. This gives

0100 (four)

Successive miles produce

0101 (five)
0110 (six)
0111 (seven)

After 8 miles, the units wheel resets and carries, the second wheel resets and carries, the third wheel resets and carries, and the fourth wheel advances by 1. The result is

1000 (eight)

The ninth mile gives

1001 (nine)

and the tenth mile produces

1010 (ten)

(Try working out a few more readings on your own.)

You should have the idea by now. Each mile advances the units wheel by 1. Whenever the units wheel runs out of digits, it resets and carries. Whenever the second wheel runs out of digits, it resets and carries. And so for the other wheels.

Binary Numbers

A binary odometer displays binary numbers, strings of 0s and 1s. The number 0001 stands for 1, 0010 for 2, 0011 for 3, and so forth. Binary numbers are long when large amounts are involved. For instance, 101010 represents decimal 42. As another example, 111100001111 stands for decimal 3,855.

Computer circuits are like binary odometers; they count and work with binary numbers. Therefore, you have to learn to count with binary numbers, to convert them to decimal numbers, and to do binary arithmetic. Then you will be ready to understand how computers operate.

A final point. When a decimal odometer shows 0036, we can drop the leading 0s and read the number as 36. Similarly, when a binary odometer indicates 0011, we can drop the leading 0s and read the number as 11. With the leading 0s omitted, the binary numbers are 0, 1, 10, 11, 100, 101, and so on. To avoid confusion with decimal numbers, read the binary numbers like this: zero, one, one-zero, one-one, one-zero-zero, one-zero-one, etc.

1-3 NUMBER CODES

People used to count with pebbles. The numbers 1, 2, 3 looked like ●, ●●, ●●●. Larger numbers were worse: seven appeared as ●●●●●●●.

Codes

From the earliest times, people have been creating codes that allow us to think, calculate, and communicate. The decimal numbers are an example of a code (see Table 1-1). It's an old idea now, but at the time it was as revolutionary; 1 stands for ●, 2 for ●●, 3 for ●●●, and so forth.

Table 1-1 also shows the binary code. 1 stands for ●, 10 for ●●, 11 for ●●●, and so on. A binary number and a decimal number are equivalent if each represents the same amount of pebbles. Binary 10 and decimal 2 are equivalent because each represents ●●. Binary 101 and decimal 5 are equivalent because each stands for ●●●●●

TABLE 1-1. NUMBER CODES

Decimal	Pebbles	Binary
0	None	0
1	●	1
2	●●	10
3	●●●	11
4	●●●●	100
5	●●●●●	101
6	●●●●●●	110
7	●●●●●●●	111
8	●●●●●●●●	1000
9	●●●●●●●●●	1001

Equivalence is the common ground between us and computers; it tells us when we're talking about the same thing. If a computer comes up with a binary answer of 101, equivalence means that the decimal answer is 5. As a start to understanding computers, memorize the binary-decimal equivalences of Table 1-1.

EXAMPLE 1-1

Figure 1-1*a* shows four light-emitting diodes (LEDs). A dark circle means that the LED is off; a light circle means it's on. To read the display, use this code:

(a) (b)

Fig. 1-1 LED display of binary numbers.

LED	Binary
Off	0
On	1

What binary number does Fig. 1-1*a* indicate? Fig. 1-1*b*?

SOLUTION

Figure 1-1*a* shows off-off-on-on. This stands for binary 0011, equivalent to decimal 3.

Figure 1-1*b* is off-on-off-on, decoded as binary 0101 and equivalent to decimal 5.

EXAMPLE 1-2

A binary odometer has four wheels. What are the successive binary numbers?

SOLUTION

As previously discussed, the first eight binary numbers are 0000, 0001, 0010, 0011, 0100, 0101, 0110, and 0111. On the next count, the three wheels on the right reset and carry; the fourth wheel advances by one. So the next eight numbers are 1000, 1001, 1010, 1011, 1100, 1101, 1110, and 1111. The final reading of 1111 is equivalent to decimal 15. The next mile resets all wheels to 0, and the cycle repeats.

Being able to count in binary from 0000 to 1111 is essential for understanding the operation of computers.

TABLE 1-2. BINARY-TO-DECIMAL EQUIVALENCES

Decimal	Binary	Decimal	Binary
0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1011
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	15	1111

Therefore, you should memorize the equivalences of Table 1-2.

1-4 WHY BINARY NUMBERS ARE USED

The word "computer" is misleading because it suggests a machine that can solve only numerical problems. But a computer is more than an automatic adding machine. It can play games, translate languages, draw pictures, and so on. To suggest this broad range of application, a computer is often referred to as a *data processor*.

Program and Data

Data means names, numbers, facts, anything needed to work out a problem. Data goes into a computer, where it is processed or manipulated to get new information. Before it goes into a computer, however, the data must be coded in binary form. The reason was given earlier: a computer's circuits can respond only to binary numbers.

Besides the data, someone has to work out a *program*, a list of instructions telling the computer what to do. These instructions spell out each and every step in the data processing. Like the data, the program must be coded in binary form before it goes into the computer.

So the two things we must input to a computer are the program and the data. These are stored inside the computer before the processing begins. Once the computer run starts, each instruction is executed and the data is processed.

Hardware and Software

The electronic, magnetic, and mechanical devices of a computer are known as *hardware*. Programs are called *software*. Without software, a computer is a pile of "dumb" metal.

An analogy may help. A phonograph is like hardware and records are like software. The phonograph is useless without records. Furthermore, the music you get depends on the record you play. A similar idea applies to computers. A computer is the hardware and programs are the software. The computer is useless without programs. The program stored in the computer determines what the computer will do; change the program and the computer processes the data in a different way.

Transistors

Computers use *integrated circuits* (ICs) with thousands of transistors, either bipolar or MOS. The parameters (β_{dc}, I_{CO}, g_m, etc.) can vary more than 50 percent with temperature change and from one transistor to the next. Yet these computer ICs work remarkably well despite the transistor variations. How is it possible?

The answer is *two-state* design, using only two points on the load line of each transistor. For instance, the common two-state design is the cutoff-saturation approach; each transistor is forced to operate at either cutoff or saturation. When a transistor is cut off or saturated, parameter variations have almost no effect. Because of this, it's possible to design reliable two-state circuits that are almost independent of temperature change and transistor variations.

Transistor Register

Here's an example of two-state design. Figure 1-2 shows a transistor register. (A *register* is a string of devices that store data.) The transistors on the left are cut off because the input base voltages are 0 V. The dark shading symbolizes the cutoff condition. The two transistors on the right have base drives of 5 V.

The transistors operate at either saturation or cutoff. A base voltage of 0 V forces each transistor to cut off, while a base voltage of 5 V drives it into saturation. Because of this two-state action, each transistor stays in a given state until the base voltage switches it to the opposite state.

Another Code

Two-state operation is universal in digital electronics. By deliberate design, all input and output voltages are either low or high. Here's how binary numbers come in: low voltage represents binary 0, and high voltage stands for binary 1. In other words, we use this code:

Voltage	Binary
Low	0
High	1

For instance, the base voltages of Fig. 1-2 are low-low-high-high, or binary 0011. The collector voltages are high-high-low-low, or binary 1100. By changing the base voltages we can store any binary number from 0000 to 1111 (decimal 0 to 15).

Bit

Bit is an abbreviation for binary digit. A binary number like 1100 has 4 bits; 110011 has 6 bits; and 11001100 has 8 bits. Figure 1-2 is a 4-bit register. To store larger binary numbers, it needs more transistors. Add two transistors and you get a 6-bit register. With four more transistors, you'd have an 8-bit register.

Nonsaturated Circuits

Don't get the idea that all two-state circuits switch between cutoff and saturation. When a bipolar transistor is heavily saturated, extra carriers are stored in the base region. If the base voltage suddenly switches from high to low, the transistor cannot come out of saturation until these extra carriers have a chance to leave the base region. The time it takes for these carriers to leave is called the *saturation delay time* t_d. Typically, t_d is in nanoseconds.

In most applications the saturation delay time is too short to matter. But some applications require the fastest possible

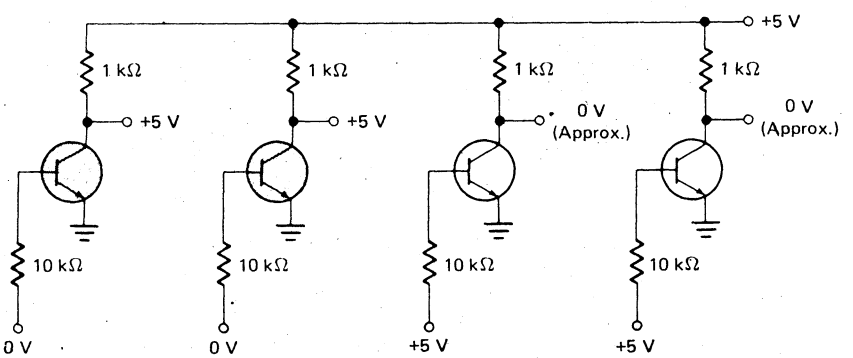


Fig. 1-2 Transistor register.

switching time. To get this maximum speed, designers have come up with circuits that switch from cutoff (or near cutoff) to a higher point on the load line (but short of saturation). These nonsaturated circuits rely on clamping diodes or heavy negative feedback to overcome transistor variations.

Remember this: whether saturated or nonsaturated circuits are used, the transistors switch between distinct points on the load line. This means that all input and output voltages are easily recognized as low or high, binary 0 or binary 1.

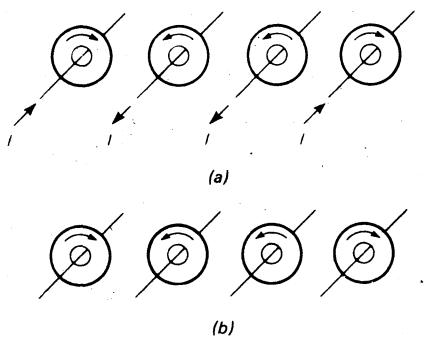


Fig. 1-3 Core register.

Magnetic Cores

In some digital computers, magnetic cores store binary data. Figure 1-3*a* shows a 4-bit core register. With the right-hand rule, you can see that conventional current into a wire produces a clockwise flux; reversing the current gives a counterclockwise flux. (The same result is obtained if electron-flow is assumed and the left-hand rule is used.)

The cores have rectangular hysteresis loops; this means that flux remains in a core even though the magnetizing current is removed (see Fig. 1-3*b*). This is why a core register can store binary data indefinitely. For instance, let's use the following code:

Flux	Binary
Counterclockwise	0
Clockwise	1

Then, the core register of Fig. 1-3*b* stores binary 1001, equivalent to decimal 9. By changing the magnetizing currents in Fig. 1-3*a* we can change the stored data.

To store larger binary numbers, add more cores. Two cores added to Fig. 1-3*a* result in a 6-bit register; four more cores give an 8-bit register.

The *memory* is one of the main parts of a computer. Some memories contain thousands of core registers. These registers store the program and data needed to run the computer.

Other Two-State Examples

The simplest example of a two-state device is the on-off switch. When this switch is closed, it represents binary 1; when it's open, it stands for binary 0.

Punched cards are another example of the two-state concept. A hole in a card stands for binary 1, the absence of a hole for binary 0. Using a prearranged code, a card-punch machine with a keyboard can produce a stack of cards containing the program and data needed to run a computer.

Magnetic tape can also store binary numbers. Tape recorders magnetize some points on the tape (binary 1), while leaving other points unmagnetized (binary 0). By a prearranged code, a row of points represents either a coded instruction or data. In this way, a reel of tape can store thousands of binary instructions and data for later use in a computer.

Even the lights on the control panel of a large computer are binary; a light that's on stands for binary 1, and one that's off stands for binary 0. In a 16-bit computer, for instance, a row of 16 lights allows the operator to see the binary contents in different computer registers. The operator can then monitor the overall operation and, when necessary, troubleshoot.

In summary, switches, transistors, cores, cards, tape, lights, and almost all other devices used with computers are based on two-state operation. This is why we are forced to use binary numbers when analyzing computer action.

EXAMPLE 1-3

Figure 1-4 shows a strip of magnetic tape. The black circles are magnetized points and the white circles unmagnetized points. What binary number does each horizontal row represent?

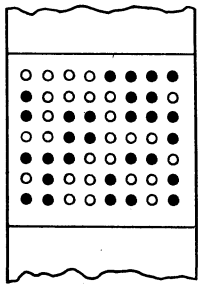


Fig. 1-4 Binary numbers on magnetic tape.

SOLUTION

The tape stores these binary numbers:

Row 1	00001111	Row 5	11100110
Row 2	10000110	Row 6	01001001
Row 3	10110111	Row 7	11001101
Row 4	00110001		

(*Note:* these binary numbers may represent either coded instructions or data.)

A string of 8 bits is called a *byte*. In this example, the magnetic tape stores 7 bytes. The first byte (row 1) is 00001111. The second byte (row 2) is 10000110. The third byte is 10110111. And so on.

A byte is the basic unit of data in computers. Most computers process data in strings of 8 bits or some multiple (16, 24, 32, and so on). Likewise, the memory stores data in strings of 8 bits or some multiple of 8 bits.

1-5 BINARY-TO-DECIMAL CONVERSION

You already know how to count to 15 using binary numbers. The next thing to learn is how to convert larger binary numbers to their decimal equivalents.

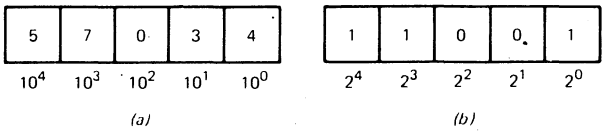


Fig. 1-5 (*a*) Decimal weights; (*b*) binary weights.

Decimal Weights

The decimal number system is an example of positional notation: each digit position has a *weight* or value. With decimal numbers the weights are units, tens, hundreds, thousands, and so on. The sum of all digits multiplied by their weights gives the total amount being represented.

For instance, Fig. 1-5*a* illustrates a decimal odometer. Below each digit is its weight. The digit on the right has a weight of 10^0 (units), the second digit has a weight of 10^1 (tens), the third digit a weight of 10^2 (hundreds), and so forth. The sum of all units multiplied by their weights is

$$(5 \times 10^4) + (7 \times 10^3) + (0 \times 10^2) + (3 \times 10^1) + (4 \times 10^0) = 50{,}000 + 7000 + 0 + 30 + 4 = 57{,}034$$

Binary Weights

Positional notation is also used with binary numbers because each digit position has a weight. Since only two digits are used, the weights are powers of 2 instead of 10. As shown in the binary odometer of Fig. 1-5*b*, these weights are 2^0 (units), 2^1 (twos), 2^2 (fours), 2^3 (eights), and 2^4 (sixteens). If longer binary numbers are involved, the weights continue in ascending powers of 2.

The decimal equivalent of a binary number equals the sum of all binary digits multiplied by their weights. For instance, the binary reading of Fig. 1-5*b* has a decimal equivalent of

$$(1 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 16 + 8 + 0 + 0 + 1 = 25$$

Binary 11001 is therefore equivalent to decimal 25.

As another example, the byte 11001100 converts to decimal as follows:

$$(1 \times 2^7) + (1 \times 2^6) + (0 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0) = 128 + 64 + 0 + 0 + 8 + 4 + 0 + 0 = 204$$

So, binary 11001100 is equivalent to decimal 204.

Fast and Easy Conversion

Here's a streamlined way to convert a binary number to its decimal equivalent:

1. Write the binary number.
2. Write the weights 1, 2, 4, 8, . . . , under the binary digits.
3. Cross out any weight under a 0.
4. Add the remaining weights.

For instance, binary 1101 converts to decimal as follows:

1.	1 1 0 1	(Write binary number)
2.	8 4 2 1	(Write weights)
3.	8 4 ~~2~~ 1	(Cross out weights)
4.	8 + 4 + 0 + 1 = 13	(Add weights)

You can compress the steps even further:

1 1 0 1	(Step 1)
8 4 ~~2~~ 1 → 13	(Steps 2 to 4)

As another example, here's the conversion of binary 1110101 in compressed form:

1 1 1 0 1 0 1	
64 32 16 ~~8~~ 4 ~~2~~ 1 → 117	

Base or Radix

The *base* or *radix* of a number system equals the number of digits it has. Decimal numbers have a base of 10 because digits 0 through 9 are used. Binary numbers have a base of 2 because only the digits 0 and 1 are used. (In terms of an odometer, the base or radix is the number of digits on each wheel.)

A subscript attached to a number indicates the base of the number. 100_2 means binary 100. On the other hand, 100_{10} stands for decimal 100. Subscripts help clarify equations where binary and decimal numbers are mixed. For instance, the last two examples of binary-to-decimal conversion can be written like this:

$$1101_2 = 13_{10}$$
$$1110101_2 = 117_{10}$$

In this book we will use subscripts when necessary for clarity.

1-6 MICROPROCESSORS

What is inside a computer? What is a microprocessor? What is a microcomputer?

Computer

The five main sections of a computer are input, memory, arithmetic and logic, control, and output. Here is a brief description of each.

Input This consists of all the circuits needed to get programs and data into the computer. In some computers the input section includes a typewriter keyboard that converts letters and numbers into strings of binary data.

Memory This stores the program and data before the computer run begins. It also can store partial solutions during a computer run, similar to the way we use a scratchpad while working out a problem.

Control This is the computer's center of gravity, analogous to the conscious part of the mind. The control section directs the operation of all other sections. Like the conductor of an orchestra, it tells the other sections what to do and when to do it.

Arithmetic and logic This is the number-crunching section of the machine. It can also make logical decisions. With control telling it what to do and with memory feeding it data, the arithmetic-logic unit (ALU) grinds out answers to number and logic problems.

Output This passes answers and other processed data to the outside world. The output section usually includes a video display to allow the user to see the processed data.

Microprocessor

The control section and the ALU are often combined physically into a single unit called the *central processing unit* (CPU). Furthermore, it's convenient to combine the input and output sections into a single unit called the *input-output* (I/O) *unit*. In earlier computers, the CPU, memory, and I/O unit filled an entire room.

With the advent of integrated circuits, the CPU, memory, and I/O unit have shrunk dramatically. Nowadays the CPU can be fabricated on a single semiconductor chip called a *microprocessor*. In other words, a microprocessor is nothing more than a CPU on a chip.

Likewise, the I/O circuits and memory can be fabricated on chips. In this way, the computer circuits that once filled a room now fit on a few chips.

Microcomputer

As the name implies, a *microcomputer* is a small computer. More specifically, a microcomputer is a computer that uses a microprocessor for its CPU. The typical microcomputer has three kinds of chips: microprocessor (usually one chip), memory (several chips), and I/O (one or more chips).

If a small memory is acceptable, a manufacturer can fabricate all computer circuits on a single chip. For instance, the 8048 from Intel Corporation is a one-chip microcomputer with an 8-bit CPU, 1,088 bytes of memory, and 27 I/O lines.

Powers of 2

Microprocessor design started with 4-bit devices, then evolved to 8- and 16-bit devices. In our later discussions of microprocessors, powers of 2 keep coming up because of the binary nature of computers. For this reason, you should study Table 1-3. It lists the powers of 2 encountered in microcomputer analysis. As shown, the abbreviation K stands for 1,024 (approximately 1,000).† Therefore, 1K means 1,024, 2K stands for 2,048, 4K for 4,096, and so on.

Some personal microcomputers have 64K memories that can store up to 65,536 bytes.

TABLE 1-3. POWERS OF 2

Powers of 2	Decimal equivalent	Abbreviation
2^0	1	
2^1	2	
2^2	4	
2^3	8	
2^4	16	
2^5	32	
2^6	64	
2^7	128	
2^8	256	
2^9	512	
2^{10}	1,024	1K
2^{11}	2,048	2K
2^{12}	4,096	4K
2^{13}	8,192	8K
2^{14}	16,384	16K
2^{15}	32,768	32K
2^{16}	65,536	64K

† The abbreviations 1K, 2K, and so on, became established before K- for *kilo-* was in common use. Retaining the capital K serves as a useful reminder that K only approximates 1,000.

1-7 DECIMAL-TO-BINARY CONVERSION

Next, you need to know how to convert from decimal to binary. After you know how it's done, you will be able to understand how circuits can be built to convert decimal numbers into binary numbers.

Double-Dabble

Double-dabble is a way of converting any decimal number to its binary equivalent. It requires successive division by 2, writing down each quotient and its remainder. The remainders are the binary equivalent of the decimal number. The only way to understand the method is to go through an example, step by step.

Here is how to convert decimal 13 to its binary equivalent.
Step 1. Divide 13 by 2, writing your work like this:

$$\begin{array}{rl} 6 & 1 \rightarrow \text{(first remainder)} \\ 2\overline{)13} & \end{array}$$

The quotient is 6 with a remainder of 1.
Step 2. Divide 6 by 2 to get

$$\begin{array}{rl} 3 & 0 \rightarrow \text{(second remainder)} \\ 2\overline{)6} & 1 \\ 2\overline{)13} & \end{array}$$

This division gives 3 with a remainder of 0.
Step 3. Again you divide by 2:

$$\begin{array}{rl} 1 & 1 \rightarrow \text{(third remainder)} \\ 2\overline{)3} & 0 \\ 2\overline{)6} & 1 \\ 2\overline{)13} & \end{array}$$

Here you get a quotient of 1 and a remainder of 1.
Step 4. One more division by 2 gives

$$\begin{array}{rl} & \text{Read down} \\ 0 & 1 \\ 2\overline{)1} & 1 \\ 2\overline{)3} & 0 \\ 2\overline{)6} & 1 \\ 2\overline{)13} & \end{array} \quad \Big\downarrow$$

In this final division, 2 does not divide into 1; therefore, the quotient is 0 with a remainder of 1.

Whenever you arrive at a quotient of 0 with a remainder of 1, the conversion is finished. The remainders when read downward give the binary equivalent. In this example, binary 1101 is equivalent to decimal 13.

Double-dabble works with any decimal number. Progressively divide by 2, writing each quotient and its remainder. When you reach a quotient of 0 and a remainder of 1, you are finished; the remainders read downward are the binary equivalent of the decimal number.

Streamlined Double-Dabble

There's no need to keep writing down 2 before each division because you're always dividing by 2. From now on, here's how to show the conversion of decimal 13 to its binary equivalent:

$$\begin{array}{rl} 0 & 1 \\ \overline{)1} & 1 \\ \overline{)3} & 0 \\ \overline{)6} & 1 \\ 2\overline{)13} & \end{array} \quad \Big\downarrow$$

EXAMPLE 1-4

Convert decimal 23 to binary.

SOLUTION

The first step in the conversion looks like this:

$$\begin{array}{rl} 11 & 1 \\ 2\overline{)23} & \end{array}$$

After all divisions, the finished work looks like this:

$$\begin{array}{rl} 0 & 1 \\ \overline{)1} & 0 \\ \overline{)2} & 1 \\ \overline{)5} & 1 \\ \overline{)11} & 1 \\ 2\overline{)23} & \end{array} \quad \Big\downarrow$$

This says that binary 10111 is equivalent to decimal 23.

1-8 HEXADECIMAL NUMBERS

Hexadecimal numbers are extensively used in microprocessor work. To begin with, they are much shorter than binary numbers. This makes them easy to write and remember. Furthermore, you can mentally convert them to binary form whenever necessary.

An Unusual Odometer

Hexadecimal means 16. The hexadecimal number system has a base or radix of 16. This means that it uses 16 digits to represent all numbers. The digits are 0 through 9, and A through F as follows: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. Hexadecimal numbers are strings of these digits like 8A5, 4CF7, and EC58.

An easy way to understand hexadecimal numbers is to visualize a hexadecimal odometer. Each wheel has 16 digits on its circumference. As it turns, it displays 0 through 9 as before. But then, instead of resetting, it goes on to display A, B, C, D, E, and F.

The idea of reset and carry applies to a hexadecimal odometer. When a wheel turns from F back to 0, it forces the next higher wheel to advance by 1. In other words, when a wheel runs out of hexadecimal digits, it resets and carries.

If used in a car, a hexadecimal odometer would count as follows. When the car is new, the odometer shows all 0s:

0000 (zero)

The next 9 miles produce readings of

0001 (one)
0002 (two)
0003 (three)
0004 (four)
0005 (five)
0006 (six)
0007 (seven)
0008 (eight)
0009 (nine)

The next 6 miles give

000A (ten)
000B (eleven)
000C (twelve)
000D (thirteen)
000E (fourteen)
000F (fifteen)

At this point the least significant wheel has run out of digits. Therefore, the next mile forces a reset-and-carry to get

0010 (sixteen)

The next 15 miles produce these readings: 0011, 0012, 0013, 0014, 0015, 0016, 0017, 0018, 0019, 001A, 001B, 001C, 001D, 001E, and 001F. Once again, the least significant wheel has run out of digits. So, the next mile results in a reset-and-carry:

0020 (thirty-two)

Subsequent readings are 0021, 0022, 0023, 0024, 0025, 0026, 0027, 0028, 0029, 002A, 002B, 002C, 002D, 002E, and 002F.

You should have the idea by now. Each mile advances the least significant wheel by 1. When this wheel runs out of hexadecimal digits, it resets and carries. And so on for the other wheels. For instance, if the odometer reading is

835F

the next reading is 8360. As another example, given

5FFF

the next hexadecimal number is 6000.

Equivalences

Table 1-4 shows the equivalences between hexadecimal, binary, and decimal digits. Memorize this table. It's essential that you be able to convert instantly from one system to another.

TABLE 1-4. EQUIVALENCES

Hexadecimal	Binary	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

1-9 HEXADECIMAL-BINARY CONVERSIONS

After you know the equivalences of Table 1-4, you can mentally convert any hexadecimal string to its binary equivalent and vice versa.

Hexadecimal to Binary

To convert a hexadecimal number to a binary number, convert each hexadecimal digit to its 4-bit equivalent, using Table 1-4. For instance, here's how 9AF converts to binary:

9	A	F
↓	↓	↓
1001	1010	1111

As another example, C5E2 converts like this:

C	5	E	2
↓	↓	↓	↓
1100	0101	1110	0010

Incidentally, for easy reading it's common practice to leave a space between the 4-bit strings. For example, instead of writing

$$C5E2_{16} = 1100010111100010_2$$

we can write

$$C5E2_{16} = 1100\ 0101\ 1110\ 0010_2$$

Binary to Hexadecimal

To convert in the opposite direction, from binary to hexadecimal, you again use Table 1-4. Here are two examples. The byte 1000 1100 converts as follows:

1000	1100
↓	↓
8	C

The 16-bit number 1110 1000 1101 0110 converts like this:

1110	1000	1101	0110
↓	↓	↓	↓
E	8	D	6

In both these conversions, we start with a binary number and wind up with the equivalent hexadecimal number.

EXAMPLE 1-5

Solve the following equation for x:

$$x_{16} = 1111\ 1111\ 1111\ 1111_2$$

SOLUTION

This is the same as asking for the hexadecimal equivalent of binary 1111 1111 1111 1111. Since hexadecimal F is equivalent to 1111, x = FFFF. Therefore,

$$FFFF_{16} = 1111\ 1111\ 1111\ 1111_2$$

EXAMPLE 1-6

As mentioned earlier, the memory contains thousands of registers (core or semiconductor) that store the program and data needed for a computer run. These memory registers are known as *memory locations*. A typical microcomputer may have up to 65,536 memory locations, each storing 1 byte.

Suppose the first 16 memory locations contain these bytes:

0011 1100
1100 1101
0101 0111
0010 1000
1111 0001
0010 1010
1101 0100
0100 0000
0111 0111
1100 0011
1000 0100
0010 1000
0010 0001
0011 1010
0011 1110
0001 1111

Convert these bytes to their hexadecimal equivalents.

SOLUTION

Here are the stored bytes and their hexadecimal equivalents:

Memory Contents	Hex Equivalents
0011 1100	3C
1100 1101	CD
0101 0111	57
0010 1000	28
1111 0001	F1

0010 1010	2A
1101 0100	D4
0100 0000	40
0111 0111	77
1100 0011	C3
1000 0100	84
0010 1000	28
0010 0001	21
0011 1010	3A
0011 1110	3E
0001 1111	1F

What's the point of this example? When talking about the contents of a computer memory, we can use either binary numbers or hexadecimal numbers. For instance, we can say that the first memory location contains 0011 1100, or we can say that it contains 3C. Either string gives the same information. But notice how much easier it is to say, write, and think 3C than it is to say, write, and think 0011 1100. In other words, hexadecimal strings are much easier for people to work with. This is why everybody working with microprocessors uses hexadecimal notation to represent particular bytes.

What we have just done is known as *chunking*, replacing longer strings of data with shorter ones. At the first memory location we chunk the digits 0011 1100 into 3C. At the second memory location we chunk the digits 1100 1101 into CD, and so on.

EXAMPLE 1-7

The typical microcomputer has a typewriter keyboard that allows you to enter programs and data; a video screen displays answers and other information.

Suppose the video screen of a microcomputer displays the hexadecimal contents of the first eight memory locations as

A7
28
C3
19
5A
4D
2C
F8

What are the binary contents of the memory locations?

SOLUTION

Convert from hexadecimal to binary to get

1010 0111
0010 1000
1100 0011
0001 1001
0101 1010
0100 1101
0010 1100
1111 1000

The first memory location stores the byte 1010 0111, the second memory location stores the byte 0010 1000, and so on.

This example emphasizes a widespread industrial practice. Microcomputers are programmed to display chunked data, often hexadecimal. The user is expected to know hexadecimal-binary conversions. In other words, a computer manufacturer assumes that you know that A7 represents 1010 0111, 28 stands for 0010 1000, and so on.

One more point. Notice that each memory location in this example stores 1 byte. This is typical of first-generation microcomputers because they use 8-bit microprocessors.

1-10 HEXADECIMAL-TO-DECIMAL CONVERSION

You often need to convert a hexadecimal number to its decimal equivalent. This section discusses methods for doing it.

Hexadecimal to Binary to Decimal

One way to convert from hexadecimal to decimal is the two-step method of converting from hexadecimal to binary and then from binary to decimal. For instance, here's how to convert hexadecimal 3C to its decimal equivalent.

Step 1. Convert 3C to its binary equivalent:

3	C
↓	↓
0011	1100

Step 2. Convert 0011 1100 to its decimal equivalent:

0	0	1	1	1	1	0	0	
~~128~~	~~64~~	32	16	8	4	~~2~~	~~1~~	→ 60

Therefore, decimal 60 is equivalent to hexadecimal 3C. As an equation,

$$3C_{16} = 0011\ 1100_2 = 60_{10}$$

Positional-Notation Method

Positional notation is also used with hexadecimal numbers because each digit position has a weight. Since 16 digits are used, the weights are the powers of 16. As shown in

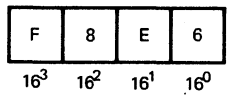


Fig. 1-6 Hexadecimal weights.

the hexadecimal odometer of Fig. 1-6, the weights are 16^0, 16^1, 16^2, and 16^3. If longer hexadecimal numbers are involved, the weights continue in ascending powers of 16.

The decimal equivalent of a hexadecimal string equals the sum of all hexadecimal digits multiplied by their weights. (In processing hexadecimal digits A through F, use 10 through 15.) For instance, the hexadecimal reading of Fig. 1-6 has a decimal equivalent of

$$\begin{aligned}(F \times 16^3) + (8 \times 16^2) + (E \times 16^1) + (6 \times 16^0) \\ = (15 \times 16^3) + (8 \times 16^2) + (14 \times 16^1) + (6 \times 16^0) \\ = 61{,}440 + 2{,}048 + 224 + 6 \\ = 63{,}718\end{aligned}$$

In other words,

$$F8E6_{16} = 63{,}718_{10}$$

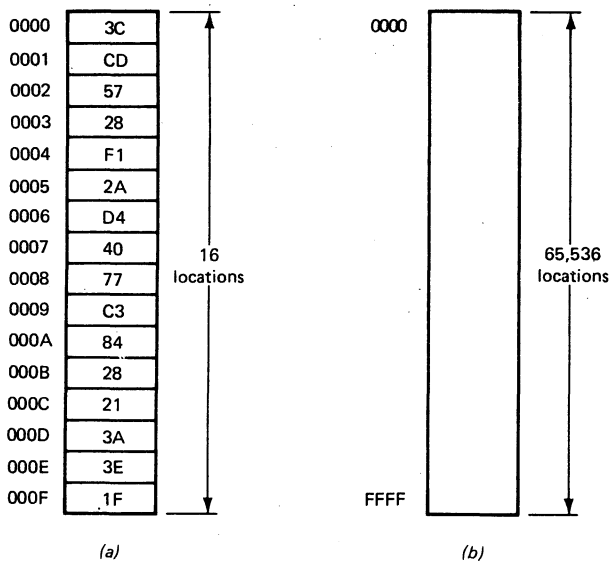


Fig. 1-7 (*a*) First 16 words in memory; (*b*) 64K memory.

Memory Locations and Addresses

As mentioned earlier, a microcomputer may have a 64K memory, meaning 65,536 memory locations, each able to store 1 byte. The different memory locations are identified by hexadecimal numbers called *addresses*. For instance, Fig. 1-7*a* shows the first 16 memory locations; their addresses are from 0000 to 000F.

The address of a memory location is different from its stored contents, just as a house address is different from the people living in the house. Figure 1-7*a* emphasizes the point. At address 0000 the stored contents are 3C (equivalent to 0011 1100). At address 0001 the stored contents are CD, at address 0002 the stored contents are 57, and so on.

Figure 1-7*b* shows how to visualize a 64K memory. The first address is 0000, and the last is FFFF.

Table of Binary-Hexadecimal-Decimal Equivalents

A 64K memory has 65,536 hexadecimal addresses from 0000 to FFFF. The equivalent binary addresses are from

0000 0000 0000 0000

to

1111 1111 1111 1111

The first 8 bits are called the *upper byte* (UB); the second 8 bits are the *lower byte* (LB). If you have to do a lot of binary-hexadecimal-decimal conversions, use the table of equivalents in Appendix 1, which shows all the values for a 64K memory.

Appendix 1 has four headings: binary, hexadecimal, UB decimal, and LB decimal. Given a 16-bit address, you convert the upper byte to its decimal equivalent (UB decimal), the lower byte to its decimal equivalent (LB decimal), and then add the two decimal equivalents. For instance, suppose you want to convert

1101 0111 1010 0010

to its decimal equivalent. The upper byte is 1101 0111, or hexadecimal D7; the lower byte is 1010 0010, or A2. Using Appendix 1, find D7 and its UB decimal equivalent

$$D7 \rightarrow 55{,}040$$

Next, find A2 and its LB decimal equivalent

$$A2 \rightarrow 162$$

Add the UB and LB decimal equivalents to get

$$55{,}040 + 162 = 55{,}202$$

This is the decimal equivalent of hexadecimal D7A2 or binary 1101 0111 1010 0010.

Once familiar with Appendix 1, you will find it enormously helpful. It is faster, more accurate, and less tiring than other methods. The only calculation required is adding the UB and LB decimal, easily done mentally, with pencil and paper, or if necessary, on a calculator. Furthermore, if you are interested in converting only the lower byte, no calculation is required, as shown in the next example.

EXAMPLE 1-8

Convert hexadecimal 7E to its decimal equivalent.

SOLUTION

When converting only a single byte, all you are dealing with is the lower byte. With Appendix 1, look up 7E and its LB decimal equivalent to get

$$7E \rightarrow 126$$

In other words, Appendix 1 can be used to convert single bytes to their decimal equivalents (LB decimal) or double bytes to their decimal equivalents (UB decimal + LB decimal).

1-11 DECIMAL-TO-HEXADECIMAL CONVERSION

One way to perform decimal-to-hexadecimal conversion is to go from decimal to binary then to hexadecimal. Another way is *hex-dabble*. The idea is to divide successively by 16, writing down the remainders. (Hex-dabble is like double-dabble except that 16 is used for the divisor instead of 2.)

Here's an example of how to convert decimal 2,479 into hexadecimal form. The first division is

```
      154    15  F
16 ) 2,479
```

The next step is

```
        9    10  A
    ) 154    15  F
16 ) 2,479
```

The final step is

```
                    Read
                    down
        0    9  9     |
    )   9    10 A     |
    ) 154    15 F     |
16 ) 2,479            |
                      v
```

Notice how similar hex-dabble is to double-dabble. Also, remainders greater than 9 have to be changed to hexadecimal digits (10 becomes A, 15 becomes F, etc.).

If you prefer, use Appendix 1 to look up the decimal-hexadecimal equivalents. The next two examples show how.

EXAMPLE 1-9

Convert decimal 141 to hexadecimal.

SOLUTION

Whenever the decimal number is between 0 and 255, all you have to do is look up the decimal number and its hexadecimal equivalent. With Appendix 1, you can see at a glance that

$$8D \leftarrow 141$$

EXAMPLE 1-10

Convert decimal 36,020 to its hexadecimal equivalent.

SOLUTION

If the decimal number is between 256 and 65,535, you need to proceed as follows. First, locate the largest UB decimal that is less than 36,020. In Appendix 1, the largest UB decimal is

$$\text{UB decimal} = 35{,}840$$

which has a hexadecimal equivalent of

$$8C \leftarrow 35{,}840$$

This is the upper byte.

Next, subtract the UB decimal from the original decimal number:

$$36{,}020 - 35{,}840 = 180$$

The difference 180 has a hexadecimal equivalent

$$B4 \leftarrow 180$$

This is the lower byte.

By combining the upper and lower bytes, we get the complete answer: 8CB4. This is the hexadecimal equivalent of 36,020.

After a little practice, you will find Appendix 1 to be one of the fastest methods of decimal-hexadecimal conversion.

1-12 BCD NUMBERS

A *nibble* is a string of 4 bits. *Binary-coded-decimal* (BCD) numbers express each decimal digit as a nibble. For instance, decimal 2,945 converts to a BCD number as follows:

$$\begin{array}{cccc} 2 & 9 & 4 & 5 \\ \downarrow & \downarrow & \downarrow & \downarrow \\ 0010 & 1001 & 0100 & 0101 \end{array}$$

As you see, each decimal digit is coded as a nibble.

Here's another example: $9{,}863_{10}$ converts like this:

$$\begin{array}{cccc} 9 & 8 & 6 & 3 \\ \downarrow & \downarrow & \downarrow & \downarrow \\ 1001 & 1000 & 0110 & 0011 \end{array}$$

Therefore, 1001 1000 0110 0011 is the BCD equivalent of $9{,}863_{10}$.

The reverse conversion is similar. For instance, 0010 1000 0111 0100 converts as follows:

$$\begin{array}{cccc} 0010 & 1000 & 0111 & 0100 \\ \downarrow & \downarrow & \downarrow & \downarrow \\ 2 & 8 & 7 & 4 \end{array}$$

Applications

BCD numbers are useful wherever decimal information is transferred into or out of a digital system. The circuits inside pocket calculators, for example, can process BCD numbers because you enter decimal numbers through the keyboard and see decimal answers on the LED or liquid-crystal display. Other examples of BCD systems are electronic counters, digital voltmeters, and digital clocks; their circuits can work with BCD numbers.

BCD Computers

BCD numbers have limited value in computers. A few early computers processed BCD numbers but were slower and more complicated than binary computers. As previously mentioned, a computer is more than a number cruncher because it must handle names and other nonnumeric data. In other words, a modern computer must be able to process *alphanumerics* (alphabet letters, numbers, and other symbols). This why modern computers have CPUs that process binary numbers rather than BCD numbers.

Comparison of Number Systems

Table 1-5 shows the four number systems we have discussed. Each number system uses strings of digits to represent quantity. Above 9, equivalent strings appear different. For instance, decimal string 128, hexadecimal string 80, binary string 1000 0000, and BCD string 0001 0010 1000 are equivalent because they represent the same number of pebbles.

Machines have to use long strings of binary or BCD numbers, but people prefer to chunk the data in either decimal or hexadecimal form. As long as we know how to convert from one number system to the next, we can always get back to the ultimate meaning, which is the number of pebbles being represented.

TABLE 1-5. NUMBER SYSTEMS

Decimal	Hexadecimal	Binary	BCD
0	0	0000 0000	0000 0000 0000
1	1	0000 0001	0000 0000 0001
2	2	0000 0010	0000 0000 0010
3	3	0000 0011	0000 0000 0011
4	4	0000 0100	0000 0000 0100
5	5	0000 0101	0000 0000 0101
6	6	0000 0110	0000 0000 0110
7	7	0000 0111	0000 0000 0111
8	8	0000 1000	0000 0000 1000
9	9	0000 1001	0000 0000 1001
10	A	0000 1010	0000 0001 0000
11	B	0000 1011	0000 0001 0001
12	C	0000 1100	0000 0001 0010
13	D	0000 1101	0000 0001 0011
14	E	0000 1110	0000 0001 0100
15	F	0000 1111	0000 0001 0101
16	10	0001 0000	0000 0001 0110
32	20	0010 0000	0000 0011 0010
64	40	0100 0000	0000 0110 0100
128	80	1000 0000	0001 0010 1000
255	FF	1111 1111	0010 0101 0101

1-13 THE ASCII CODE

To get information into and out of a computer, we need to use numbers, letters, and other symbols. This implies some kind of alphanumeric code for the I/O unit of a computer. At one time, every manufacturer had a different code, which led to all kinds of confusion. Eventually, industry settled on an input-output code known as the *American Standard Code for Information Interchange* (abbreviated ASCII). This code allows manufacturers to standardize I/O hardware such as keyboards, printers, video displays, and so on.

The ASCII (pronounced *ask'-ee*) code is a 7-bit code whose format (arrangement) is

$$X_6X_5X_4X_3X_2X_1X_0$$

where each X is a 0 or a 1. For instance, the letter A is coded as

1000001

Sometimes, a space is inserted for easier reading:

100 0001

TABLE 1-6. THE ASCII CODE

$X_3X_2X_1X_0$	$X_6X_5X_4$					
	010	**011**	**100**	**101**	**110**	**111**
0000	SP	0	@	P		p
0001	!	1	A	Q	a	q
0010	"	2	B	R	b	r
0011	#	3	C	S	c	s
0100	$	4	D	T	d	t
0101	%	5	E	U	e	u
0110	&	6	F	V	f	v
0111	'	7	G	W	g	w
1000	(	8	H	X	h	x
1001	)	9	I	Y	i	y
1010	*	:	J	Z	j	z
1011	+	;	K		k	
1100	,	<	L		l	
1101	−	=	M		m	
1110	•	>	N		n	
1111	/	?	O		o	

Table 1-6 shows the ASCII code. Read the table the same as a graph. For instance, the letter A has an $X_6X_5X_4$ of 100 and an $X_3X_2X_1X_0$ of 0001. Therefore, its ASCII code is

100 0001 (A)

Table 1-6 includes the ASCII code for lowercase letters. The letter a is coded as

110 0001 (a)

More examples are

110 0010 (b)
110 0011 (c)
110 0100 (d)

and so on.

Also look at the punctuation and mathematical symbols. Some examples are

010 0100 ($)
010 1011 (+)
011 1101 (=)

In Table 1-6, SP stands for space (blank). Hitting the space bar of an ASCII keyboard sends this into a microcomputer:

010 0000 (space)

EXAMPLE 1-11

With an ASCII keyboard, each keystroke produces the ASCII equivalent of the designated character. Suppose you type

PRINT X

What is the output of an ASCII keyboard?

SOLUTION

P (101 0000), R (101 0010), I (100 1001), N (100 1110), T (101 0100), space (010 0000), X (101 1000).

GLOSSARY

address Each memory location has an address, analogous to a house address. Using addresses, we can tell the computer where desired data is stored.

alphanumeric Letters, numbers, and other symbols.

base The number of digits (basic symbols) in a number system. Decimal has a base of 10, binary a base of 2, and hexadecimal a base of 16. Also called the radix.

bit An abbreviation for binary digit.

byte A string of 8 bits. The byte is the basic unit of binary information. Most computers process data with a length of 8 bits or some multiple of 8 bits.

central processing unit The control section and the arithmetic-logic section. Abbreviated CPU.

chip An integrated circuit.

chunking Replacing a longer string by a shorter one.

data Names, numbers, and any other information needed to solve a problem.

digital Pertains to anything in the form of digits, for example, digital data.

hardware The electronic, magnetic, and mechanical devices used in a computer.

hexadecimal A number system with a base of 16. Hexadecimal numbers are used in microprocessor work.

input-output Abbreviated I/O. The input and output sections of a computer are often lumped into one unit known as the I/O unit.

microcomputer A computer that uses a microprocessor for its central processing unit (CPU).

microprocessor A CPU on a chip. It contains the control and arithmetic-logic sections. Sometimes abbreviated MPU (microprocessor unit).

nibble A string of 4 bits. Half of a byte.

program A sequence of instructions that tells the computer how to process the data. Also known as software.

register A group of electronic, magnetic, or mechanical devices that store digital data.

software Programs.

string A group of digits or other symbols.

SELF-TESTING REVIEW

Read each of the following and provide the missing words. Answers appear at the beginning of the next question.

1. Binary means _____. Binary numbers have a base of 2. The digits used in a binary number system are _____ and _____.
2. (*two; 0, 1*) Names, numbers, and other information needed to solve a problem are called _____. The _____ is a sequence of instructions that tells the computer how to process the data.
3. (*data, program*) Computer ICs work reliably because they are based on _____ design. When a transistor is cut off or saturated, transistor _____ have almost no effect.
4. (*two-state, variations*) A _____ is a group of devices that store digital data. _____ is an abbreviation for binary digit. A byte is a string of _____ bits.
5. (*register, Bit, 8*) The control and arithmetic-logic sections are called the _____ (CPU). A microprocessor is a CPU on a chip. A microcomputer is a computer that uses a _____ for its CPU.
6. (*central processing unit, microprocessor*) The abbreviation K indicates units of approximately 1,000 or precisely 1,024. Therefore, 1K means 1,024, 2K means 2,048, 4K means _____, and 64K means _____.
7. (*4,096, 65,536*) The hexadecimal number system is widely used in analyzing and programming _____. The hexadecimal digits are 0 to 9 and A to _____. The main advantage of hexadecimal numbers is the ease of conversion from hexadecimal to _____ and vice versa.
8. (*microprocessors, F, binary*) A typical microcomputer may have up to 65,536 registers in its memory. Each of these registers, usually called a _____, stores 1 byte. Such a memory is specified as a 64-kilobyte memory, or simply a _____ memory.
9. (*memory location, 64K*) Binary-coded-decimal (BCD) numbers express each decimal digit as a _____ BCD numbers are useful whenever _____ information is transferred into or out of a digital system. Equipment using BCD numbers includes pocket calculators, electronic counters, and digital voltmeters.
10. (*nibble, decimal*) The ASCII code is a 7-bit code for _____ (letters, numbers, and other symbols).
11. (*alphanumerics*) With the typical microcomputer, you enter the program and data with typewriter keyboard that converts each character into ASCII code.

PROBLEMS

1-1. How many bytes are there in each of these numbers?
   a. 1100 0101
   b. 1011 1001 0110 1110
   c. 1111 1011 0111 0100 1010

1-2. What are the equivalent decimal numbers for each of the following binary numbers: 10, 110, 111, 1011, 1100, and 1110?

1-3. What is the base for each of these numbers?
   a. 348_{10}
   b. $1100\ 0101_2$
   c. 2312_5
   d. $F4C3_{16}$

1-4. Write the equation

$$2 + 2 = 4$$

using binary numbers.

1-5. What is the decimal equivalent of 2^{10}? What does 4K represent? Express 8,192 in K units.

1-6. A 4-bit register has output voltages of high-low-high-low. What is the binary number stored in the register? The decimal equivalent?

Fig. 1-8 An 8-bit LED display.

1-7. Figure 1-8 shows an 8-bit LED display. A light circle means that a LED is ON (binary 1) and a dark circle means a LED is OFF (binary 0). What is the binary number being displayed? The decimal equivalent?

1-8. Convert the following binary numbers to decimal numbers:
   a. 00111
   b. 11001
   c. 10110
   d. 11110

1-9. Solve the following equation for x:

$$x_{10} = 11001001_2$$

1-10. An 8-bit transistor register has this output:

low-high-low-high-low-high-low-high

What is the equivalent decimal number being stored?

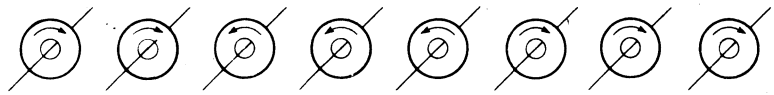


Fig. 1-9 An 8-bit core register.

1-11. In Fig. 1-9 clockwise flux stands for binary 1 and counterclockwise flux for binary 0. What is the binary number stored in the 8-bit core register? Convert this byte to an equivalent decimal number.

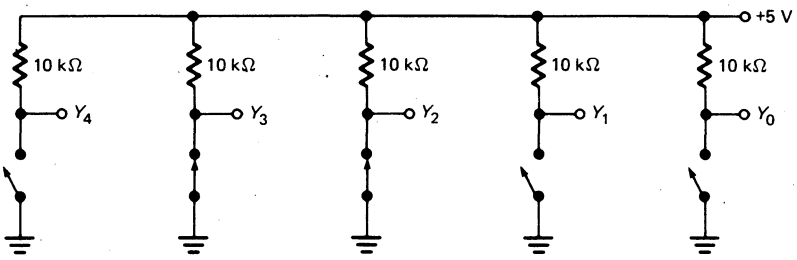


Fig. 1-10 A 5-bit switch register.

1-12. Figure 1-10 shows a 5-bit switch register. By opening and closing the switches you can set up different binary numbers. As usual, high output voltage stands for binary 1 and low output voltage for binary 0. What is the binary number stored in the switch register? The equivalent decimal number?

1-13. Convert decimal 56 to its binary equivalent.

1-14. Convert 72_{10} to a binary number.

1-15. An 8-bit transistor register stores decimal 150. What is the binary output of the register?

1-16. How would you set the switches of Fig. 1-10 to get a decimal output of 27?

1-17. A hexadecimal odometer displays F52A. What are the next six readings?

1-18. The reading on a hexadecimal odometer is 27FF. What is the next reading? Miles later, you see a reading of 8AFC. What are the next six readings?

1-19. Convert each of the following hexadecimal numbers to binary:

a. FF
b. ABC
c. CD42
d. F329

1-20. Convert each of these binary numbers to an equivalent hexadecimal number:

a. 1110 1000
b. 1100 1011
c. 1010 1111 0110
d. 1000 1011 1101 0110

1-21. Here is a program written for the 8085 microprocessor:

Address	Hex Contents
2000	3E
2001	0E
2002	D3
2003	20
2004	76

Convert the hex contents to equivalent binary numbers.

1-22. Convert each of these hexadecimal numbers to its decimal equivalent:

a. FF
b. A4
c. 9B
d. 3C

1-23. Convert the following hexadecimal numbers to their decimal equivalents:

a. 0FFF
b. 3FFF
c. 7FE4
d. B3D8

1-24. A microcomputer has memory locations from 0000 to 0FFF. Each memory location stores 1 byte. In decimal, how many bytes can the microcomputer store in its memory? How many kilobytes is this?

1-25. Suppose a microcomputer has memory locations from 0000 to 3FFF, each storing 1 byte. How

many bytes can the memory store? Express this in kilobytes.

1-26. A microcomputer has a 32K memory. How many bytes does this represent? If 0000 stands for the first memory location, what is the hexadecimal notation for the last memory location?

1-27. If a microcomputer has a 64K memory, what are the hexadecimal notations for the first and last memory locations?

1-28. Convert the following decimal numbers to hexadecimal:
- a. 4,095
- b. 16,383
- c. 32,767
- d. 65,535

1-29. Convert each of the following decimal numbers to hexadecimal numbers:
- a. 238
- b. 7,547
- c. 15,359
- d. 47,285

1-30. How many nibbles are there in each of the following:
- a. 1000 0111
- b. 1001 0000 0100 0011
- c. 0101 1001 0111 0010 0110 0110

1-31. If the numbers in Prob. 1-30 are BCD numbers, what are the equivalent decimal numbers?

1-32. What is the ASCII code for each of the following:
- a. 7
- b. W
- c. f
- d. y

1-33. Suppose you type LIST with an ASCII keyboard. What is the binary output as you strike each letter?

Gates

2

For centuries mathematicians felt there was a connection between mathematics and logic, but no one before George Boole could find this missing link. In 1854 he invented symbolic logic, known today as *boolean algebra*. Each variable in boolean algebra has either of two values: true or false. The original purpose of this two-state algebra was to solve logic problems.

Boolean algebra had no practical application until 1938, when Claude Shannon used it to analyze telephone switching circuits. He let the variables represent closed and open relays. In other words, Shannon came up with a new application for boolean algebra. Because of Shannon's work, engineers realized that boolean algebra could be applied to computer electronics.

This chapter introduces the *gate*, a circuit with one or more input signals but only one output signal. Gates are digital (two-state) circuits because the input and output signals are either low or high voltages. Gates are often called *logic circuits* because they can be analyzed with boolean algebra.

2-1 INVERTERS

An *inverter* is a gate with only one input signal and one output signal; th output state is always the opposite of the input state.

Transistor Inverter

Figure 2-1 shows a transistor inverter. This common-emitter amplifier switches between cutoff and saturation. When V_{IN} is low (approximately 0 V), the transistor cuts off and V_{OUT} is high. On the other hand, a high V_{IN} saturates the transistor, forcing V_{OUT} to go low.

Table 2-1 summarizes the operation. A low input produces a high output, and a high input results in a low output. Table 2-2 gives the same information in binary form; binary 0 stands for low voltage and binary 1 for high voltage.

An inverter is also called a NOT gate because the output is not the same as the input. The output is sometimes called the *complement* (opposite) of the input.

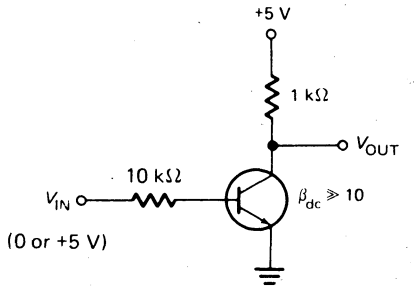


Fig. 2-1 Example of inverter design.

TABLE 2-1

V_{IN}	V_{OUT}
Low	High
High	Low

TABLE 2-2

V_{IN}	V_{OUT}
0	1
1	0

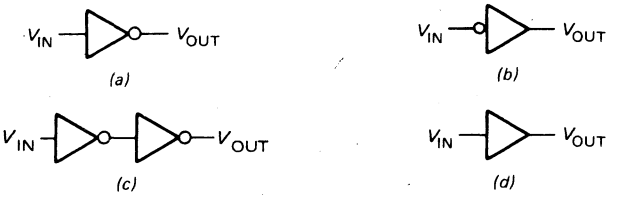


Fig. 2-2 Logic symbols: (*a*) inverter; (*b*) another inverter symbol; (*c*) double inverter; (*d*) buffer.

Inverter Symbol

Figure 2-2*a* is the symbol for an inverter of any design. Sometimes a schematic diagram will use the alternative symbol shown in Fig. 2-2*b*; the bubble (small circle) is on

the input side. Whenever you see either of these symbols, remember that the output is the complement of the input.

Noninverter Symbol

If you cascade two inverters (Fig. 2-2*c*), you get a noninverting amplifier. Figure 2-2*d* is the symbol for a noninverting amplifier. Regardless of the circuit design, the action is always the same: a low input voltage produces a low output voltage, and a high input voltage results in a high output voltage.

The main use of noninverting amplifier is buffering (isolating) two other circuits. More will be said about buffers in a later chapter.

EXAMPLE 2-1

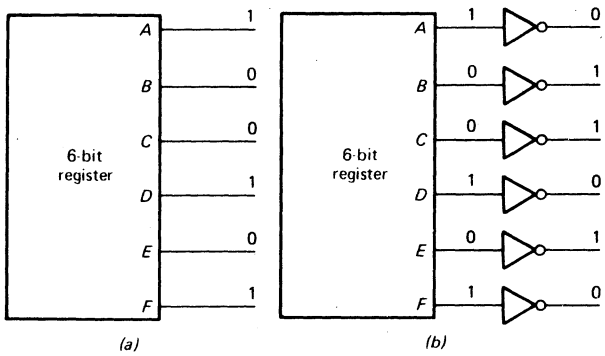


Fig. 2-3 Example 2-1.

Figure 2-3*a* has an output, *A* to *F*, of 100101. Show how to complement each bit.

SOLUTION

Easy. Use an inverter on each signal line (Fig. 2-3*b*). The final output is now 011010.

A *hex inverter* is a commercially available IC containing six separate inverters. Given a 6-bit register like Fig. 2-3*a*, we can connect a hex inverter to complement each bit as shown in Fig. 2-3*b*.

One more point. In Fig. 2-3*a* the bits may represent a coded instruction, number, letter, etc. To convey this variety of meaning, a string of bits is often called a binary word or simply a *word*. In Fig. 2-3*b* the word 100101 is complemented to get the word 011010.

2-2 OR GATES

The OR gate has two or more input signals but only one output signal. If any input signal is high, the output signal is high.

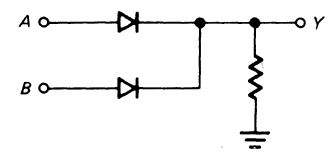


Fig. 2-4 A 2-input diode OR gate.

Diode OR Gate

Figure 2-4 shows one way to build an OR gate. If both inputs are low, the output is low. If either input is high, the diode with the high input conducts and the output is high. Because of the two inputs, we call this circuit a 2-input OR gate.

Table 2-3 summarizes the action; binary 0 stands for low voltage and binary 1 for high voltage. Notice that one or more high inputs produce a high output; this is why the circuit is called an OR gate.

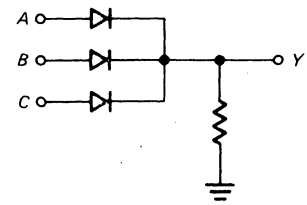


Fig. 2-5 A 3-input diode OR gate.

More than Two Inputs

Figure 2-5 shows a 3-input OR gate. If all inputs are low, all diodes are off and the output is low. If 1 or more inputs are high, the output is high.

Table 2-4 summarizes the action. A table like this is called a *truth table;* it lists all the input possibilities and the corresponding outputs. When constructing a truth table, always list the input words in a binary progression as shown (000, 001, 010, . . . , 111); this guarantees that all input possibilities will be accounted for.

An OR gate can have as many inputs as desired; add one diode for each additional input. Six diodes result in a 6-

TABLE 2-3. TWO INPUT OR GATE

A	*B*	*Y*
0	0	0
0	1	1
1	0	1
1	1	1

TABLE 2-4. THREE-INPUT OR GATE

A	*B*	*C*	*Y*
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

input OR gate, nine diodes in a 9-input OR gate. No matter how many inputs, the action of any OR gate is summarized like this: one or more high inputs produce a high output.

Bipolar transistors and MOSFETs can also be used to build OR gates. But no matter what devices are used, OR gates always produce a high output when one or more inputs are high. Figure 2-6 shows the logic symbols for 2-, 3-, and 4-input OR gates.

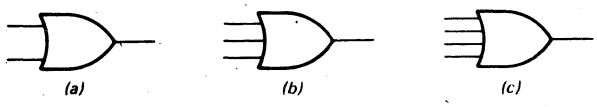


Fig. 2-6 OR-gate symbols.

EXAMPLE 2-2

Show the truth table of a 4-input OR gate.

SOLUTION

Let Y stand for the output bit and A, B, C, D for input bits. Then the truth table has input words of 0000, 0001, 0010, . . . , 1111, as shown in Table 2-5. As expected, output Y is 0 for input word 0000; Y is 1 for all other input words.

As a check, the number of input words in a truth table always equals 2^n, where n is the number of input bits. A 2-input OR gate has a truth table with 2^2 or 4 input words; a 3-input OR gate has 2^3 or 8 input words; and a 4-input OR gate has 2^4 or 16 input words.

TABLE 2-5. FOUR-INPUT OR GATE

A	*B*	*C*	*D*	*Y*
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

EXAMPLE 2-3

How many inputs words are in the truth table of an 8-input OR gate? Which input words produce a high output?

SOLUTION

The input words are 0000 0000, 0000 0001, . . . , 1111 1111. With the formula of the preceding example, the total number of input words is $2^n = 2^8 = 256$.

In any OR gate, 1 or more high inputs produce a high output. Therefore, the input word of 0000 0000 results in a low output; all other input words produce a high output.

EXAMPLE 2-4

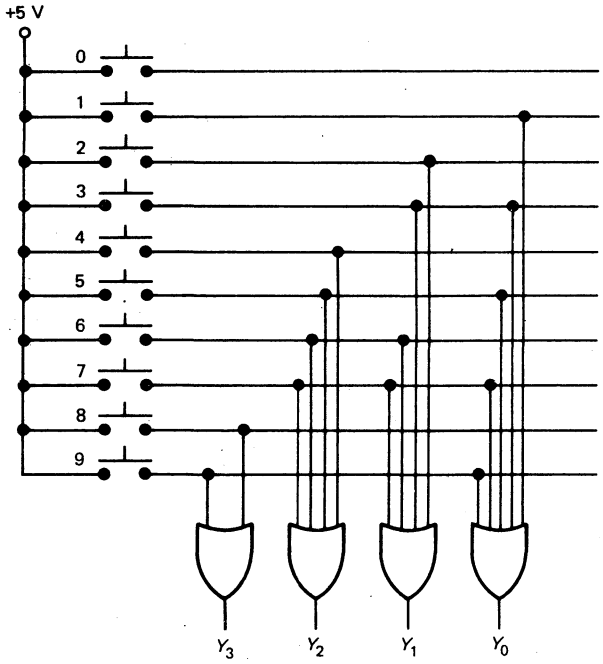


Fig. 2-7 Decimal-to-binary encoder.

The switches of Fig. 2-7 are push-button switches like those of a pocket calculator. The bits out of the OR gates form a 4-bit word, designated $Y_3Y_2Y_1Y_0$. What does the circuit do?

SOLUTION

Figure 2-7 is a decimal-to-binary *encoder*, a circuit that converts decimal to binary. For instance, when push button 3 is pressed, the Y_1 and Y_0 OR gates have high inputs; therefore, the output word is

$$Y_3Y_2Y_1Y_0 = 0011$$

If button 5 is keyed, the Y_2 and Y_0 OR gates have high inputs and the output word becomes

$$Y_3Y_2Y_1Y_0 = 0101$$

When switch 9 is pressed,

$$Y_3Y_2Y_1Y_0 = 1001$$

Check the other input switches to convince yourself that the output word always equals the binary equivalent of the switch being pressed.

2-3 AND GATES

The AND gate has two or more input signals but only one output signal. All inputs must be high to get a high output.

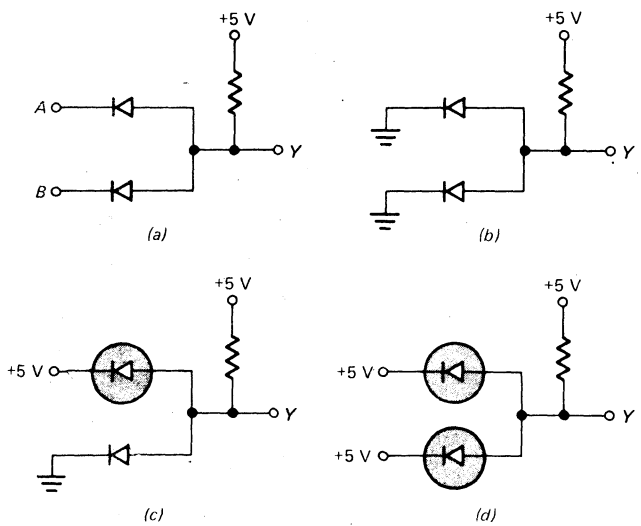


Fig. 2-8 A 2-input AND gate. (*a*) circuit; (*b*) both inputs low; (*c*) 1 low input, 1 high; (*d*) both inputs high.

Diode AND Gate

Figure 2-8*a* shows one way to build an AND gate. In this circuit the inputs can be either low (ground) or high (+5 V). When both inputs are low (Fig. 2-8*b*), both diodes conduct and pull the output down to a low voltage. If one of the inputs is low and the other high (Fig. 2-8*c*), the diode with the low input conducts and this pulls the output down to a low voltage. The diode with the high input, on the other hand, is reverse-biased or cut off, symbolized by the dark shading in Fig. 2-8*c*.

When both inputs are high (Fig. 2-8*d*), both diodes are cut off. Since there is no current in the resistor, the supply voltage pulls the output up to a high voltage (+5 V).

TABLE 2-6. TWO-INPUT AND GATE

A	*B*	*Y*
0	0	0
0	1	0
1	0	0
1	1	1

Table 2-6 summarizes the action. As usual, binary zero stands for low voltage and binary 1 for high voltage. As you see, *A and B* must be high to get a high output; this is why the circuit is called an AND gate.

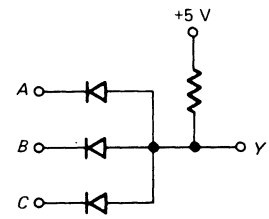


Fig. 2-9 A 3-input AND gate.

More than Two Inputs

Figure 2-9 is a 3-input AND gate. If all inputs are low, al diodes conduct and pull the output down to a low voltage Even one conducting diode will pull the output down to a low voltage; therefore, the only way to get a high outpu is to have all inputs high. When all inputs are high, al diodes are nonconducting and the supply voltage pulls th output up to a high voltage.

Table 2-7 summarizes the 3-input AND gate. The outpu is 0 for all input words except 111. That is, all inputs mus be high to get a high output.

AND gates can have as many inputs as desired; add on diode for each additional input. Eight diodes, for instance result in an 8-input AND gate; sixteen diodes in a 16-inpu

TABLE 2-7. THREE-INPUT AND GATE

A	*B*	*C*	*Y*
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

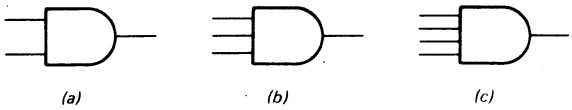


Fig. 2-10 AND-gate symbols.

AND gate. No matter how many inputs an AND gate has, the action can be summarized like this: All inputs must be high to get a high output.

Figure 2-10 shows the logic symbols for 2-, 3-, and 4-input AND gates.

EXAMPLE 2-5

Describe the truth table of an 8-input AND gate.

SOLUTION

The input words are from 0000 0000 to 1111 1111, following the binary progression. The total number of input words is

$$2^n = 2^8 = 256$$

The first 255 input words produce a 0 output. Only the last word, 1111 1111, results in a 1 output. This is because all inputs must be high to get a high output.

EXAMPLE 2-6

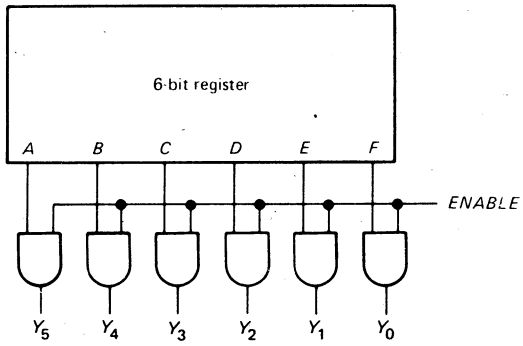


Fig. 2-11 Using AND gates to block or transmit data.

The 6-bit register of Fig. 2-11 stores the word ABCDEF. The *ENABLE* input can be low or high. What does the circuit do?

SOLUTION

One use of AND gates is to transmit data when certain conditions are satisfied. In Fig. 2-11 a low *ENABLE* blocks the register contents from the final output, but a high *ENABLE* transmits the register contents.

For instance, when

$$ENABLE = 0$$

each AND gate has a low *ENABLE* input. No matter what the register contents, the output of each AND gate must be low. Therefore, the final word is

$$Y_5Y_4Y_3Y_2Y_1Y_0 = 000000$$

As you see, a low *ENABLE* blocks the register contents from the final output.

On the other hand, when

$$ENABLE = 1$$

the output of each AND gate depends on the data inputs (*A*, *B*, *C*, . . .); a low data input results in a low output, and a high data input in a high output. For example, if ABCDEF = 100100, a high *ENABLE* gives

$$Y_5Y_4Y_3Y_2Y_1Y_0 = 100100$$

In general, a high *ENABLE* transmits the register contents to the final output to get

$$Y_5Y_4Y_3Y_2Y_1Y_0 = ABCDEF$$

2-4 BOOLEAN ALGEBRA

As mentioned earlier, Boole invented two-state algebra to solve logic problems. This new algebra had no practical use until Shannon applied it to telephone switching circuits. Today boolean algebra is the backbone of computer circuit analysis and design.

Inversion Sign

In boolean algebra a variable can be either a 0 or a 1. For digital circuits, this means that a signal voltage can be either low or high. Figure 2-12 is an example of a digital circuit because the input and output voltages are either low or high. Furthermore, because of the inversion, *Y* is always the complement of *A*.

Fig. 2-12 Inverter.

A word equation for Fig. 2-12 is

$$Y = \text{NOT } A \qquad (2\text{-}1)$$

If A is 0,

$$Y = \text{NOT } 0 = 1$$

On the other hand, if A is 1,

$$Y = \text{NOT } 1 = 0$$

In boolean algebra, the overbar stands for the NOT operation. This means that Eq. 2-1 can be written

$$Y = \overline{A} \tag{2-2}$$

Read this as "Y equals NOT A" or "Y equals the complement of A." Equation 2-2 is the standard way to write the output of an inverter.

Using the equation is easy. Given the value of A, substitute and solve for Y. For instance, if A is 0,

$$Y = \overline{A} = \overline{0} = 1$$

because NOT 0 is 1. On the other hand, if A is 1,

$$Y = \overline{A} = \overline{1} = 0$$

because NOT 1 is 0.

Fig. 2-13 OR gate.

OR Sign

A word equation for Fig. 2-13 is

$$Y = A \text{ OR } B \tag{2-3}$$

Given the inputs, you can solve for the output. For instance, if $A = 0$ and $B = 0$,

$$Y = 0 \text{ OR } 0 = 0$$

because 0 comes out of an OR gate when both inputs are 0s.

As another example, if $A = 0$ and $B = 1$,

$$Y = 0 \text{ OR } 1 = 1$$

because 1 comes out of an OR gate when either input is 1. Similarly, if $A = 1$ and $B = 0$,

$$Y = 1 \text{ OR } 0 = 1$$

If $A = 1$ and $B = 1$,

$$Y = 1 \text{ OR } 1 = 1$$

In boolean algebra the + sign stands for the OR operation. In other words, Eq. 2-3 can be written

$$Y = A + B \tag{2-4}$$

Read this as "Y equals A OR B." Equation 2-4 is the standard way to write the output of an OR gate.

Given the inputs, you can substitute and solve for the output. For instance, if $A = 0$ and $B = 0$,

$$Y = A + B = 0 + 0 = 0$$

If $A = 0$ and $B = 1$,

$$Y = A + B = 0 + 1 = 1$$

because 0 ORed with 1 results in 1. If $A = 1$ and $B = 0$,

$$Y = A + B = 1 + 0 = 1$$

If both inputs are high,

$$Y = A + B = 1 + 1 = 1$$

because 1 ORed with 1 gives 1.

Don't let the new meaning of the + sign bother you. There's nothing unusual about symbols having more than one meaning. For instance, "pot" may mean a cooking utensil, a flower container, the money wagered in a card game, a derivative of *cannabis sativa* and so forth; the intended meaning is clear from the sentence it's used in. Similarly, the + sign may stand for ordinary addition or OR addition; the intended meaning comes across in the way it's used. If we're talking about decimal numbers, + means ordinary addition, but when the discussion is about logic circuits, + stands for OR addition.

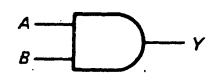


Fig. 2-14 AND gate.

AND Sign

A word equation for Fig. 2-14 is

$$Y = A \text{ AND } B \tag{2-5}$$

In boolean algebra the multiplication sign stands for the AND operation. Therefore, Eq. 2-5 can be written

$$Y = A \cdot B$$

or simply

$$Y = AB \tag{2-6}$$

Read this as "Y equals A AND B." Equation 2-6 is the standard way to write the output of an AND gate.

Given the inputs, you can substitute and solve for the output. For instance, if both inputs are low,

$$Y = AB = 0 \cdot 0 = 0$$

because 0 ANDed with 0 gives 0. If A is low and B is high,

$$Y = AB = 0 \cdot 1 = 0$$

because 0 comes out of an AND gate if any input is 0. If A is 1 and B is 0,

$$Y = AB = 1 \cdot 0 = 0$$

When both inputs are high,

$$Y = AB = 1 \cdot 1 = 1$$

because 1 ANDed with 1 gives 1.

Decision-Making Elements

The inverter, OR gate, and AND gate are often called *decision-making elements* because they can recognize some input words while disregarding others. A gate recognizes a word when its output is high; it disregards a word when its output is low. For example, the AND gate disregards all words with one or more 0s; it recognizes only the word whose bits are all 1s.

Notation

In later equations we need to distinguish between bits that are ANDed and bits that are part of a binary word. To do this we will use italic (slanted) letters (A, B, Y, etc.) for ANDed bits and roman (upright) letters (A, B, Y, etc.) for bits that form a word.

For example, $Y_3Y_2Y_1Y_0$ stands for the logical product (ANDing) of Y_3, Y_2, Y_1, and Y_0. If $Y_3 = 1$, $Y_2 = 0$, $Y_1 = 0$, and $Y_0 = 1$, the product $Y_3Y_2Y_1Y_0$ will reduce as follows:

$$Y_3Y_2Y_1Y_0 = 1 \cdot 0 \cdot 0 \cdot 1 = 0$$

In this case, the italic letters represent bits that are being ANDed.

On the other hand, $Y_3Y_2Y_1Y_0$ is our notation for a 4-bit word. With the Y values just given, we can write

$$Y_3Y_2Y_1Y_0 = 1001$$

In this equation, we are not dealing w h bits that are ANDed; instead, we are dealing with bits that are part of a word.

The distinction between italic and roman notation become clearer when we get to computer analysis.

Positive and Negative Logic

A final point. *Positive logic* means that 1 stands for the more positive of the two voltage levels. *Negative logic* means that 1 stands for the more negative of the two voltage levels. For instance, if the two voltage levels are 0 and -5 V, positive logic would have 1 stand for 0 V and 0 for -5 V, whereas negative logic would have 1 stand for -5 V and 0 for 0 V.

Ordinarily, people use positive logic with positive supply voltages and negative logic with negative supply voltages. Throughout this book, we will be using positive logic.

EXAMPLE 2-7

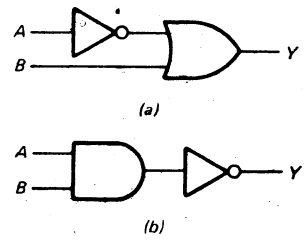


Fig. 2-15 Logic circuits.

What is the boolean equation for Fig. 2-15*a*? The output if both inputs are high?

SOLUTION

A is inverted before it reaches the OR gate; therefore, the upper input to the OR gate is $\overline{A}$. The final output is

$$Y = \overline{A} + B$$

This is the boolean equation for Fig. 2-15*a*.

To find the output when both inputs are high, either of two approaches can be used. First, you can substitute directly into the foregoing equation and solve for Y

$$Y = \overline{A} + B = \overline{1} + 1 = 0 + 1 = 1$$

Alternatively, you can analyze the operation of Fig. 2-15*a* like this. If both inputs are high, the inputs to the OR gate are 0 and 1. Now, 0 ORed with 1 gives 1. Therefore, the final output is high.

EXAMPLE 2-8

What is the boolean equation for Fig. 2-15*b*? If both inputs are high, what is the output?

SOLUTION

The AND gate forms the logical product AB, which is inverted to get

$$Y = \overline{AB}$$

Read this as "Y equals NOT AB" or "Y equals the complement of AB."

If both inputs are high, direct substitution into the equation gives

$$Y = \overline{AB} = \overline{1 \cdot 1} = \overline{1} = 0$$

Note the order of operations: the ANDing is done first, then the inversion.

Instead of using the equation, you can analyze Fig. 2-15*b* as follows. If both inputs are high, the AND gate has a high output. Therefore, the final output is low.

EXAMPLE 2-9

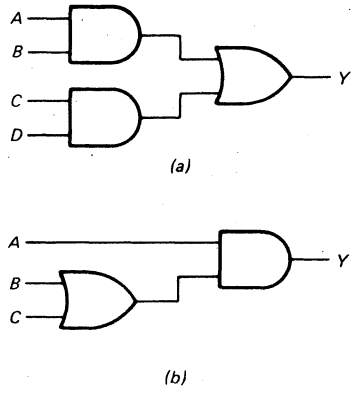


Fig. 2-16 Logic circuits.

What is the boolean equation for Fig. 2-16*a*? The truth table? Which input words does the circuit recognize?

SOLUTION

The upper AND gate forms the logical product AB, and the lower AND gate gives CD. ORing these products results in

$$Y = AB + CD$$

Read this as "Y equals AB OR CD."

Next, look at Fig. 2-16*a*. The final output is high if the OR gate has one or more high inputs. This happens when AB is 1, CD is 1, or both are 1s. In turn, AB is 1 when

$$A = 1 \quad \text{and} \quad B = 1$$

TABLE 2-8. TRUTH TABLE FOR $Y = AB + CD$

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

CD is 1 when

$$C = 1 \quad \text{and} \quad D = 1$$

Both products are 1s when

$$A = 1 \quad B = 1 \quad C = 1 \quad \text{and} \quad D = 1$$

Therefore, the final output is high when A and B are 1s, when C and D are 1s, or when all inputs are 1s.

Table 2-8 summarizes the foregoing analysis. From this it's clear that the circuit recognizes these input words: 0011, 0111, 1011, 1100, 1101, 1110, and 1111.

EXAMPLE 2-10

Write the boolean equation for Fig. 2-16*b*. If all inputs are high, what is the output?

SOLUTION

The OR gate forms the logical sum $B + C$. This sum is ANDed with A to get

$$Y = A(B + C)$$

(Parentheses indicate ANDing.)

One way to find the output when all inputs are high is to substitute and solve as follows:

$$Y = A(B + C) = 1(1 + 1) = 1(1) = 1$$

Alternatively, you can analyze Fig. 2-16*b* like this. If all inputs are high, the OR gate has a high output; therefore, both inputs to the AND gate are high. Since all high inputs to an AND gate result in a high output, the final output is high.

EXAMPLE 2-11

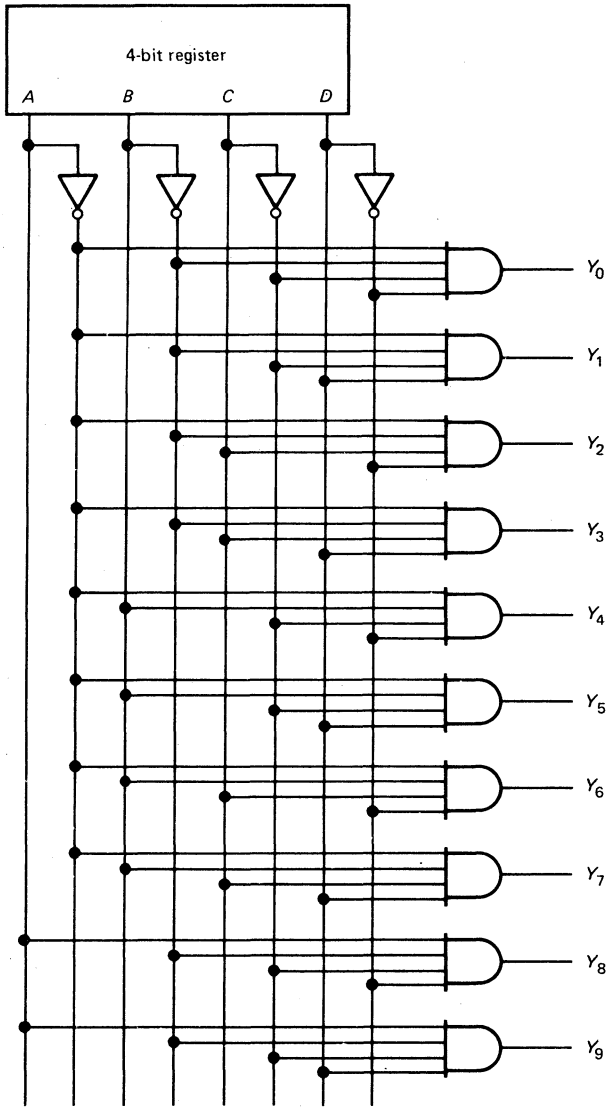


Fig. 2-17 A 1-of-10 decoder.

What is the boolean equation for each Y output in Fig. 2-17?

SOLUTION

Each AND gate forms the logical product of its input signals. The inputs to the top AND gate are $\overline{A}$, $\overline{B}$, $\overline{C}$ and $\overline{D}$; therefore,

$$Y_0 = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$$

The inputs to the next AND gate are $\overline{A}$, $\overline{B}$, $\overline{C}$ and D; this means that

$$Y_1 = \overline{A}\,\overline{B}\,\overline{C}D$$

Analyzing the remaining gates gives

$$Y_2 = \overline{A}\,\overline{B}C\,\overline{D}$$
$$Y_3 = \overline{A}\,\overline{B}CD$$
$$Y_4 = \overline{A}B\,\overline{C}\,\overline{D}$$
$$Y_5 = \overline{A}B\,\overline{C}D$$
$$Y_6 = \overline{A}BC\,\overline{D}$$
$$Y_7 = \overline{A}BCD$$
$$Y_8 = A\,\overline{B}\,\overline{C}\,\overline{D}$$
$$Y_9 = A\,\overline{B}\,\overline{C}D$$

EXAMPLE 2-12

What does the circuit of Fig. 2-17 do?

SOLUTION

This is a binary-to-decimal decoder, a circuit that converts from binary to decimal. For instance, when the register contents are 0011, the Y_3 AND gate has all high inputs; therefore, Y_3 is high. Furthermore, register contents of 0011 mean that all other AND gates have at least one low input. As a result, all other AND gates have low outputs. (Analyze the circuit to convince yourself.)

If the register contents change to 0100, only the Y_4 AND gate has all high inputs; therefore, only Y_4 is high. If the register contents change to 0111, Y_7 is the only high output.

In general, the subscript of the high output equals the decimal equivalent of the binary number stored in the register. This is why the circuit is called a *binary-to-decimal* decoder.

The circuit of this example is also called a 4-line–to–10-line decoder because there are 4 input lines and 10 output lines. Another name for it is a 1-of-10 decoder because only 1 of 10 output lines has a high voltage.

GLOSSARY

AND *gate* A logic circuit whose output is high only when all inputs are high.

boolean algebra Originally known as symbolic logic, this modern algebra uses the set of numbers 0 and 1. The

operations OR, AND, and NOT are sometimes called *union, intersection,* and *inversion.* Boolean algebra is ideally suited to digital circuit analysis.

complement The output of an inverter.

gate A logic circuit with one or more input signals but only one output signal.

inverter A gate with only 1 input and 1 output. The output is always the complement of the input. Also known as a NOT gate.

logic circuit A circuit whose input and output signals are two-state, either low or high voltages. The basic logic circuits are OR, AND, and NOT gates.

OR *gate* A logic circuit with 2 or more inputs and only 1 output; 1 or more high inputs produce a high output.

truth table A table that shows all input and output possibilities for a logic circuit. The input words are listed in binary progression.

word A string of bits that represent a coded instruction or data.

SELF-TESTING REVIEW

Read each of the following and provide the missing words. Answers appear at the beginning of the next question.

1. A gate is a logic circuit with one or more input signals but only _____ output signal. These signals are either _____ or high.
2. (*one, low*) An inverter is a gate with only _____ input; the output is always in the opposite state from the input. An inverter is also called a _____ gate. Sometimes the output is referred to as the complement of the input.
3. (*1,* NOT) The OR gate has two or more input signals. If any input is _____, the output is high. The number of input words in a truth table always equals _____, where *n* is the number of input bits.
4. (*high,* 2^n) The _____ gate has two or more input signals. All inputs must be high to get a high output.
5. (AND) In boolean algebra, the overbar stands for the NOT operation, the plus sign stands for the _____ operation, and the times sign for the _____ operation.
6. (OR, AND) The inverter, OR gate, and AND gate are called decision-making elements because they can recognize some input _____ while disregarding others. A gate recognizes a word when its output is _____.
7. (*words, high*) A binary-to-decimal decoder is also called a 4-line–to–10-line decoder because it has 4 input lines and 10 output lines. Another name for it is the 1-of-10 decoder because only 1 of its 10 output lines is high at a time.

PROBLEMS

2-1. How many inputs signals can a gate have? How many output signals?

2-2. If you cascade seven inverters, does the overall circuit act like an inverter or noninverter?

2-3. Double inversion occurs when two inverters are cascaded. Does such a connection act like an inverter or noninverter?

2-4. The contents of the 6-bit register in Fig. 2-3*b* change to 101010. What is the decimal equivalent of the register contents? The decimal equivalent out of the hex inverter?

2-5. An OR gate has 6 inputs. How many input words are in its truth table? What is the only input word that produces a 0 output?

2-6. Figure 2-18 shows a hexadecimal encoder, a circuit that converts hexadecimal to binary. Pressing each push-button switch results in a different output word $Y_3Y_2Y_1Y_0$. Starting with switch 0, what are the output words? (NOTE: The new symbol in Fig. 2-18 is another way to draw an OR gate.

2-7. In Fig. 2-18 what switches would you press to produce

0011 1001 1100 1111

(Work from left to right.)

2-8. What is the 4-bit output in Fig. 2-18 when switch A is pressed? Switch 4? Switch E? Switch 6?

2-9. An AND gate has 7 inputs. How many input words are in its truth table? What is the only input word that produces a 1 output?

2-10. Visualize the register contents of Fig. 2-19 as the word $A_7A_6 \cdots A_0$, and the final output as the word $Y_7Y_6 \cdots Y_0$. What is the output word for each of the following conditions:

a. $A_7A_6 \cdots A_0 = 1100\ 1010$, *ENABLE* = 0.
b. $A_7A_6 \cdots A_0 = 0101\ 1101$, *ENABLE* = 1.
c. $A_7A_6 \cdots A_0 = 1111\ 0000$, *ENABLE* = 1.
d. $A_7A_6 \cdots A_0 = 1010\ 1010$, *ENABLE* = 0.

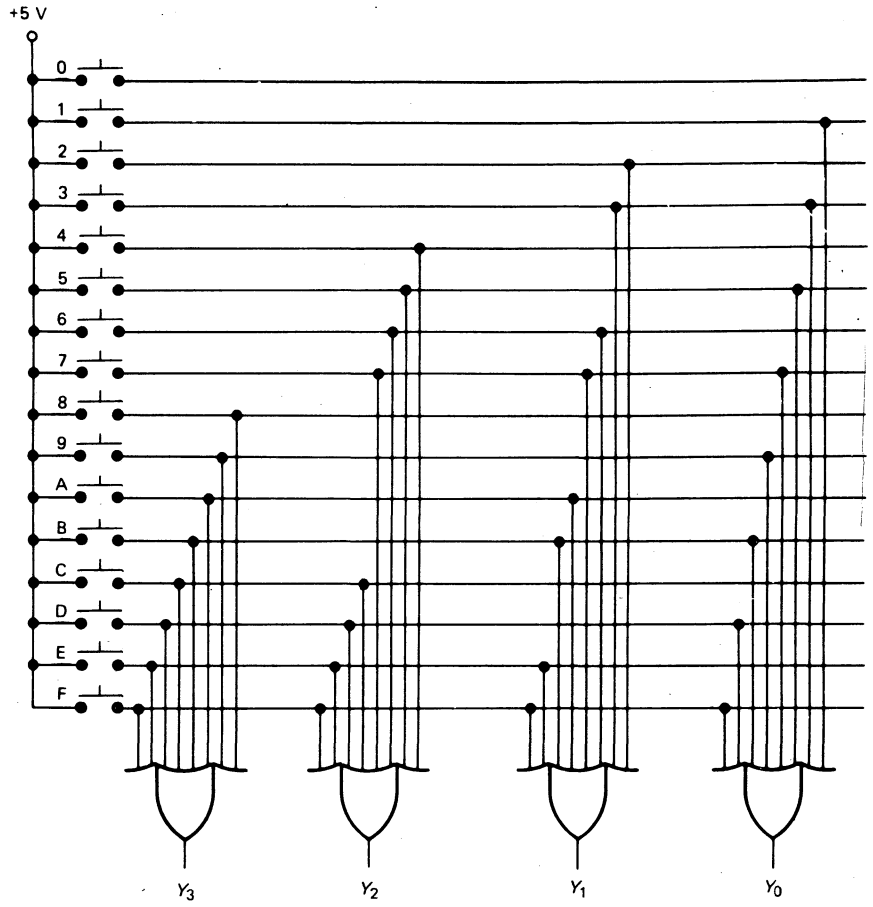


Fig. 2-18 Hexadecimal encoder.

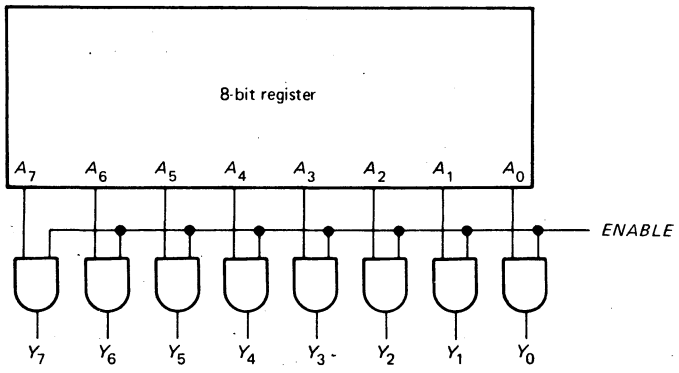


Fig. 2-19

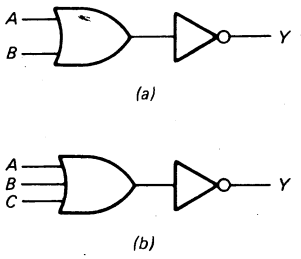


Fig. 2-20

2-11. The 8-bit register of Fig. 2-19 stores 59_{10}. What is the decimal equivalent of the final output word if *ENABLE* = 0? If *ENABLE* = 1?

2-12. Answer these questions:

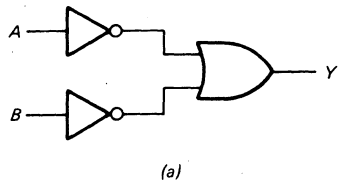
a. What input words does a 6-input OR gate recognize? What word does it disregard?

b. What input word does an 8-input AND gate recognize? What words does it disregard?

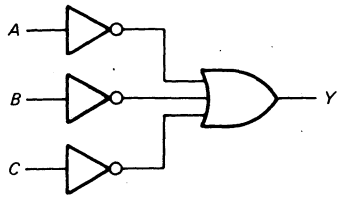
2-13. What is the boolean equation for Fig. 2-20*a*? The output if both inputs are high?

2-14. If all inputs are high in Fig. 2-20*b*, what is the output? The boolean equation for the circuit? What is the only ABC input word the circuit recognizes?

2-15. If you constructed the truth table for Fig. 2-20*b*, how many input words would it contain?

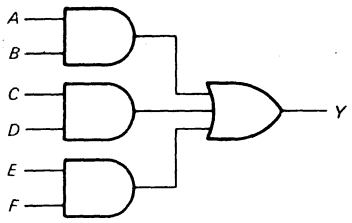


(a)

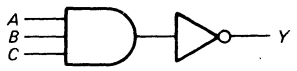


(b)

Fig. 2-21



(a)



(b)

Fig. 2-22

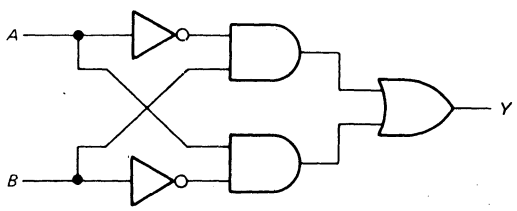


Fig. 2-23

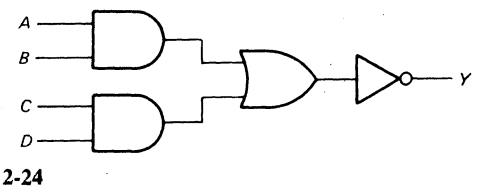


Fig. 2-24

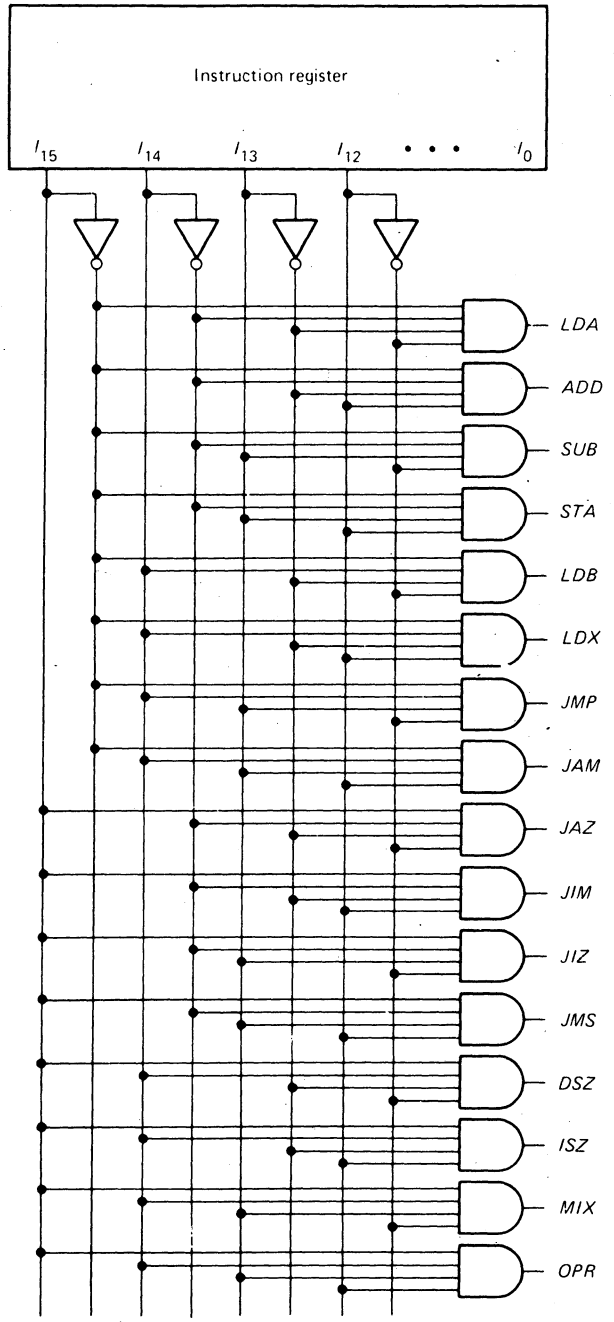


Fig. 2-25 A 1-of-16 decoder.

2-16. What is the boolean equation for Fig. 2-21*a*? The output if both inputs are high?

2-17. If all inputs are high in Fig. 2-21*b*, what is the output? What is the boolean equation of the circuit? What ABC input words does the circuit recognize? What is the only word it disregards?

2-18. What is the boolean equation for Fig. 2-22*a*? The output if all inputs are 1s? If you were to construct the truth table, how many input words would it have?

2-19. Write the boolean equation for Fig. 2-22*b*. If all inputs are 1s, what is the output?

2-20. If both inputs are high in Fig. 2-23, what is the output? What is the boolean equation for the circuit? Describe the truth table.

2-21. What is the boolean equation for Fig. 2-24? How many ABCD input words are in the truth table? Which input words does the circuit recognize?

2-22. Because of the historical connection between boolean algebra and logic, some people use the words "true" and "false" instead of "high" and "low" when discussing logic circuits. For instance, here's how an AND gate can be described. If any input is false, the output is false; if all inputs are true, the output is true.

a. If both inputs are false in Fig. 2-23, what is the output?

b. What is the output in Fig. 2-23 if one input is false and the other true?

c. In Fig. 2-23 what is the output if all inputs are true?

2-23. Figure 2-25 shows a 1-of-16 decoder. The signals coming out of the decoder are labeled *LDA*, *ADD*, *SUB*, and so on. The word formed by the 4 leftmost register bits is called the **OP CODE.** As an equation,

$$\textbf{OP CODE} = I_{15}I_{14}I_{13}I_{12}$$

a. If *LDA* is high, what does **OP CODE** equal?

b. If *ADD* is high, what does it equal?

c. When **OP CODE** = 1001, which of the output signals is high?

d. Which output signal is high if **OP CODE** = 1111?

2-24. In Fig. 2-25, list the **OP CODE** words and the corresponding high output signals. (Start with 0000 and proceed in binary to 1111.)

2-25. In the following equations the equals sign means "is equivalent to." Classify each of the following as positive or negative logic:

a. 0 = 0 V and 1 = +5 V.

b. 0 = +5 V and 1 = 0 V.

c. 0 = −5 V and 1 = 0 V.

d. 0 = 0 V and 1 = −5 V.

More Logic Gates

3

This chapter introduces NOR and NAND gates, devices that are widely used in industry. You will also learn about De Morgan's theorems; they help you to rearrange and simplify logic circuits.

3-1 NOR GATES

The NOR gate has two or more input signals but only one output signal. All inputs must be low to get a high output. In other words, the NOR gate recognizes only the input word whose bits are all 0s.

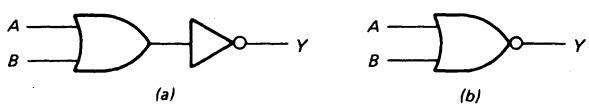


Fig. 3-1 NOR gate: (*a*) logical meaning; (*b*) standard symbol.

Two-Input Gate

Figure 3-1*a* shows the logical structure of a NOR gate, which is an OR gate followed by an inverter. Therefore, the final output is NOT the OR of the inputs. Originally called a NOT-OR gate, the circuit is now referred to as a NOR gate.

Figure 3-1*b* is the standard symbol for a NOR gate. Notice that the inverter triangle has been deleted and the small circle or bubble moved to the OR-gate output. The bubble is a reminder of the inversion that follows the ORing.

With Fig. 3-1*a* and *b* the following ideas are clear. If both inputs are low, the final output is high. If one input is low and the other high, the output is low. And if both inputs are high, the output is low.

Table 3-1 summarizes the circuit action. As you see, the NOR gate recognizes only the input word whose bits are all 0s. In other words, all inputs must be low to get a high output.

TABLE 3-1. TWO-INPUT NOR GATE

A	*B*	$\overline{A + B}$
0	0	1
0	1	0
1	0	0
1	1	0

Incidentally, the boolean equation for a 2-input NOR gate is

$$Y = \overline{A + B} \tag{3-1}$$

Read this as "*Y* equals NOT *A* OR *B*." If you use this equation, remember that the ORing is done first, then the inversion.

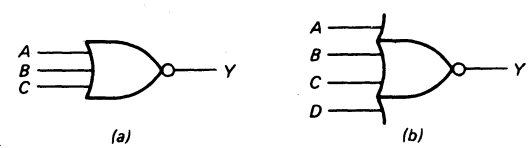


Fig. 3-2 NOR gates: (*a*) 3-input; (*b*) 4-input.

Three-Input Gate

Regardless of how many inputs a NOR gate has, it is still logically equivalent to an OR gate followed by an inverter. For instance, Fig. 3-2*a* shows a 3-input NOR gate. The 3 inputs are ORed, and the result is inverted. Therefore, the boolean equation is

$$Y = \overline{A + B + C} \tag{3-2}$$

The analysis of Fig. 3-2*a* goes like this. If all inputs are low, the result of ORing is low; therefore, the final output

TABLE 3-2. THREE-INPUT NOR GATE

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

is high. If one or more inputs are high, the result of ORing is high; so the final output is low.

Table 3-2 summarizes the action of a 3-input NOR gate. As you see, the circuit recognizes only the input word whose bits are 0s. In other words, all inputs must be low to get a high output.

Four-Input Gate

Figure 3-2*b* is the symbol for a 4-input NOR gate. The inputs are ORed, and the result is inverted. For this reason, the boolean equation is

$$Y = \overline{A + B + C + D} \qquad (3\text{-}3)$$

The corresponding truth table has input words from 0000 to 1111. Word 0000 gives a 1 output; all other words produce a 0 output. (For practice, you should construct the truth table of the 4-input NOR gate.)

3-2 DE MORGAN'S FIRST THEOREM

Most mathematicians ignored boolean algebra when it first appeared; some even ridiculed it. But Augustus De Morgan saw that it offered profound insights. He was the first to acclaim Boole's great achievement.

Always a warm and likable man, De Morgan himself had paved the way for boolean algebra by discovering two important theorems. This section introduces the first theorem.

The First Theorem

Figure 3-3*a* is a 2-input NOR gate, analyzed earlier. As you recall, the boolean equation is

$$Y = \overline{A + B}$$

and Table 3-3 is the truth table.

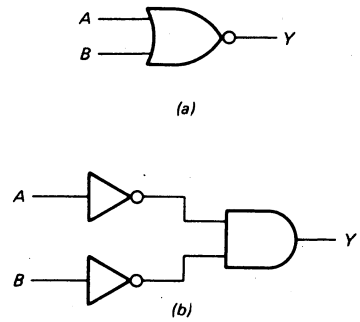


Fig. 3-3 De Morgan's first theorem: (*a*) NOR gate; (*b*) AND gate with inverted inputs.

Figure 3-3*b* has the inputs inverted before they reach the AND gate. Therefore, the boolean equation is

$$Y = \overline{A}\,\overline{B}$$

If both inputs are low in Fig. 3-3*b*, the AND gate has high inputs; therefore, the final output is high. If one or more inputs are high, one or more AND-gate inputs must be low and the final output is low. Table 3-4 summarizes these ideas.

TABLE 3-3

A	B	$\overline{A + B}$
0	0	1
0	1	0
1	0	0
1	1	0

TABLE 3-4

A	B	$\overline{A}\,\overline{B}$
0	0	1
0	1	0
1	0	0
1	1	0

Compare Tables 3-3 and 3-4. They're identical. This means that the two circuits are logically equivalent; given the same inputs, the outputs are the same. In other words, the circuits of Fig. 3-3 are interchangeable.

De Morgan discovered the foregoing equivalence long before logic circuits were invented. His first theorem says

$$\overline{A + B} = \overline{A}\,\overline{B} \qquad (3\text{-}4)$$

The left member of this equation represents Fig. 3-3*a*; the right member, Fig. 3-3*b*. Equation 3-4 says that Fig. 3-3*a* and *b* are equivalent (interchangeable).

Bubbled AND Gate

Figure 3-4*a* shows an AND gate with inverted inputs. This circuit is so widely used that the abbreviated logic symbol of Fig. 3-4*b* has been adopted. Notice that the inverter triangles have been deleted and the bubbles moved to the

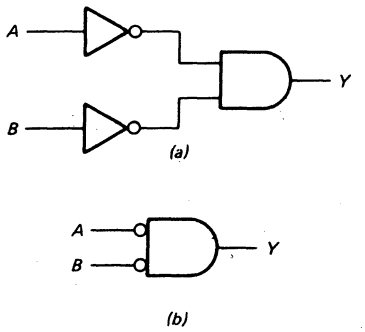


Fig. 3-4 AND gate with inverted inputs: (*a*) circuit; (*b*) abbreviated symbol.

AND-gate inputs. From now on, we will refer to Fig. 3-4*b* as a *bubbled* AND *gate;* the bubbles are a reminder of the inversion that takes place before ANDing.

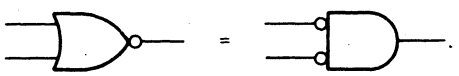


Fig. 3-5 De Morgan's first theorem.

Figure 3-5 is a graphic summary of De Morgan's first theorem. A NOR gate and a bubbled AND gate are equivalent. As shown later, because the circuits are interchangeable, you can often reduce complicated logic circuits to simpler forms.

More than Two Inputs

When 3 inputs are involved, De Morgan's first theorem is written

$$\overline{A + B + C} = \overline{A}\,\overline{B}\,\overline{C} \qquad (3\text{-}5)$$

For 4 inputs

$$\overline{A + B + C + D} = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} \qquad (3\text{-}6)$$

In both cases, the theorem says that the complement of a sum equals the product of the complements.

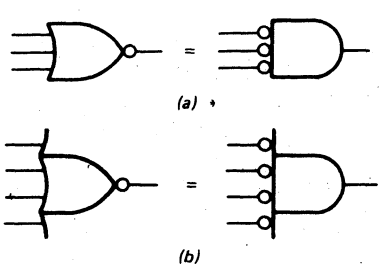


Fig. 3-6 De Morgan's first theorem: (*a*) 3-input circuits; (*b*) 4-input circuits.

Here's what really counts. Equation 3-5 says that a 3-input NOR gate and a 3-input bubbled AND gate are equivalent (see Fig. 3-6*a*). Equation 3-6 means that a 4-input NOR gate and a 4-input bubbled AND gate are equivalent (Fig. 3-6*b*). Memorize these equivalent circuits; they are a visual statement of De Morgan's first theorem.

Notice in Fig. 3-6*b* how the input edges of the NOR gate and the bubbled AND gate have been extended. This is common drafting practice when there are many input signals. The same idea applies to any type of gate.

EXAMPLE 3-1

Prove that Fig. 3-7*a* and *c* are equivalent.

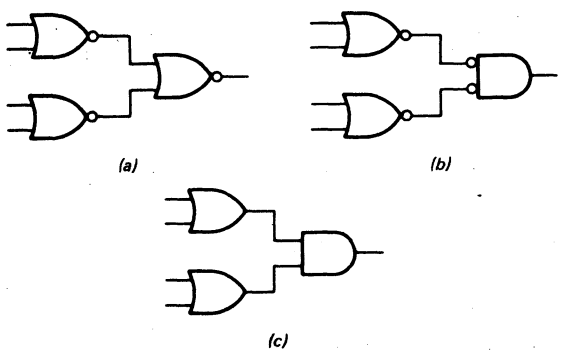


Fig. 3-7 Equivalent De Morgan circuits.

SOLUTION

The final NOR gate in Fig. 3-7*a* is equivalent to a bubbled AND gate. This allows us to redraw the circuit as shown in Fig. 3-7*b*.

Double inversion produces noninversion; therefore, each double inversion in Fig. 3-7*b* cancels out, leaving the simplified circuit of Fig. 3-7*c*. Figure 3-7*a* and *c* are therefore equivalent.

Remember the idea. Given a logic circuit, you can replace any NOR gate by a bubbled AND gate. Then any double inversion (a pair of bubbles in a series path) cancels out. Sometimes you wind up with a simpler logic circuit than you started with; sometimes not.

But the point remains. De Morgan's first theorem enables you to rearrange a logic circuit with the hope of finding a simpler equivalent circuit or perhaps getting more insight into how the original circuit works.

3-3 NAND GATES

The NAND gate has two or more input signals but only one output signal. All input signals must be high to get a low output.

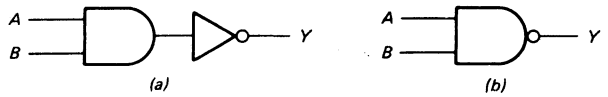


Fig. 3-8 NAND gate: (*a*) logical meaning; (*b*) standard symbol.

Two-Input Gate

Figure 3-8*a* shows the logical structure of a NAND gate, an AND gate followed by an inverter. Therefore, the final output is NOT the AND of the inputs. Originally called a NOT-AND gate, the circuit is now referred to as a NAND gate.

Figure 3-8*b* is the standard symbol for a NAND gate. The inverter triangle has been deleted and the bubble moved to the AND-gate output. If one or more inputs are low, the result of ANDing is low; therefore, the final inverted output is high. Only when all inputs are high does the ANDing produce a high signal; then the final output is low.

Table 3-5 summarizes the action of a 2-input NAND gate. As shown, the NAND gate recognizes any input word with one or more 0s. That is, one or more low inputs produce a high output. The boolean equation for a 2-input NAND gate is

$$Y = \overline{AB} \tag{3-7}$$

Read this as "*Y* equals NOT *AB*." If you use this equation, remember that the ANDing is done first then the inversion.

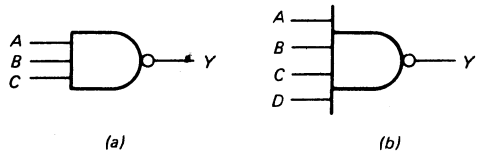


Fig. 3-9 NAND gates: (*a*) 3-input; (*b*) 4-input.

Three-Input Gate

Regardless of how many inputs a NAND gate has, it's still logically equivalent to an AND gate followed by an inverter. For example, Fig. 3-9*a* shows a 3-input NAND gate. The inputs are ANDed, and the product is inverted. Therefore, the boolean equation is

$$Y = \overline{ABC} \tag{3-8}$$

Here is the analysis of Fig. 3-9*a*. If one or more inputs are low, the result of ANDing is low; therefore, the final output is high. If all inputs are high, the ANDing gives a high signal; so the final output is low.

Table 3-6 is the truth table for a 3-input NAND gate. As indicated, the circuit recognizes words with one or more 0s. This means that one or more low inputs produce a high output.

TABLE 3-5. TWO-INPUT NAND GATE

A	*B*	$\overline{AB}$
0	0	1
0	1	1
1	0	1
1	1	0

TABLE 3-6. THREE-INPUT NAND GATE

A	*B*	*C*	$\overline{ABC}$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Four-Input Gate

Figure 3-9*b* is the symbol for a 4-input NAND gate. The inputs are ANDed, and the result is inverted. Therefore, the boolean equation is

$$Y = \overline{ABCD} \tag{3-9}$$

If you construct the truth table, you will have input words from 0000 to 1111. All words from 0000 through 1110 produce a 1 output; only the word 1111 gives a 0 output.

3-4 DE MORGAN'S SECOND THEOREM

The proof of De Morgan's second theorem is similar to the proof given for the first theorem. What follows is a brief explanation.

The Second Theorem

When two inputs are used, De Morgan's second theorem says that

$$\overline{AB} = \overline{A} + \overline{B} \tag{3-10}$$

In words, the complement of a product equals the sum of the complements. The left member of this equation represents a NAND gate (Fig. 3-10*a*); the right member stands

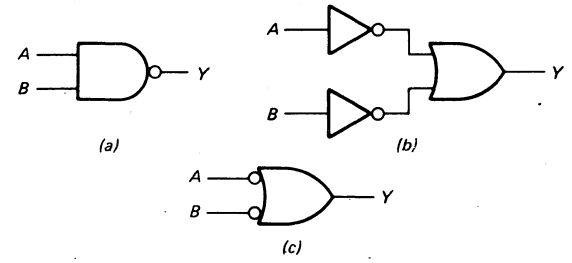


Fig. 3-10 De Morgan's second theorem: (*a*) NAND gate; (*b*) OR gate with inverted inputs; (*c*) bubbled OR gate.

for an OR gate with inverted inputs (Fig. 3-10*b*). Therefore, De Morgan's second theorem boils down to the fact that Fig. 3-10*a* and *b* are equivalent.

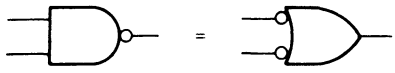


Fig. 3-11 De Morgan's second theorem.

Bubbled OR Gate

The circuit of Fig. 3-10*b* is so widely used that the abbreviated logic symbol of Fig. 3-10*c* has been adopted. From now on we will refer to Fig. 3-10*c* as a *bubbled* OR gate; the bubbles are a reminder of the inversion that takes place before ORing.

Figure 3-11 is a visual statement of De Morgan's second theorem: a NAND gate and a bubbled OR gate are equivalent. This equivalence allows you to replace one circuit by the other whenever desired. This may lead to a simpler logic circuit or give you more insight into how the original circuit works.

More than Two Inputs

When 3 inputs are involved, De Morgan's second theorem is written

$$\overline{ABC} = \overline{A} + \overline{B} + \overline{C} \qquad (3\text{-}11)$$

If 4 inputs are used,

$$\overline{ABCD} = \overline{A} + \overline{B} + \overline{C} + \overline{D} \qquad (3\text{-}12)$$

These equations say that the complement of a product equals the sum of the complements.

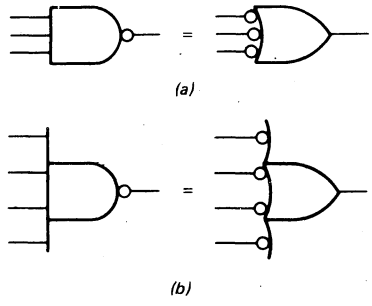


Fig. 3-12 De Morgan's second theorem: (*a*) 3-input circuits; (*b*) 4-input circuits.

Figure 3-12 is a visual summary of the second theorem. Whether 3 or 4 inputs are involved, a NAND gate and a bubbled OR gate are equivalent (interchangeable).

EXAMPLE 3-2

Prove that Fig. 3-13*a* and *c* are equivalent.

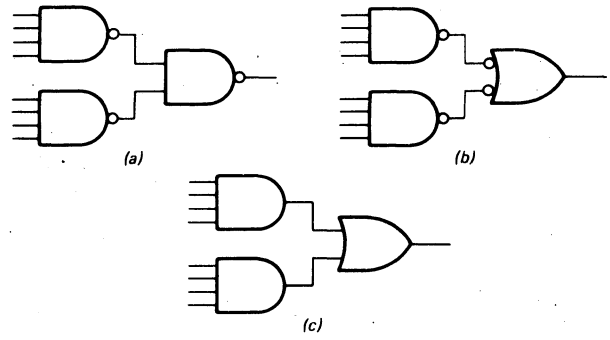


Fig. 3-13 Equivalent circuits.

SOLUTION

Replace the final NAND gate in Fig. 3-13*a* by a bubbled OR gate. This gives Fig. 3-13*b*. The double inversions cancel out, leaving the simplified circuit of Fig. 3-13*c*. Figure 3-13*a* and *c* are therefore equivalent. Driven by the same inputs, either circuit produces the same output as the other. So if you're loaded with NAND gates, build Fig. 3-13*a*. If your shelves are full of AND and OR gates, build Fig. 3-13*c*.

Incidentally, most people find Fig. 3-13*b* easier to analyze than Fig. 3-13*a*. For this reason, if you build Fig. 3-13*a*, draw the circuit like Fig. 3-13*b*. Anyone who sees Fig. 3-13*b* on a schematic diagram knows that the bubbled OR gate is the same as a NAND gate and that the built-up circuit is two NAND gates working into a NAND gate.

EXAMPLE 3-3

Figure 3-14 shows a circuit called a *control matrix*. At first, it looks complicated, but on closer inspection it is relatively simple because of the repetition of NAND gates. De Morgan's theorem tells us that NAND gates driving NAND gates are equivalent to AND gates driving OR gates.

The upper set of inputs T_1 to T_6 are called *timing signals;* only one of them is high at a time. T_1 goes high first, then T_2, then T_3, and so on. These signals control the rate and sequence of computer operations.

The lower set of inputs *LDA*, *ADD*, *SUB*, and *OUT* are computer instructions; only one of them is high at a time. The outputs C_P, E_P, L_M, . . . , to L_O control different registers in the computer.

Answer the following questions about the control matrix:

a. Which outputs are high when T_1 is high?
b. If T_4 and *LDA* are high, which outputs are high?
c. When T_6 and *SUB* are high, which outputs are high?

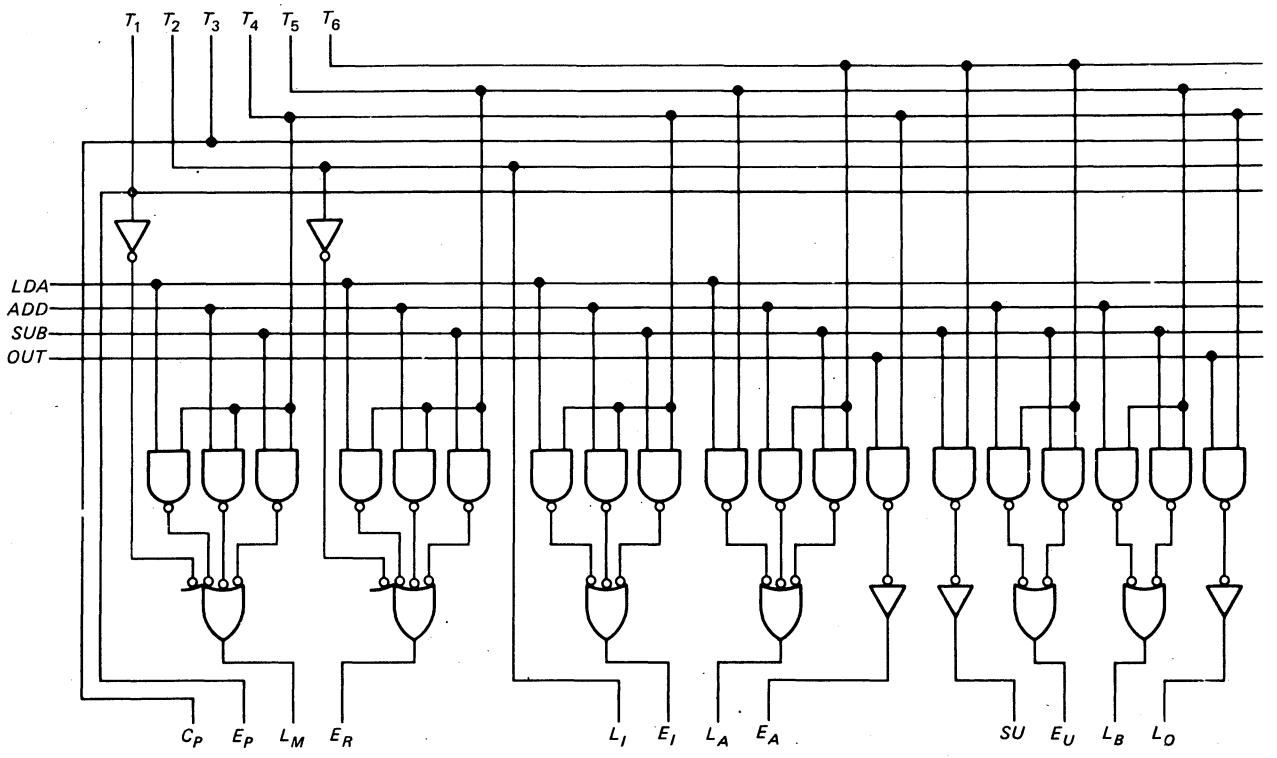


Fig. 3-14 Control matrix.

SOLUTION

a. Visualize T_1 high. You can quickly check out each gate and realize that E_P and L_M are the only high outputs.
b. This time T_4 and *LDA* are high. Check each gate and you can see that L_M and E_I are the only high outputs.
c. When T_6 and *SUB* are high, the high outputs are L_A, S_U, and E_U.

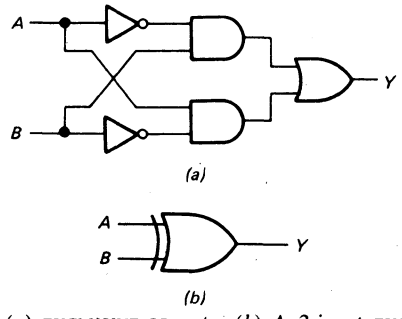


Fig. 3-15 (*a*) EXCLUSIVE-OR gate. (*b*) A 2-input EXCLUSIVE-OR gate.

3-5 EXCLUSIVE-OR GATES

An OR gate recognizes words with one or more 1s. The EXCLUSIVE-OR gate is different; it recognizes only words that have an odd number of 1s.

Two Inputs

Figure 3-15*a* shows one way to build an EXCLUSIVE-OR gate, abbreviated XOR. The upper AND gate forms the product $\overline{A}B$, and the lower AND gate gives $A\overline{B}$. Therefore, the boolean equation is

$$Y = \overline{A}B + A\overline{B} \qquad (3\text{-}13)$$

Here's what the circuit does. In Fig. 3-15*a* two low inputs mean both AND gates have low outputs; so the final output is low. If A is low and B is high, the upper AND gate has a high output; therefore, the final output is high. Likewise, a high A and low B result in a final output that is high. If both inputs are high, both AND gates have low outputs and the final output is low.

Table 3-7 shows the truth table for a 2-input EXCLUSIVE-OR gate. The output is high when A or B is high but not both; this is why the circuit is known as an EXCLUSIVE-OR gate. In other words, the output is a 1 only when the inputs are different.

TABLE 3-7. TWO-INPUT XOR GATE

A	B	$\overline{A}B + A\overline{B}$
0	0	0
0	1	1
1	0	1
1	1	0

Logic Symbol and Boolean Sign

Figure 3-15*b* is the standard symbol for a 2-input XOR gate. Whenever you see this symbol, remember the action: the inputs must be different to get a high output.

A word equation for Fig. 3-15*b* is

$$Y = A \text{ XOR } B \tag{3-14}$$

In boolean algebra the sign $\oplus$ stands for XOR addition. This means that Eq. 3-14 can be written

$$Y = A \oplus B \tag{3-15}$$

Read this as "Y equals A XOR B."

Given the inputs, you can substitute and solve for the output. For instance, if both inputs are low,

$$Y = 0 \oplus 0 = 0$$

because 0 XORed with 0 gives 0. If one input is low and the other high,

$$Y = 0 \oplus 1 = 1$$

because 0 XORed with 1 produces 1. And so on.

Here's a summary of the four possible XOR additions:

$$0 \oplus 0 = 0$$
$$0 \oplus 1 = 1$$
$$1 \oplus 0 = 1$$
$$1 \oplus 1 = 0$$

Remember these four results; we will be using XOR addition when we get to arithmetic circuits.

Four Inputs

In Fig. 3-16*a* the upper gate produces $A \oplus B$, while the lower gate gives $C \oplus D$. The final gate XORs both of these sums to get

$$Y = (A \oplus B) \oplus (C \oplus D) \tag{3-16}$$

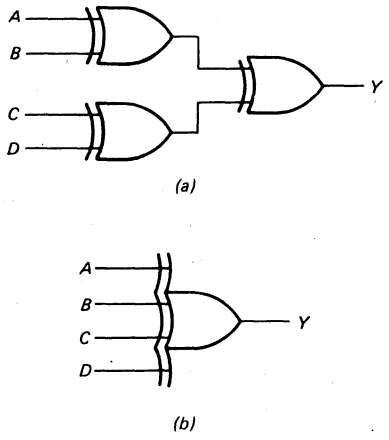


Fig. 3-16 A 4-input EXCLUSIVE-OR gate: (*a*) circuit with 2-input XOR gates; (*b*) logic symbol.

It's possible to substitute input values into the equation and solve for the output. For instance, if A through C are low and D is high,

$$\begin{aligned} Y &= (0 \oplus 0) \oplus (0 \oplus 1) \\ &= 0 \oplus 1 \\ &= 1 \end{aligned}$$

One way to get the truth table is to plow through all the input possibilities.

Alternatively, you can analyze Fig. 3-16*a* as follows. If all inputs are 0s, the first two gates have 0 outputs; so the final gate has a 0 output. If A to C are 0s and D is a 1, the upper gate has a 0 output, the lower gate has a 1 output, and the final gate has a 1 output. In this way, you can analyze the circuit action for all input words.

Table 3-8 summarizes the action. Here is an important property: each input word with an odd number of 1s produces a 1 output. For instance, the first input word to produce a 1 output is 0001; this word has an odd number of 1s. The next word with a 1 output is 0010; again an odd number of 1s. A 1 output also occurs for these words: 0100, 0111, 1000, 1011, 1101, and 1110, all of which have an odd number of 1s.

The circuit of Fig. 3-16*a* recognizes words with an odd number of 1s; it disregards words with an even number of 1s. Figure 3-16*a* is a 4-input XOR gate. In this book, we will use the abbreviated symbol of Fig. 3-16*b* to represent a 4-input XOR gate. When you see this symbol, remember the action: the circuit recognizes words with an odd number of 1s.

Any Number of Inputs

Using 2-input XOR gates as building blocks, we can make XOR gates with any number of inputs. For example, Fig.

TABLE 3-8. FOUR-INPUT XOR GATE

Comment	*A*	*B*	*C*	*D*	*Y*
Even	0	0	0	0	0
Odd	0	0	0	1	1
Odd	0	0	1	0	1
Even	0	0	1	1	0
Odd	0	1	0	0	1
Even	0	1	0	1	0
Even	0	1	1	0	0
Odd	0	1	1	1	1
Odd	1	0	0	0	1
Even	1	0	0	1	0
Even	1	0	1	0	0
Odd	1	0	1	1	1
Even	1	1	0	0	0
Odd	1	1	0	1	1
Odd	1	1	1	0	1
Even	1	1	1	1	0

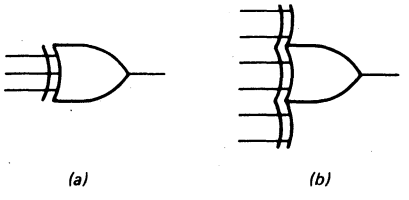


Fig. 3-17 XOR gates: (*a*) 3-input; (*b*) 6-input.

3-17*a* shows the abbreviated symbol for a 3-input XOR gate, and Fig. 3-17*b* is the symbol for a 6-input XOR gate. The final output of any XOR gate is the XOR sum of the inputs:

$$Y = A \oplus B \oplus C \cdots \qquad (3\text{-}17)$$

What you have to remember for practical work is this: an XOR gate, no matter how may inputs, recognizes only words with an odd number of 1s.

Parity

Even parity means a word has an even number of 1s. For instance, 110011 has even parity because it contains four 1s. *Odd parity* means a word has an odd number of 1s. As an example, 110001 has odd parity because it contains three 1s.

Here are two more examples:

1111 0000 1111 0011 (Even parity)
1111 0000 1111 0111 (Odd parity)

The first word has even parity because it contains ten 1s; the second word has odd parity because it contains eleven 1s.

XOR gates are ideal for testing the parity of a word. XOR gates recognize words with an odd number of 1s. Therefore, even-parity words produce a low output and odd-parity words produce a high output.

EXAMPLE 3-4

What is the output of Fig. 3-18 for each of these input words?

a. 1010 1100 1000 1100
b. 1010 1100 1000 1101

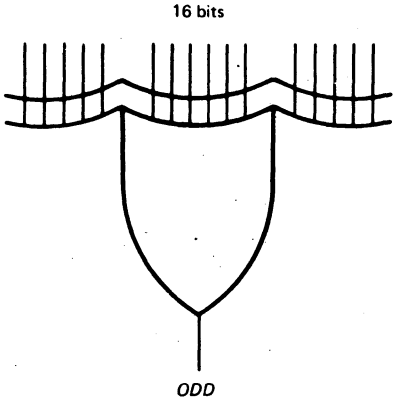


Fig. 3-18 Odd-parity tester.

SOLUTION

a. The word has seven 1s, an odd number. Therefore, the output signal is

$$ODD = 1$$

b. The word has eight 1s, an even number. Now

$$ODD = 0$$

This is an example of an odd-parity tester. An even-parity word produces a low output. An odd-parity word results in a high output.

EXAMPLE 3-5

The 7-bit register of Fig. 3-19 stores the letter A in ASCII form. What does the 8-bit output word equal?

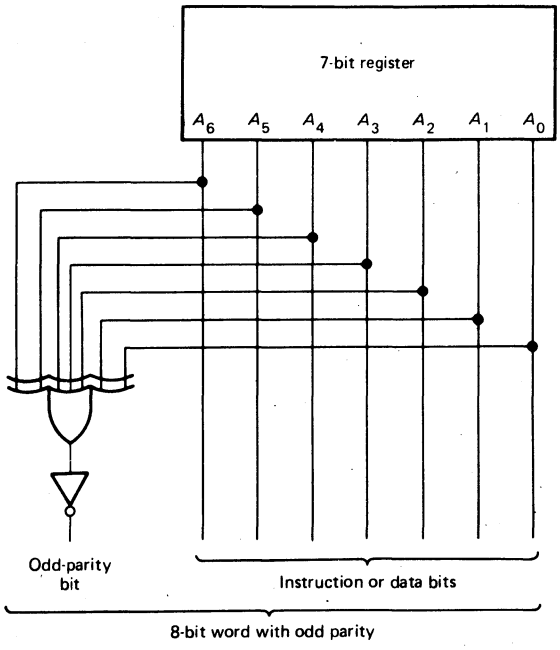


Fig. 3-19 Odd-parity generator.

SOLUTION

The ASCII code for letter A is

100 0001

(see Table 1-6 for the ASCII code). This word has an even parity, which means that the XOR gate has a 0 output. Because of the inverter, the overall output of the circuit is the 8-bit word

1100 0001

Notice that this has odd parity.

The circuit is called an *odd-parity generator* because it produces an 8-bit output word with odd parity. If the register word has even parity, 0 comes out of the XOR gate and the odd-parity bit is 1. On the other hand, if the register word has odd parity, a 1 comes out of the XOR gate and the odd-parity bit is 0. No matter what the register contents, the odd-parity bit and the register bits form a new 8-bit word that has odd parity.

What is the practical application? Because of transients, noise, and other disturbances, 1-bit errors sometimes occur in transmitted data. For instance, the letter A may be transmitted over phone lines in ASCII form:

100 0001 (A)

Somewhere along the line, one of the bits may be changed. If the X_1 bit changes, the received data will be

100 0011 (C)

Because of the 1-bit error, we receive letter C when letter A was actually sent.

One solution is to transmit an odd-parity bit along with the data word and have an XOR gate test each received word for odd parity. For instance, with a circuit like Fig. 3-19 the letter A would be transmitted as

1100 0001

An XOR gate will test this word when it is received. If no error has occurred, the XOR gate will recognize the word. On the other hand, if a 1-bit error has crept in, the XOR gate will disregard the received word and the data can be rejected.

A final point. When errors come, they are usually 1-bit errors. This is why the method described catches most of the errors in transmitted data.

EXAMPLE 3-6

What does the circuit of Fig. 3-20 do?

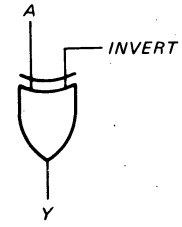


Fig. 3-20

SOLUTION

When *INVERT* = 0 and $A = 0$,

$$Y = 0 \oplus 0 = 0$$

When *INVERT* = 0 and $A = 1$,

$$Y = 0 \oplus 1 = 1$$

In either case, the output is the same as A; that is,

$$Y = A$$

for a low *INVERT* signal.

On the other hand, when *INVERT* = 1 and $A = 0$,

$$Y = 1 \oplus 0 = 1$$

When *INVERT* = 1 and $A = 1$,

$$Y = 1 \oplus 1 = 0$$

This time, the output is the complement of A. As an equation,

$$Y = \overline{A}$$

for a high *INVERT* signal.

To summarize, the circuit of Fig. 3-20 does either of two things. It transmits A when *INVERT* is 0 and $\overline{A}$ when INVERT is 1.

3-6 THE CONTROLLED INVERTER

The preceding example suggests the idea of a *controlled inverter*, a circuit that transmits a binary word or its *1's complement*.

The 1's Complement

Complement each bit in a word and the new word you get is the 1's complement. For instance, given

1100 0111

the 1's complement is

0011 1000

Each bit in the original word is inverted to get the 1's complement.

The Circuit

The XOR gates of Fig. 3-21 form a *controlled inverter* (sometimes called a programmed inverter). This circuit can transmit the register contents or the 1's complement of the register contents. As demonstrated in Example 3-6, each XOR gate acts like this. A low *INVERT* results in

$$Y_n = A_n$$

and a high *INVERT* gives

$$Y_n = \overline{A}_n$$

So each bit is either transmitted or inverted before reaching the final output.

Visualize the register contents as a word $A_7A_6 \cdots A_0$ and the final output as a word $Y_7Y_6 \cdots Y_0$. Then a low *INVERT* means

$$Y_7Y_6 \cdots Y_0 = A_7A_6 \cdots A_0$$

On the other hand, a high *INVERT* results in

$$Y_7Y_6 \cdots Y_0 = \overline{A}_7\overline{A}_6 \cdots \overline{A}_0$$

As a concrete example, suppose the register word is

$$A_7A_6 \cdots A_0 = 1110\ 0110$$

Then, a low *INVERT* gives an output word of

$$Y_7Y_6 \cdots Y_0 = 1110\ 0110$$

and a high *INVERT* produces

$$Y_7Y_6 \cdots Y_0 = 0001\ 1001$$

The controlled inverter of Fig. 3-21 is important. Later you will see how it is used in solving arithmetic and logic problems. For now, all you need to remember is the key idea. The output word from a controlled inverter equals the

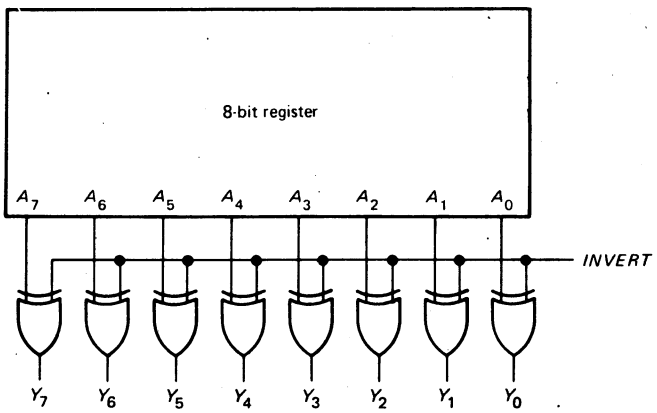


Fig. 3-21 Controlled inverter.

input word when *INVERT* is low; the output word equals the 1's complement when *INVERT* is high.

Boldface Notation

After you understand an idea, it simplifies discussions and equations if you use a symbol, letter, or other sign to represent the idea. From now on, boldface letters will stand for binary words.

For instance, instead of writing

$$A_7A_6 \cdots A_0 = 1110\ 0110$$

we can write

$$\mathbf{A} = 1110\ 0110$$

Likewise, instead of

$$Y_7Y_6 \cdots Y_0 = 0001\ 1001$$

the simpler equation

$$\mathbf{Y} = 0001\ 1001$$

can be used.

This is another example of chunking. We are replacing long strings like $A_7A_6 \cdots A_0$ and $Y_7Y_6 \cdots Y_0$ by **A** and **Y**. This chunked notation will be convenient when we get to computer analysis.

This is how to summarize the action of a controlled inverter:

$$\mathbf{Y} = \begin{cases} \mathbf{A} & \text{when } \textit{INVERT} = 0 \\ \overline{\mathbf{A}} & \text{when } \textit{INVERT} = 1 \end{cases}$$

(*Note:* A boldface letter with an overbar means that each bit in the word is complemented; if **A** is a word, $\overline{\mathbf{A}}$ is its 1's complement.)

3-7 EXCLUSIVE-NOR GATES

The EXCLUSIVE-NOR gate, abbreviated XNOR, is logically equivalent to an XOR gate followed by an inverter. For example, Fig. 3-22*a* shows a 2-input XNOR gate. Figure 3-22*b* is an abbreviated way to draw the same circuit.

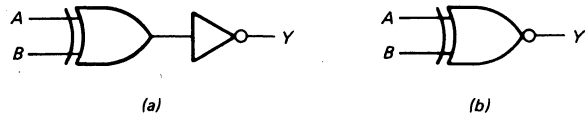


Fig. 3-22 A 2-input XNOR gate: (*a*) circuit; (*b*) abbreviated symbol.

TABLE 3-9. TWO-INPUT XNOR GATE

A	*B*	*Y*
0	0	1
0	1	0
1	0	0
1	1	1

Because of the inversion on the output side, the truth table of an XNOR gate is the complement of an XOR truth table. As shown in Table 3-9, the output is high when the inputs are the same. For this reason, the 2-input XNOR gate is ideally suited for *bit comparison*, recognizing when two input bits are identical. (Example 3-7 tells you more about bit comparison.)

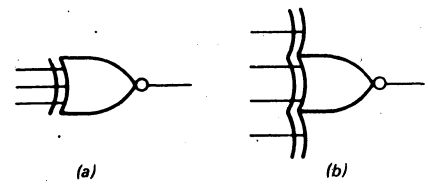


Fig. 3-23 XNOR gates: (*a*) 3-input; (*b*) 4-input.

Figure 3-23*a* is the symbol for a 3-input XNOR gate, and Fig. 3-23*b* is the 4-input XNOR gate. Because of the inversion on the output side, these XNOR gates perform the complementary function of XOR gates. Instead of recognizing odd-parity words, XNOR gates recognize even-parity words.

EXAMPLE 3-7

What does the circuit of Fig. 3-24 do?

SOLUTION

The circuit is a *word comparator;* it recognizes two identical words. Here is how it works. The leftmost XNOR gate compares A_5 and B_5; if they are the same, Y_5 is a 1. The second XNOR gate compares A_4 and B_4; if they are the same, Y_4 is a 1. In turn, the remaining XNOR gates compare the bits that are left, producing a 1 output for equal bits and a 0 output for unequal bits.

If the words **A** and **B** are identical, all XNOR gates have high outputs and the AND gate has a high *EQUAL*. If words **A** and **B** differ in one or more bit positions, the AND gate has a low *EQUAL*.

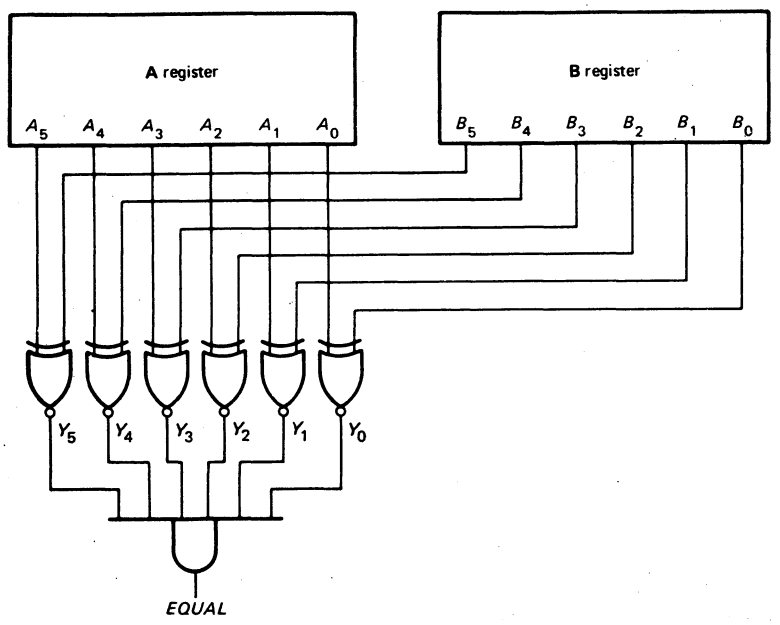


Fig. 3-24 Word comparator.

GLOSSARY

controlled inverter This circuit produces the 1's complement of the input word. One application is binary subtraction. It is sometimes called a programmed inverter.

De Morgan's theorems The first theorem says that a NOR gate is equivalent to a bubbled AND gate. The second theorem says that a NAND gate is equivalent to a bubbled OR gate.

even parity An even number of 1s in a binary word.

NAND *gate* Equivalent to an AND gate followed by an inverter. All inputs must be high to get a low output.

NOR *gate* Equivalent to an OR gate followed by an inverter. All inputs must be low to get a high output.

odd parity An odd number of 1s in a binary word.

parity generator A circuit that produces either an odd- or even-parity bit to go along with the data.

XNOR *gate* Equivalent to an EXCLUSIVE-OR gate followed by an inverter. The output is high only when the input word has even parity.

XOR *gate* An EXCLUSIVE-OR gate. It has a high output only when the input word has odd parity. For a 2-input XOR gate, the output is high only when the inputs are different.

SELF-TESTING REVIEW

Read each of the following and provide the missing words. Answers appear at the beginning of the next question.

1. A NOR gate has two or more input signals. All inputs must be _____ to get a high output. A NOR gate recognizes only the input word whose bits are _____. The NOR gate is logically equivalent to an OR gate followed by an _____.
2. (*low, 0s, inverter*) De Morgan's first theorem says that a NOR gate is equivalent to a bubbled _____ gate.
3. (AND) A NAND gate is equivalent to an AND gate followed by an inverter. All inputs must be _____ to get a low output. De Morgan's second theorem says that a NAND gate is equivalent to a bubbled _____ gate.
4. (*high*, OR) An XOR gate recognizes only words with an _____ number of 1s. The 2-input XOR gate has a high output only when the input bits are _____. XOR gates are ideal for testing parity because even-parity words produce a _____ output and odd-parity words produce a _____ output.
5. (*odd, different, low, high*) An odd-parity generator produces an odd-parity bit to go along with the data.

The parity of the transmitted data is _____. An XOR gate can test each received word for parity, rejecting words with _____ parity.

6. (*odd, even*) A controlled inverter is a logic circuit that transmits a binary word or its _____ complement.
7. (*1's*) The EXCLUSIVE-NOR gate is equivalent to an XOR gate followed by an inverter. Because of this, even-parity words produce a high output.

PROBLEMS

3-1. In Fig. 3-25*a* the two inputs are connected together. If *A* is low, what is *Y*? If *A* is high, what is *Y*? Does the circuit act like a noninverter or an inverter?

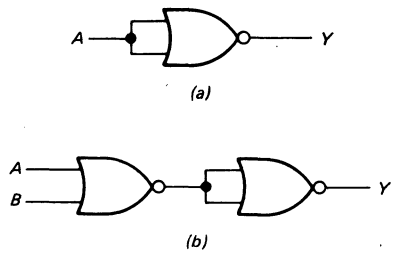


Fig. 3-25

3-2. What is the output in Fig. 3-25*b* if both inputs are low? If one is low and the other high? If both are high? Does the circuit act like an OR gate or an AND gate?

3-3. Figure 3-26 shows a NOR-gate *crossbar switch*. If all *X* and *Y* inputs are high, which of the *Z* outputs is high? If all inputs are high except X_1 and Y_2, which *Z* output is high? If X_2 and Y_0 are low and all other inputs are high, which *Z* output is high?

3-4. In Fig. 3-26, you want Z_7 to be 1 and all other *Z* outputs to be 0. What values must the *X* and *Y* inputs have?

3-5. The outputs in Fig. 3-27 are cross-coupled back to the inputs of the NOR gates. If $R = 0$ and $S = 1$, what do Q and $\overline{Q}$ equal?

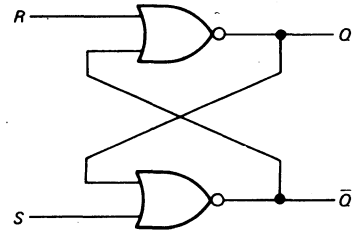


Fig. 3-27 Cross-coupled NOR gates.

3-6. If $R = 1$ and $S = 0$ in Fig. 3-27, what does Q equal? $\overline{Q}$?

3-7. Prove that Fig. 3-28*a* and *b* are equivalent.

3-8. What is the output in Fig. 3-28*a* if all inputs are 0s. If all inputs are 1s?

3-9. What is the output in Fig. 3-28*b* if all inputs are 0s. If all inputs are 1s?

3-10. A NOR has 6 inputs. How many input words are in its truth table? What is the only input word that produces a 1 output?

3-11. In Fig. 3-28*a* how many input words are there in the truth table?

3-12. What is the output in Fig. 3-29 if all inputs are low? If all inputs are high?

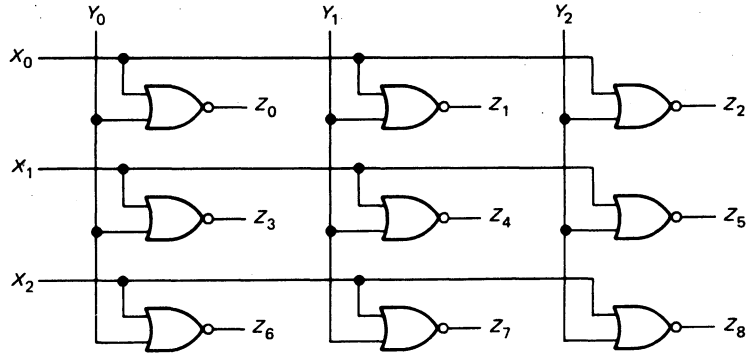


Fig. 3-26 NOR-gate crossbar switch.

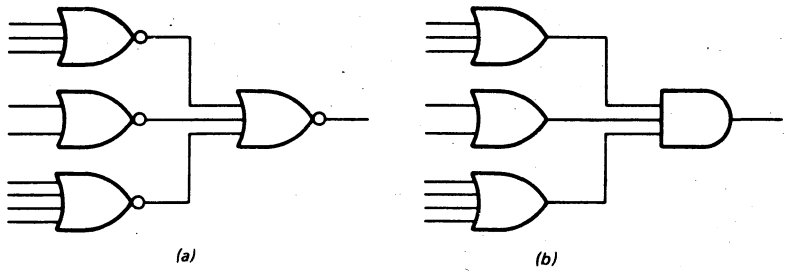


Fig. 3-28

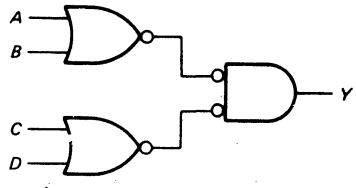


Fig. 3-29

3-13. How many words are in the truth table of Fig. 3-29. What is the value of Y for each of the following?
- a. ABCD = 0011
- b. ABCD = 0110
- c. ABCD = 1001
- d. ABCD = 1100

3-14. Which ABCD input words does the circuits of Fig. 3-29 recognize?

3-15. In Fig. 3-30*a* the two inputs are connected together. If $A = 0$ what does Y equal? If $A = 1$, what does Y equal? Does the circuit act like a noninverter or an inverter?

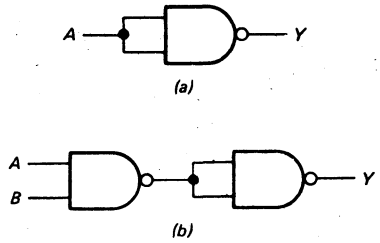


Fig. 3-30

3-16. What is the output in Fig. 3-30*b* if both inputs are low? If one input is low and the other high? If both are high? Does the circuit act like an OR gate or an AND gate?

3-17. Suppose the NOR gates of Fig. 3-26 are replaced by NAND gates. Then you've got a NAND-gate crossbar switch.
- a. If all X and Y inputs are low, which Z output is low?
- b. If all inputs are low except X_2 and Y_1, which Z output is low?
- c. If all inputs are low except X_0 and Y_2, which Z output is low?
- d. To get a low Z_8 output, which inputs must be high?

3-18. In Fig. 3-31, what are the outputs if $R = 0$ and $S = 1$?

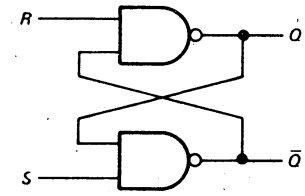


Fig. 3-31 Cross-coupled NAND gates.

3-19. If $R = 1$ and $S = 0$ in Fig. 3-31, what does Q equal? $\overline{Q}$?

3-20. What is the output in Fig. 3-32*a* if all inputs are 0s? If all inputs are 1s?

3-21. How many input words are there in the truth table of Fig. 3-32*a*?

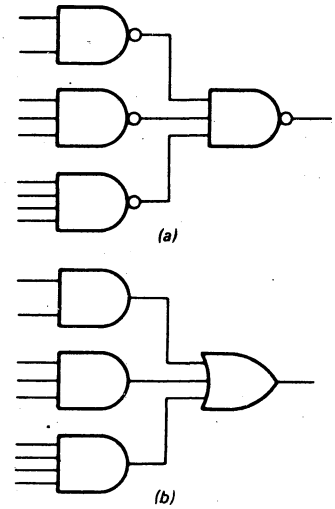


Fig. 3-32

3-22. Prove that Fig. 3-32*a* and *b* are equivalent.

3-23. What is the output in Fig. 3-33 if all inputs are low? If they are all high?

3-24. How many words are in the truth table of Fig. 3-33? What does Y equal for each of the following:

a. ABCDE = 00111
b. ABCDE = 10110
c. ABCDE = 11010
d. ABCDE = 10101

3-25. In Fig. 3-34 the inputs are T_4, *JMP*, *JAM*, *JAZ*, A_M, and A_Z; the output is L_P. What is the output for each of these input conditions?

a. All inputs are 0s.
b. All inputs are low except T_4 and *JMP*.
c. All inputs are low except T_4, *JAZ*, and A_Z.
d. The only high inputs are T_4, *JAM*, AND A_M.

3-26. Figure 3-35 shows the control matrix discussed in Example 3-3. Only one of the timing signals T_1 to T_6 is high at a time. Also, only one of the instructions, *LDA* to *OUT*, is high at a time. Which are the high outputs for each of the following conditions?

a. T_1 high
b. T_2 high
c. T_3 high
d. T_4 and *LDA* high
e. T_5 and *LDA* high
f. T_4 and *ADD* high
g. T_5 and *ADD* high
h. T_6 and *ADD* high
i. T_4 and *SUB* high
j. T_5 and *SUB* high
k. T_6 and *SUB* high
l. T_4 and *OUT* high

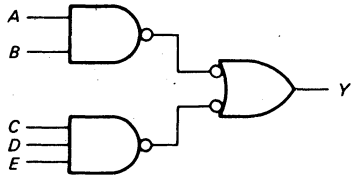


Fig. 3-33

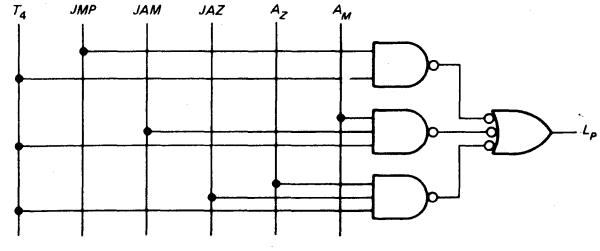


Fig. 3-34

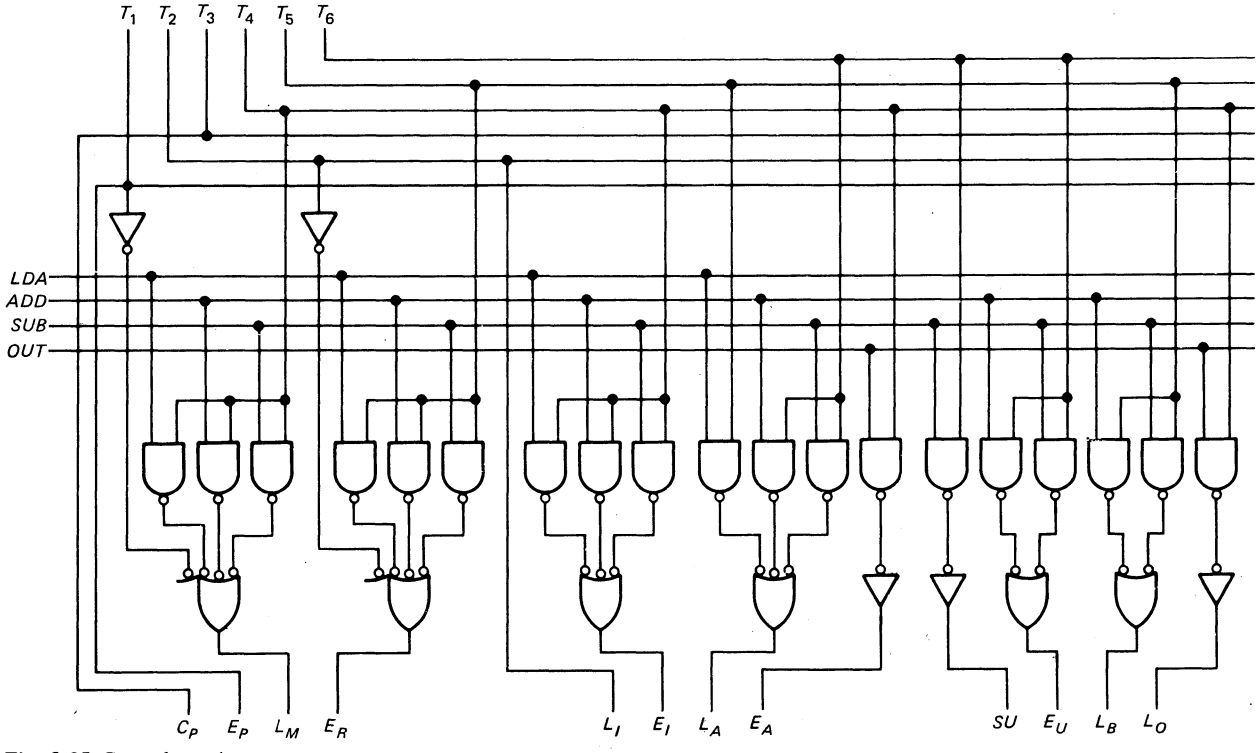


Fig. 3-35 Control matrix.

3-27. Figure 3-36 shows a binary–to–Gray-code converter. (Gray code is a special code used in analog-to-digital conversions.) The input word is $X_4X_3 \cdots X_0$, and the output word is $Y_4Y_3 \cdots Y_0$. What does the output word equal for each of these inputs?

a. $X_4X_3 \cdots X_0 = 10011$
b. $X_4X_3 \cdots X_0 = 01110$
c. $X_4X_3 \cdots X_0 = 10101$
d. $X_4X_3 \cdots X_0 = 11100$

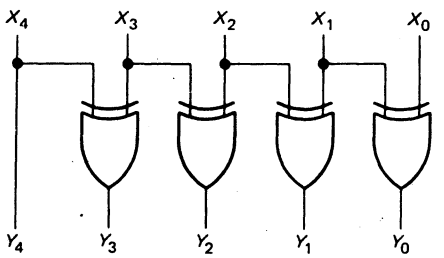


Fig. 3-36 Binary–to–Gray-code converter.

3-28. How many input words are there in the truth table of an 8-input XOR gate?

3-29. How can you modify Fig. 3-19 so that it produces an 8-bit output word with even parity?

3-30. In the controlled inverter of Fig. 3-21, what is the output word *Y* for each of these conditions?

a. **A** = 1100 1111 and *INVERT* = 0
b. **A** = 0101 0001 and *INVERT* = 1
c. **A** = 1110 1000 and *INVERT* = 1
d. **A** = 1010 0101 and *INVERT* = 0

3-31. The inputs *A* and *B* of Fig. 3-37 produce outputs of *CARRY* and *SUM*. What are the values of *CARRY* and *SUM* for each of these inputs?

a. *A* = 0 and *B* = 0
b. *A* = 0 and *B* = 1
c. *A* = 1 and *B* = 0
d. *A* = 1 and *B* = 1

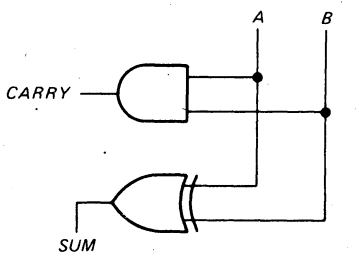


Fig. 3-37

3-32. In Fig. 3-37, what is the boolean equation for *CARRY*? For *SUM*?

3-33. What is the 1's complement for each of these numbers?

a. 1100 0011
b. 1010 1111 0011
c. 1110 0001 1010 0011
d. 0000 1111 0010 1101

3-34. What is the output of a 16-input XNOR gate for each of these input words?

a. 0000 0000 0000 1111
b. 1111 0101 1110 1100
c. 0101 1100 0001 0011
d. 1111 0000 1010 0110

TTL Circuits

4

In 1964 Texas Instruments introduced *transistor-transistor logic* (TTL), a widely used family of digital devices. TTL is fast, inexpensive, and easy to use. This chapter concentrates on TTL because once you are familiar with it, you can branch out to other logic families and technologies.

4-1 DIGITAL INTEGRATED CIRCUITS

Using advanced photographic techniques, a manufacturer can produce miniature circuits on the surface of a *chip* (a small piece of semiconductor material). The finished network is so small you need a microscope to see the connections. Such a circuit is called an *integrated circuit* (IC) because the components (transistors, diodes, resistors) are an integral part of the chip. This is different from a discrete circuit, in which the components are individually connected during assembly.

Levels of Integration

Small-scale integration (SSI) refers to ICs with fewer than 12 gates on the same chip. *Medium-scale integration* (MSI) means from 12 to 100 gates per chip. And *large-scale integration* (LSI) refers to more than 100 gates per chip. The typical microcomputer has its microprocessor, memory, and I/O circuits on LSI chips; a number of SSI and MSI chips are used to support the LSI chips.

Technologies and Families

The two basic technologies for manufacturing digital ICs are *bipolar* and *MOS*. The first fabricates bipolar transistors on a chip; the second, MOSFETS. Bipolar technology is preferred for SSI and MSI because it is faster. MOS technology dominates the LSI field because more MOSFETs can be packed on the same chip area.

A digital family is a group of compatible devices with the same logic levels and supply voltages ("compatible" means that you can connect the output of one device to the input of another). Compatibility permits a large number of different combinations.

Bipolar Families

In the bipolar category are these basic families:

DTL	Diode-transistor logic
TTL	Transistor-transistor logic
ECL	Emitter-coupled logic

DTL uses diodes and transistors; this design, once popular, is now obsolete. TTL uses transistors almost exclusively; it has become the most popular family of SSI and MSI chips. ECL, the fastest logic family, is used in high-speed applications.

MOS Families

In the MOS category are these families:

PMOS	p-Channel MOSFETs
NMOS	n-Channel MOSFETs
CMOS	Complementary MOSFETs

PMOS, the oldest and slowest type, is becoming obsolete. NMOS dominates the LSI field, being used for microprocessors and memories. CMOS, a push-pull arrangement of n- and p-channel MOSFETs, is extensively used where low power consumption is needed, as in pocket calculators, digital wristwatches, etc.

4-2 7400 DEVICES

The 7400 series, a line of TTL circuits introduced by Texas Instruments in 1964, has become the most widely used of all bipolar ICs. This TTL family contains a variety of SSI and MSI chips that allow you to build all kinds of digital circuits and systems.

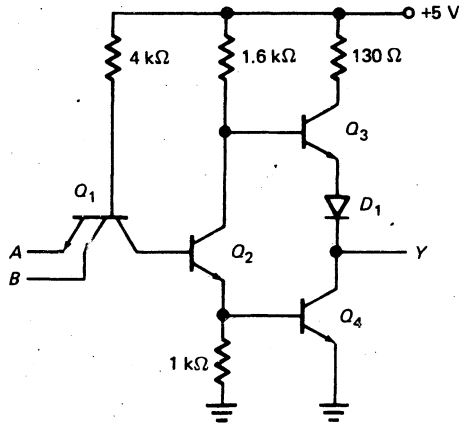


Fig. 4-1 Standard TTL NAND gate.

Standard TTL

Figure 4-1 shows a TTL NAND gate. The multiple-emitter input transistor is typical of all the gates and circuits in the 7400 series. Each emitter acts like a diode; therefore, Q_1 and the 4-kΩ resistor act like a 2-input AND gate. The rest of the circuit inverts the signal; therefore, the overall circuit acts like a 2-input NAND gate.

The output transistors (Q_3 and Q_4) form a totem-pole connection, typical of most TTL devices. Either one or the other is on. When Q_3 is on, the output is high; when Q_4 is on, the output is low. The advantage of a totem-pole connection is its low output impedance.

Ideally, the input voltages *A* and *B* are either low (grounded) or high (5 V). If *A* or *B* is low, Q_1 saturates. This reduces the base voltage of Q_2 to almost zero. Therefore, Q_2 cuts off, forcing Q_4 to cut off. Under these conditions, Q_3 acts like an emitter follower and couples a high voltage to the output.

On the other hand, when both *A* and *B* are high, the collector diode of Q_1 goes into forward conduction; this forces Q_2 and Q_4 into saturation, producing a low output. Table 4-1 summarizes all input and output conditions.

Incidentally, without diode D_1 in the circuit, Q_3 would conduct slightly when the output is low. To prevent this, the diode is inserted; its voltage drop keeps the base-emitter diode of Q_3 reverse-biased. In this way, only Q_4 conducts when the output is low.

TABLE 4-1. TWO-INPUT NAND GATE

A	*B*	*Y*
0	0	1
0	1	1
1	0	1
1	1	0

Totem-Pole Output

Why are totem-pole transistors used? Because they produce a low output impedance. Either Q_3 acts like an emitter follower (high output) or Q_4 is saturated (low output). Either way, the output impedance is very low. This is important because it reduces the switching speed. In other words, when the output changes from low to high, or vice versa, the low output impedance implies a short *RC* time constant; this short time constant means that the output voltage can change quickly from one state to the other.

Propagation Delay Time and Power Dissipation

Two quantities needed for our later discussions are power dissipation and propagation delay time. A standard TTL gate has a power dissipation of about 10 mW. It may vary from this value because of signal levels, tolerances, etc., but on the average, it's 10 mW per gate.

The propagation delay time is the amount of time it takes for the output of a gate to change after the inputs have changed. The propagation delay time of a TTL gate is in the vicinity of 10 ns.

Device Numbers

By varying the design of Fig. 4-1 manufacturers can alter the number of inputs and the logic function. The multiple-emitter inputs and the totem-pole outputs are still used, no matter what the design. (The only exception is an open collector, discussed later.)

Table 4-2 lists some of the 7400-series TTL gates. For instance, the 7400 is a chip with four 2-input NAND gates in one package. Similarly, the 7402 has four 2-input NOR gates, the 7404 has six inverters, and so on.

TABLE 4-2. STANDARD TTL

Device number	Description
7400	Quad 2-input NAND gates
7402	Quad 2-input NOR gates
7404	Hex inverter
7408	Quad 2-input AND gates
7410	Triple 3-input NAND gates
7411	Triple 3-input AND gates
7420	Dual 4-input NAND gates
7421	Dual 4-input AND gates
7427	Triple 3-input NOR gates
7430	8-input NAND gate
7486	Quad 2-input XOR gates

5400 Series

Any device in the 7400 series works over a temperature range of 0° to 70°C and over a supply range of 4.75 to 5.25 V. This is adequate for commercial applications. The 5400 series, developed for the military applications, has the same logic functions as the 7400 series, except that it works over a temperature range of −55 to 125°C and over a supply range of 4.5 to 5.5 V. Although 5400-series devices can replace 7400-series devices, they are rarely used commercially because of their much higher cost.

High-Speed TTL

The circuit of Fig. 4-1 is called *standard TTL*. By decreasing the resistances a manufacturer can lower the internal time constants; this decreases the propagation delay time. The smaller resistances, however, increase the power dissipation. This variation is known as *high-speed TTL*. Devices of this type are numbered 74H00, 74H01, 74H02, and so on. A high-speed TTL gate has a power dissipation around 22 mW and a propagation delay time of approximately 6 ns.

Low-Power TTL

By increasing the internal resistances a manufacturer can reduce the power dissipation of TTL gates. Devices of this type are called *low-power TTL* and are numbered 74L00, 74L01, 74L02, etc. These devices are slower than standard TTL because of the larger internal time constants. A low-power TTL gate has a power dissipation of approximately 1 mW and a propagation delay time around 35 ns.

Schottky TTL

With standard TTL, high-speed TTL, and low-power TTL, the transistors go into saturation causing extra carriers to flood the base. If you try to switch this transistor from saturation to cutoff, you have to wait for the extra carriers to flow out of the base; the delay is known as the *saturation delay time*.

One way to reduce saturation delay time is with Schottky TTL. The idea is to fabricate a Schottky diode along with each bipolar transistor of a TTL circuit, as shown in Fig. 4-2. Because the Schottky diode has a forward voltage of only 0.4 V, it prevents the transistor from saturating fully.

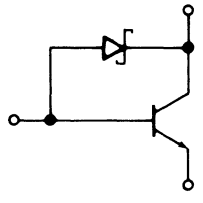


Fig. 4-2 Schottky diode prevents transistor saturation.

This virtually eliminates saturation delay time, which means better switching speed. This variation is called *Schottky TTL*; the devices are numbered 74S00, 74S01, 74S02, and so forth.

Schottky TTL devices are very fast, capable of operating reliably at 100 MHz. The 74S00 has a power dissipation around 20 mW per gate and a propagation delay time of approximately 3 ns.

Low-Power Schottky TTL

By increasing internal resistances as well as using Schottky diodes manufacturers have come up with the best compromise between low power and high speed: *low-power Schottky TTL*. Devices of this type are numbered 74LS00, 74LS01, 74LS02, etc. A low-power Schottky gate has a power dissipation of around 2 mW and a propagation delay time of approximately 10 ns, as shown in Table 4-3.

Standard TTL and low-power Schottky TTL are the mainstays of the digital designer. In other words, of the five TTL types listed in Table 4-3, standard TTL and low-power Schottky TTL have emerged as the favorites of the digital designers. You will see them used more than any other bipolar types.

4-3 TTL CHARACTERISTICS

7400-series devices are guaranteed to work reliably over a temperature range of 0 to 70°C and over a supply range of 4.75 to 5.25 V. In the discussion that follows, *worst case* means that the parameters (characteristics like maximum input current, minimum output voltage, and so on) are measured under the worst conditions of temperature and voltage—maximum temperature and minimum voltage for some parameters, minimum temperature and maximum voltage for others, or whatever combination produces the worst values.

Floating Inputs

When a TTL input is low or grounded, a current I_E (conventional direction) exists in the emitter, as shown in

TABLE 4-3. TTL POWER-DELAY VALUES

Type	Power, mW	Delay time, ns
Low-power	1	35
Low-power Schottky	2	10
Standard	10	10
High-speed	22	6
Schottky	20	3

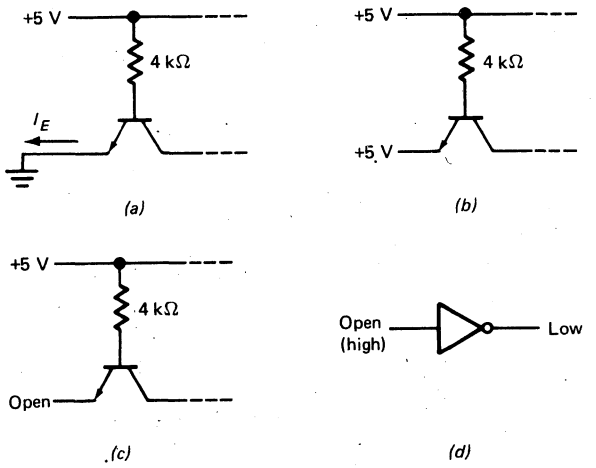


Fig. 4-3 Open or floating input is the same as a high input.

Fig. 4-3*a*. On the other hand, when a TTL input is high (Fig. 4-3*b*), the emitter diode cuts off and the emitter current is approximately zero.

When a TTL input is floating (unconnected), as shown in Fig. 4-3*c*, no emitter current is possible. Therefore, a floating TTL input is equivalent to a high input. In other words, Fig. 4-3*c* produces the same output as Fig. 4-3*b*. This is important to remember. In building circuits *any floating TTL input will act like a high input.*

Figure 4-3*d* emphasizes the point. The input is floating and is equivalent to a high input; therefore, the output of the inverter is low.

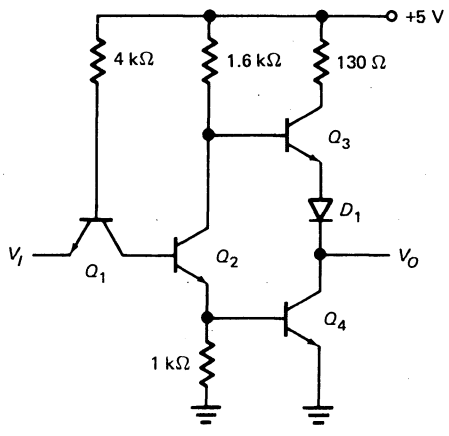


Fig. 4-4 TTL inverter.

Worst-Case Input Voltages

Figure 4-4 shows a TTL inverter with an input voltage of V_I and an output voltage of V_O. When V_I is 0 V (grounded), the output voltage is high. With TTL devices, we can raise V_I to 0.8 V and still have a high output. The maximum low-level input voltage is designated V_{IL}. Data sheets list this worst-case low input as

$$V_{IL} = 0.8 \text{ V}$$

Take the other extreme. Suppose V_I is 5 V in Fig. 4-4. This is a high input; therefore, the output of the inverter is low. V_I can decrease all the way down to 2 V, and the output will still be high. Data sheets list this worst-case high input as

$$V_{IH} = 2 \text{ V}$$

In other words, any input voltage from 2 to 5 V is a high input for TTL devices.

Worst-Case Output Voltages

Ideally, 0 V is the low output, and 5 V is the high output. We cannot attain these ideal values because of internal voltage drops. When the output is low in Fig. 4-4, Q_4 is saturated and has a small voltage drop across it. With TTL devices, any voltage from 0 to 0.4 V is a low output.

When the output is high, Q_3 acts like an emitter follower. Because of the drop across Q_3, D_1, and the 130-Ω resistor, the output is less than 5 V. With TTL devices, a high output is between 2.4 and 3.9 V, depending on the supply voltage, temperature, and load.

This means that the worst-case output values are

$$V_{OL} = 0.4 \text{ V} \qquad V_{OH} = 2.4 \text{ V}$$

Table 4-4 summarizes the worst-case values. Remember that they are valid over the temperature range (0 to 70°C) and supply range (4.75 to 5.25 V).

Compatibility

The values shown in Table 4-4 indicate that TTL devices are compatible. This means that the output of a TTL device can drive the input of another TTL device, as shown in Fig. 4-5*a*. To be specific, Fig. 4-5*b* shows a low TTL output (0 to 0.4 V). This is low enough to drive the second TTL device because any input less than 0.8 V is a low input.

TABLE 4-4. TTL STATES (WORST CASE)

	Output, V	Input, V
Low	0.4	0.8
High	2.4	2

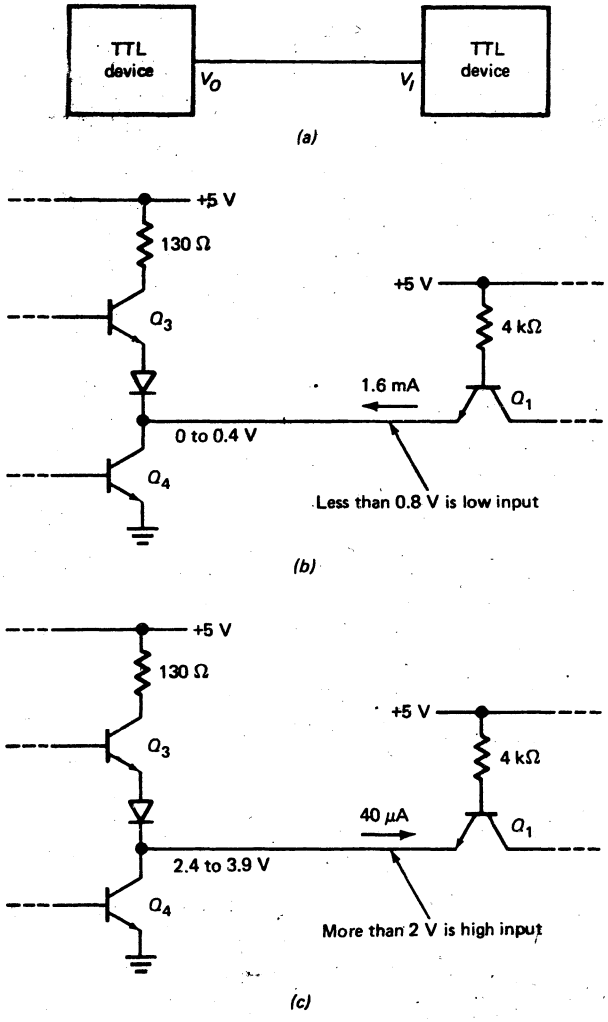


Fig. 4-5 Sourcing and sinking current.

Similarly, Fig. 4-5*c* shows a high TTL output (2.4 to 3.9 V). This is more than enough to drive the second TTL because any input greater than 2 V is a high input.

Noise Margin

In the worst case, there is a margin of 0.4 V between the driver and the load in Fig. 4-5*b* and *c*. This difference, called the *noise margin,* represents protection against noise. In other words, the connecting wire between a TTL driver and a TTL load may pick up stray noise voltages. As long as these induced voltages are less than 0.4 V, we get no false triggering of the TTL load.

Sourcing and Sinking

When a standard TTL output is low (Fig. 4-5*b*), an emitter current of approximately 1.6 mA (worst case) exists in the direction shown. The charges flow from the emitter of Q_1 to the collector of Q_4. Because it is saturated, Q_4 acts like a *current sink;* charges flow through it to ground like water flowing down a drain.

On the other hand, when a standard TTL output is high (Fig. 4-5*c*), a reverse emitter current of 40 μA (worst case) exists in the direction shown. Charges flow from Q_3 to the emitter of Q_1. In this case, Q_3 is acting like a *source.*

Data sheets lists the worst-case input currents as

$$I_{IL} = -1.6 \text{ mA} \quad I_{IH} = 40 \text{ μA}$$

The minus sign indicates that the current is out of the device; plus means the current is into the device. All data sheets use this convention.

Standard Loading

A TTL device can source current (high output) or it can sink current (low output). Data sheets of standard TTL devices indicate that any 7400-series device can sink up to 16 mA, designated as

$$I_{OL} = 16 \text{ mA}$$

and can source up to 400 μA, designated

$$I_{OH} = -400 \text{ μA}$$

(Again, a minus sign means that the current is out of the device and a plus sign means that it's into the device.)

A single TTL load has a low-level input current of 1.6 mA (Fig., 4-5*b*) and a high-level input current of 40 μA (Fig. 4-5*c*). Since the maximum output currents are 10 times as large, we can connect up to 10 TTL emitters to any TTL output.

Figure 4-6*a* illustrates a low output. Here you see the TTL driver sinking 16 mA, the sum of 10 TTL load currents. In this state, the output voltage is guaranteed to be 0.4 V or less. If you try connecting more than 10 emitters, the output voltage may rise above 0.4 V.

Figure 4-6*b* shows a high output with the driver sourcing 400 μA for 10 TTL loads of 40 μA each. For this maximum loading, the output voltage is guaranteed to be 2.4 V or more under worst-case conditions.

Loading Rules

The maximum number of TTL emitters that can be reliably driven under worst-case conditions is called the *fanout.* With standard TTL, the fanout is 10, as shown in Fig. 4-6. Sometimes, we may want to use a standard TTL device to drive low-power Schottky devices. In this case, the fanout increases because low-power Schottky devices have less input current.

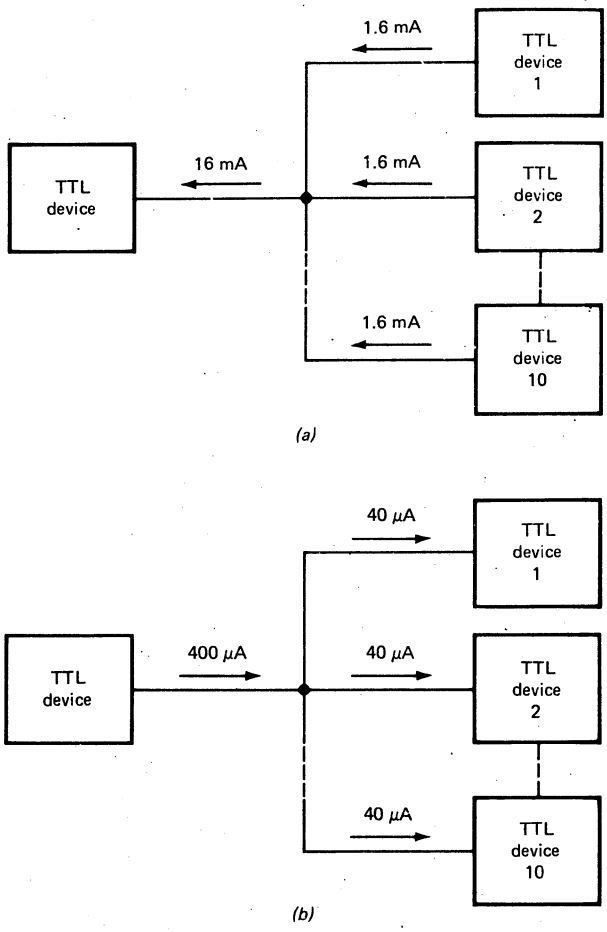


Fig. 4-6 Fanout of standard TTL devices: (*a*) low output; (*b*) high output.

By examining data sheets for the different TTL types we can calculate the fanout for all possible combinations. Table 4-5 summarizes these fanouts, which may be useful if you ever have to mix TTL types.

Read Table 4-5 as follows. The series numbers have been abbreviated; 74 stands for 7400 series, 74H for 74H00 series, and so forth. Drivers are on the left and loads on the right. Pick the driver, pick the load, and read the fanout at the intersection of the two. For instance, the fanout of a standard device (74) driving low-power Schottky devices (74LS) is 20. As another example, the fanout of a low-power device (74L) driving high-speed devices (74H) is only 1.

TABLE 4-5. FANOUTS

TTL driver	TTL load				
	74	74H	74L	74S	74LS
74	10	8	40	8	20
74H	12	10	50	10	25
74L	2	1	20	1	10
74S	12	10	100	10	50
74LS	5	4	40	4	20

4-4 TTL OVERVIEW

Let's take a look at the logic functions available in the 7400 series. This overview will give you an idea of the variety of gates and circuits found in the TTL family. As guide, Appendix 2 lists some of the 7400-series devices. You will find it useful when looking for a device number or logic function.

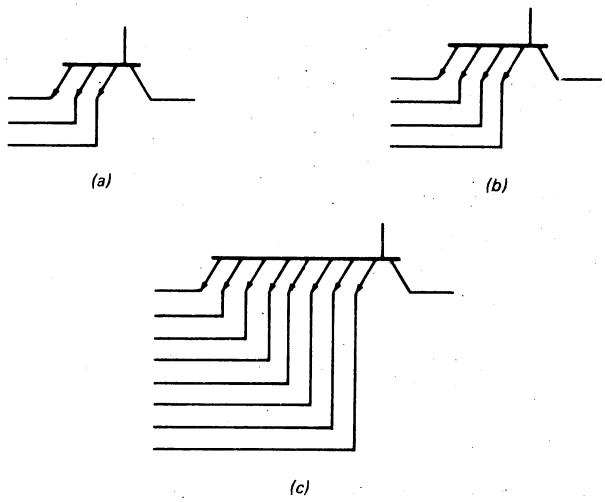


Fig. 4-7 Three, four, and eight inputs.

NAND Gates

To begin with, the NAND gate is the backbone of the entire series. All devices in the 7400 series are derived from the 2-input NAND gate shown in Fig. 4-1. To produce 3-, 4-, and 8-input NAND gates the manufacturer uses 3-, 4-, and 8-emitter transistors, as shown in Fig. 4-7. Because they are so basic, NAND gates are the least expensive devices in the 7400 series.

NOR Gates

To get other logic functions the manufacturer modifies the basic NAND-gate design. For instance, Fig. 4-8 shows a 2-input NOR gate. Q_1, Q_2, Q_3, and Q_4 are the same as in the basic design. Q_5 and Q_6 have been added to produce ORing. Notice that Q_2 and Q_6 are in parallel, the key to the ORing followed by inversion to get NORing.

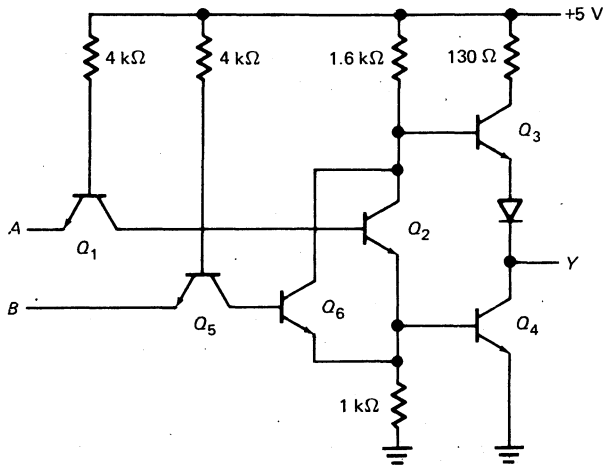


Fig. 4-8 TTL OR gate.

When A and B are both low, Q_1 and Q_5 are saturated; this cuts off Q_2 and Q_6. Then Q_3 acts like an emitter follower and we get a high output.

If A or B or both are high, Q_1 or Q_5 or both are cut off, forcing Q_2 or Q_6 or both to turn on. When this happens, Q_4 saturates and pulls the output down to a low voltage.

With more transistors, manufacturers can produce 3- and 4-input NOR gates. (A TTL 8-input NOR gate is not available.)

AND and OR Gates

To produce the AND function, another common-emitter stage is inserted before the totem-pole output of the basic NAND gate design. The extra inversion converts the NAND gate to an AND gate. Similarly, another CE stage can be inserted before the totem-pole output of Fig. 4-8; this converts the NOR gate to an OR gate.

Buffer-Drivers

A *buffer* is a device that isolates two other devices. Typically, a buffer has a high input impedance and a low output impedance. In terms of digital ICs, this means a low input current and a high output current.

Since the output current of a standard TTL gate can be 10 times the input current, a basic gate does a certain amount of buffering (isolating). But it's only when the manufacturer optimizes the design for high output currents that we call a device a buffer or driver.

As an example, the 7437 is a quad 2-input NAND buffer, meaning four 2-input NAND gates optimized to get high output currents. Each gate has the following worst-case values of input and output currents:

$$I_{IL} = -1.6\text{ mA} \qquad I_{IH} = 40\ \mu\text{A}$$
$$I_{OL} = 48\text{ mA} \qquad I_{OH} = -1.2\text{ mA}$$

The input currents are the same as those of a standard NAND gate, but the output currents are 3 times as high, which means that the 7437 can drive heavier loads.

Appendix 2 includes several other buffer-drivers.

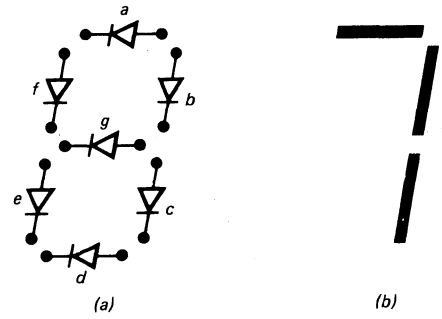


Fig. 4-9 Seven-segment display.

Encoders and Decoders

A number of TTL chips are available for encoding and decoding data. For instance, the 74147 is a decimal-to-BCD encoder. It has 10 input lines (decimal) and 4 output lines (BCD). As another example, the 74154 is a 1-of-16 decoder. It has 4 input lines (binary) and 16 output lines (hexadecimal).

Seven-segment decoders (7446, 7447, etc.) are useful for decimal displays. They convert a BCD nibble into an output that can drive a seven-segment display. Figure 4-9*a* illustrates the idea behind a seven-segment LED display. It has seven separate LEDs that allow you to display any digit between 0 and 9. To display a 7, the decoder will turn on LEDs *a*, *b*, and *c* (Fig. 4-9*b*).

Seven-segment displays are not limited to decimal numbers. For instance, in some microprocessor trainers, seven-segment displays are used to indicate hexadecimal digits. Digits A, C, E, and F are displayed in uppercase form; digit B is shown as a lowercase b (LEDs *c*, *d*, *e*, *f*, *g*); and digit D as a lowercase d (LEDs *b*, *c*, *d*, *e*, *g*).

Schmitt Triggers

When a computer is running, the outputs of gates are rapidly switching from one state to another. If you look at these signals with an oscilloscope, you see signals that ideally resemble rectangular waves like Fig. 4-10*a*.

When digital signals are transmitted and later received, they are often corrupted by noise, attenuation, or other factors and may wind up looking like the ragged waveform shown in Fig. 4-10*b*. If you try to use these nonrectangular signals to drive a gate or other digital device, you get unreliable operation.

This is where the *Schmitt trigger* comes in. It designed to clean up ragged looking pulses, producing almost vertical

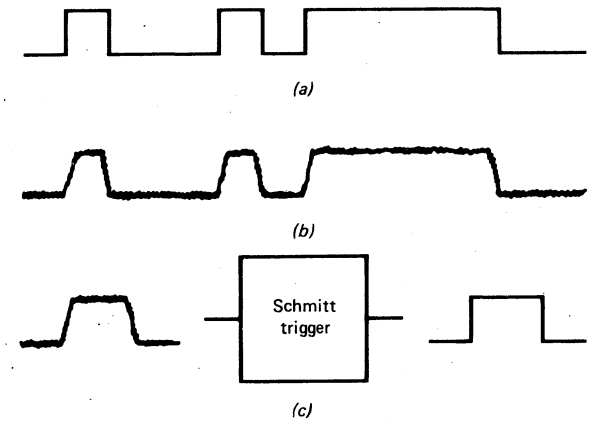


Fig. 4-10 Schmitt trigger produces rectangular output.

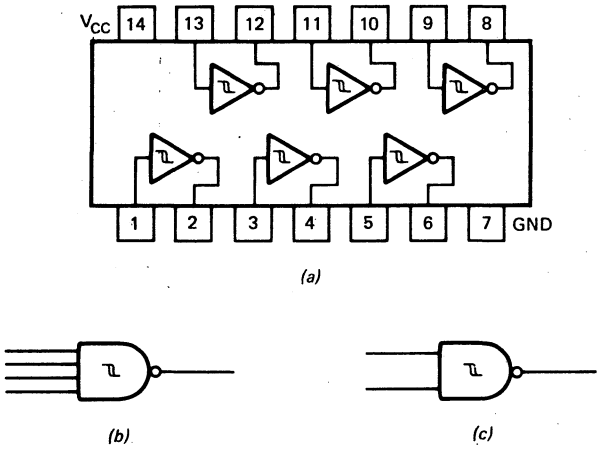


Fig. 4-11 (*a*) Hex Schmitt-trigger inverters; (*b*) 4-input NAND Schmitt trigger; (*c*) 2-input NAND Schmitt trigger.

transitions between the low and high state, and vice versa (Fig. 4-10*c*). In other words, the Schmitt trigger produces a rectangular output, regardless of the input waveform.

The 7414 is a hex Schmitt-trigger inverter, meaning six Schmitt-trigger inverters in one package like Fig. 4-11*a*. Notice the hysteresis symbol inside each inverter; it designates the Schmitt-trigger function.

Two other TTL Schmitt triggers are available. The 7413 is a dual 4-input NAND Schmitt trigger, two Schmitt-trigger gates like Fig. 4-11*b*. The 74132 is a quad 2-input NAND Schmitt trigger, four Schmitt-trigger gates like Fig. 4-11*c*.

Other Devices

The 7400 series also includes a number of other devices that you will find useful, such as AND-OR-INVERT gates (discussed in the next section), latches and flip-flops (Chap. 7), registers and counters (Chap. 8), and memories (Chap. 9).

4-5 AND-OR-INVERT GATES

Figure 4-12*a* shows an AND-OR circuit. Figure 4-12*b* shows the De Morgan equivalent circuit, a NAND-NAND network. In either case, the boolean equation is

$$Y = AB + CD \tag{4-1}$$

Since NAND gates are the preferred TTL gates, we would build the circuit of Fig. 4-12*b*. NAND-NAND circuits like this are important because with them you can build any desired logic circuit (discussed in Chap. 5).

TTL Devices

Is there any TTL device with the output given by Eq. 4-1? Yes, there are some AND-OR gates but they are not easily derived from the basic NAND-gate design. The gate that is easy to derive and comes close to having an expression like Eq. 4-1 is the AND-OR-INVERT gate shown in Fig. 4-12*c*. In other words, a variety of circuits like this are available on chips. Because of the inversion, the output has an equation of

$$Y = \overline{AB + CD} \tag{4-2}$$

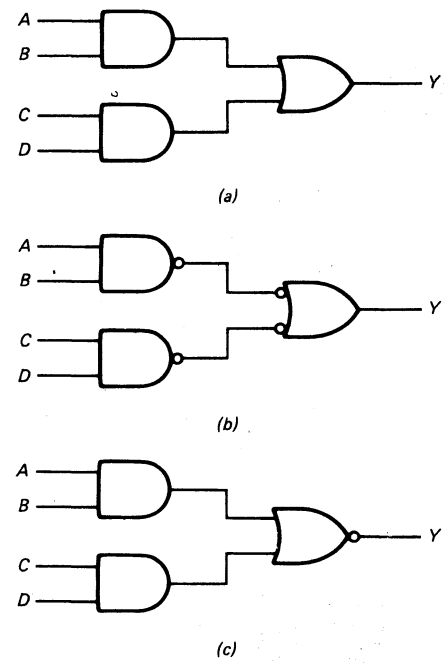


Fig. 4-12 (*a*) AND-OR circuit; (*b*) NAND-NAND circuit; (*c*) AND-OR-INVERT circuit.

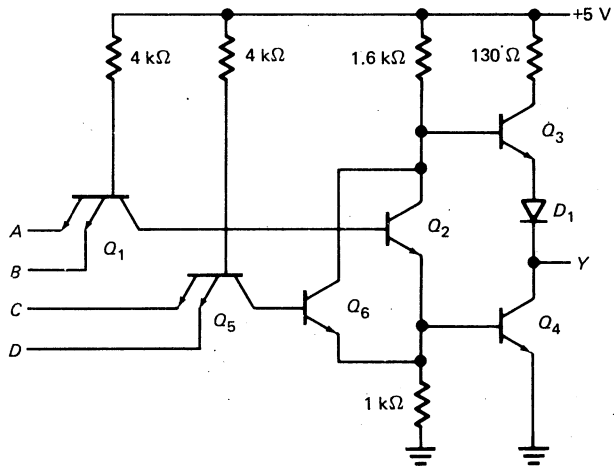


Fig. 4-13 AND-OR-INVERT schematic diagram.

Figure 4-13 shows the schematic diagram of a TTL AND-OR-INVERT gate. Q_1, Q_2, Q_3, and Q_4 form the basic 2-input NAND gate of the 7400 series. By adding Q_5 and Q_6 we convert the basic NAND gate to an AND-OR-INVERT gate.

Q_1 and Q_5 act like 2-input AND gates; Q_2 and Q_6 produce ORing and inversion. Because of this, the circuit is logically equivalent to Fig. 4-12*c*.

In Table 4-6, listing the AND-OR-INVERT gates available in the 7400 series, 2-wide means two AND gates across, 4-wide means four AND gates across, and so on. For instance, the 7454 is a 2-input 4-wide AND-OR-INVERT gate like Fig. 4-14*a*; each AND gate has two inputs (2-input) and there are four AND gates (4-wide). Figure 4-14*b* shows the 7464; it is a 2-2-3-4-input 4-wide AND-OR-INVERT gate.

When we want the output given by Eq. 4-1, we can connect the output of a 2-input 2-wide AND-OR-INVERT gate to another inverter. This cancels out the internal inversion, giving us the equivalent of an AND-OR circuit (Fig. 4-12*a*) or a NAND-NAND network (Fig. 4-12*b*).

Expandable AND-OR-INVERT Gates

The widest AND-OR-INVERT gate available in the 7400 series is 4-wide. What do we do when we need a 6- or 8-wide circuit? One solution is to use an *expandable* AND-OR-INVERT gate.

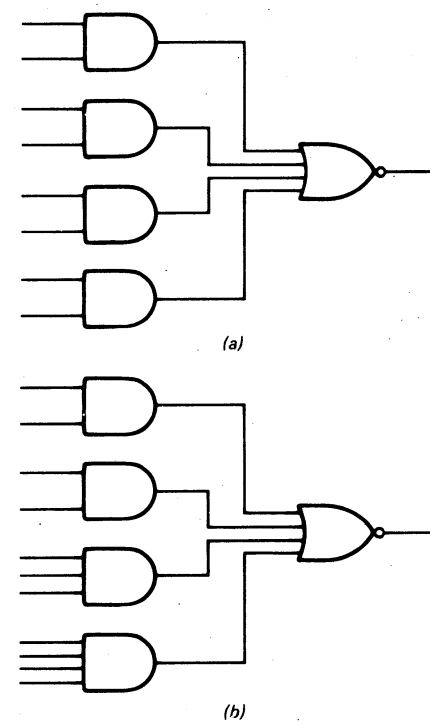


Fig. 4-14 Examples of AND-OR-INVERT circuits.

Figure 4-15*a* shows the schematic diagram of an expandable AND-OR-INVERT gate. The only difference between this and the preceding AND-OR-INVERT gate (Fig. 4-13) is collector and emitter tie points brought outside the package. Since Q_2 and Q_6 are the key to the ORing operation, we are being given access to the internal ORing function. By connecting other gates to these new inputs we can expand the width of the AND-OR-INVERT gate.

Figure 4-15*b* shows the logic symbol for an expandable AND-OR-INVERT gate. The arrow input represents the emitter, and the bubble stands for the collector. Table 4-7 lists the expandable AND-OR-INVERT gates in the 7400 series.

Expanders

What do we connect to the collector and emitter inputs of an expandable gate? The output of an *expander* like Fig. 4-16*a*. The input transistor acts like a 4-input AND gate. The output transistor is a phase splitter; it produces two

TABLE 4-6. AND-OR-INVERT GATES

Device	Description
7451	Dual 2-input 2-wide
7454	2-input 4-wide
7459	Dual 2-3 input 2-wide
7464	2-2-3-4 input 4-wide

TABLE 4-7. EXPANDABLE AND-OR-INVERT GATES

Device	Description
7450	Dual 2-input 2-wide
7453	2-input 4-wide
7455	4-input 2-wide

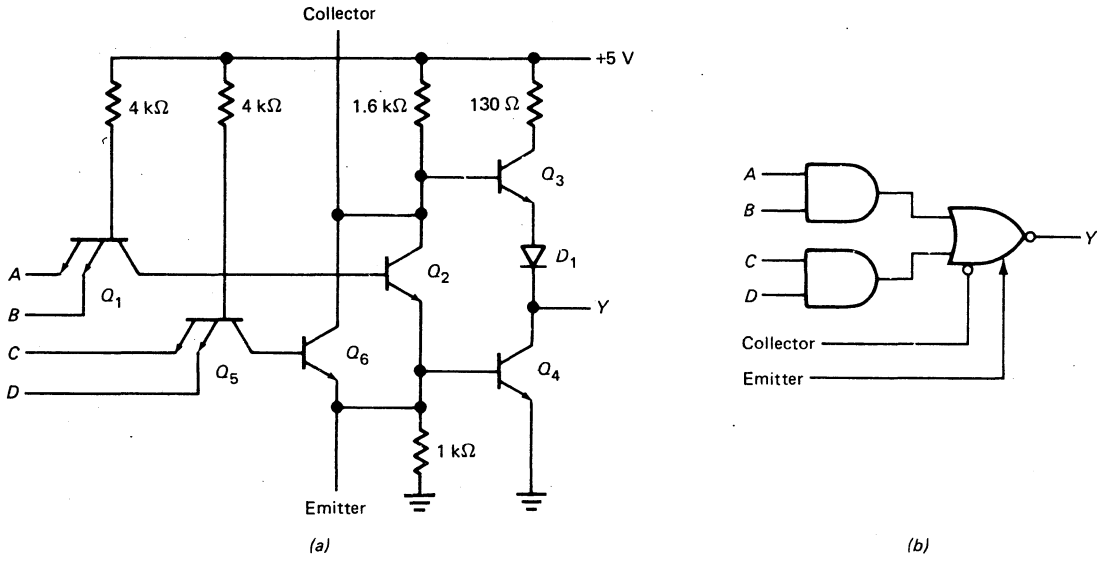


Fig. 4-15 (*a*) Expandable AND-OR-INVERT gate; (*b*) logic symbol.

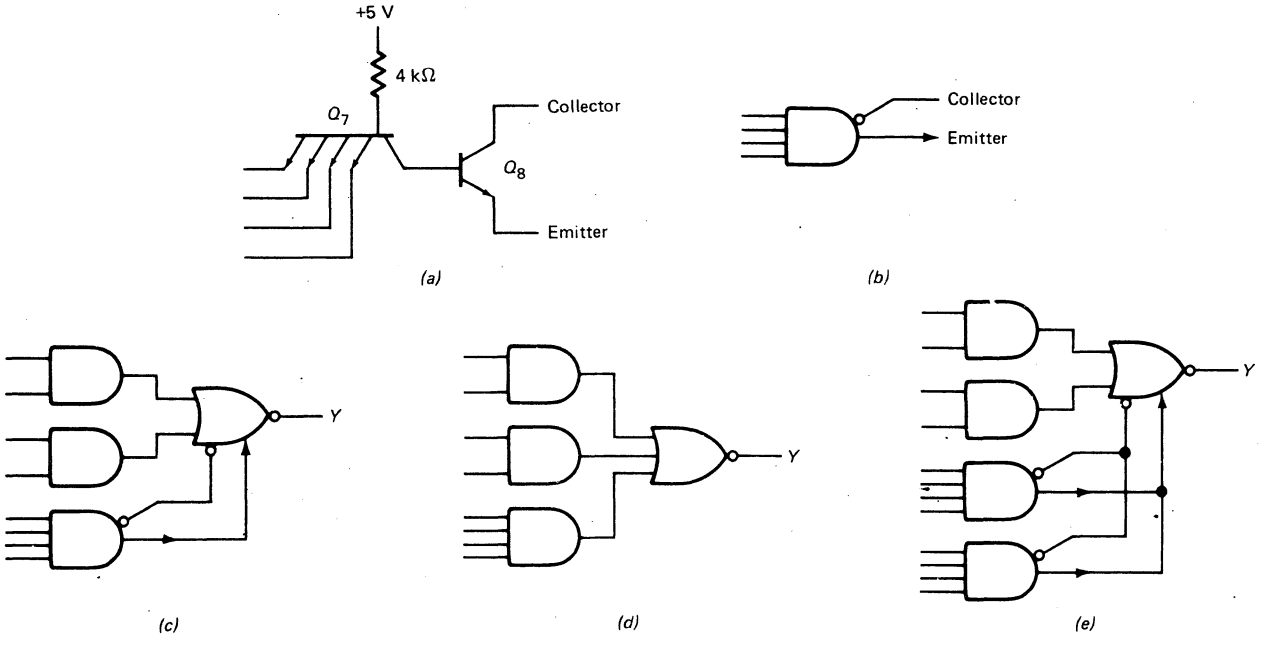


Fig. 4-16 (*a*) Expander; (*b*) symbol for expander; (*c*) expander driving expandable AND-OR-INVERT gate; (*d*) AND-OR-INVERT circuit; (*e*) expandable AND-OR-INVERT with two expanders.

output signals, one in phase (emitter) and the other inverted (collector). Figure 4-16*b* shows the symbol of a 4-input expander.

Visualize the outputs of Fig. 4-16*a* connected to the collector and emitter inputs of Fig. 4-15*a*. Then Q_8 is in parallel with Q_2 and Q_6. Figure 4-16*c* shows the logic circuit. This means that the expander outputs are being ORed with the signals of the AND-OR-INVERT gate. In other words, Fig. 4-16*c* is equivalent to the AND-OR-INVERT circuit of Fig. 4-16*d*.

We can connect more expanders. Figure 4-16*e* shows two expanders driving the expandable gate. Now we have a 2-2-4-4-input 4-wide AND-OR-INVERT circuit.

The 7460 is a dual 4-input expander. The 7450, a dual expandable AND-OR-INVERT gate, is designed for use with up to four 7460 expanders. This means that we can add two more expanders in Fig. 4-16*e* to get a 2-2-4-4-4-4-input 6-wide AND-OR-INVERT circuit.

4-6 OPEN-COLLECTOR GATES

Instead of a totem-pole output, some TTL devices have an *open-collector* output. This means they use only the lower transistor of a totem-pole pair. Figure 4-17*a* shows a 2-input NAND gate with an open-collector output. Because the collector of Q_4 is open, a gate like this won't work properly until you connect an external *pull-up* resistor, shown in Fig. 4-17*b*.

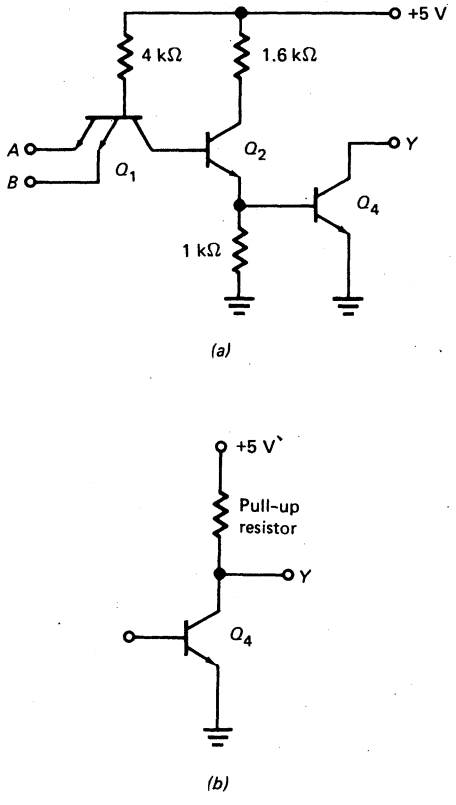


Fig. 4-17 Open-collector TTL: (*a*) circuit; (*b*) with pull-up resistor.

The outputs of open-collector gates can be wired together and connected to a common pull-up resistor. This is known as WIRE-OR. The big disadvantage of open-collector gates is their slow switching speed.

Open-collector gates are virtually obsolete because a new device called the *three-state switch* appeared in the early 1970s. Section 8-8 discusses three-state switches in detail.

4-7 MULTIPLEXERS

Multiplex means "many into one." A *multiplexer* is a circuit with many inputs but only one output. By applying control signals we can steer any input to the output.

Data Selection

Figure 4-18 shows a 16-to-1 multiplexer, also called a *data selector*. The input data bits are D_0 to D_{15}. Only one of these is transmitted to the output. Control word ABCD determines which data bit is passed to the output. For instance, when

$$\text{ABCD} = 0000$$

the upper AND gate is enabled but all other AND gates are disabled. Therefore, data bit D_0 is transmitted to the output, giving

$$Y = D_0$$

If the control word is changed to

$$\text{ABCD} = 1111$$

the bottom gate is enabled and all other gates are disabled. In this case,

$$Y = D_{15}$$

Boolean Function Generator

Digital design often starts with a truth table. The problem then is to come up with an equivalent logic circuit. Multiplexers give us a simple way to transform a truth table into an equivalent logic circuit. The idea is to use input data bits that are equal to the desired output bits of the truth table.

For example, look at the truth table of Table 4-8. When the input word ABCD is 0000, the output is 0; when ABCD

TABLE 4-8

A	*B*	*C*	*D*	*Y*
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

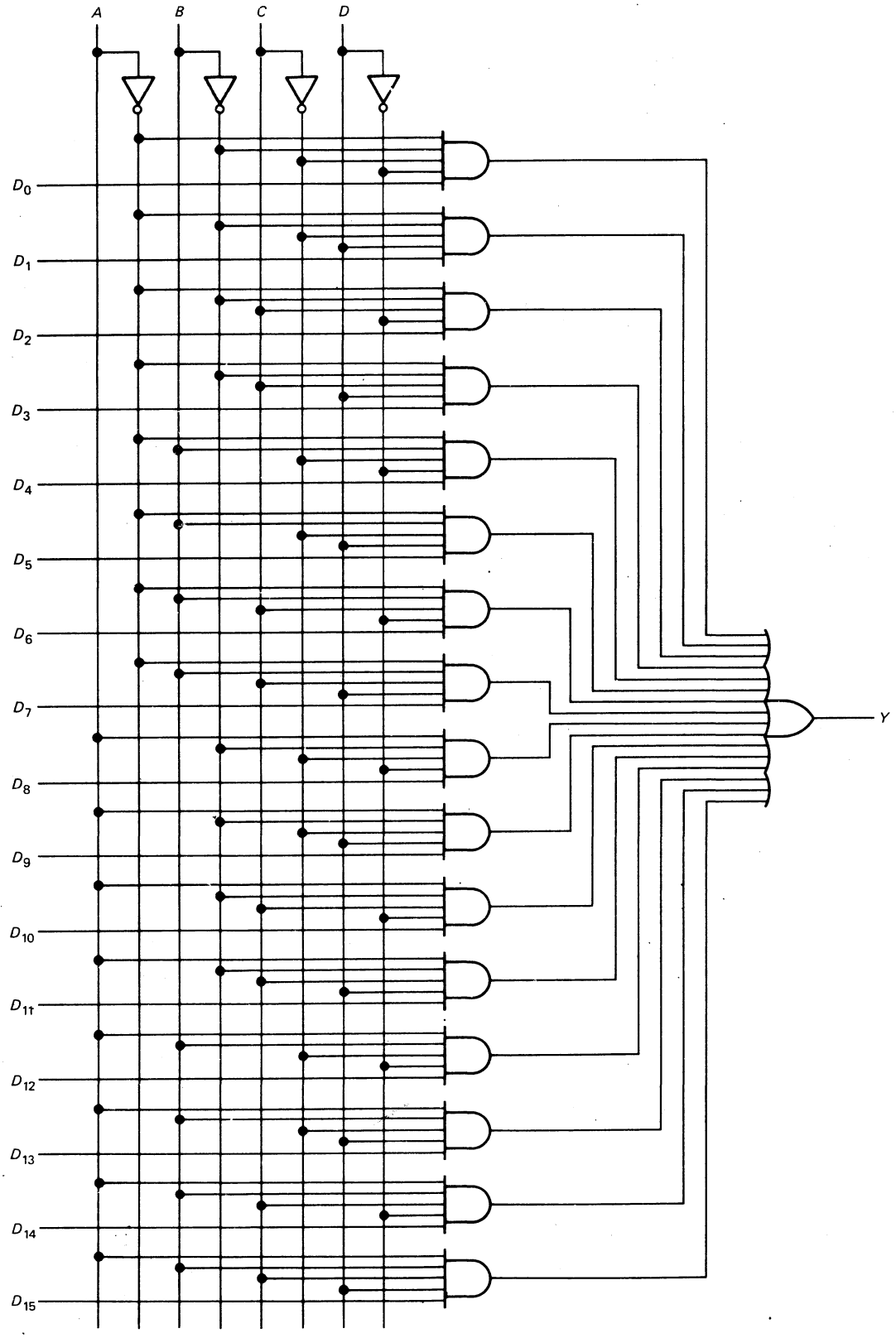


Fig. 4-18 A 16-to-1 multiplexer.

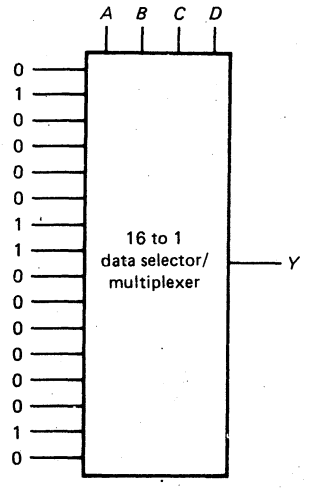


Fig. 4-19 Generating a boolean function.

= 0001, the output is 1; when ABCD = 0010, the output is 0; and so on. Figure 4-19 shows how to set up a multiplexer with the foregoing truth table. When ABCD = 0000, data bit 0 is steered to the output; when ABCD = 0001, data bit 1 is steered to the output; when ABCD = 0010, data bit 0 is steered to the output; and so forth. As a result, the truth table of this circuit is the same as Table 4-8.

Universal Logic Circuit

The 74150 is a 16-to-1 multiplexer. This TTL device is a universal logic circuit because you can use it to get the hardware equivalent of any four-variable truth table. In other words, by changing the input data bits the same IC can be made to generate thousands of different truth tables.

Multiplexing Words

Figure 4-20 illustrates a *word multiplexer* that has two input words and one output word. The input word on the left is $L_3L_2L_1L_0$ and the one on the right is $R_3R_2R_1R_0$. The control signal labeled *RIGHT* selects the input word that will be transmitted to the output. When *RIGHT* is low, the four NAND gates on the left are activated; therefore,

$$\mathbf{OUT} = L_3L_2L_1L_0$$

When *RIGHT* is high,

$$\mathbf{OUT} = R_3R_2R_1R_0$$

The 74157 is TTL multiplexer with an equivalent circuit like Fig. 4-20. Appendix 2 lists other multiplexers available in the 7400 series.

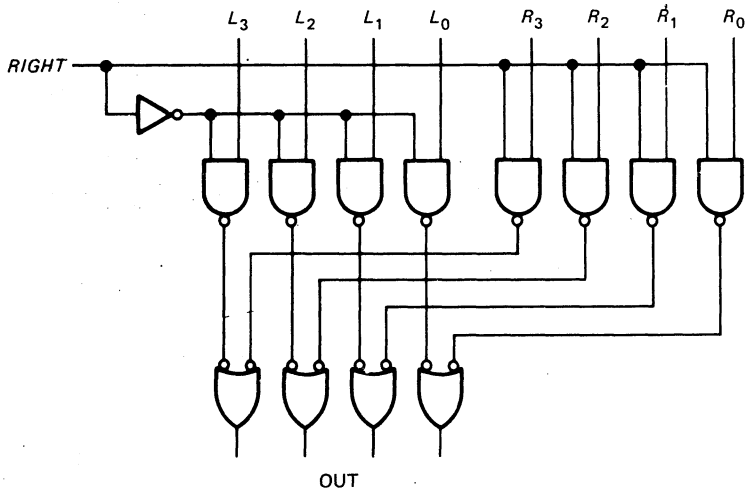


Fig. 4-20 Nibble multiplexer.

GLOSSARY

bipolar Having two types of charge carriers: free electrons and holes.

chip A small piece of semiconductor material. Sometimes, chip refers an IC device including its pins.

fanout The maximum number of TTL loads that a TTL device can drive reliably over the specified temperature range.

low-power Schottky TTL A modification of standard TTL

in which larger resistances and Schottky diodes are used. The increased resistances decrease the power dissipation, and the Schottky diodes increase the speed.

multiplexer A circuit with many inputs but only one output. Control signals select which input reaches the output.

noise margin The amount of noise voltage that causes unreliable operation. With TTL it is 0.4 V. As long as noise voltages induced on connecting lines are less than 0.4 V, the TTL devices will work reliably.

saturation delay time The time delay encountered when a transistor tries to come out of the saturation region. When the base drive switches from high to low, a transistor cannot instantaneously come out of saturation; extra carriers that flooded the base region must first flow out of the base.

Schmitt trigger A digital circuit that produces a rectangular output from any input large enough to drive the Schmitt trigger. The input waveform may be sinusoidal, triangular, distorted, and so on. The output is always rectangular.

sink A place where something is absorbed. When saturated, the lower transistor in a totem-pole output acts like a current sink because conventional charges flow through the transistor to ground.

source A place where something originates. The upper transistor of a totem-pole output acts like a source because charges flow out of its emitter into the load.

standard TTL The initial TTL design with resistance values that produce a power dissipation of 10 mW per gate and a propagation delay time of 10 ns.

SELF-TESTING REVIEW

Read each of the following and provide the missing words. Answers appear at the beginning of the next question.

1. Small-scale integration, abbreviated _____, refers to fewer than 12 gates on the same chip. Medium-scale integration (MSI) means 12 to 100 gates per chip. And large-scale integration (LSI) refers to more than _____ gates per chip.
2. (*SSI, 100*) The two basic technologies for digital ICs are bipolar and MOS. Bipolar technology is preferred for _____ and _____, whereas MOS technology is better suited to LSI. The reason MOS dominates the LSI field is that more _____ can be fabricated on the same chip area.
3. (*SSI, MSI, MOSFETs*) Some of the bipolar families include DTL, TTL, and ECL. _____ has become the most widely used bipolar family. _____ is the fastest logic family; it's used in high-speed applications.
4. (*TTL, ECL*) Some of the MOS families are PMOS, NMOS, and CMOS. _____ dominates the LSI field, and _____ is used extensively where lowest power consumption is necessary.
5. (*NMOS, CMOS*) The 7400 series, also called standard TTL, contains a variety of SSI and _____ chips that allow us to build all kinds of digital circuits and systems. Standard TTL has a multiple-emitter input transistor and a _____ output. The totem-pole output produces a low output impedance in either state.
6. (*MSI, totem-pole*) Besides standard TTL, there is high-speed TTL, low-power TTL, Schottky TTL, and low-power _____ TTL. Standard TTL and low-power _____ TTL have become the favorites of digital designers, used more than any other bipolar families.
7. (*Schottky, Schottky*) 7400-series devices are guaranteed to work reliably over a _____ range of 0 to 70°C and over a voltage range of 4.75 to 5.25 V. A floating TTL input has the same effect as a _____ input.
8. (*temperature, high*) A _____ TTL device can sink up to 16 mA and can source up to 400 μA. The maximum number of TTL loads a TTL device can drive is called the _____. With standard TTL, the fanout equals _____.
9. (*standard, fanout, 10*) A buffer is a device that isolates other devices. Typically, a buffer has a high input impedance and a _____ output impedance. In terms of digital ICs, this means a _____ input current and a high output current capability.
10. (*low, low*) A Schmitt trigger is a digital circuit that produces a _____ output regardless of the input waveform. It is used to clean up ragged looking pulses that have been distorted during transmission from one place to another.
11. (*rectangular*) A multiplexer is a circuit with many inputs but only one output. It is also called a data selector because data can be steered from one of the inputs to the output. A 74150 is a 16-to-1 multiplexer. With this TTL device you can implement the logic circuit for any four-variable truth table.

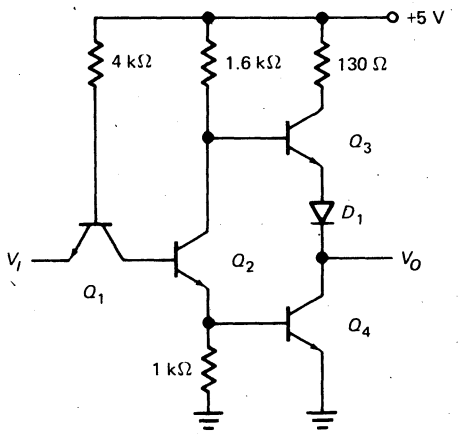


Fig. 4-21

4-1. In Fig. 4-21 a grounded input means that almost the entire supply voltage appears across the 4-kΩ resistor. Allowing 0.7 V for the emitter-base voltage of Q_1, how much input emitter current is there with a grounded input? The supply voltage can be as high as 5.25 V and the 4-kΩ resistance can be a low as 3.28 kΩ. What is the input emitter current in this case?

4-2. What is the fanout of a 74S00 device when it drives low-power TTL loads?

4-3. What is the fanout of a low-power Schottky device driving standard TTL devices?

4-4. Section 4-4 gave the input and output currents for a 7437 buffer. What is the fanout of a 7437 when it drives standard TTL loads?

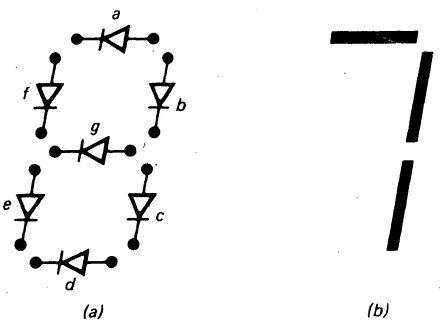
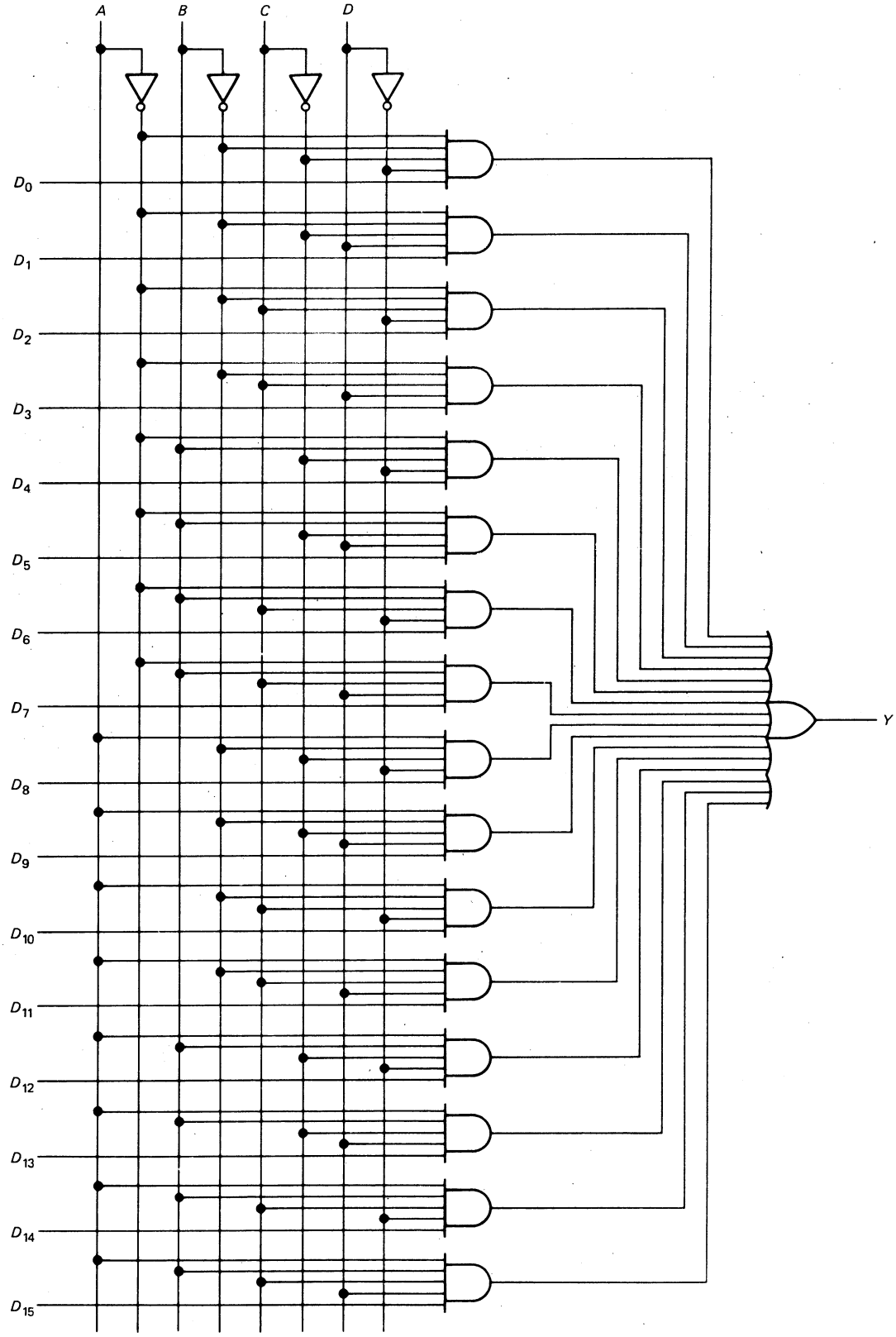


Fig. 4-22

4-5. A seven-segment decoder is driving a LED display like Fig. 4-22*a*. Which LEDs are on when digit 8 appears? Which LEDs are on when digit 4 appears?

4-6. Section 4-7 described the 74150, a 16-to-1 multiplexer. Refer to Fig. 4-23 and indicate the values the D_0 to D_{15} inputs of a 74150 should have to reproduce the following truth table: The output is high when ABCD = 0000, 0100, 0111, 1100, and 1111; the output is low for all other inputs.



ig. 4-23

Boolean Algebra and Karnaugh Maps

5

This chapter discusses boolean algebra and *Karnaugh maps*, topics needed by the digital designer. Digital design usually begins by specifying a desired output with a truth table. The question then is how to come up with a logic circuit that has the same truth table. Boolean algebra and Karnaugh maps are the tools used to transform a truth table into a practical logic circuit.

5-1 BOOLEAN RELATIONS

What follows is a discussion of basic relations in boolean algebra. Many of these relations are the same as in ordinary algebra, which makes remembering them easy.

Commutative, Associative, and Distributive Laws

Given a 2-input OR gate, you can transpose the input signals without changing the output (see Fig. 5-1*a*). In boolean terms

$$A + B = B + A \qquad (5\text{-}1)$$

Similarly, you can transpose the input signals to a 2-input AND gate without affecting the output (Fig. 5-1*b*). The boolean equivalent of this is

$$AB = BA \qquad (5\text{-}2)$$

The foregoing relations are called *commutative laws*.

The next group of rules are called the *associative laws*. The associative law for ORing is

$$A + (B + C) = (A + B) + C \qquad (5\text{-}3)$$

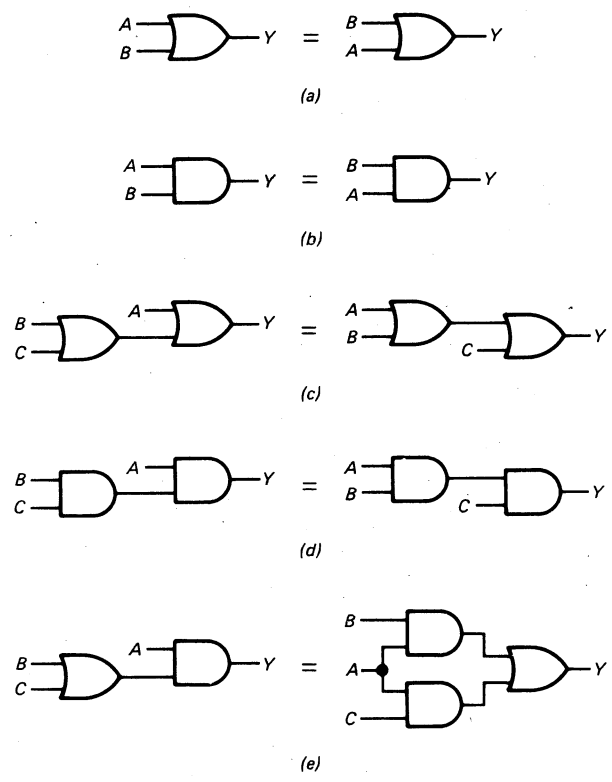


Fig. 5-1 Commutative, associative, and distributive laws.

Figure 5-1*c* illustrates this rule. The idea is that how you group variables in an ORing operation has no effect on the output. For either gate in Fig. 5-1*c* the output is

$$Y = A + B + C$$

Similarly, the associative law for ANDing is

$$A(BC) = (AB)C \tag{5-4}$$

Figure 5-1*d* illustrates this rule. How you group variables in ANDing operations has no effect on the output. For either gate of Fig. 5-1*d* the output is

$$Y = ABC$$

The *distributive law* states that

$$A(B + C) = AB + AC \tag{5-5}$$

This is easy to remember because it's identical to ordinary algebra. Figure 5-1*e* shows the meaning in terms of gates.

OR Operations

The next four boolean relations are about OR operations. Here is the first:

$$A + 0 = A \tag{5-6}$$

This says that a variable ORed with 0 equals the variable. For better grasp of this idea, look at Fig. 5-2*a*. (The solid arrow stands for "implies.") The two cases on the left imply the case on the right. In other words, if the variable is 0, the output is 0 (left gate); if the variable is 1, the output is 1 (middle gate); therefore, a variable ORed with 0 equals the variable (right gate).

Another boolean relation is

$$A + A = A \tag{5-7}$$

which is illustrated in Fig. 5-2*b*. You can see what happens. If A is 0, the output is 0; if A is 1, the output is 1; therefore, a variable ORed with itself equals the variable.

Figure 5-2*c* shows the next boolean rule:

$$A + 1 = 1 \tag{5-8}$$

In a nutshell, if one input to an OR gate is 1, the output is 1 regardless of the other input.

Finally, we have

$$A + \overline{A} = 1 \tag{5-9}$$

shown in Fig. 5-2*d*. In this case, a variable ORed with its complement equals 1.

AND Operations

The first AND relation to know about is

$$A \cdot 1 = A \tag{5-10}$$

illustrated in Fig. 5-3*a*. If A is 0, the output is 0; if A is 1, the output is 1; therefore, a variable ANDed with 1 equals the variable.

Another relation is

$$A \cdot A = A \tag{5-11}$$

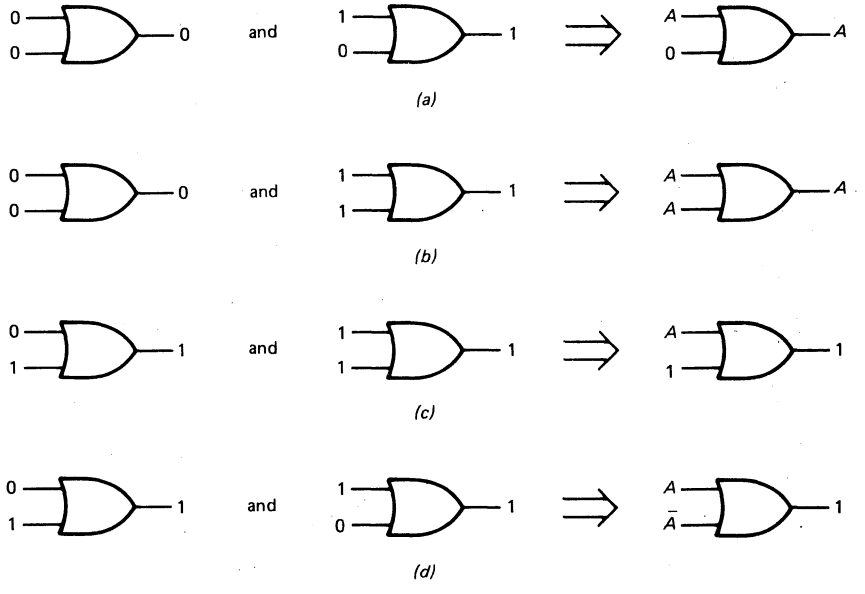


Fig. 5-2 OR relations.

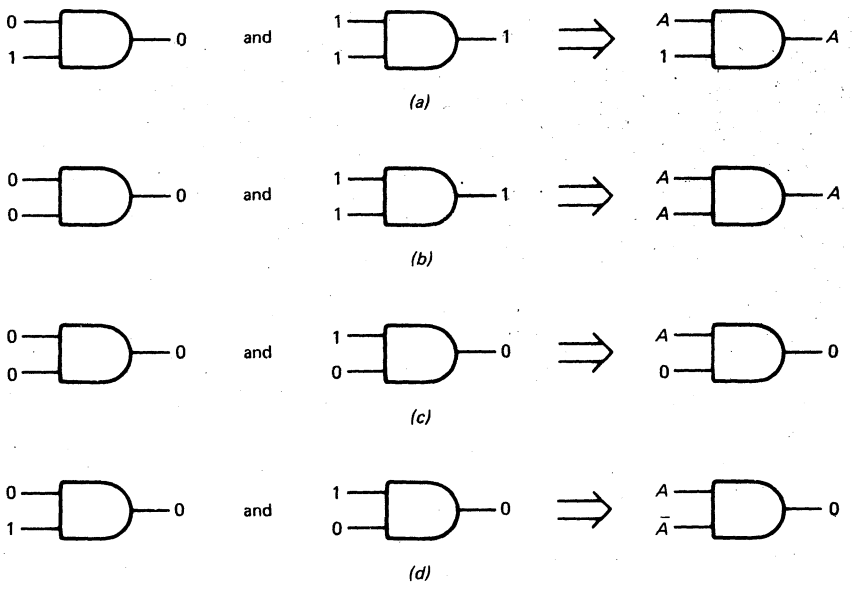


Fig. 5-3 AND relations.

shown in Fig. 5-3*b*. In this case, a variable ANDed with itself equals the variable.

Figure 5-3*c* illustrates this relation

$$A \cdot 0 = 0 \tag{5-12}$$

The rule is clear. If one input to an AND gate is 0, the output is 0 regardless of the other input.

The last AND rule is

$$A \cdot \overline{A} = 0 \tag{5-13}$$

As shown in Fig. 5-3*d*, a variable ANDed with its complement produces a 0 output.

Double Inversion and De Morgan's Theorems

The double-inversion rule is

$$\overline{\overline{A}} = A \tag{5-14}$$

which says that the double complement of a variable equals the variable. Finally, there are the De Morgan theorems discussed in Chap. 3:

$$\overline{A + B} = \overline{A}\,\overline{B} \tag{5-15}$$

$$\overline{AB} = \overline{A} + \overline{B} \tag{5-16}$$

You should memorize Eqs. 5-1 to 5-16 because they are used frequently in design work.

Duality Theorem

We state the *duality theorem* without proof. Starting with a boolean relation, you can derive another boolean relation by

1. Changing each OR sign to an AND sign
2. Changing each AND sign to an OR sign
3. Complementing each 0 and 1

For instance, Eq. 5-6 says that

$$A + 0 = A$$

The dual relation is

$$A \cdot 1 = A$$

This is obtained by changing the OR sign to an AND sign, and by complementing the 0 to get a 1.

The duality theorem is useful because it sometimes produces a new boolean relation. For example, Eq. 5-5 states that

$$A(B + C) = AB + AC$$

By changing each OR and AND operation we get the dual relation

$$A + BC = (A + B)(A + C)$$

This is a new boolean relation, not previously discussed. (If you want to prove it, construct the truth table for the

left and right members of the equation. The two truth tables will be identical.)

Summary

For future reference, here are some boolean relations and their duals:

$A + B = B + A$	$AB = BA$
$A + (B + C) = (A + B) + C$	$A(BC) = (AB)C$
$A(B + C) = AB + AC$	$A + BC = (A + B)(A + C)$
$A + 0 = A$	$A \cdot 1 = A$
$A + 1 = 1$	$A \cdot 0 = 0$
$A + A = A$	$AA = A$
$A + \overline{A} = 1$	$A\overline{A} = 0$
$\overline{\overline{A}} = A$	$\overline{\overline{A}} = A$
$\overline{A + B} = \overline{A}\,\overline{B}$	$\overline{AB} = \overline{A} + \overline{B}$
$A + AB = A$	$A(A + B) = A$
$A + \overline{A}B = A + B$	$A(\overline{A} + B) = AB$

5-2 SUM-OF-PRODUCTS METHOD

Digital design often starts by constructing a truth table with a desired output (0 or 1) for each input condition. Once you have this truth table, you transform it into an equivalent logic circuit. This section discusses the *sum-of-products method*, a way of deriving a logic circuit from a truth table.

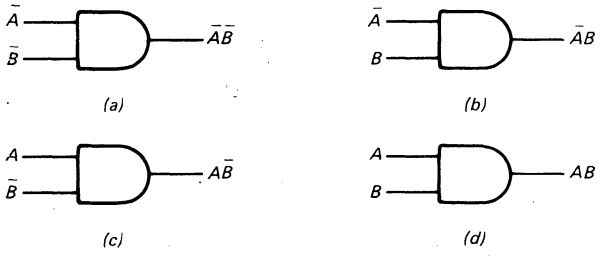


Fig. 5-4 Fundamental products.

Fundamental Products

Figure 5-4 shows the four possible ways to AND two input signals and their complements. In Fig. 5-4*a* the inputs are $\overline{A}$ and $\overline{B}$. Therefore, the output is

$$Y = \overline{A}\,\overline{B}$$

The output is high only when $A = 0$ and $B = 0$.

Figure 5-4*b* shows another possibility. Here the inputs are $\overline{A}$ and B; so the output is

$$Y = \overline{A}B$$

TABLE 5-1. TWO VARIABLES

A	B	Fundamental product
0	0	$\overline{A}\,\overline{B}$
0	1	$\overline{A}B$
1	0	$A\overline{B}$
1	1	AB

In this case, the output is 1 only when $A = 0$ and $B = 1$. In Fig. 5-4*c* the inputs are A and $\overline{B}$. The output

$$Y = A\overline{B}$$

is high only when $A = 1$ and $B = 0$. Finally, in Fig. 5-4*d* the inputs are A and B. The output

$$Y = AB$$

is 1 only when $A = 1$ and $B = 1$.

Table 5-1 summarizes the four possible ways to AND two signals in complemented or uncomplemented form. The logical products $\overline{A}\,\overline{B}$, $\overline{A}B$, $A\overline{B}$, and AB are called *fundamental products* because each produces a high output for its corresponding input. For instance, $\overline{A}\,\overline{B}$ is a 1 when A is 0 and B is 0, $\overline{A}B$ is a 1 when A is 0 and B is 1, and so forth.

Three Variables

A similar idea applies to three signals in complemented and uncomplemented form. Given A, B, C, and their complements, there are eight fundamental products: $\overline{A}\,\overline{B}\,\overline{C}$, $\overline{A}\,\overline{B}C$, $\overline{A}B\overline{C}$, $\overline{A}BC$, $A\overline{B}\,\overline{C}$, $A\overline{B}C$, $AB\overline{C}$, and ABC. Table 5-2 lists each input possibility and its fundamental product. Again notice this property: each fundamental product is high for the corresponding input. This means that $\overline{A}\,\overline{B}\,\overline{C}$ is a 1 when A is 0, B is 0, and C is 0; $\overline{A}\,\overline{B}C$ is a 1 when A is 0, B is 0, and C is 1; and so on.

TABLE 5-2. THREE VARIABLES

A	B	C	Fundamental product
0	0	0	$\overline{A}\,\overline{B}\,\overline{C}$
0	0	1	$\overline{A}\,\overline{B}C$
0	1	0	$\overline{A}B\overline{C}$
0	1	1	$\overline{A}BC$
1	0	0	$A\overline{B}\,\overline{C}$
1	0	1	$A\overline{B}C$
1	1	0	$AB\overline{C}$
1	1	1	ABC

Four Variables

When there are 4 input variables, there are 16 possible input conditions, 0000 to 1111. The corresponding fundamental products are from $\bar{A}\bar{B}\bar{C}\bar{D}$ through $ABCD$. Here is a quick way to find the fundamental product for any input condition. Whenever the input variable is 0, the same variable is complemented in the fundamental product. For instance, if the input condition is 0110, the fundamental product is $\bar{A}BC\bar{D}$. Similarly, if the input is 0100, the fundamental product is $\bar{A}B\bar{C}\bar{D}$.

Deriving a Logic Circuit

To get from a truth table to an equivalent logic circuit OR the fundamental products for each input condition that produces a high output. For example, suppose you have a truth table like Table 5-3. The fundamental products are listed for each high output. By ORing these products you get the boolean equation

$$Y = \bar{A}B\bar{C} + A\bar{B}C + AB\bar{C} + ABC \qquad (5\text{-}17)$$

This equation implies four AND gates driving an OR gate. The first AND gate has inputs of $\bar{A}$, B, and $\bar{C}$; the second AND gate has inputs of A, $\bar{B}$, and C; the third AND gate has inputs of A, B, and $\bar{C}$; the fourth AND gate has inputs of A, B, and C. Figure 5-5 shows the corresponding logic circuit. This AND-OR circuit has the same truth table as Table 5-3.

As another example of the sum-of-products method, look at Table 5-4. Find each output 1 and write its fundamental product. The resulting products are $\bar{A}\bar{B}CD$, $\bar{A}BCD$, and $A\bar{B}\bar{C}D$. This means that the boolean equation is

$$Y = \bar{A}\bar{B}CD + \bar{A}BCD + A\bar{B}\bar{C}D \qquad (5\text{-}18)$$

This equation implies that three AND gates are driving an OR gate. The first AND gate has inputs of $\bar{A}$, $\bar{B}$, C, and D; the second has inputs of $\bar{A}$, B, C, and D; the third has inputs of A, $\bar{B}$, $\bar{C}$, and D. Figure 5-6 is the equivalent logic circuit.

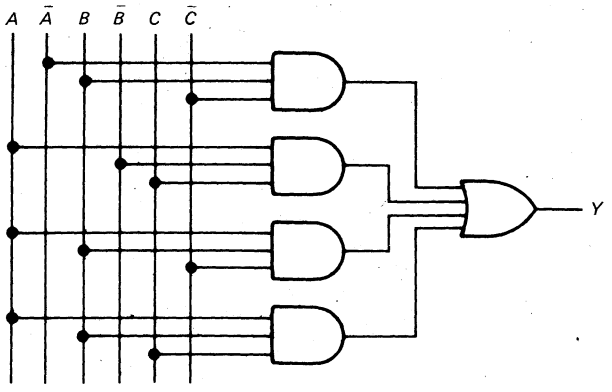


Fig. 5-5 Sum-of-products circuit.

TABLE 5-3

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	$1 \rightarrow \bar{A}B\bar{C}$
0	1	1	0
1	0	0	0
1	0	1	$1 \rightarrow A\bar{B}C$
1	1	0	$1 \rightarrow AB\bar{C}$
1	1	1	$1 \rightarrow ABC$

TABLE 5-4

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

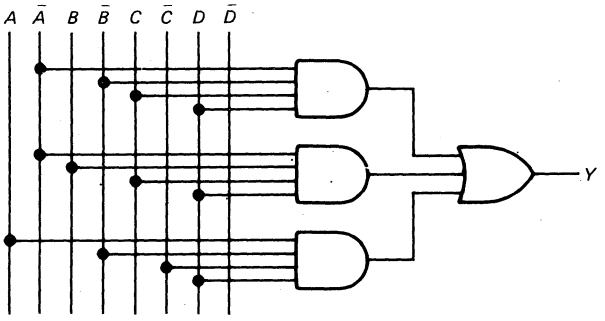


Fig. 5-6

The sum-of-products method always works. You OR the fundamental products of each high output in the truth table. This gives an equation which you can transform into an AND-OR network that is the circuit equivalent of the truth table.

5-3 ALGEBRAIC SIMPLIFICATION

After obtaining a sum-of-products equation as described in the preceding section, the thing to do is to simplify the circuit if possible. One way to do this is with boolean algebra. Here is the approach. Starting with the boolean equation for the sum-of-products circuit, you try to rearrange and simplify the equation as much as possible using the boolean rules of Sec. 5-1. The simplified boolean equation means a simpler logic circuit. This section will give you examples.

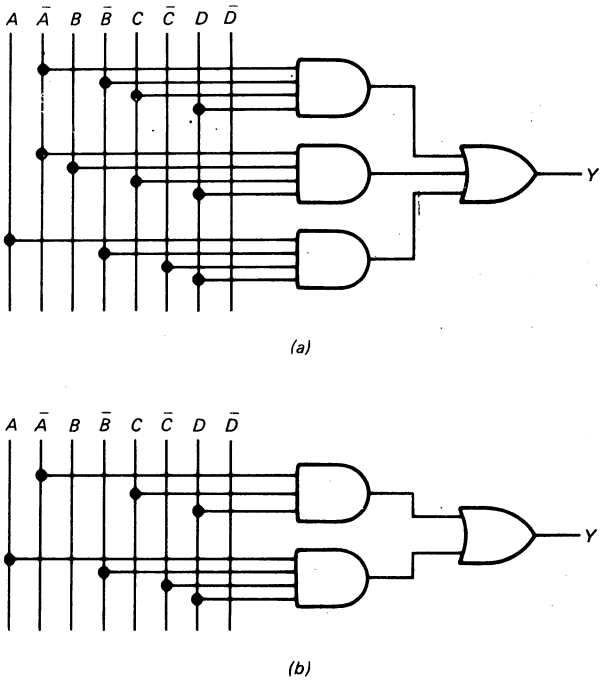


Fig. 5-7

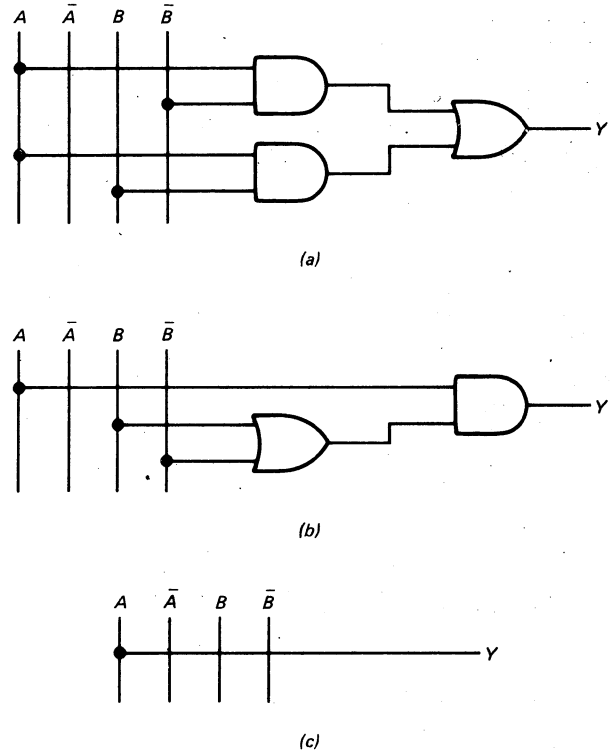


Fig. 5-8

Gate Leads

A preliminary guide for comparing the simplicity of one logic circuit with another is to count the number of *input gate leads;* the circuit with fewer input gate leads is usually easier to build. For instance, the AND-OR circuit of Fig. 5-7*a* has a total of 15 input gate leads (4 on each AND gate and 3 on the OR gate). The AND-OR circuit of Fig. 5-7*b*, on the other hand, has a total of 9 input gate leads. The AND-OR circuit of Fig. 5-7*b* is simpler than the AND-OR circuit of Fig. 5-7*a* because it has fewer input gate leads.

A *bus* is a group of wires carrying digital signals. The 8-bit bus of Fig. 5-7*a* transmits variables A, B, C, D and their complements $\overline{A}$, $\overline{B}$, $\overline{C}$, and $\overline{D}$. In the typical microcomputer, the microprocessor, memory, and I/O units exchange data by means of buses.

Factoring to Simplify

One way to reduce the number of input gate leads is to factor the boolean equation if possible. For instance, the boolean equation

$$Y = A\overline{B} + AB \qquad (5\text{-}19)$$

has the equivalent logic circuit shown in Fig. 5-8*a*. This circuit has six input gate leads. By factoring Eq. 5-19 we get

$$Y = A(\overline{B} + B)$$

The equivalent logic circuit for this is shown in Fig. 5-8*b*; it has only four input gate leads.

Recall that a variable ORed with its complement always equals 1; therefore,

$$Y = A(\overline{B} + B) = A \cdot 1 = A$$

To get this output, all we need is a connecting wire from the input to the output, as shown in Fig. 5-8*c*. In other words, we don't need any gates at all.

Another Example

Here is another example of how factoring can simplify a boolean equation and its corresponding logic circuit. Suppose we are given

$$Y = AB + AC + BD + CD \qquad (5\text{-}20)$$

In this equation, two variables at a time are being ANDed. The logical products are then ORed to get the final output. Figure 5-9*a* shows the corresponding logic circuit. It has 12 input gate leads.

We can factor and rearrange Eq. 5-20 as

$$Y = A(B + C) + D(B + C)$$

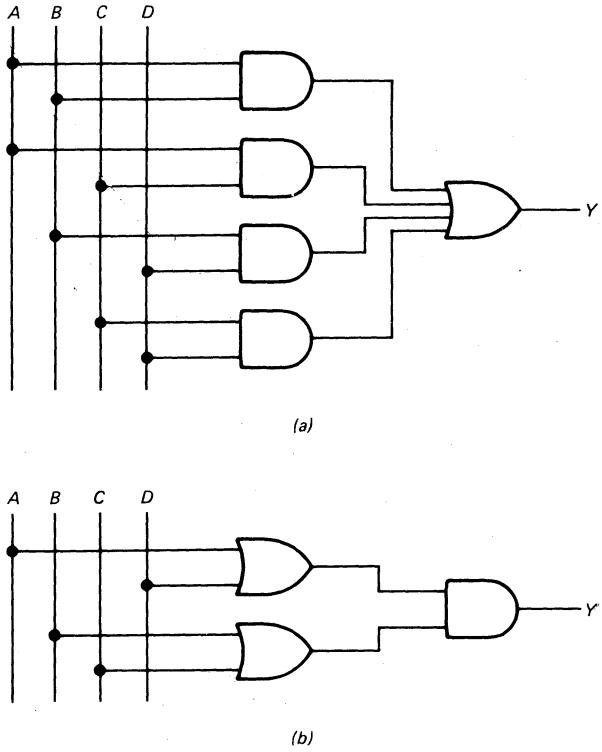


Fig. 5-9

or as

$$Y = (A + D)(B + C) \qquad (5\text{-}21)$$

In this case, the variables are first ORed, then the logical sums are ANDed. Figure 5-9*b* illustrates the logic circuit. Notice it has only six input gate leads and is simpler than the circuit of Fig. 5-9*a*.

Final Example

In Sec. 5-2 we derived this sum-of-products equation from a truth table:

$$Y = \overline{A}\overline{B}CD + \overline{A}BCD + A\overline{B}\overline{C}D \qquad (5\text{-}22)$$

Figure 5-7*a* shows the sum-of-products circuit. It has 15 input gate leads. We can factor the equation as

$$Y = \overline{A}CD(\overline{B} + B) + A\overline{B}\overline{C}D$$

or as

$$Y = \overline{A}CD + A\overline{B}\overline{C}D \qquad (5\text{-}23)$$

Figure 5-7*b* shows the equivalent logic circuit; it has only nine input gate leads.

In general, one approach in digital design is to transform a truth table into a sum-of-products equation, which you then simplify as much as possible to get a practical logic circuit.

5-4 KARNAUGH MAPS

Many engineers and technicians don't simplify equations with boolean algebra. Instead, they use a method based on *Karnaugh maps*. This section tells you how to construct a Karnaugh map.

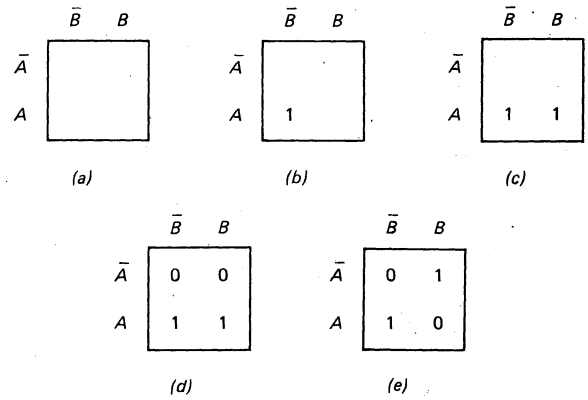


Fig. 5-10 Two-variable Karnaugh map.

Two-Variable Map

Suppose you have a truth table like Table 5-5. Here's how to construct the Karnaugh map. Begin by drawing Fig. 5-10*a*. Note the order of the variables and their complements; the vertical column has $\overline{A}$ followed by A, and the horizontal row has $\overline{B}$ followed by B.

Next, look for output 1s in Table 5-5. The first 1 output to appear is for the input of $A = 1$ and $B = 0$. The fundamental product for this is $A\overline{B}$. Now, enter a 1 on the Karnaugh map as shown in Fig. 5-10*b*. This 1 represents the product $A\overline{B}$ because the 1 is in the A row and the $\overline{B}$ column.

Similarly, Table 5-5 has an output 1 appearing for an input of $A = 1$ and $B = 1$. The fundamental product for this is AB. When you enter a 1 on the Karnaugh map to represent AB, you get the map of Fig. 5-10*c*.

The final step in the construction of the Karnaugh map is to enter 0s in the remaining spaces. Figure 5-10*d* shows how the Karnaugh map looks in its final form.

Here's another example of a two-variable map. In the truth table of Table 5-6, the fundamental products are $\overline{A}B$ and $A\overline{B}$. When 1s are entered on the Karnaugh map for these products and 0s for the remaining spaces, the completed map looks like Fig. 5-10*e*.

TABLE 5-5

A	B	Y
0	0	0
0	1	0
1	0	1
1	1	1

TABLE 5-6

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

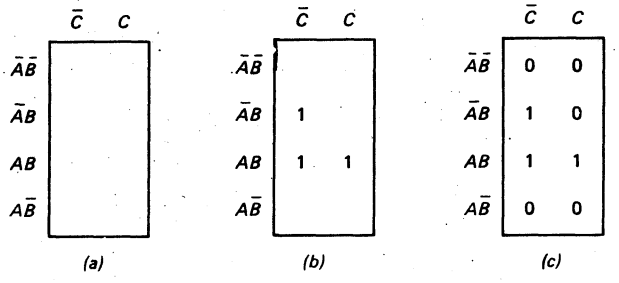


Fig. 5-11 Three-variable Karnaugh map.

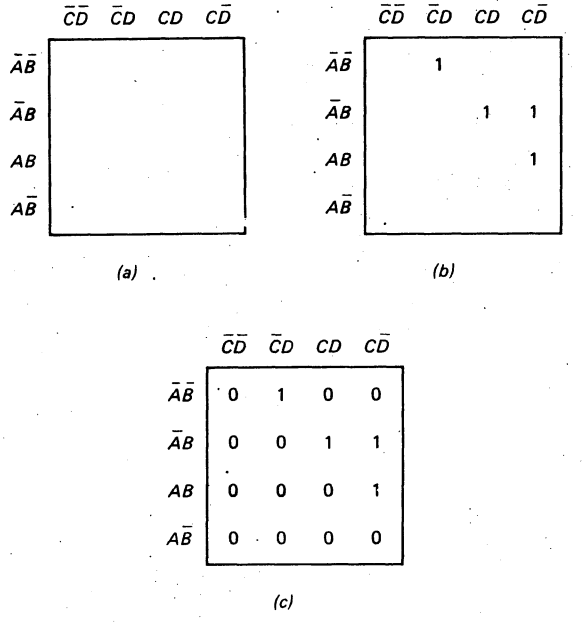


Fig. 5-12 Four-variable Karnaugh map.

Three-Variable Map

Suppose you have a truth table like Table 5-7. Begin by drawing Fig. 5-11*a*. It is especially important to notice the order of the variables and their complements. The vertical column is labeled $\bar{A}\bar{B}$, $\bar{A}B$, AB, and $A\bar{B}$. This order is not a binary progression; instead it follows the order of 00, 01, 11, and 10. The reason for this is explained in the derivation of the Karnaugh method; briefly, it's done so that only one variable changes from complemented to uncomplemented form (or vice versa).

Next, look for output 1s in Table 5-7. The fundamental products for these 1 outputs are $\bar{A}B\bar{C}$, $AB\bar{C}$, and ABC. Enter these 1s on the Karnaugh map (Fig. 5-11*b*). The final step is to enter 0s in the remaining spaces (Fig. 5-11*c*). This Karnaugh map is useful because it shows the fundamental products needed for the sum-of-products circuit.

TABLE 5-7

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Four-Variable Map

Many MSI circuits process binary words of 4 bits each (nibbles). For this reason, logic circuits are often designed to handle four variables (or their complements). This is why the four-variable map is the most important.

Here's an example of constructing a four-variable map. Suppose you have the truth table of Table 5-8. The first step is to draw the blank map of Fig. 5-12*a*. Again, notice the progression. The vertical column is labeled $\bar{A}\bar{B}$, $\bar{A}B$,

TABLE 5-8

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

AB, and $A\overline{B}$. The horizontal row is labeled $\overline{C}\overline{D}$, $\overline{C}D$, CD, and $C\overline{D}$.

In Table 5-8 the output 1s have these fundamental products: $\overline{A}B\overline{C}D$, $\overline{A}BCD$, $\overline{A}BCD$, and $ABC\overline{D}$. After entering 1s on the Karnaugh map, you will have Fig. 5-12*b*. The final step of filling in 0s results in the completed map of Fig. 5-12*c*.

5-5 PAIRS, QUADS, AND OCTETS

There is a way of using the Karnaugh map to get simplified logic circuits. But before you can understand how this is done, you will have to learn the meaning of *pairs*, *quads*, and *octets*.

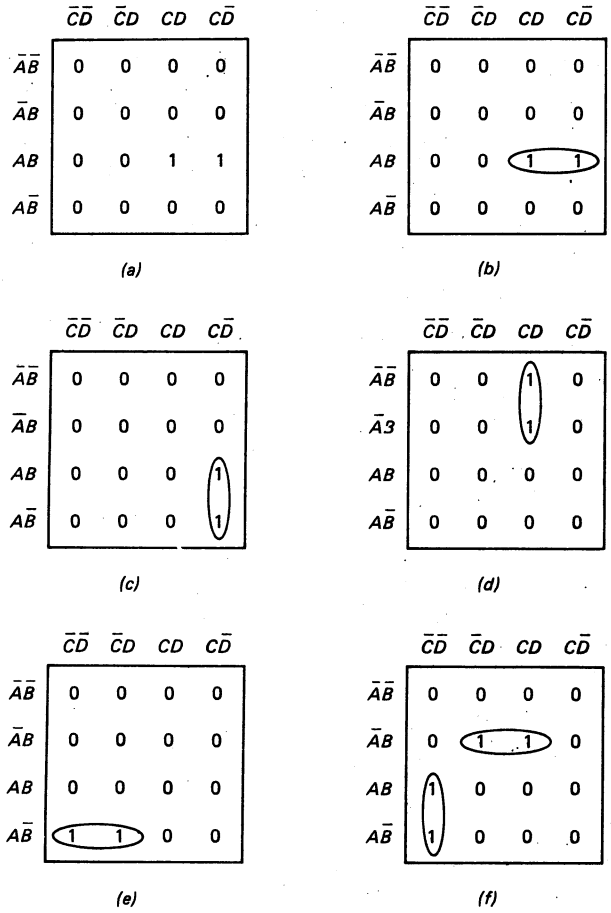


Fig. 5-13 Pairs on a Karnaugh map.

Pairs

The map of Fig. 5-13*a* contains a *pair* of 1s that are horizontally adjacent. The first 1 represents the product $ABCD$; the second 1 stands for the product $ABC\overline{D}$. As we move from the first 1 to the second 1, only one variable goes from uncomplemented to complemented form (D to $\overline{D}$). The other variables don't change form (A, B, and C remain uncomplemented). Whenever this happens, you can eliminate the variable that changes form.

Algebraic Proof

The sum-of-products equation corresponding to Fig. 5-13*a* is

$$Y = ABCD + ABC\overline{D}$$

which factors into

$$Y = ABC(D + \overline{D})$$

Since D is ORed with $\overline{D}$, the equation reduces to

$$Y = ABC$$

A pair of adjacent 1s is like those of Fig. 5-13*a* always means that the sum-of-products equation will have a variable and a complement that drop out.

For easy identification, it is customary to encircle a pair of adjacent 1s, as shown in Fig. 5-13*b*. Then when you look at the map, you can tell at a glance that one variable and its complement will drop out of the boolean equation. In other words, an encircled pair of 1s like those of Fig. 5-13*b* no longer stands for the ORing of two separate products, $ABCD$ and $ABC\overline{D}$. The encircled pair should be visualized instead as representing a single reduced product ABC.

Here's another example. Figure 5-13*c* shows a pair of 1s that are vertically adjacent. These 1s correspond to the product $ABC\overline{D}$ and $A\overline{B}C\overline{D}$. Notice that only one variable changes from uncomplemented to complemented form (B to $\overline{B}$); all other variables retain their original form. Therefore, B and $\overline{B}$ drop out. This means that the encircled pair of Fig. 5-13*c* represents $AC\overline{D}$.

From now on, whenever you see a pair of adjacent 1s, eliminate the variable that goes from complemented to uncomplemented form. A glance at Fig. 5-13*d* indicates that B changes form; therefore, the pair of 1s represents $\overline{A}CD$. Likewise, D changes form in Fig. 5-13*e*; so the pair of 1s stands for $A\overline{B}\overline{C}$.

If more than one pair exists on a Karnaugh map, you can OR the simplified products to get the boolean equation. For instance, the lower pair of Fig. 5-13*f* represents $A\overline{C}\overline{D}$. The upper pair stands for $\overline{A}BD$. The corresponding boolean equation for this map is

$$Y = A\overline{C}\overline{D} + \overline{A}BD$$

The Quad

A *quad* is a group of four 1s that are end to end, as shown in Fig. 5-14*a*, or in the form of a square, as shown in Fig.

5-14*b*. When you see a quad, always encircle it because it leads to a simpler product. In fact, a quad means that two variables and their complements drop out of the boolean equation.

Here's why a quad eliminates two variables. Visualize the four 1s of Fig. 5-14*a* as two pairs (Fig. 5-14*c*). The first pair represents $AB\overline{C}$; the second pair stands for ABC. The boolean equation for these two pairs is

$$Y = AB\overline{C} + ABC$$

This factors into

$$Y = AB(\overline{C} + C)$$

which reduces to

$$Y = AB$$

So the quad of Fig. 5-14*a* represents a product where two variables and their complements drop out.

A similar proof applies to all quads. There's no need to go through the algebra again. Merely determine which variables go from complemented to uncomplemented form; these are the variables that drop out.

For instance, look at the quad of Fig. 5-14*b*. Pick any 1 as a starting point. When you move horizontally, D is the variable that changes form. When you move vertically, B changes form. Therefore, the simplified equation is

$$Y = AC$$

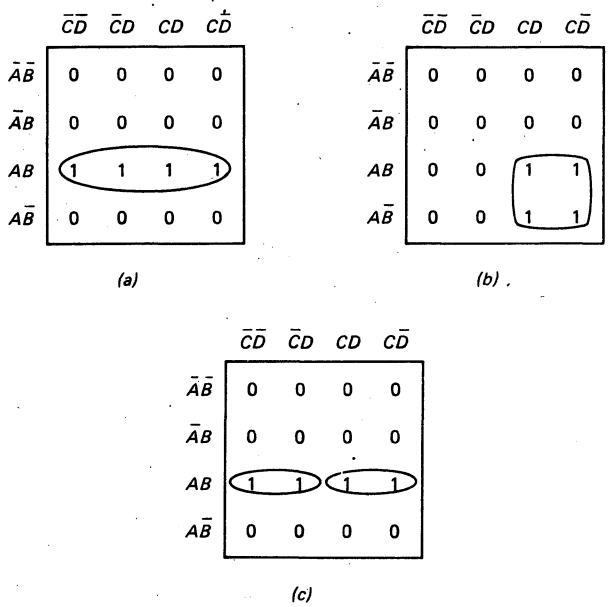


Fig. 5-14 Quads on a Karnaugh map.

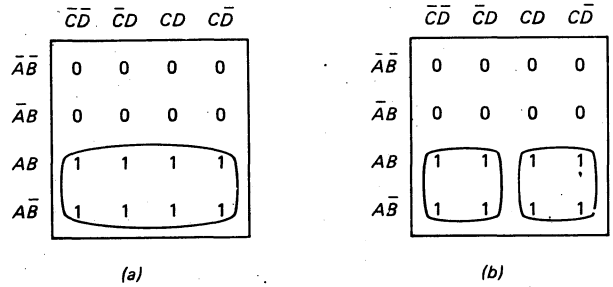


Fig. 5-15 Octets on a Karnaugh map.

The Octet

An *octet* is a group of eight adjacent 1s like those of Fig. 5-15*a*. An octet always eliminates three variables and their complements. Here's why. Visualize the octet as two quads (Fig. 5-15*b*). The equation for these two quads is

$$Y = A\overline{C} + AC$$

Factoring gives

$$Y = A(\overline{C} + C)$$

But this reduces to

$$Y = A$$

So the octet of Fig. 5-15*a* means that three variables and their complements drop out of the corresponding product.

A similar proof applies to any octet. From now on, don't bother with the algebra. Just step through the 1s of the octet and determine which three variables change form. These are the variables that drop out.

5-6 KARNAUGH SIMPLIFICATIONS

You have seen how a pair eliminates one variable, a quad eliminates two variables, and an octet eliminates three variables. Because of this, you should encircle the octets first, the quads second, and the pairs last. In this way, the greatest simplification takes place.

An Example

Suppose you've translated a truth table into the Karnaugh map shown in Fig. 5-16*a*. Look for octets first. There are none. Next, look for quads. There are two. Finally, look for pairs. There is one. If you do it correctly, you arrive at Fig. 5-16*b*.

The pair represents the simplified product $\overline{A}\overline{B}D$, the lower quad stands for $A\overline{C}$, and the quad on the right

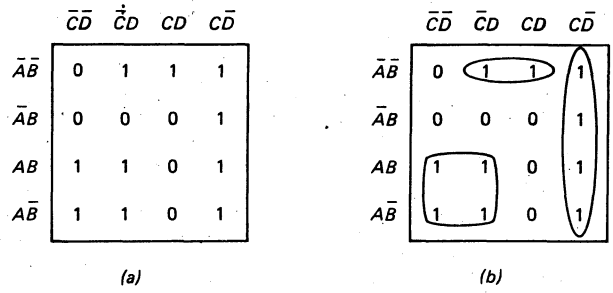


Fig. 5-16

represents $C\bar{D}$. By ORing these simplified products, you get the boolean equation for the map

$$Y = \bar{A}\bar{B}D + A\bar{C} + C\bar{D} \qquad (5\text{-}24)$$

Overlapping Groups

When you encircle groups, you are allowed to use the same 1 more than once. Figure 5-17*a* illustrates the idea. The simplified equation for the overlapping groups is

$$Y = A + B\bar{C}D \qquad (5\text{-}25)$$

It is valid to encircle the 1s as shown in Fig. 5-17*b*, but then the isolated 1 results in a more complicated equation:

$$Y = A + \bar{A}B\bar{C}D$$

This requires a more complicated logic circuit than Eq. 5-25. So always overlap groups if possible; that is, use the 1s more than once to get the largest groups you can.

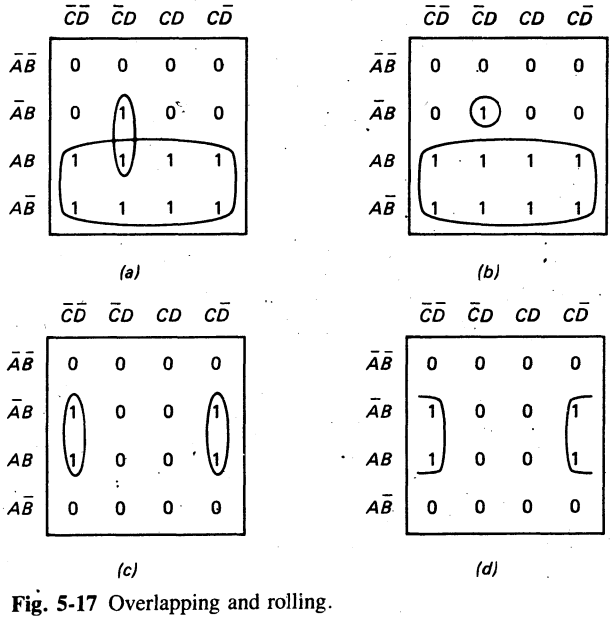


Fig. 5-17 Overlapping and rolling.

Rolling the Map

Another thing to know about is rolling. In Fig. 5-17*c*, the pairs result in the equation

$$Y = B\bar{C}\bar{D} + BC\bar{D} \qquad (5\text{-}26)$$

Visualize picking up the Karnaugh map and rolling it so that the left side touches the right side. If you're visualizing correctly, you will realize the two pairs actually form a quad. To indicate this, draw half circles around each pair, as shown in Fig. 5-17*d*. From this viewpoint, the quad of Fig. 5-17*d* has the equation

$$Y = B\bar{D} \qquad (5\text{-}27)$$

Why is rolling valid? Because Eq. 5-26 can be simplified to Eq. 5-27. Here's the proof. Start with Eq. 5-26:

$$Y = B\bar{C}\bar{D} + BC\bar{D}$$

This factors into

$$Y = B\bar{D}(\bar{C} + C)$$

which reduces to

$$Y = B\bar{D}$$

This final equation represents a rolled quad like Fig. 5-17*d*. Therefore, 1s on the edges of a Karnaugh map can be grouped with 1s on opposite edges.

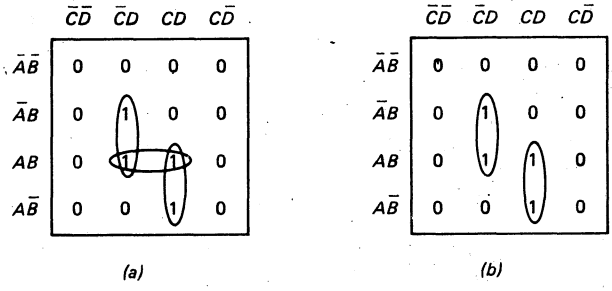


Fig. 5-18 Redundant group.

Redundant Groups

After you finish encircling groups, there is one more thing to do before writing the simplified boolean equation: eliminate any group whose 1s are completely overlapped by other groups. (A group whose 1s are all overlapped by other groups is called a *redundant group*.)

Here is an example. Suppose you have encircled the three pairs shown in Fig. 5-18*a*. The boolean equation then is

$$Y = B\bar{C}D + ABD + ACD$$

At this point, you should check to see if there are any redundant groups. Notice that the 1s in the inner pair are completely overlapped by the outside pairs. Because of this, the inner pair is a redundant pair and can be eliminated to get the simpler map of Fig. 5-18*b*. The equation for this map is

$$Y = B\bar{C}D + ACD$$

Since this is a simpler equation, it means a simpler logic circuit. This is why you should eliminate redundant groups if they exist.

Summary

Here's a summary of how to use the Karnaugh map to simplify logic circuits:

1. Enter a 1 on the Karnaugh map for each fundamental product that corresponds to 1 output in the truth table. Enter 0s elsewhere.
2. Encircle the octets, quads, and pairs. Remember to roll and overlap to get the largest groups possible.
3. If any isolated 1s remain, encircle them.
4. Eliminate redundant groups if they exist.
5. Write the boolean equation by ORing the products corresponding to the encircled groups.
6. Draw the equivalent logic circuit.

EXAMPLE 5-1

What is the simplified boolean equation for the Karnaugh map of Fig. 5-19*a*?

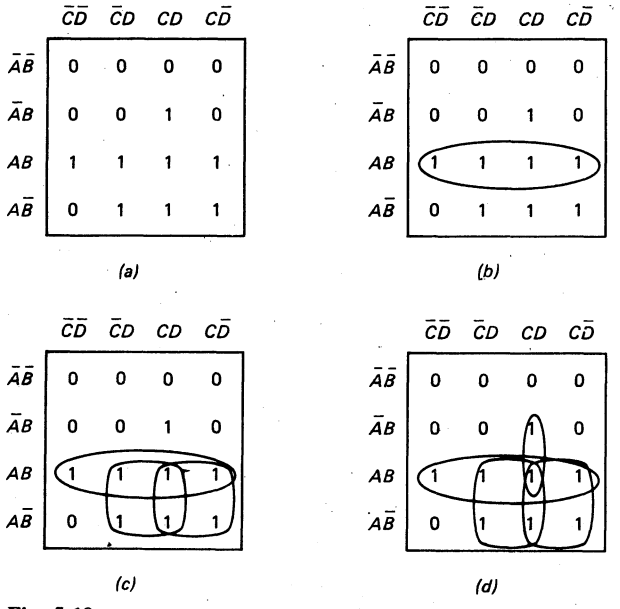


Fig. 5-19

SOLUTION

There are no octets, but there is a quad, as shown in Fig. 5-19*b*. By overlapping we can find two more quads (Fig. 5-19*c*). Finally, overlapping gives us the pair of Fig. 5-19*d*.

The horizontal quad of Fig. 5-19*d* corresponds to a simplified product of AB. The square quad on the right corresponds to AC, while the one on the left stands for AD. The pair represents BCD. By ORing these products we get the simplified equation

$$Y = AB + AC + AD + BCD \qquad (5\text{-}28)$$

Figure 5-20 shows the equivalent logic circuit.

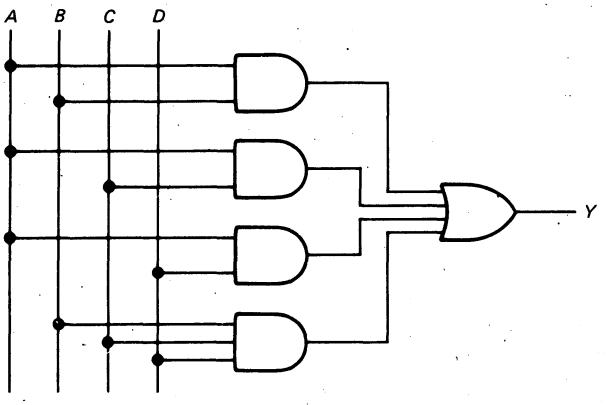


Fig. 5-20

EXAMPLE 5-2

As you know from Chap. 4, the NAND gate is the least expensive gate in the 7400 series. Because of this, AND-OR circuits are usually built as equivalent NAND-NAND circuits.

Convert the AND-OR circuit of Fig. 5-20 to a NAND-NAND circuit using 7400-series devices.

SOLUTION

Replace each AND gate of Fig. 5-20 by a NAND gate and replace the final OR gate by a NAND gate. Figure 5-21 is the De Morgan equivalent of Fig. 5-20. As shown, we can build the circuit with a 7400, a 7410, and a 7420.

5-7 DON'T-CARE CONDITIONS

Sometimes, it doesn't matter what the output is for a given input word. To indicate this, we use an X in the truth table instead of a 0 or a 1. For instance, look at Table 5-9. The

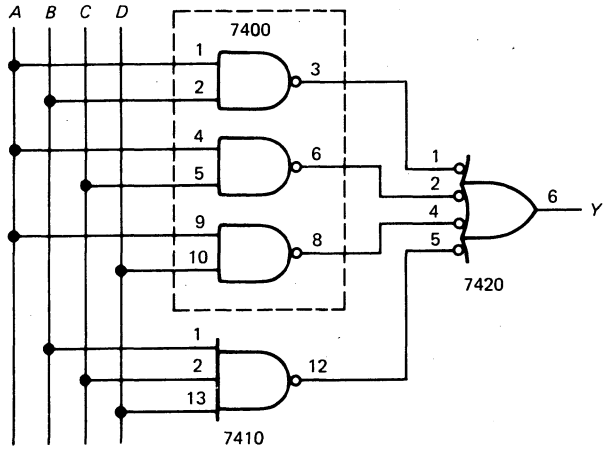


Fig. 5-21 NAND-NAND circuit using TTL gates.

output is an X for any input word from 1000 through 1111. The X's are called *don't cares* because they can be treated either as 0s or 1s, whichever leads to a simpler circuit.

Figure 5-22*a* shows the Karnaugh map for Table 5-9. X's are used for $A\overline{B}\,\overline{C}\,\overline{D}$, $A\overline{B}\,\overline{C}D$, $A\overline{B}C\overline{D}$, $A\overline{B}CD$, $AB\overline{C}\,\overline{D}$, $AB\overline{C}D$, $ABC\overline{D}$, and $ABCD$ because these are don't cares in the truth table. Figure 5-22*b* shows the most efficient way to encircle the groups. Notice two crucial ideas. First, we visualize all X's as 1s and try to form the largest groups that include the real 1s. This gives us three quads. Second, we visualize all remaining X's as 0s. In this way, the X's are used to the best advantage. We are free to do this because the don't cares can be either 0s or 1s, whichever we prefer.

TABLE 5-9

A	*B*	*C*	*D*	*Y*
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	X
1	0	0	1	X
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

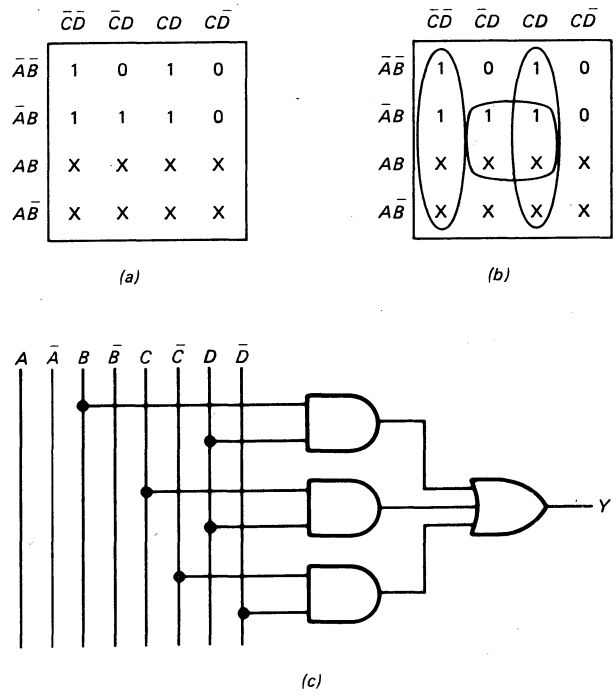


Fig. 5-22 Don't cares.

Figure 5-22*b* implies the simplified boolean equation

$$Y = BD + \overline{C}\,\overline{D} + CD$$

Figure 5-22*c* is the simplified logic circuit. This AND-OR network has nine input gate leads.

EXAMPLE 5-3

Recall that BCD numbers express each decimal digit as a nibble: 0 to 9 are encoded as 0000 to 1001. Especially important, nibbles 1010 to 1111 are never used in a BCD system.

Table 5-10 shows a truth table for use in a BCD system. As you see, don't cares appear for 1010 through 1111. Construct the Karnaugh map and show the simplified logic circuit.

SOLUTION

Figure 5-23*a* illustrates the Karnaugh map. The largest group we can form is the pair shown in Fig. 5-23*b*. The boolean equation is

$$Y = BCD$$

Figure 5-23*c* is the simplified logic circuit.

TABLE 5-10

A	*B*	*C*	*D*	*Y*
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

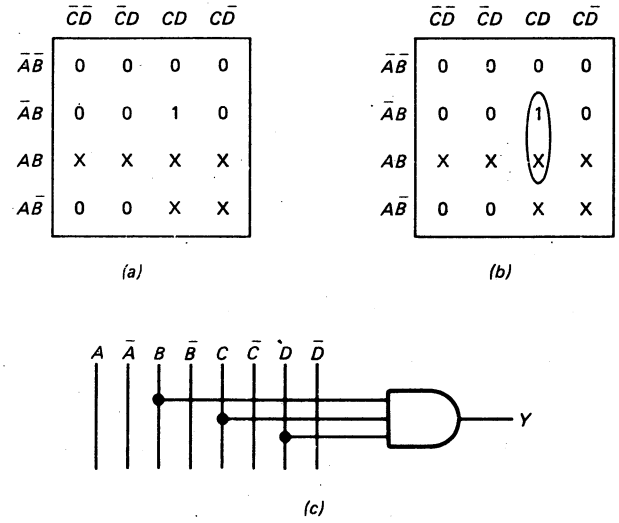


Fig. 5-23 Don't cares in a BCD system.

GLOSSARY

bus A group of wires carrying digital signals.
don't care An output that may be either low or high without affecting the operation of the system.
fundamental product The logical product of variables and complements that produces a high output for a given input condition.
Karnaugh map A graphical display of the fundamental products in a truth table.
octet A group of eight adjacent 1s on a Karnaugh map.
pair A group of two adjacent 1s on a Karnaugh map. These 1s may be horizontally or vertically aligned.
quad A group of four adjacent 1s on a Karnaugh map.
redundant group A group of 1s on a Karnaugh map all of which are overlapped by other groups.
sum-of-products circuit An AND-OR circuit obtained by ORing the fundamental products that produce output 1s in a truth table.

SELF-TESTING REVIEW

Read each of the following and provide the missing words. Answers appear at the beginning of the next question.

1. Digital design often starts by constructing a _____ table. By ORing the _____ products, you get a sum-of-products equation.
2. (*truth, fundamental*) A preliminary guide for comparing the simplicity of logic circuits is to count the number of input _____ leads.
3. (*gate*) A bus is a group of _____ carrying digital signals. In the typical microcomputer, the microprocessor, memory, and I/O units communicate via buses.
4. (*wires*) One way to simplify the sum-of-products equation is to use boolean algebra. Another way is the _____ map.
5. (*Karnaugh*) A pair eliminates one variable, a _____ eliminates two variables, and an octet eliminates _____ variables. Because of this, you should encircle the _____ first, the quads next, and the pairs last.
6. (*quad, three, octets*) NAND-NAND circuits are equivalent to AND-OR circuits. This is important because _____ gates are the least expensive gates in the 7400 series.
7. (*NAND*) When a truth table has don't cares, we enter X's on the Karnaugh map. These can be treated as 0s or 1s, whichever leads to a simpler logic circuit.

PROBLEMS

5-1. What are the fundamental products for each of the inputs words ABCD = 0010, ABCD = 1101, ABCD = 1110?

5-2. A truth table has output 1s for each of these inputs:

a. ABCD = 0011
b. ABCD = 0101
c. ABCD = 1000
d. ABCD = 1101

What are the fundamental products?

5-3. Draw the logic circuit for this boolean equation:

$$Y = \overline{A}\overline{B}\overline{C}D + A\overline{B}\overline{C}D + AB\overline{C}D + ABC\overline{D}$$

5-4. Output 1s appear in the truth table for these input conditions: ABCD = 0001, ABCD = 0110, and ABCD = 1110. What is the sum-of-products equation?

5-5. Draw the AND-OR circuit for

$$Y = A\overline{B}\overline{C}\overline{D} + AB\overline{C}\overline{D} + ABCD$$

How many input gate leads does this circuit have?

5-6. A truth table has output 1s for these inputs: ABCD = 0011, ABCD = 0110, ABCD = 1001, and ABCD = 1110. Draw the Karnaugh map showing the fundamental products.

5-7. A truth table has four input variables. The first eight outputs are 0s, and the last eight outputs are 1s. Draw the Karnaugh map.

5-8. Draw the Karnaugh map for the Y_3 output of Table 5-11. Simplify as much as possible; then draw the logic circuit.

5-9. Use the Karnaugh map to work out the simplified logic circuit for the Y_2 output of Table 5-11.

5-10. Repeat Prob. 5-9 for the Y_1 output.

5-11. Repeat Prob. 5-9 for the Y_0 output.

5-12. Use the Karnaugh map to work out the simplified logic circuit for the Y_3 output of Table 5-12.

5-13. Repeat Prob. 5-12 for the Y_2 output.

5-14. Repeat Prob. 5-12 for the Y_1 output.

5-15. Repeat Prob. 5-12 for Y_0 output.

TABLE 5-11

A	B	C	D	Y_3	Y_2	Y_1	Y_0
0	0	0	0	1	0	1	0
0	0	0	1	0	1	0	1
0	0	1	0	0	1	1	1
0	0	1	1	1	0	0	1
0	1	0	0	0	0	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	1	1	0
0	1	1	1	1	1	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	0	1
1	0	1	0	1	0	1	1
1	0	1	1	0	1	0	0
1	1	0	0	0	1	1	0
1	1	0	1	1	0	1	0
1	1	1	0	1	1	0	0
1	1	1	1	1	1	0	1

TABLE 5-12

A	B	C	D	Y_3	Y_2	Y_1	Y_0
0	0	0	0	1	0	1	0
0	0	0	1	0	1	0	1
0	0	1	0	0	1	1	1
0	0	1	1	1	0	0	1
0	1	0	0	0	0	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	1	1	0
0	1	1	1	1	1	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	0	1
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

Arithmetic-Logic Units

6

The arithmetic-logic unit (ALU) is the number-crunching part of a computer. This means not only arithmetic operations but logic as well (OR, AND, NOT, and so forth). In this chapter you will learn how the ALU adds and subtracts binary numbers. Later chapters will discuss the logic operations.

6-1 BINARY ADDITION

ALUs don't process decimal numbers; they process binary numbers. Before you can understand the circuits inside an ALU, you must learn how to add binary numbers. There are five basic cases that must be understood before going on.

Case 1

When no pebbles are added to no pebbles, the total is no pebbles. As a word equation,

None + none = none

With binary numbers, this equation is written as

0 + 0 = 0

Case 2

If no pebbles are added to one pebble, the total is one pebble:

None + ● = ●

In terms of binary numbers,

0 + 1 = 1

Case 3

Addition is commutative. This means you can transpose the numbers of the preceding case to get

● + none = ●

or

1 + 0 = 1

Case 4

Next, one pebble added to one pebble gives two pebbles:

● + ● = ●●

As a binary equation,

1 + 1 = 10

To avoid confusion with decimal numbers, read this as "one plus one equals one-zero." An alternative way of reading the equation is "one plus one equals zero, carry one."

Case 5

One pebble plus one pebble plus one pebble gives a total of three pebbles:

● + ● + ● = ●●●

The binary equation is

1 + 1 + 1 = 11

Read this as "one plus one plus one equals one-one." Alternatively, "one plus one plus one equals one, carry one."

Rules to Remember

The foregoing cases are all you need for more complicated binary addition. Therefore, memorize these five rules:

$$0 + 0 = 0 \quad (6\text{-}1)$$
$$0 + 1 = 1 \quad (6\text{-}2)$$
$$1 + 0 = 1 \quad (6\text{-}3)$$
$$1 + 1 = 10 \quad (6\text{-}4)$$
$$1 + 1 + 1 = 11 \quad (6\text{-}5)$$

Larger Binary Numbers

Column-by-column addition applies to binary numbers as well as decimal. For example, suppose you have this problem in binary addition:

$$\begin{array}{r} 11100 \\ +\ 11010 \\ \hline ? \end{array}$$

Start with the least significant column to get

$$\begin{array}{r} 11100 \\ +\ 11010 \\ \hline 0 \end{array}$$

Here, 0 + 0 gives 0.

Next, add the bits of the second column as follows:

$$\begin{array}{r} 11100 \\ +\ 11010 \\ \hline 10 \end{array}$$

This time, 0 + 1 results in 1.

The third column gives

$$\begin{array}{r} 11100 \\ +\ 11010 \\ \hline 110 \end{array}$$

In this case, 1 + 0 produces 1.

The fourth column results in

$$\begin{array}{r} 11100 \\ +\ 11010 \\ \hline 0110 \end{array} \quad \text{(carry 1)}$$

As you see, 1 + 1 equals 0 with a carry of 1.

Finally, the last column gives

$$\begin{array}{r} 11100 \\ +\ 11010 \\ \hline 110110 \end{array}$$

Here, 1 + 1 + 1 (carry) produces 11, recorded as 1 with a carry to the next higher column.

EXAMPLE 6-1

Add the binary numbers 01010111 and 00110101.

SOLUTION

This is the problem:

$$\begin{array}{r} 01010111 \\ +\ 00110101 \\ \hline ? \end{array}$$

If you add the bits column by column as previously demonstrated, you will get

$$\begin{array}{r} 01010111 \\ +\ 00110101 \\ \hline 10001100 \end{array}$$

Expressed in hexadecimal numbers, the foregoing addition is

$$\begin{array}{r} 57 \\ +\ 35 \\ \hline 8C \end{array}$$

For clarity, we can use subscripts:

$$\begin{array}{r} 57_{16} \\ +\ 35_{16} \\ \hline 8C_{16} \end{array}$$

In microprocessor work, it is more convenient to use the letter H to signify hexadecimal numbers. In other words, the usual way to express the foregoing addition is

$$\begin{array}{r} 57H \\ +\ 35H \\ \hline 8CH \end{array}$$

6-2 BINARY SUBTRACTION

To subtract binary numbers, we need to discuss four cases.

Case 1: $0 - 0 = 0$
Case 2: $1 - 0 = 1$
Case 3: $1 - 1 = 0$
Case 4: $10 - 1 = 1$

The last result represents

●● − ● = ●

which makes sense.

To subtract larger binary numbers, subtract column by column, borrowing from the next higher column when necessary. For instance, in subtracting 101 from 111, proceed like this:

$$\begin{array}{r} 7 \\ -\ 5 \\ \hline 2 \end{array} \qquad \begin{array}{r} 111 \\ -\ 101 \\ \hline 010 \end{array}$$

Starting on the right, 1 − 1 gives 0; then, 1 − 0 is 1; finally, 1 − 1 is 0.

Here is another example: subtract 1010 from 1101.

$$\begin{array}{r} 13 \\ -\ 10 \\ \hline 3 \end{array} \qquad \begin{array}{r} 1101 \\ -\ 1010 \\ \hline 0011 \end{array}$$

In the least significant column, 1 − 0 is 1. In the second column, we have to borrow from the next higher column; then, 10 − 1 is 1. In the third column, 0 (after borrow) − 0 is 0. In the fourth column, 1 − 1 = 0.

Direct subtraction like the foregoing has been used in computers; however, it is possible to subtract in a different way. Later sections of this chapter will show you how.

6-3 HALF-ADDERS

Figure 6-1 is a *half-adder*, a logic circuit that adds 2 bits. Notice the outputs: *SUM* and *CARRY*. The boolean equations for these outputs are

$$SUM = A \oplus B \tag{6-6}$$
$$CARRY = AB \tag{6-7}$$

The *SUM* output is *A* XOR *B*; the *CARRY* output is *A* AND *B*. Therefore, *SUM* is a 1 when *A* and *B* are different; *CARRY* is a 1 when *A* and *B* are 1s.

Table 6-1 summarizes the operation. When *A* and *B* are 0s, the *SUM* is 0 with a *CARRY* of 0. When *A* is 0 and *B* is 1, the *SUM* is 1 with a *CARRY* of 0. When *A* is 1 and *B* is 0, the *SUM* equals 1 with a *CARRY* of 0. Finally, when *A* is 1 and *B* is 1, the *SUM* is 0 with a *CARRY* of 1.

The logic circuit of Fig. 6-1 does electronically what we do mentally when we add 2 bits. Applications for the half-adder are limited. What we need is a circuit that can add 3 bits at a time.

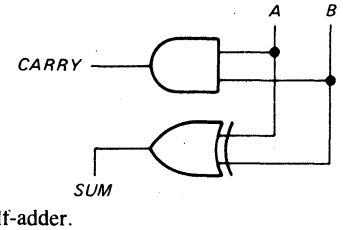


Fig. 6-1 Half-adder.

TABLE 6-1. HALF-ADDER

A	*B*	*CARRY*	*SUM*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

6-4 FULL ADDERS

Figure 6-2 shows a *full adder*, a logic circuit that can add 3 bits. Again there are two outputs, *SUM* and *CARRY*. The boolean equations are

$$SUM = A \oplus B \oplus C \tag{6-8}$$
$$CARRY = AB + AC + BC \tag{6-9}$$

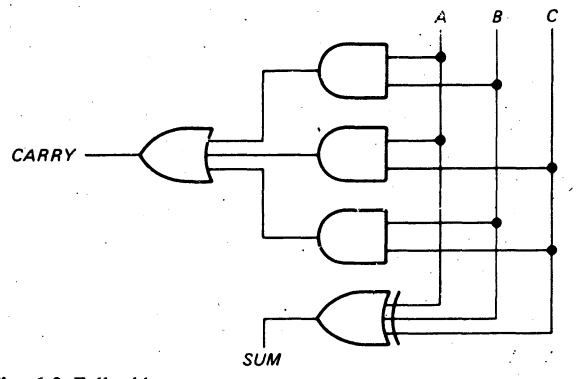


Fig. 6-2 Full adder.

In this case, *SUM* equals *A* XOR *B* XOR *C*; *CARRY* equals *AB* OR *AC* OR *BC*. Therefore, *SUM* is 1 when the number of input 1s is odd; *CARRY* is a 1 when two or more inputs are 1s.

Table 6-2 summarizes the circuit action. *A*, *B*, and *C* are the bits being added. If you check each entry, you will see that the circuit adds 3 bits at a time and comes up with the correct answer.

TABLE 6-2. FULL ADDER

A	*B*	*C*	*CARRY*	*SUM*
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Here's the point. The circuit of Fig. 6-2 does electronically what we do mentally when we add 3 bits. The full adder can be cascaded to add large binary numbers. The next section tells you how.

6-5 BINARY ADDERS

Figure 6-3 shows a *binary adder,* a logic circuit that can add two binary numbers. The block on the right (labeled HA) represents a half-adder. The inputs are A_0 and B_0; the outputs are S_0 *(SUM)* and C_1 *(CARRY)*. All other blocks are full adders (abbreviated FA). Each of these full adders has three inputs (A_n, B_n, and C_n) and two outputs.

The circuit adds two binary numbers. In other words, it carries out the following addition:

$$\begin{array}{r} A_3A_2A_1A_0 \\ +\ B_3B_2B_1B_0 \\ \hline C_4S_3S_2S_1S_0 \end{array}$$

Here's an example. Suppose **A** = 1100 and **B** = 1001. Then the problem is

$$\begin{array}{r} 1100 \\ +\ 1001 \\ \hline ? \end{array}$$

Figure 6-4 shows the binary adder with the same inputs, 1100 and 1001. The half-adder produces a sum of 1 and carry of 0, the first full adder produces a sum of 0 and a carry of 0, the second full adder produces a sum of 1 and a carry of 0, and the third full adder produces a sum of 0 and a carry of 1. The overall output is 10101, the same answer we would get with pencil and paper.

By using more full adders, we can build binary adders of any length. For example, to add 16-bit numbers, we need 1 half-adder and 15 full adders. From now on, we will use the abbreviated symbol of Fig. 6-5 to represent a binary adder of any length. Notice the solid arrows, the standard way to indicate words in motion. In Fig. 6-5, words **A** and **B** are added to get a sum of **S** plus a final *CARRY*.

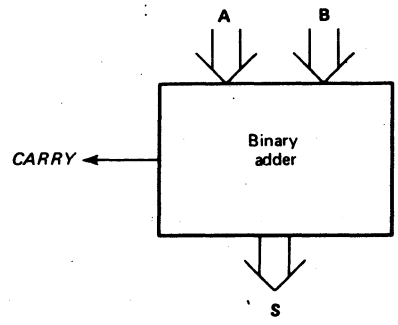


Fig. 6-5 Symbol for binary adder.

EXAMPLE 6-2

Find the output in Fig. 6-5 if the two input words are

A = 0000 0001 0000 1100
B = 0000 0000 0100 1001

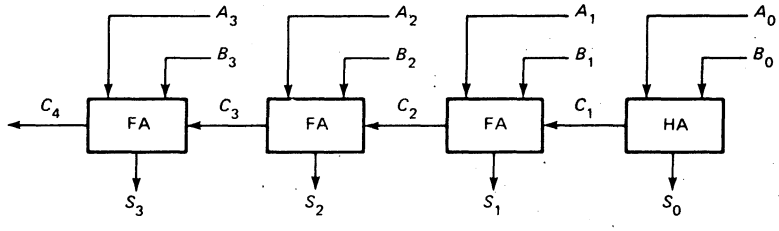


Fig. 6-3 Binary adder.

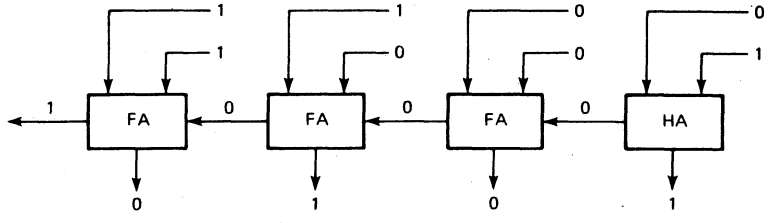


Fig. 6-4 Adding 12 and 9 to get 21.

SOLUTION

The binary adder adds the two inputs to get

$$\begin{array}{r} 0000\ 0001\ 0000\ 1100 \\ +\ 0000\ 0000\ 0100\ 1001 \\ \hline 0000\ 0001\ 0101\ 0101 \end{array}$$

In hexadecimal form, the foregoing addition is

$$\begin{array}{r} 010CH \\ +\ 0049H \\ \hline 0155H \end{array}$$

6-6 SIGNED BINARY NUMBERS

The negative decimal numbers are -1, -2, -3, and so on. One way to code these as binary numbers is to convert the *magnitude* (1, 2, 3, . . .) to its binary equivalent and prefix the sign. With this approach, -1, -2, and -3 becomes -001, -010, and -011. It's customary to use 0 for the + sign and 1 for the − sign. Therefore, -001, -010, and -011 are coded as 1001, 1010, and 1011.

The foregoing numbers have the *sign bit* followed by the magnitude bits. Numbers in this form are called *signed binary numbers* or *sign-magnitude numbers*. For larger decimal numbers you need more than 4 bits. But the idea is still the same: the leading bit represents the sign and the remaining bits stand for the magnitude.

EXAMPLE 6-3

Express each of the following as 16-bit signed binary numbers.

a. +7
b. −7
c. +25
d. −25

SOLUTION

a. +7 = 0000 0000 0000 0111
b. −7 = 1000 0000 0000 0111
c. +25 = 0000 0000 0001 1001
d. −25 = 1000 0000 0001 1001

No subscripts are used in these equations because it's clear from the context that decimal numbers are being expressed in binary form. Nevertheless, you can use subscripts if you prefer. The first equation can be written as

$$+7_{10} = 0000\ 0000\ 0000\ 0111_2$$

the next equation as

$$-7_{10} = 1000\ 0000\ 0000\ 0111_2$$

and so forth.

EXAMPLE 6-4

Convert the following signed binary numbers to decimal numbers:

a. 0000 0000 0000 1001
b. 1000 0000 0000 1111
c. 1000 0000 0011 0000
d. 0000 0000 1010 0101

SOLUTION

As usual, the leading bit gives the sign and the remaining bits give the magnitude.

a. 0000 0000 0000 1001 = +9
b. 1000 0000 0000 1111 = −15
c. 1000 0000 0011 0000 = −48
d. 0000 0000 1010 0101 = +165

6-7 2's COMPLEMENT

Sign-magnitude numbers are easy to understand, but they require too much hardware for addition and subtraction. This has led to the widespread use of complements for binary arithmetic.

Definition

Recall that a high invert signal to a controlled inverter produces the 1's complement. For instance, if

$$\mathbf{A} = 0111 \qquad (6\text{-}10a)$$

the 1's complement is

$$\overline{\mathbf{A}} = 1000 \qquad (6\text{-}10b)$$

The 2's complement is defined as the new word obtained by adding 1 to 1's complement. As an equation,

$$\mathbf{A}' = \overline{\mathbf{A}} + 1 \qquad (6\text{-}11)$$

where $\mathbf{A}'$ = 2's complement
$\overline{\mathbf{A}}$ = 1's complement

Here are some examples. If

$$\mathbf{A} = 0111$$

the 1's complement is

$$\overline{A} = 1000$$

and the 2's complement is

$$A' = 1001$$

In terms of a binary odometer, the 2's complement is the next reading after the 1's complement.

Another example. If

$$A = 0000\ 1000$$

then

$$\overline{A} = 1111\ 0111$$

and

$$A' = 1111\ 1000$$

Double Complement

If you take the 2's complement twice, you get the original word back. For instance, if

$$A = 0111$$

the 2's complement is

$$A' = 1001$$

If you take the 2's complement of this, you get

$$A'' = 0111$$

which is the original word.

In general, this means that

$$A'' = A \qquad (6\text{-}12)$$

Read this as "the double complement of A equals A." Because of this property, the 2's complement of a binary number is equivalent to the negative of a decimal number. This idea is explained in the following discussion.

Back to the Odometer

Chapter 1 used an odometer to introduce binary numbers. The discussion was about positive numbers only. But odometer readings can also indicate negative numbers. Here's how.

If a car has a binary odometer, all bits eventually reset to 0s. A few readings before and after a complete reset look like this:

1101	
1110	
1111	
0000	(RESET)
0001	
0010	
0011	

1101 is the reading 3 miles before reset, 1110 occurs 2 miles before reset, and 1111 indicates 1 mile before reset. Then, 0001 is the reading 1 mile after reset, 0010 occurs 2 miles after reset, and 0011 indicates 3 miles after reset.

"Before" and "after" are synonymous with "negative" and "positive." Figure 6-6 illustrates this idea with the number line learned in basic algebra: 0 marks the origin, positive decimal numbers are on the right, and negative decimal numbers are on the left. The odometer readings are the binary equivalent of positive and negative decimal numbers: 1101 is the binary equivalent of -3, 1110 stands for -2, 1111 for -1; 0000 for 0; 0001 for $+1$; 0010 for $+2$, and 0011 for $+3$.

The odometer readings of Fig. 6-6 demonstrate how positive and negative numbers are stored in a typical microcomputer. Positive decimal numbers are expressed in sign-magnitude form, but negative decimal numbers are represented as 2's complements. As before, positive numbers have a leading sign bit of 0, and negative numbers have a leading sign bit of 1.

2's Complement Same as Decimal Sign Change

Taking the 2's complement of a binary number is the same as changing the sign of the equivalent decimal number. For example, if

$$A = 0001 \qquad (+1 \text{ in Fig. 6-6})$$

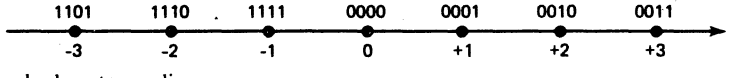


Fig. 6-6 Decimal numbers and odometer readings.

taking the 2's complement gives

$$A' = 1111 \quad (-1 \text{ in Fig. 6-6})$$

Similarly, if

$$A = 0010 \quad (+2 \text{ in Fig. 6-6})$$

then the 2's complement is

$$A' = 1110 \quad (-2 \text{ in Fig. 6-6})$$

Again, if

$$A = 0011 \quad (+3 \text{ in Fig. 6-6})$$

the 2's complement is

$$A' = 1101 \quad (-3 \text{ in Fig. 6-6})$$

The same principle applies to binary numbers of any length: taking the 2's complement of any binary number is the same as changing the sign of the equivalent decimal number. As will be shown later, this property allows us to use a binary adder for both addition and subtraction.

Summary

Here are the main things to remember about 2's complement representation:

1. The leading bit is the sign bit; 0 for plus, 1 for minus.
2. Positive decimal numbers are in sign-magnitude form.
3. Negative decimal numbers are in 2's-complement form.

EXAMPLE 6-5

What is the 2's complement of this word?

$$A = 0011\ 0101\ 1001\ 1100$$

SOLUTION

The 2's complement is

$$A' = 1100\ 1010\ 0110\ 0100$$

EXAMPLE 6-6

What is the binary form of +5 and −5 in 2's-complement representation? Express the answers as 8-bit numbers.

SOLUTION

Decimal +5 is expressed in sign-magnitude form:

$$+5 = 0000\ 0101$$

On the other hand, −5 appears as the 2's complement:

$$-5 = 1111\ 1011$$

EXAMPLE 6-7

What is the 2's-complement representation of −24 in a 16-bit microcomputer?

SOLUTION

Start with the positive form:

$$+24 = 0000\ 0000\ 0001\ 1000$$

Then take the 2's complement to get the negative form:

$$-24 = 1111\ 1111\ 1110\ 1000$$

EXAMPLE 6-8

What decimal number does this represent in 2's-complement representation?

$$1111\ 0001$$

SOLUTION

Start by taking the 2's complement to get

$$0000\ 1111$$

This represents +15. Therefore, the original number is

$$1111\ 0001 = -15$$

6-8 2's-COMPLEMENT ADDER-SUBTRACTER

Early computers used signed binary for both positive and negative numbers. This led to complicated arithmetic circuits. Then, engineers discovered that the 2's-complement representation could greatly simplify arithmetic hardware.

This is why 2's-complement adder-subtracters are now the most widely used arithmetic circuits.

Addition

Figure 6-7 shows a 2's-complement adder-subtracter, a logic circuit that can add or subtract binary numbers. Here's how it works. When *SUB* is low, the *B* bits pass through the controlled inverter without inversion. Therefore, the full adders produce the sum

$$\mathbf{S = A + B} \tag{6-13}$$

Incidentally, as indicated in Fig. 6-7, the final *CARRY* is not used. This is because S_3 is the sign bit and S_2 to S_0 are the numerical bits. The final *CARRY* therefore has no significance at this time.

Subtraction

When *SUB* is high, the controlled inverter produces the 1's complement. Furthermore, the high *SUB* adds a 1 to the first full adder. This addition of 1 to the 1's complement forms the 2's complement of **B.** In other words, the controlled inverter produces $\overline{\mathbf{B}}$, and adding 1 results in $\mathbf{B}'$

The output of the full adders is

$$\mathbf{S = A + B'} \tag{6-14}$$

which is equivalent to

$$\mathbf{S = A - B} \tag{6-15}$$

because the 2's complement is equivalent to a sign change.

EXAMPLE 6-9

A 7483 is a TTL circuit with four full adders. This means that it can add nibbles (4-bit numbers).

Figure 6-8 shows a TTL adder-subtracter. The *CARRY* out (pin 14) of the least significant nibble is used as the *CARRY* in (pin 13) for the most significant nibble. This allows the two 7483s to add 8-bit numbers. Two 7486s form the controlled inverter needed for subtraction.

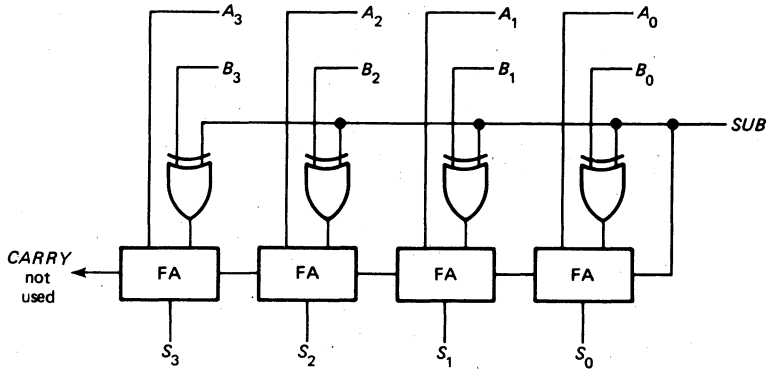


Fig. 6-7 A 2's-complement adder-2-subtracter.

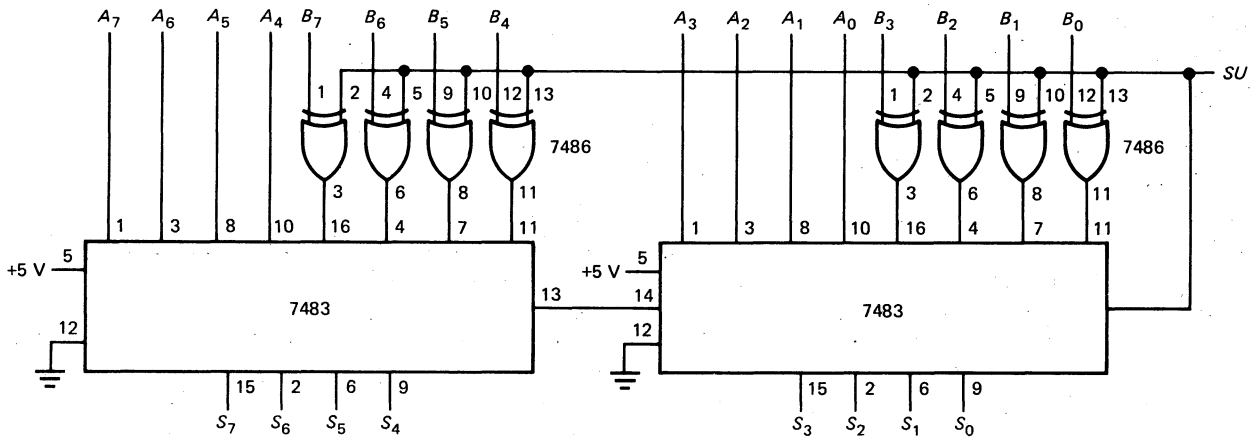


Fig. 6-8 TTL adder-2-subtracter.

Suppose the circuit has these inputs:

$$\mathbf{A} = 0001\ 1000$$
$$\mathbf{B} = 0001\ 0000$$

If *SUB* = 0, what is the output of the adder-subtracter?

SOLUTION

When *SUB* is 0, the adder-subtracter adds the two inputs as follows:

$$\begin{array}{r} 0001\ 1000 \\ +\ 0001\ 0000 \\ \hline 0010\ 1000 \end{array}$$

Therefore, the output is 0010 1000. Notice that the decimal equivalent of the foregoing addition is

$$\begin{array}{r} 24 \\ +\ 16 \\ \hline 40 \end{array}$$

EXAMPLE 6-10

Repeat the preceding example for *SUB* = 1.

SOLUTION

When *SUB* is 1, the adder-subtracter subtracts the inputs by adding the 2's complement as follows:

$$\begin{array}{r} 0001\ 1000 \\ +\ 1111\ 0000 \\ \hline 0000\ 1000 \end{array}$$

The decimal equivalent is

$$\begin{array}{r} 24 \\ +\ -16 \\ \hline 8 \end{array}$$

EXAMPLE 6-11

In Fig. 6-8, what are the largest positive and negative sums we can get?

SOLUTION

The largest positive output is

$$0111\ 1111$$

which represents decimal +127. The largest negative output is

$$1000\ 0000$$

which represents −128. With 8 bits, therefore, all answers must lie between −128 and +127. If you try to add numbers with a sum outside this range, you get an *overflow* into the sign-bit position, causing an error.

Chapter 12 discusses the overflow problem in more detail. All you have to remember for now is that an overflow or error will occur if the true sum lies outside the range of −128 to +127.

GLOSSARY

ALU Arithmetic-logic unit. The ALU carries out arithmetic and logic operations.

binary adder A logic circuit that can add two binary numbers.

full adder A logic circiut that can add 3 bits.

half-adder A logic circuit that adds 2 bits.

overflow In 2's-complement representation, a carry into the sign-bit position, which results in an error. For an 8-bit adder-substracter, the true sum must lie between −128 and +127 to avoid overflow.

signed binary A system in which the leading bit represents the sign and the remaining bits the magnitude of the number. Also called sign magnitude.

2's complement The new number you get when you take the 1's complement and then add 1.

SELF-TESTING REVIEW

Read each of the following and provide the missing words. Answers appear at the beginning of the next question.

1. The ALU carries out arithmetic and _____ operations (OR, AND, NOT, etc.). It processes _____ numbers rather than decimal numbers.
2. (*logic, binary*) A half-adder adds _____ bits. A full adder adds _____ bits, producing a SUM and a _____.
3. (*two, three,* CARRY) A binary adder is a logic cicuit that can add _____ binary numbers at a time. The 7483 is a TTL binary adder. It can add two 4-bit binary numbers.
4. (*two*) With signed binary numbers, also known as sign-magnitude numbers, the leading bit stands for the _____ and the remaining bits for the _____.
5. (*sign, magnitude*) Signed binary numbers require too much hardware. This has led to the use of _____ complements to represent negative numbers. To get the 2's complement of a binary number, you first take the _____ complement, then add _____.
6. (*2's, 1's, 1*) If you take the 2's complement twice, you get the original binary number back. Because of this property, taking the _____ complement of a binary number is equivalent to changing the sign of a decimal number.
7. (*2's*) In a microcomputer positive numbers are represented in _____ form and negative numbers in 2's-complement form. The leading bit still represents the _____.
8. (*sign-magnitude, sign*) A 2's-complement adder-subtracter can add or subtract binary numbers. Sign-magnitude numbers represent _____ decimal numbers, and 2's complements stand for _____ decimal numbers. You can tell one from the other by the leading bit, which represents the _____.
9. (*positive, negative, sign*) With 2's-complement representation and an 8-bit adder-subtracter no overflow is possible if the true sum is between −128 and +127.

PROBLEMS

6-1. Add these 8-bit numbers:
   a. 0001 0000 and 0000 1000
   b. 0001 1000 and 0000 1100
   c. 0001 1100 and 0000 1110
   d. 0010 1000 and 0011 1011
   After you have each binary sum, convert it to hexadecimal form.

6-2. Add these 16-bit numbers:

$$\begin{array}{r} 1000\ 0001\ 1100\ 1001 \\ +\ 0011\ 0011\ 0001\ 0111 \end{array}$$

Express the answer in hexadecimal form.

6-3. In each of the following, convert to binary to do the addition, then convert the answer back to hexadecimal:
   a. 2CH + 4FH = ?
   b. 5EH + 1AH = ?
   c. 3BH + 6DH = ?
   d. A5H + 2CH = ?

6-4. Convert each of the following decimal numbers to an 8-bit sign-magnitude number:
   a. +27
   b. −27
   c. +80
   d. −80
   After you have the sign-magnitude numbers, convert them to hexadecimal form.

6-5. Convert each of these sign-magnitude numbers to its decimal equivalent:
   a. 0001 1110
   b. 1000 0111
   c. 1001 1100
   d. 0011 0001

6-6. The following hexadecimal numbers represent sign-magnitude numbers. Convert each to its decimal equivalent.
   a. 8FH
   b. 3AH
   c. 7FH
   d. FFH

6-7. Find the 2's complements:
   a. 0000 0111
   b. 1111 1111
   c. 1111 1101
   d. 1110 0001
   Express your answers in hexadecimal form.

6-8. Convert each of the following to binary. Then take the 2's complement:
   a. 4CH
   b. 8DH
   c. CBH
   d. FFH

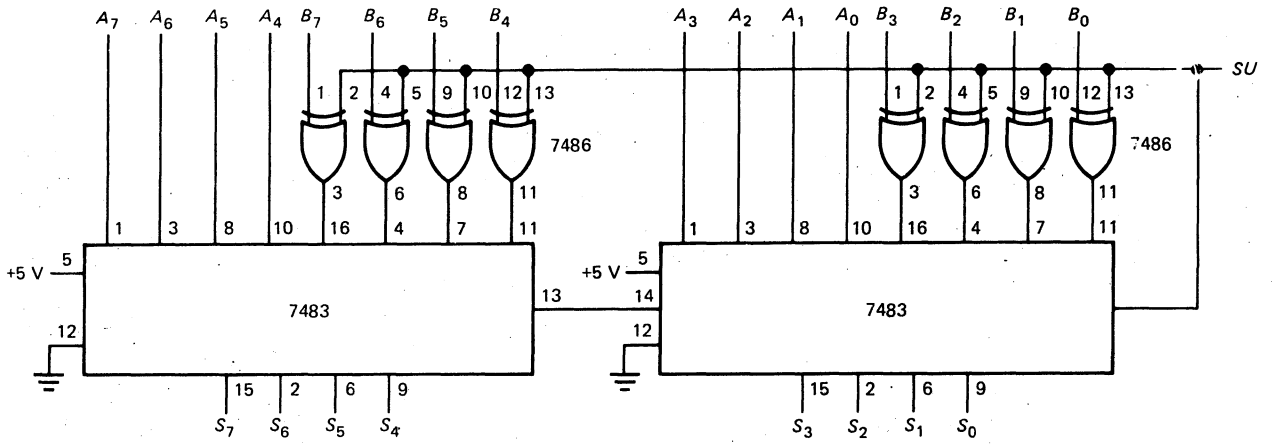


Fig. 6-9

After you have the 2's complements, convert them to hexadecimal form.

6-9. An 8-bit microprocessor uses 2's-complement representation. How do the following decimal numbers appear:

a. −19
b. −48
c. +37
d. −33

Express your answers in binary and hexadecimal form.

6-10. The output of an ALU is EEH. What decimal number does this represent in 2's-complement representation?

6-11. Suppose the inputs to Fig. 6-9 are **A** = 3CH and **B** = 5FH. What is the output for a low *SUB?* A high *SUB?* Express your final answers in hexadecimal form.

6-12. In Fig. 6-9 which of the following inputs cause an overflow when *SUB* is low?

a. 2DH and 4BH
b. 8FH and C3H
c. 5EH and B8H
d. 23H and 14H

Flip-Flops

7

Gates are decision-making elements. As shown in the preceding chapter, they can perform binary addition and subtraction. But decision-making elements are not enough. A computer also needs *memory elements*, devices that can store a binary digit. This chapter is about memory elements called *flip-flops*.

7-1 *RS* LATCHES

A flip-flop is a device with two stable states; it remains in one of these states until triggered into the other. The *RS* latch, discussed in this section, is one of the simplest flip-flops.

Transistor Latch

In Fig. 7-1*a* each collector drives the opposite base through a 100-kΩ resisitor. In a circuit like this, one of the transistors is saturated and the other is cut off.

For instance, if the right transistor is saturated, its collector voltage is approximately 0 V. This means that there is no base drive for the left transistor, so it cuts off and its collector voltage approaches +5 V. This high voltage produces enough base current in the right transistor to sustain its saturation. The overall circuit is *latched* with the left transistor cut off (dark shading) and the right transistor saturated. Q is approximately 0 V.

By a similar argument, if the left transistor is saturated, the right transistor is cut off. Figure 7-1*b* illustrates this other state. Q is approximately 5 V for this condition.

Output Q can be low or high, binary 0 or 1. If latched as shown in Fig. 7-1*a*, the circuit is storing a binary 0 because

$$Q = 0$$

On the other hand, when latched as shown in Fig. 7-1*b*, the circuit stores a binary 1 because

$$Q = 1$$

Control Inputs

To control the bit stored in the latch, we can add the inputs shown in Fig. 7-1*c*. These control inputs will be either low (0 V) or high (+5 V). A high *set* input S forces the left transistor to saturate. As soon as the left transistor saturates, the overall circuit latches and

$$Q = 1$$

Once set, the output will remain a 1 even though the S input goes back to 0 V.

A high *reset* input R drives the right transistor into saturation. Once this happens, the circuit latches and

$$Q = 0$$

The output stays latched in the 0 state, even though the R input returns to a low.

In Fig. 7-1*c*, Q represents the stored bit. A complementary output $\overline{Q}$ is available from the collector of the left transistor This may or may not be used, depending on the application

Truth Table

Table 7-1 summarizes the operation of the transistor latch. With both control inputs low, no change can occur in the output and the circuit remains latched in its last state. This condition is called the *inactive state* because nothing changes.

TABLE 7-1. TRANSISTOR LATCH

R	*S*	*Q*	Comments
0	0	NC	No change
0	1	1	Set
1	0	0	Reset
1	1	*	Race

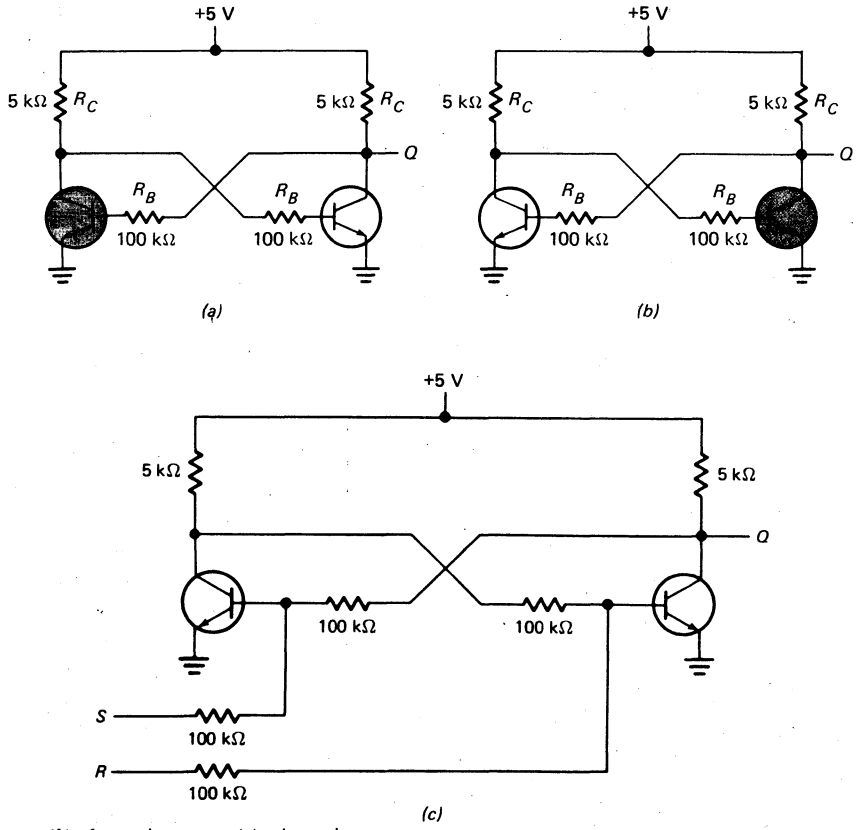


Fig. 7-1 (*a*) Latched state; (*b*) alternative state; (*c*) trigger inputs.

When R is low and S is high, the circuit sets the Q output to a high. On the other hand, if R is high and S is low, the Q output resets to a low.

Race Condition

Look at the last entry in Table 7-1. R and S are high simultaneously. This is called a *race condition;* it is never used because it leads to unpredictable operation.

Here's why. If both control inputs are high, both transistors saturate. When the R and S inputs return to low, both transistors try to come out of saturation. It is a race between the transistors to see which one desaturates first. The faster transistor (the one with the shorter saturation delay time) will win the race and latch the circuit. If the faster transistor is on the left side of Fig. 7-1*c*, the Q output will be low. If the faster transistor is on the right side, the Q output will go high. In mass production, either transistor can be faster; therefore, the Q output is unpredictable. This is why the race condition must be avoided.

Here's how to recognize a race condition. If simultaneously changing both inputs to a memory element leads to an unpredictable output, you've got a race condition. With the transistor latch, $R = 1$ and $S = 1$ is a race condition because simultaneously returning R and S to 0 forces Q into a random state.

From now on, an asterisk in a truth table (see Table 7-1) indicates a race condition, sometimes called a forbidden or invalid state.

NOR Latches

A discrete circuit like Fig. 7-1*c* is rarely used because we are in the age of integrated circuits. Nowadays, you build RS latches with NOR gates or NAND gates.

Figure 7-2*a* shows how it's done with NOR gates. Figure 7-2*b* is the De Morgan equivalent. As shown in Table 7-2, a low R and a low S give us the inactive state; the circuit stores or remembers. A low R and a high S represent the set state, while a high R and a low S give the reset state. Finally, a high R and a high S produce a race condition; therefore, we must avoid $R = 1$ and $S = 1$ when using a NOR latch.

Figure 7-2*c* is a *timing diagram;* it shows how the input signals interact to produce the output signal. As you see, the Q output goes high when S goes high. Q remains high after S goes low. Q returns to low when R goes high, and stays low after R returns to low.

TABLE 7-2. NOR LATCH

R	S	Q	Comment
0	0	NC	No change
0	1	1	Set
1	0	0	Reset
1	1	*	Race

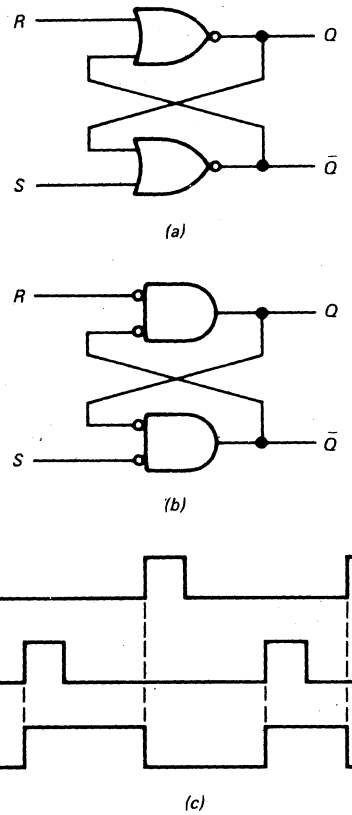


Fig. 7-2 (*a*) NOR latch; (*b*) De Morgan equivalent; (*c*) timing diagram.

TABLE 7-3. NAND LATCH

R	S	Q	Comment
0	0	*	Race
0	1	1	Set
1	0	0	Reset
1	1	NC	No change

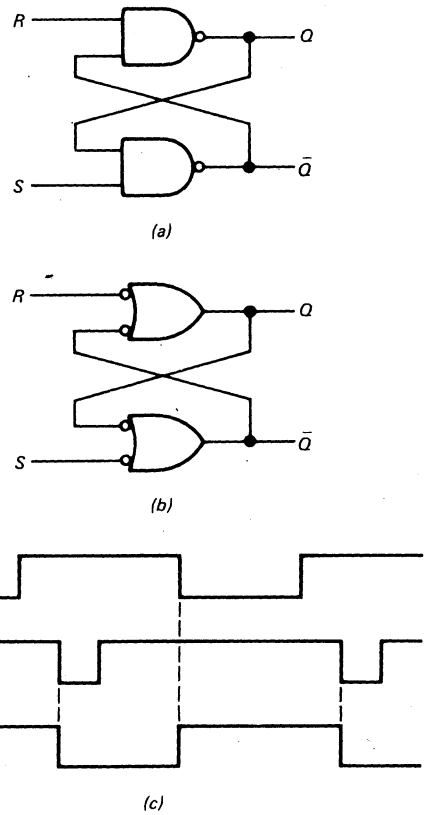


Fig. 7-3 (*a*) NAND latch; (*b*) De Morgan equivalent; (*c*) timing diagram.

NAND Latches

If you prefer using NAND gates, you can build an *RS* latch as shown in Fig. 7-3*a*. Sometimes it is convenient to draw the De Morgan equivalent shown in Fig. 7-3*b*. In either case, a low *R* and a high *S* set *Q* to high; a high *R* and a low *S* reset *Q* to low.

Because of the NAND-gate inversion, the inactive and race conditions are reversed. In other words, $R = 1$ and $S = 1$ becomes the inactive state; $R = 0$ and $S = 0$ becomes the race condition (see Table 7-3). Therefore, whenever you use a NAND latch, you must avoid having both inputs low at the same time. (To remember the race condition for a NAND latch, glance at Fig. 7-3*b*. If $R = 0$ and $S = 0$, then $Q = 1$ and $\overline{Q} = 1$; both outputs are the same, indicating an invalid condition.)

Figure 7-3*c* shows the timing diagram for a NAND latch. *R* and *S* are normally high to avoid the race condition. Only one of them goes low at any time. As you see, the *Q* output goes high whenever *R* goes low; the *Q* output goes low whenever *S* goes low.

Switch Debouncers

RS latches are often used as *switch debouncers*. Whenever you throw a switch from the open to the closed position, the contacts bounce and the switch alternately makes and breaks for a few milliseconds before finally settling in the closed position. One way to eliminate the effects of contact bounce is to use an *RS* latch in conjunction with the switch. The following example explains the idea.

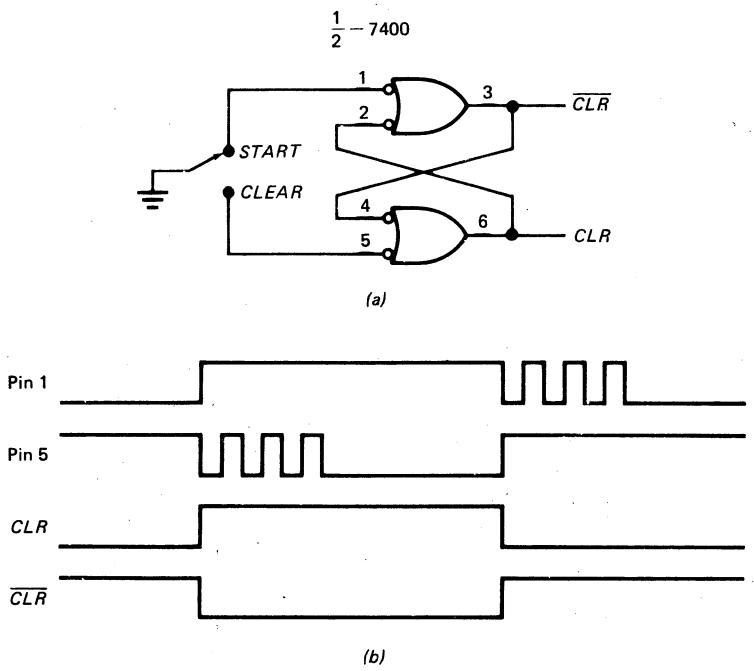


Fig. 7-4 Switch debouncer.

EXAMPLE 7-1

Figure 7-4*a* shows a switch debouncer. What does it do?

SOLUTION

As discussed in Chap. 4, floating TTL inputs are equivalent to high inputs. With the switch in the START position, pin 1 is low and pin 5 is high; therefore, $\overline{CLR}$ is high and *CLR* is low. When the switch is thrown to the CLEAR position, pin 1 goes high, as shown in Fig. 7-4*b*. Because of contact bounce, pin 5 goes alternately low and high for a few milliseconds before settling in the low state, symbolized by the ideal pulses of Fig. 7-4*b*.The first time pin 5 goes low, the latch sets, *CLR* going high and $\overline{CLR}$ going low. Subsequent bounces have no effect on *CLR* and $\overline{CLR}$ because the latch stays set.

Similarly, when the switch is thrown back to START, pin 1 bounces low and high for a while. The first time pin 1 goes low, *CLR* goes back to low and $\overline{CLR}$ to high. Later bounces have no effect on *CLR* and $\overline{CLR}$.

Registers need clean signals like *CLR* and $\overline{CLR}$ of Fig. 7-4*b* to operate properly. If the bouncing signals on pins 1 and 5 drove the registers, the operation would be erratic. This is why you often see *RS* latches used as switch debouncers.

7-2 LEVEL CLOCKING

Computers use thousands of flip-flops. To coordinate the overall action, a square-wave signal called the *clock* is sent to each flip-flop. This signal prevents the flip-flops from changing states until the right time.

Clocked Latch

In Fig. 7-5*a* a pair of NAND gates drive a NAND latch. *S* and *R* signals drive the input gates. To avoid confusion, the inner control signals are labeled *R'* and *S'*. The NAND latch works as previously described; a low *R'* and a high *S'* set *Q* to 1, whereas a high *R'* and a low *S'* reset *Q* to 0. Furthermore, a low *R'* and *S'* represent the race condition; therefore, *R'* and *S'* are normally high when the latch is inactive. Because of the inversion through the input NAND gates, the *S* input has to drive the upper NAND input and the *R* input must drive the lower NAND input.

Double Inversions Cancel

When analyzing the operation of this and similar circuits, remember that a double inversion (two bubbles in a series path) cancels out; this makes it appear as though two AND gates drove OR gates, as shown in Fig. 7-5*b*. In this way, you can see at a glance that a high *S* and high *CLK* force

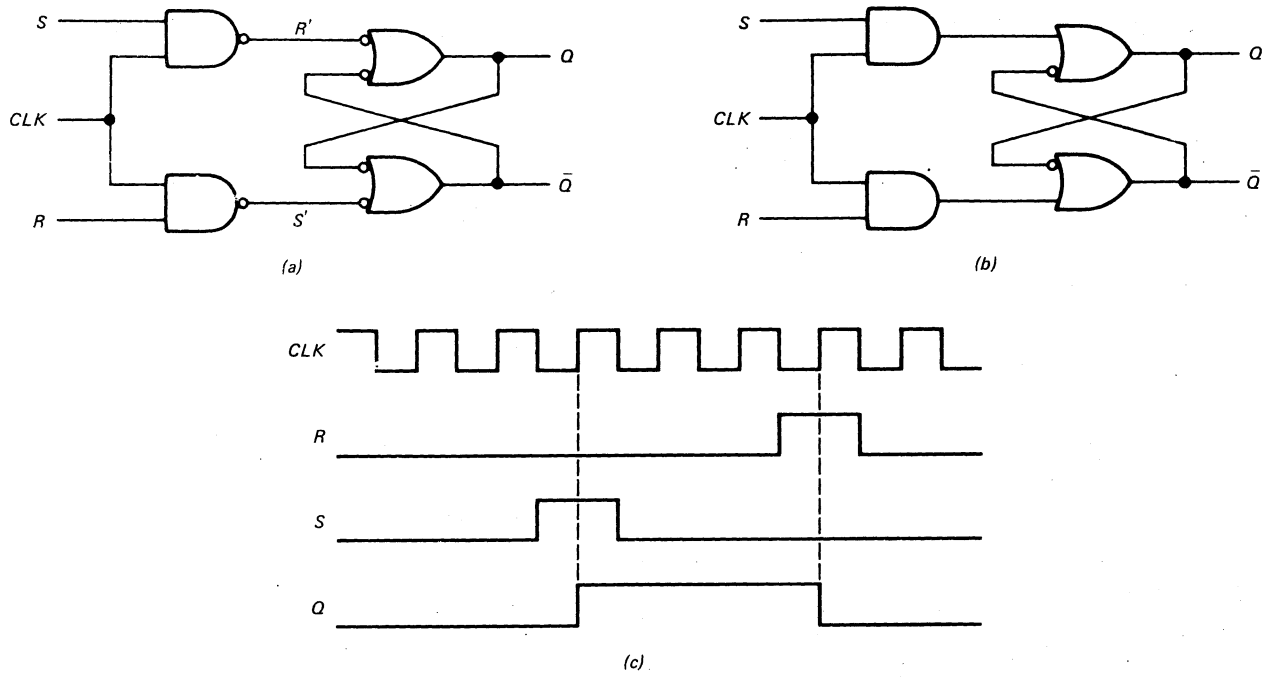


Fig. 7-5 (*a*) Clocked latch; (*b*) equivalent circuit; (*c*) timing diagram.

Q to go high. In other words, even though you are looking at Fig. 7-5*a*, in your mind you should see Fig. 7-5*b*.

Positive Clocking

In Fig. 7-5*a* the clock is a square-wave signal. Because the clock (abbreviated CLK) drives both NAND gates, a low *CLK* prevents *S* and *R* from controlling the latch. If a high *S* and a low *R* drive the gate inputs, the latch must wait until the clock goes high before *Q* can be set to 1. Similarly, given a low *S* and a high *R*, the latch must wait for a high *CLK* before *Q* can reset to 0. This is an example of *positive clocking*, making a latch wait until the clock signal is high before the output can change.

Negative clocking is similar. Visualize an inverter between CLK and the input gates of Fig. 7-5*a*. In this case, the latch must wait until *CLK* is low before the output can change.

Positive and negative clocking are often called *level clocking* because the flip-flop responds to the level (high or low) of the clock signal. Level clocking is the simplest way to control flip-flops with a clock. Later, we will discuss more advanced methods called edge triggering and master-slave clocking.

Race Condition

What about the race condition? When the clock is low in Fig. 7-5*a*, *R'* and *S'* are high, which is a stable condition. The only way to get a race condition is to have a high *CLK*, high *R*, and high *S*. Therefore, normal operation of this circuit requires that *R* and *S* never both be high when the clock goes high.

Timing Diagram and Truth Table

Figure 7-5*c* shows the timing diagram. *Q* goes high when *S* is high and *CLK* goes high. *Q* returns to the low state when *R* is high and *CLK* goes high. Using a common *CLK* signal to drive many flip-flops allows us to synchronize the operation of the different sections of a computer.

Table 7-4 summarizes the operation of the clocked NAND latch. When the clock is low, the output is latched in its last state. When the clock goes high, the circuit will set if *S* is high or reset if *R* is high. *CLK*, *R*, and *S* all high is a race condition, which is never used deliberately.

TABLE 7-4. CLOCKED NAND LATCH

CLK	*R*	*S*	*Q*
0	0	0	NC
0	0	1	NC
0	1	0	NC
0	1	1	NC
1	0	0	NC
1	0	1	1
1	1	0	0
1	1	1	*

7-3 *D* LATCHES

Since the *RS* flip-flop is susceptible to a race condition, we will modify the design to eliminate the possibility of a race condition. The result is a new kind of flip-flop known as a *D latch*.

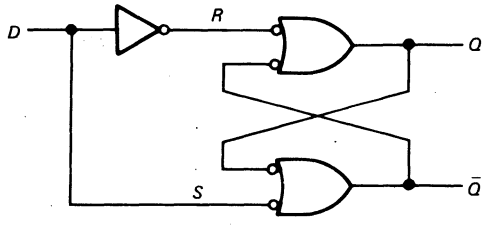


Fig. 7-6 *D* latch.

Unclocked

Figure 7-6 shows one way to build a *D* latch. Because of the inverter, data bit *D* drives the *S* input of a NAND latch and the complement $\overline{D}$ drives the *R* input. Therefore, a high *D* sets the latch, and a low *D* resets it. Table 7-5 summarizes the operation of the *D* latch. Especially important, there is no race condition in this truth table. The inverter guarantees that *S* and *R* will always be in opposite states; therefore, it's impossible to set up a race condition in the *D* latch.

The *D* latch of Fig. 7-6 is unclocked; it will set or reset as soon as *D* goes high or low. An unclocked flip-flop like this is almost never used.

TABLE 7-5. UNCLOCKED *D* LATCH

D	*Q*
0	0
1	1

TABLE 7-6. CLOCKED *D* LATCH

CLK	*D*	*Q*
0	X	NC
1	0	0
1	1	1

Clocked

Figure 7-7*a* is level-clocked. A low *CLK* disables the input gates and prevents the latch from changing states. In other words, while *CLK* is low, the latch is in the inactive state and the circuit stores or remembers. When *CLK* is high, *D* controls the output. A high *D* sets the latch, while a low *D* resets it.

Table 7-6 summarizes the operation. X represents a don't-care condition; it stands for either 0 or 1. While *CLK* is low, the output cannot change, no matter what *D* is. When *CLK* is high, however, the output equals the input

$$Q = D$$

Figure 7-7*b* shows a timing diagram. If the clock is low, the circuit is latched and the *Q* output cannot be changed. While the clock is high, however, *Q* equals *D*; when *D* goes high, *Q* goes high; when *D* goes low, *Q* goes low. The latch is *transparent*, meaning that the output follows the value of *D* while the clock is high.

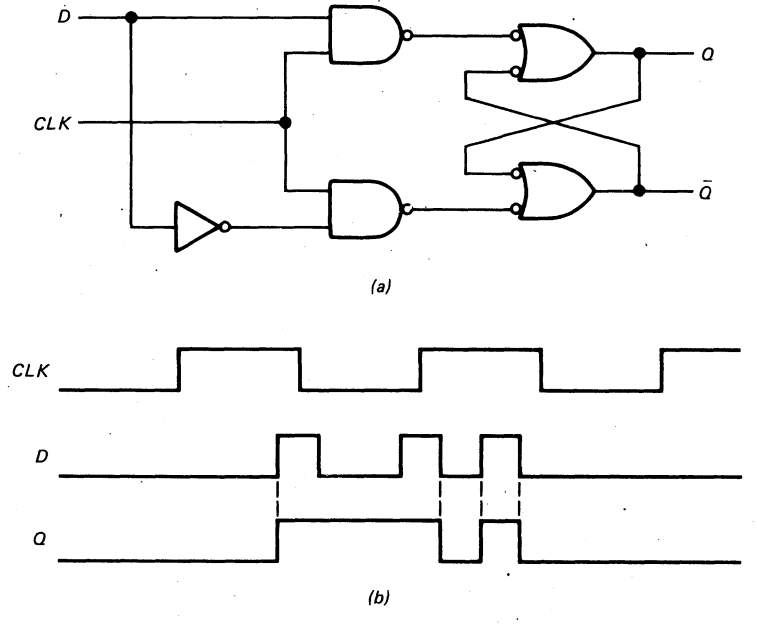


Fig. 7-7 Clocked *D* latch.

Disadvantage

Because the *D* latch is level-clocked, it has a serious disadvantage. While the clock is high, the output follows the value of *D*. Transparent latches may be all right in some applications but not in the computer circuits we will be discussing. To be truly useful, the circuit of Fig. 7-7*a* needs a slight modification.

7-4 EDGE-TRIGGERED *D* FLIP-FLOPS

Now we're ready to talk about the most common type of *D* flip-flop. What a practical computer needs is a *D* flip-flop that samples the data bit at a unique instant.

Edge Triggering

Figure 7-8*a* shows an *RC* circuit at the input of a *D* flip-flop. By deliberate design, the *RC* time constant is much smaller than the clock's pulse width. Because of this, the capacitor can charge fully when *CLK* goes high; this exponential charging produces a narrow positive voltage spike across the resistor. Later, the trailing edge of the clock pulse results in a narrow negative spike.

The narrow positive spike enables the input gates for an instant; the narrow negative spike does nothing. The effect is to activate the input gates during the positive spike, equivalent to sampling the value of *D* for an instant. At this unique time, *D* and its complement hit the flip-flop inputs, forcing *Q* to set or reset.

TABLE 7-7. EDGE-TRIGGERED *D* FLIP-FLOP

CLK	*D*	*Q*
0	X	NC
1	X	NC
↓	X	NC
↑	0	0
↑	1	1

This kind of operation is called *edge triggering* because the flip-flop responds only when the clock is changing states. The triggering in Fig. 7-8*a* occurs on the positive-going edge of the clock; this is why it's referred to as *positive-edge triggering*.

Figure 7-8*b* illustrates the action. The crucial idea is that the output changes only on the rising edge of the clock. In other words, data is stored only on the positive-going edge.

Table 7-7 summarizes the operation of the positive-edge-triggered *D* flip-flop. The up and down arrows represent the rising and falling edges of the clock. The first three entries indicate that there's no output change when the clock is low, high, or on its negative edge. The last two entries indicate an output change on the positive edge of the clock. In other words, input data *D* is stored only on the positive-going edge of the clock.

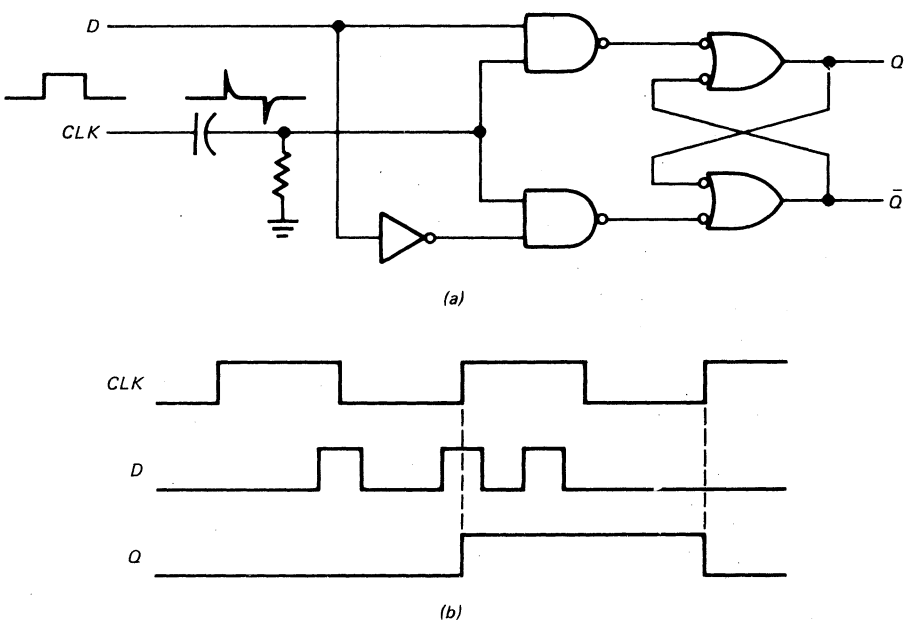


Fig. 7-8 Edge-triggered *D* flip-flop.

Edge Triggering versus Level Clocking

When a circuit is edge-triggered, the output can change only on the rising (or falling) edge of the clock. But when the circuit is level-clocked, the output can change while the clock is high (or low). With edge triggering, the output can change only at one instant during the clock cycle; with level clocking, the output can change during an entire half cycle of the clock.

Preset and Clear

When power is first applied, flip-flops come up in random states. To get some computers started, an operator has to push a master reset button. This sends a *clear* (reset) signal to all flip-flops. Also, it is necessary in some computers to *preset* (synonymous with "set") certain flip-flops before a computer run.

Figure 7-9 shows how to include both functions in a *D* flip-flop. The edge triggering is the same as previously described. In addition, the AND gates allow us to slip in a low *PRESET* or low *CLEAR* when desired. A low *PRESET* forces *Q* to equal 1; a low *CLEAR* resets *Q* to 0.

Table 7-8 summarizes the circuit action. When *PRESET* and *CLEAR* are both low, we get a race condition; therefore, *PRESET* and *CLEAR* should be kept high when inactive. Take *PRESET* low by itself and you set the flip-flop; take *CLEAR* low by itself and you reset the flip-flop. As shown in the remaining entries, the output changes only on the positive-going edge of the clock.

Preset is sometimes called *direct set*, and clear is sometimes called *direct reset*. The word "direct" means unclocked. For instance, the clear signal may come from a push button; regardless of what the clock is doing, the output will reset when the operator pushes the clear button.

The preset and clear inputs override the other inputs; they have first priority. For example, when *PRESET* goes low, the *Q* output goes high and stays there no matter what the *D* and *CLK* inputs are doing. The output will remain high as long as *PRESET* is low. Therefore, the normal procedure in presetting is to take the *PRESET* low temporarily, then return it to high. Similarly, for the clear function: take *CLEAR* low briefly to reset the flip-flop, then take it back to high to allow the circuit to operate.

TABLE 7-8. *D* FLIP-FLOP WITH PRESET AND CLEAR

PRESET	*CLEAR*	*CLK*	*D*	*Q*
0	0	X	X	*
0	1	X	X	1
1	0	X	X	0
1	1	0	X	NC
1	1	1	X	NC
1	1	↓	X	NC
1	1	↑	0	0
1	1	↑	1	1

Direct-Coupled Edge-Triggered *D* Flip-Flop

Integrated *D* flip-flops do not use *RC* circuits to get narrow spikes because capacitors are difficult to fabricate on a chip. Instead, a variety of direct-coupled designs is used. As an example, Fig. 7-10 shows a positive-edge-triggered *D* flip-flop. This direct-coupled circuit has no capacitors, only NAND gates. The analysis is too long and complicated to go into here, but the idea is the same as previously discussed. The circuit responds only during the brief instant the clock switches from low to high. That is, data bit *D* is stored only on the positive-going edge of the clock.

Logic Symbol

Figure 7-11 is the symbol of a positive-edge-triggered *D* flip-flop. The *CLK* input has a small triangle, a reminder of the edge triggering. When you see this schematic symbol, remember what it means: the *D* input is stored on the rising edge of the clock.

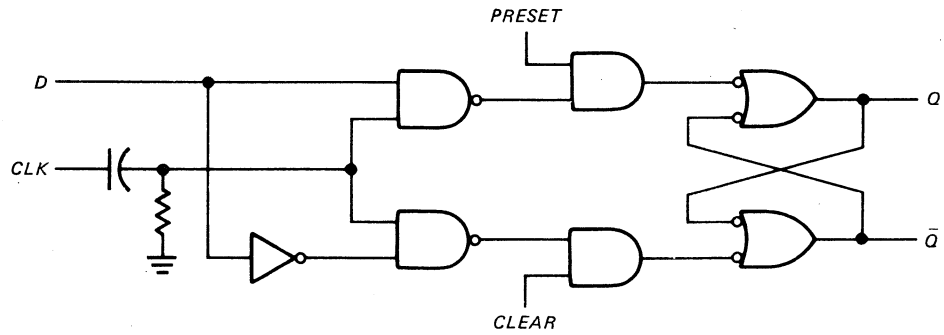


Fig. 7-9 Edge-triggered *D* flip-flop with preset and clear.

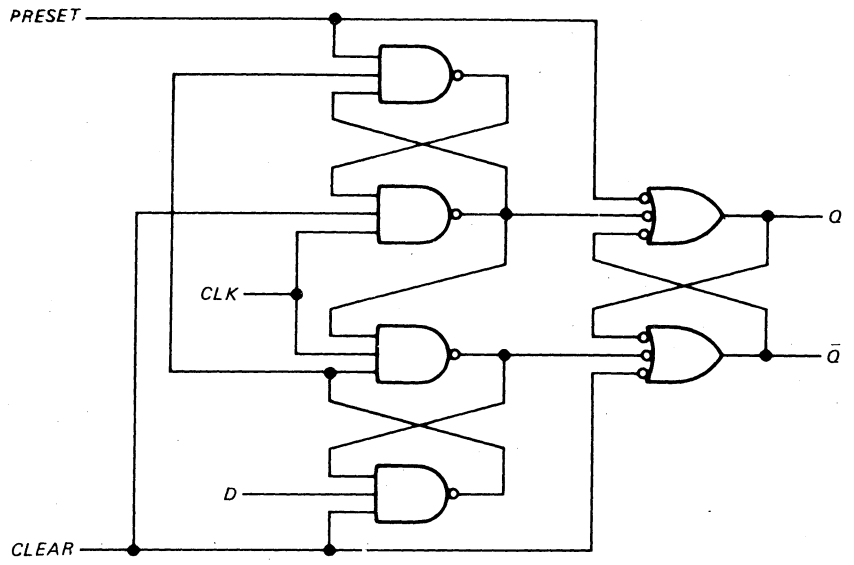


Fig. 7-10 Direct-coupled edge-triggered *D* flip-flop.

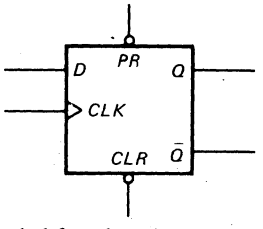


Fig. 7-11 Logic symbol for edge-triggered *D* flip-flop.

Figure 7-11 also includes preset (*PR*) and clear (*CLR*) inputs. The bubbles indicate an *active low state*. In other words, the preset and clear inputs are high when inactive. To preset the flip-flop, the preset input must go low temporarily and then be returned to high. Similarly, to reset the flip-flop, the clear input must go low, then back to high.

The same idea applies to circuits discussed later. A bubble at an input means an active low state: the input has to go low to produce an effect. When no bubble is present, the input has to go high to have an effect.

Propagation Delay Time

Diodes and transistors cannot switch states instantaneously. It always takes a small amount of time to turn a diode on or off. Likewise, it takes a time for a transistor to switch from saturation to cutoff or vice versa. For bipolar diodes and transistors, switching time is in the nanosecond region.

Switching time is the main cause of *propagation delay time* t_p. This represents the amount of time it takes for the output of a gate or flip-flop to change states. For instance, if the data sheet of a *D* flip-flop indicates a t_p of 10 ns, it takes approximately 10 ns for *Q* to change states after *D* has been sampled by the clock edge.

Propagation delay time is so small that it's negligible in many applications, but in high-speed circuits you have to take it into account. If a flip-flop has a t_p of 10 ns, this means that you have to wait 10 ns before the output can trigger another circuit.

Setup Time

Stray capacitance at the *D* input (plus other factors) makes it necessary for data bit *D* to be at the input before the *CLK* edge arrives. The *setup time* t_{setup} is the minimum length of time the data bit must be present before the *CLK* edge hits.

For instance, if the data sheet of a *D* flip-flop indicates a t_{setup} of 15 ns, the data bit to be stored must be at the *D* input at least 15 ns before the *CLK* edge arrives; otherwise, the IC manufacturer does not guarantee correct sampling and storing.

Hold Time

Furthermore, data bit *D* has to be held long enough for the internal transistors to switch states. Only after the transition is assured can we allow data bit *D* to change. *Hold time* t_{hold} is the minimum length of time the data bit must be present after the *CLK* edge has struck.

For example, if t_{setup} is 15 ns and t_{hold} is 5 ns, the data bit has to be at the *D* input at least 15 ns before the *CLK* edge arrives and held at least 5 ns after the *CLK* edge hits.

7-5 EDGE-TRIGGERED JK FLIP-FLOPS

The next chapter shows you how to build a counter, the electronic equivalent of a binary odometer. When it comes to circuits that count, the *JK flip-flop* is the ideal memory element to use.

Circuit

Figure 7-12*a* shows one way to build a *JK* flip-flop. As before, an *RC* circuit with a short time constant converts the rectangular *CLK* pulse to narrow spikes. Because of the double inversion through the NAND gates, the circuit is positive-edge-triggered. In other words, the input gates are enabled only on the rising edge of the clock.

Inactive

The J and K inputs are control inputs; they determine what the circuit will do on the positive clock edge. When J and K are low, both input gates are disabled and the circuit is inactive at all times including the rising edge of the clock.

Reset

When J is low and K is high, the upper gate is disabled; so there's no way to set the flip-flop. The only possibility is reset. When Q is high, the lower gate passes a reset trigger as soon as the positive clock edge arrives. This forces Q to become low. Therefore, $J = 0$ and $K = 1$ means that a rising clock edge resets the flip-flop.

Set

When J is high and K is low, the lower gate is disabled; so it's impossible to reset the flip-flop. But you can set the flip-flop as follows. When Q is low, $\overline{Q}$ is high; therefore, the upper gate passes a set trigger on the positive clock edge. This drives Q into the high state. That is, $J = 1$ and $K = 0$ means that the next positive clock edge sets the flip-flop.

Toggle

When J and K are both high, it is possible to set or reset the flip-flop, depending on the current state of the output. If Q is high, the lower gate passes a reset trigger on the

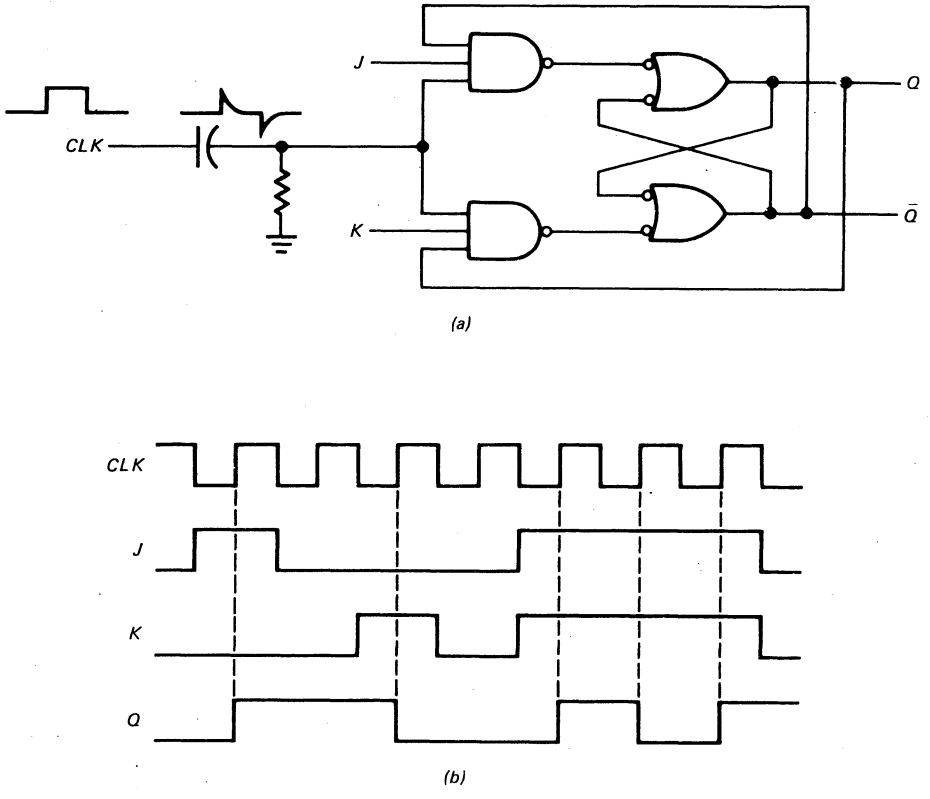


Fig. 7-12 (*a*) Edge-triggered *JK* flip-flop; (*b*) timing diagram.

TABLE 7-9. POSITIVE-EDGE-TRIGGERED *JK* FLIP-FLOP

CLK	*J*	*K*	*Q*
0	X	X	NC
1	X	X	NC
↓	X	X	NC
X	0	0	NC
↑	0	1	0
↑	1	0	1
↑	1	1	Toggle

next positive clock edge. On the other hand, when Q is low, the upper gate passes a set trigger on the next positive clock edge. Either way, Q changes to the complement of the last state. Therefore, $J = 1$ and $K = 1$ means that the flip-flop will *toggle* on the next positive clock edge. ("Toggle" means switch to opposite state.)

Timing Diagram

The timing diagram of Fig. 7-12*b* is a visual summary of the action. When J is high and K is low, the rising clock edge sets Q to high. On the other hand, when J is low and K is high, the rising clock edge resets Q to low. When J and K are high simultaneously, the output toggles on each rising clock edge.

Truth Table

Table 7-9 summarizes the operation. The circuit is inactive when the clock is low, high, or on its negative edge. Likewise, the circuit is inactive when J and K are both low. Output changes occur only on the rising edge of the clock, as indicated by the last three entries of the table. The output either resets, sets, or toggles.

Racing

The *JK* flip-flop shown in Fig. 7-12*a* has to be edge-triggered to avoid oscillations. Why? Assume that the circuit is level-clocked. In other words, assume that we remove the *RC* circuit and run the clock straight into the gates. With a high J, high K, and high *CLK*, the output will toggle. New outputs are then fed back to the input gates. After two propagation times (input and output gates), the output toggles again. And once more, new outputs return to the input gates. In this way, the output can toggle repeatedly as long as the clock is high. That is, we get oscillations during the positive half cycle of the clock. Toggling more than once during a clock cycle is called *racing*.

Now assume that we put the *RC* circuit back in and return to edge triggering. Propagation delay time prevents the *JK* flip-flop from racing. Here's why. In Fig. 7-12*a* the outputs change after the positive clock edge has struck. By the time the new Q and $\overline{Q}$ signals return to the input gates, the positive spikes have decayed to zero. This is why we get only one toggle during each clock cycle.

For instance, if the total propagation delay time from input to output is 20 ns, the outputs change approximately 20 ns after the rising edge of the clock. If the spikes are narrower than 20 ns, the returning Q and $\overline{Q}$ arrive too late to cause false triggering.

Symbols

As previously mentioned, capacitors are too difficult to fabricate on a chip. This is why manufacturers prefer direct-coupled designs for edge-triggered *JK* flip-flops. Such designs are too complicated to reproduce here, but you can find them in manufacturers' IC data books.

Figure 7-13*a* is the standard symbol for a positive-edge-triggered *JK* flip-flop of any design.

Figure 7-13*b* is the symbol for a *JK* flip-flop with the preset and clear functions. As usual, *PR* and *CLR* have active low states. This means that they are normally high and taken low temporarily to preset or clear the circuit.

Figure 7-13*c* is another commercially available *JK* flip-flop. The bubble on the clock input is the standard way to indicate negative-edge triggering. As shown in Table 7-10, the output can change only on the *falling* edge of the clock. The timing diagram of Fig. 7-13*d* emphasizes this negative-edge triggering.

7-6 JK MASTER-SLAVE FLIP-FLOP

Figure 7-14 shows a *JK master-slave* flip-flop, another way to avoid racing. A master-slave flip-flop is a combination of two clocked latches; the first is called the *master*, and the second is the *slave*. Notice that the master is positively

TABLE 7-10. NEGATIVE-EDGE-TRIGGERED *JK* FLIP-FLOP

CLK	*J*	*K*	*Q*
0	X	X	NC
1	X	X	NC
↑	X	X	NC
X	0	0	NC
↓	0	1	0
↓	1	0	1
↓	1	1	Toggle

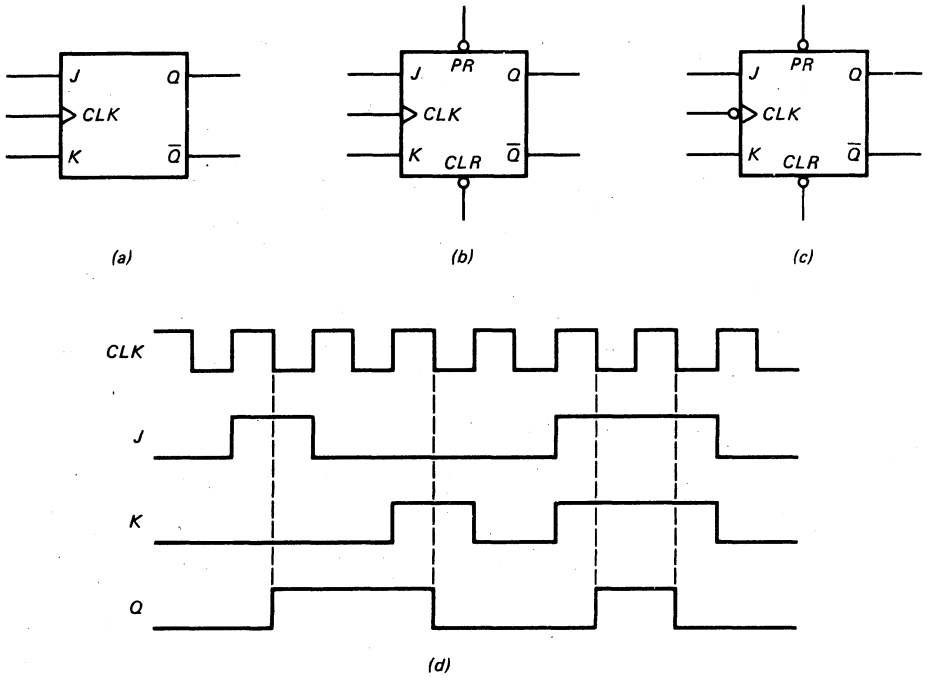


Fig. 7-13 (*a*) Positive-edge triggering; (*b*) active low preset and clear; (*c*) negative-edge triggering; (*d*) timing diagram.

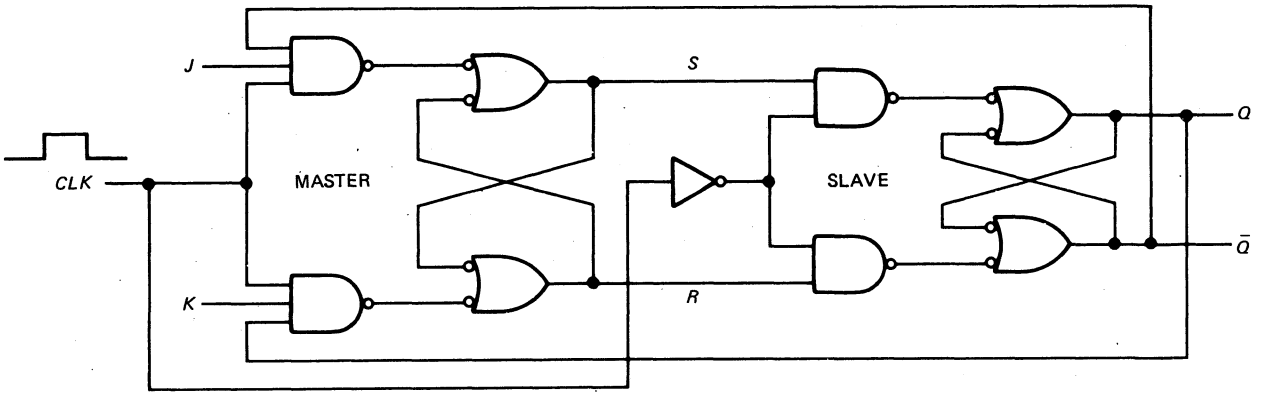


Fig. 7-14 Master-slave *JK* flip-flop.

clocked but the slave is negatively clocked. This implies the following:

1. While the clock is high, the master is active and the slave is inactive.
2. While the clock is low, the master is inactive and the slave is active.

Set

To start the analysis, let's assume low Q and high $\overline{Q}$. For an input condition of high J, low K, and high CLK, the master goes into the set state, producing high S and low R. Nothing happens to the Q and $\overline{Q}$ outputs because the slave is inactive while the clock is high. When the clock goes low, however, the high S and low R force the slave into the set state, producing a high Q and a low $\overline{Q}$.

There are two distinct steps in setting the final Q output. First, the master is set while the clock is high. Second, the slave is set while the clock is low. This action is sometimes called *cocking* and *triggering*. You cock the master during the positive half cycle of the clock, and you trigger the slave during the negative half cycle of the clock.

Reset

When the slave is set, Q is high and $\overline{Q}$ is low. For the input condition of low J, high K, and high CLK, the master will reset, forcing S to go low and R to go high. Again, no changes can occur in Q and $\overline{Q}$ because the slave is inactive while the clock is high. When the clock returns to the low state, the low S and high R force the slave to reset; this forces Q to go low and $\overline{Q}$ to go high.

Again, notice the cocking and triggering. This is the key idea behind the master-slave flip-flop. Every action of the master with a high CLK is copied by the slave when the clock goes low.

Toggle

If the J and K inputs are both high, the master toggles once while the clock is high; the slave then toggles once when the clock goes low. No matter what the master does, the slave copies it. If the master toggles into the set state, the slave toggles into the set state. If the master toggles into the reset state, the slave toggles into the reset state.

Level Clocking

The master-slave flip-flop is level-clocked in Fig. 7-14. While the clock is high, therefore, any changes in J and K can affect the S and R outputs. For this reason, you normally keep J and K constant during the positive half cycle of the clock. After the clock goes low, the master becomes inactive and you can allow J and K to change.

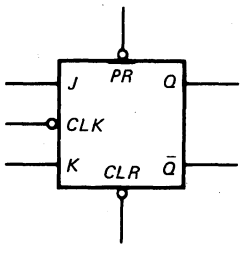


Fig. 7-15 (*a*) Symbol for master-slave *JK* flip-flop; (*b*) timing diagram.

Symbol

Figure 7-15 shows the symbol for a *JK* master-slave flip-flop with preset and clear functions. The bubble on the *CLK* input reminds us that the output changes when the clock goes low.

Truth Table

Table 7-11 summarizes the operation of a *JK* master-slave flip-flop. A low *PR* and low *CLR* produces a race condition; therefore, *PR* and *CLR* are normally kept at a high voltage when inactive. To clear, you take *CLR* low; to preset, you take *PR* low. In either case, you return them to high when ready to run.

TABLE 7-11. MASTER-SLAVE FLIP-FLOP

PR	*CLR*	*CLK*	*J*	*K*	*Q*
0	0	X	X	X	*
0	1	X	X	X	1
1	0	X	X	X	0
1	1	X	0	0	NC
1	1	⊓	0	1	0
1	1	⊓	1	0	1
1	1	⊓	1	1	Toggle

As before, low J and low K produce an inactive state, regardless of the what the clock is doing. If K goes high by itself, the next clock pulse resets the flip-flop. If J goes high by itself, the next clock pulse sets the flip-flop. When J and K are both high, each clock pulse produces one toggle.

EXAMPLE 7-2

Figure 7-16*a* shows a *clock generator*. What does it do when $\overline{HLT}$ is high?

SOLUTION

To begin with, the 555 is an IC that can generate a rectangular output when connected as shown in Fig. 7-16*a*. The frequency of the output is

$$f = \frac{1.44}{(R_A + 2R_B)C}$$

The duty cycle (ratio of high state to period) is

$$D = \frac{R_A + R_B}{R_A + 2R_B}$$

With the values shown in Fig. 7-16*a* the frequency of the output is

$$f = \frac{1.44}{(36\,\text{k}\Omega + 36\,\text{k}\Omega)(0.01\,\mu\text{F})} = 2\,\text{kHz}$$

and the duty cycle is

$$D = \frac{36\,\text{k}\Omega + 18\,\text{k}\Omega}{36\,\text{k}\Omega + 36\,\text{k}\Omega} = 0.75$$

which is equivalent to 75 percent.

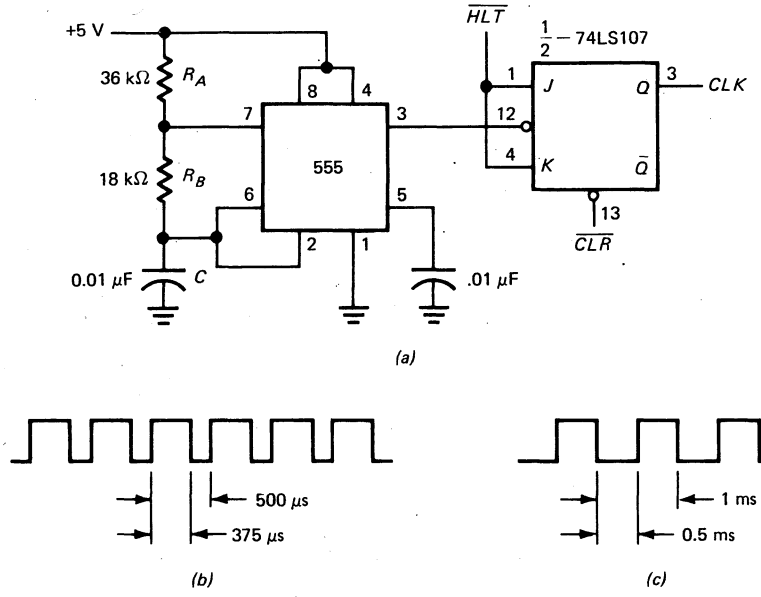


Fig. 7-16 Clock generator: (*a*) circuit; (*b*) 555 output; (*c*) *JK* flip-flop output.

Figure 7-16*b* illustrates how the output (pin 3) of the 555 looks. Note how the signal is high for 75 percent of the cycle. This unsymmetrical output drives the clock input of a *JK* master-slave flip-flop.

The *JK* master-slave flip-flop toggles once per input cycle; therefore, its output has a frequency of 1 kHz and a duty cycle of 50 percent. One of the reasons for using the flip-flop is to get the symmetrical output shown in Fig. 7-16*c*.

Another reason for using the flip-flop is to control the starting phase of the clock. A computer run starts with $\overline{CLR}$ going momentarily low, then back to high. This resets the flip-flop, forcing *CLK* to go low. Therefore, the starting phase of the *CLK* signal is always low. You will see the clock generator of Fig. 7-16*a* again in Chap. 10; remember that the *CLK* signal has a frequency of 1 kHz, a duty cycle of 50 percent, and starting phase of low.

GLOSSARY

contact bounce The making and breaking of contacts for a few milliseconds after a switch closes.

edge triggering Changing the output state of a flip-flop on the rising or falling edge of a clock pulse.

flip-flop A two-state circuit that can remain in either state indefinitely. Also called a bistable multivibrator. An external trigger can change the output state.

hold time The minimum amount of time the input signals must be held constant after the clock edge has struck. After a clock edge strikes a flip-flop, the internal transistors need time to change from one state to another. The input control signals (*D*, or *J* and *K*) must be held constant while these internal transistors are switching over.

latch The simplest type of flip-flop, consisting of two cross-coupled NAND or NOR latches.

level clocking A type of triggering in which the output of a flip-flop responds to the level (high or low) of the clock signal. With positive level clocking, for example, the output can change at any time during the positive half cycle.

master-slave triggering A type of triggering using two cascaded latches called the master and the slave. The master is cocked during the positive half cycle of the clock, and the slave is triggered during the negative half cycle.

propagation delay time The time it takes for the output of a gate or flip-flop to change after the inputs have changed.

race condition An undesirable condition which may exist in a system when two or more inputs change simultaneously. If the final output depends on which input changes first, a race condition exists.

setup time The minimum amount of time the inputs to a flip-flop must be present before the clock edge arrives.

toggle Change of the output to the opposite state in a *JK* flip-flop.

SELF-TESTING REVIEW

Read each of the following and provide the missing words. Answers appear at the beginning of the next question.

1. A flip-flop is a _____ element that stores a binary digit as a low or high voltage. With an *RS* latch a high *S* and a low *R* sets the output to _____; a low *S* and a high *R* _____ the output to low.
2. (*memory, high, reset*) With a NAND latch a low *R* and a low *S* produce a _____ condition. This is why *R* and *S* are kept high when inactive. One use for latches is switch debouncers; they eliminate the effects of _____ bounce.
3. (*race, contact*) Computers use thousands of flip-flops. To coordinate the overall action, a common signal called the _____ is sent to each flip-flop. With positive clocking the clock signal must be _____ for the flip-flop to respond. Positive and negative clocking are also called level clocking because the flip-flop responds to the _____ of the clock, either high or low.
4. (*clock, high, level*) In a *D* latch, data bit *D* drives the *S* input of a latch, and the complement $\overline{D}$ drives the *R* input; therefore, a high *D* _____ the latch and a low *D* resets it. Since *R* and *S* are always in opposite states in a *D* latch, the _____ condition is impossible.
5. (*sets, race*) With a positive-edge-triggered *D* flip-flop, the data bit is sampled and stored on the _____ edge of the clock pulse. Preset and clear inputs are often called _____ set and _____ reset. These inputs override the other inputs; they have first priority. When preset goes low, the *Q* output goes _____ and stays there no matter what the *D* and *CLK* inputs are doing.
6. (*rising, direct, direct, high*) In a flip-flop, propagation delay time is the amount of time it takes for the _____ to change after the clock edge has struck. Setup time is the amount of time an input signal must be present _____ the clock edge strikes. Hold time is the amount of time an input signal must be present _____ the clock edge strikes.
7. (*output, before, after*) In a positive-edge-triggered *JK* flip-flop, a low *J* and a low *K* produce the _____ state. A high *J* and a high *K* mean that the output will _____ on the rising edge of the clock.
8. (*inactive, toggle*) With a *JK* master-slave flip-flop the master is cocked when the clock is _____, and the slave is triggered when the clock is _____. This type of flip-flop is usually level-clocked instead of edge-triggered. For this reason, *J* and *K* are normally kept _____ while the clock is high.
9. (*high, low, constant*) Since capacitors are too difficult to fabricate on an IC chip, manufacturers rely on various direct-coupled designs for *D* flip-flops and *JK* flip-flops.

PROBLEMS

7-1. The waveforms of Fig. 7-17 drive a clocked *RS* latch (Fig. 7-5*a*). If *Q* is low before time *A*,

a. At what point does *Q* become a 1?
b. When does *Q* reset to 0?

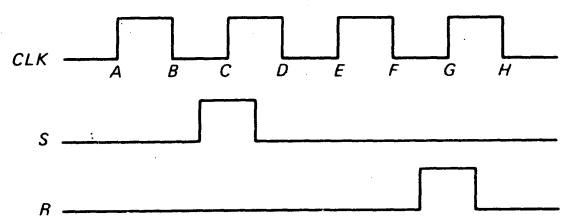


Fig. 7-17

7-2. A *D* flip-flop has these specifications:

$$t_{setup} = 10 \text{ ns}$$
$$t_{hold} = 5 \text{ ns}$$
$$t_p = 30 \text{ ns}$$

a. How far ahead of the rising clock edge must the data bit be applied to the *D* input to ensure correct storage?
b. After the rising clock edge, how long must you wait before letting the data bit change?
c. How long after the rising clock edge will *Q* change?

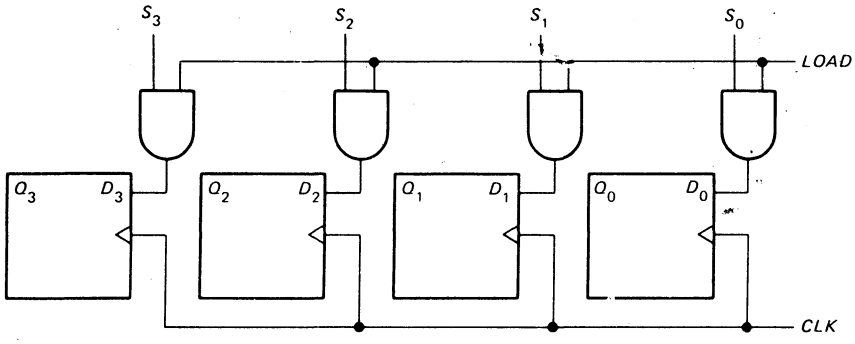


Fig. 7-18

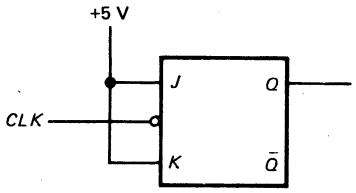


Fig. 7-19

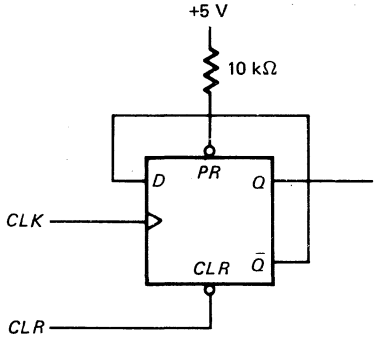


Fig. 7-20

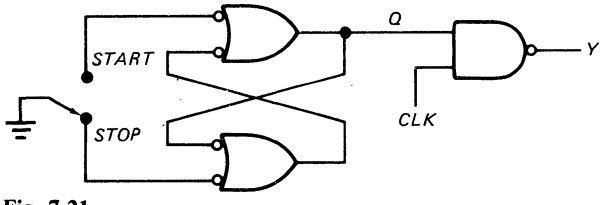


Fig. 7-21

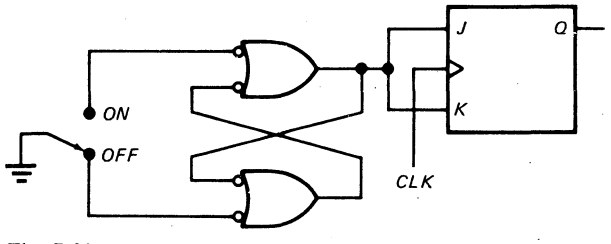


Fig. 7-22

7-3. In Fig. 7-18, the data word to be stored is

$$S = 1001$$

a. If *LOAD* is low, what does Q equal after the positive clock edge?
b. If *LOAD* is high, what does Q equal after the positive clock edge.

7-4. The clock of Fig. 7-19 has a frequency of 1 MHz, and the flip-flop has a propagation delay time of 25 ns.
a. What is the period of the clock?
b. The frequency of the Q output? Its period?
c. How long after the negative clock edge does the Q output change?

7-5. The clock has a frequency of 6 MHz in Fig. 7-19. What is the frequency of the Q output? This circuit is sometimes called a divide-by-2 circuit. Explain why.

7-6. In Fig. 7-20, *CLR* is taken low temporarily, then high. Draw the timing diagram. If the clock has a frequency of 1 MHz, what is the frequency of the Q output? Is this a divide-by-2 circuit?

7-7. Figure 7-21 shows a NAND latch used as a switch debouncer. With the switch in the STOP position, what do Q and Y equal? If the switch is thrown to the START position, what do Q and Y equal?

7-8. The clock has a frequency of 1 MHz in Fig. 7-22. With the switch in the OFF position, what is the frequency of the Q output? If the switch is thrown to the ON position, what is the frequency of the Q output?

Registers and Counters

8

A *register* is a group of memory elements that work together as a unit. The simplest registers do nothing more than store a binary word; others modify the stored word by shifting its bits left or right or by performing other operations to be discussed in this chapter. A *counter* is a special kind of register, designed to count the number of clock pulses arriving at its input. This chapter discusses some basic registers and counters used in microcomputers.

8-1 BUFFER REGISTERS

A *buffer register* is the simplest kind of register; all it does is store a digital word.

Basic Idea

Figure 8-1 shows a buffer register built with positive-edge-triggered *D* flip-flops. The *X* bits set up the flip-flops for loading. Therefore, when the first positive clock edge arrives, the stored word becomes $Q_3Q_2Q_1Q_0 = X_3X_2X_1X_0$. In chunked notation,

$$\mathbf{Q} = \mathbf{X}$$

The circuit is too primitive to be of any use. What it needs is some control over the *X* bits, some way of holding them off until we're ready to store them.

Controlled

Figure 8-2 is more like it. This is a controlled buffer register with an active-high *CLR*. Therefore, when *CLR* goes high, all flip-flops reset and the stored word becomes

$$\mathbf{Q} = 0000$$

When *CLR* returns low, the register is ready for action.

LOAD is a control input; it determines what the circuit does. When *LOAD* is low, the *X* bits cannot reach the flip-flops. At the same time, the inverted signal $\overline{LOAD}$ is high; this forces each flip-flop output to feed back to its data input. When each rising clock edge arrives, data is circulated or retained. In other words, the register contents are unchanged when *LOAD* is low.

When *LOAD* goes high, the *X* bits are transmitted to the data inputs. After a short setup time, the flip-flops are ready for loading. With the arrival of the positive clock edge, the *X* bits are loaded and the stored word becomes

$$Q_3Q_2Q_1Q_0 = X_3X_2X_1X_0$$

If *LOAD* returns to low, the foregoing word is stored indefinitely; this means that the *X* bits can change without affecting the stored word.

EXAMPLE 8-1

Chapter 10 discusses the *SAP* (simple as possible) computer. This educational computer has three generations, SAP-1, SAP-2, and SAP-3. Figure 8-3 shows the output register of the SAP-1 computer. The 74LS173 chips are controlled buffer registers, similar to Fig. 8-2. What does the circuit do?

SOLUTION

To begin with, it is an 8-bit buffer register built with TTL chips. Each chip handles 4 bits of input word **X**. The upper nibble $X_7X_6X_5X_4$ goes to pins 14, 13, 12, and 11 of C22; the lower nibble $X_3X_2X_1X_0$ goes to pins 14, 13, 12, and 11 of the C23.

Output word **Q** drives an 8-bit LED display. The upper nibble $Q_7Q_6Q_5Q_4$ comes out of pins 3, 4, 5, and 6 of C22; the lower nibble $Q_3Q_2Q_1Q_0$ comes out of pins 3, 4, 5, and 6 of C23. The typical high-state output of a 74LS173 is 3.5 V, and the typical LED drop is 1.5 V. Since each current-limiting resistance is 1 kΩ, the high-state current is approximately 2 mA for each output pin.

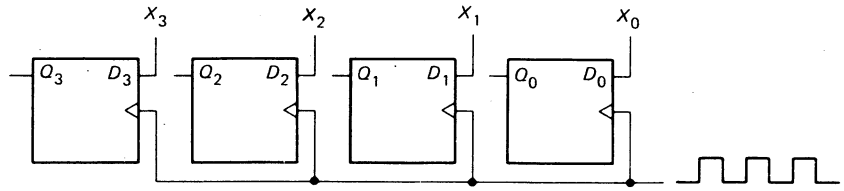


Fig. 8-1 Buffer register.

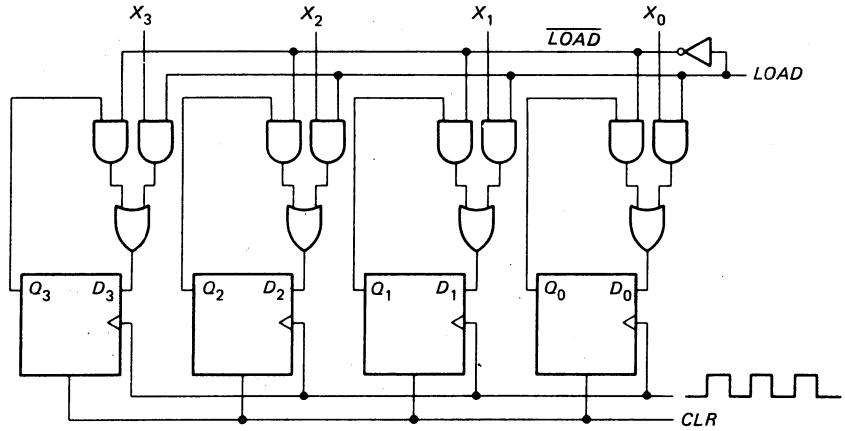
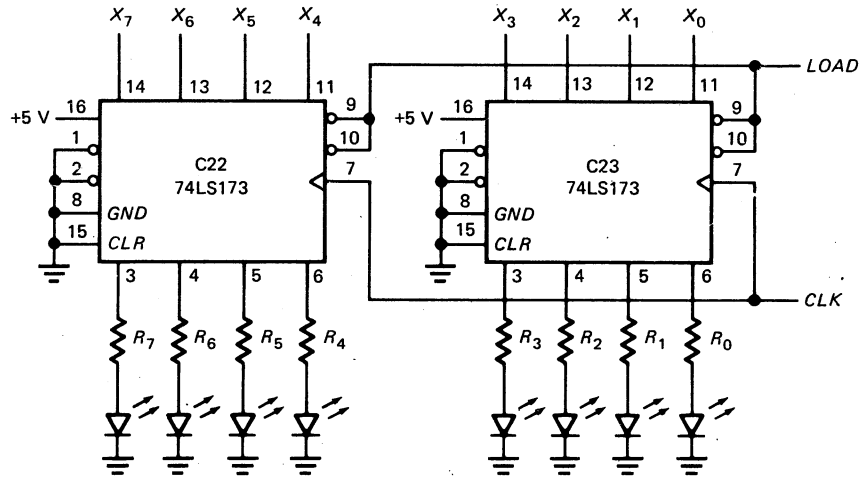


Fig. 8-2 Controlled buffer register.



Note: All resistors are 1 kΩ.

Fig. 8-3 SAP-1 output register.

The 74LS173 requires a 5-V supply for pin 16 and a ground return on pin 8. The SAP-1 output register never needs clearing; this is why the CLR input (pin 15) is made inactive by tying it to ground. In a 74LS173, pins 9 and 10 are separate *LOAD* controls. Because SAP-1 needs only a single *LOAD* control, pins 9 and 10 are tied together. The bubbles on pins 9 and 10 indicate an active low state; this means that *LOAD* must be low for the positive clock edge to store the input word. See Appendix 3 for a more detailed description of the 74LS173.

The action of the circuit is straightforward. While *LOAD* is high, the register contents are unchanged even though the clock is running. To change the stored word, *LOAD* must go low. Then the next rising clock edge loads the *X* bits into the register. As soon as this happens, the LED display shows the new contents.

8-2 SHIFT REGISTERS

A *shift register* moves the stored bits left or right. This bit shifting is essential for certain arithmetic and logic operations used in microcomputers.

Shift Left

Figure 8-4 is a *shift-left* register. As shown, D_{in} sets up the right flip-flop, Q_0 sets up the second flip-flop, Q_1 the third, and so on. When the next positive clock edge strikes, therefore, the stored bits move one position to the left.

As an example, here's what happens with $D_{in} = 1$ and

$$\mathbf{Q} = 0000$$

All data inputs except the one on the right are 0s. The arrival of the first rising clock edge sets the right flip-flop, and the stored word becomes

$$\mathbf{Q} = 0001$$

This new word means D_1 now equals 1, as well as D_0. When the next positive clock edge hits, the Q_1 flip-flop sets and the register contents become

$$\mathbf{Q} = 0011$$

The third positive clock edge results in

$$\mathbf{Q} = 0111$$

and the fourth rising clock edge gives

$$\mathbf{Q} = 1111$$

Hereafter, the stored word is unchanged as long as $D_{in} = 1$.

Suppose D_{in} is now changed to 0. Then, successive clock pulses produce these register contents:

$$\mathbf{Q} = 1110$$
$$\mathbf{Q} = 1100$$
$$\mathbf{Q} = 1000$$
$$\mathbf{Q} = 0000$$

As long as $D_{in} = 0$, subsequent clock pulses have no further effect.

The timing diagram of Fig. 8-5 summarizes the foregoing discussion.

Shift Right

Figure 8-6 is a *shift-right register*. As shown, each Q output sets up the D input of the preceding flip-flop. When the rising clock edge arrives, the stored bits move one position to the right.

Here's an example with $D_{in} = 1$ and

$$\mathbf{Q} = 0000$$

All data inputs except the one on the left are 0s. The first positive clock edge sets the left flip-flop and the stored word becomes

$$\mathbf{Q} = 1000$$

With the appearance of this word, D_3 and D_2 are 1s. The second rising clock edge gives

$$\mathbf{Q} = 1100$$

The third clock pulse gives

$$\mathbf{Q} = 1110$$

and the fourth clock pulse gives

$$\mathbf{Q} = 1111$$

8-3 CONTROLLED SHIFT REGISTERS

A *controlled shift register* has control inputs that determine what it does on the next clock pulse.

SHL Control

Figure 8-7 shows how the shift-left operation can be controlled. *SHL* is the control signal. When *SHL* is low, the inverted signal $\overline{SHL}$ is high. This forces each flip-flop output to feed back to its data input. Therefore, the data is retained in each flip-flop as the clock pulses arrive. In this way, a digital word can be stored indefinitely.

When *SHL* goes high, D_{in} sets up the right flip-flop, Q_0 sets up the second flip-flop, Q_1 the third flip-flop, and so on. In this mode, the circuit acts like a shift-left register. Each positive clock edge shifts the stored bits one position to the left.

Serial Loading

Serial loading means storing a word in the shift register by entering 1 bit per clock pulse. To store a 4-bit word, we need four clock pulses. For instance, here's how to serially store the word

$$\mathbf{X} = 1010$$

With *SHL* high in Fig. 8-7, make $D_{in} = 1$ for the first clock pulse, $D_{in} = 0$ for the second clock pulse, $D_{in} = 1$

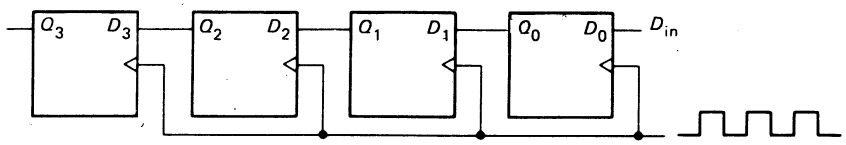


Fig. 8-4 Shift-left register.

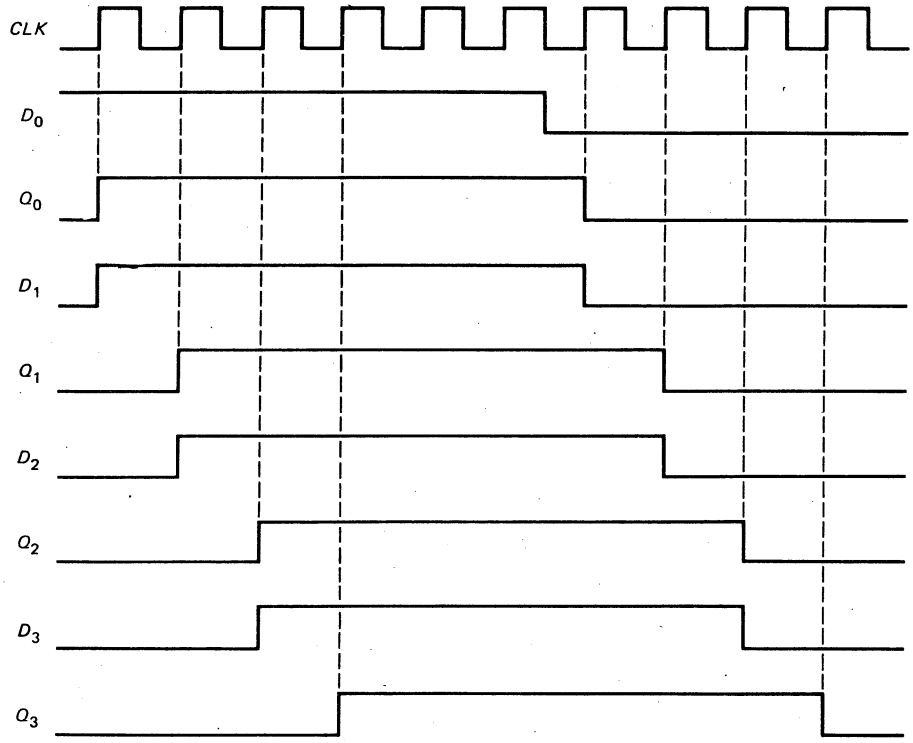


Fig. 8-5 Shift-left timing diagram.

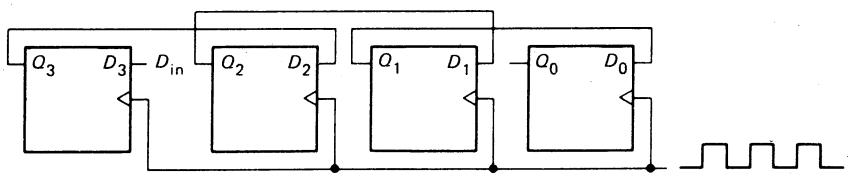


Fig. 8-6 Shift-right register.

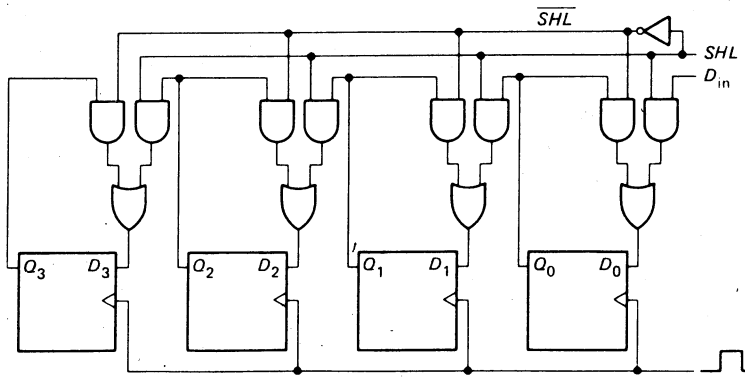


Fig. 8-7 Controlled shift register.

for the third clock pulse, and $D_{in} = 0$ for the fourth clock pulse. If the register is clear before the first clock pulse, the successive register contents look like this:

Q = 0001 (D_{in} = 1: first clock pulse)
Q = 0010 (D_{in} = 0: second clock pulse)
Q = 0101 (D_{in} = 1: third clock pulse)
Q = 1010 (D_{in} = 0: fourth clock pulse)

In this way, data is entered serially into the right end of the register and shifted left until all 4 bits have been stored. After the last bit is entered, *SHL* is taken low to freeze the register contents.

Parallel Loading

Figure 8-8 is another step in the evolution of shift registers. The circuit can load *X* bits directly into the flip-flops, the same as a buffer register. This kind of entry is called *parallel* or *broadside loading;* it takes only one clock pulse to store a digital word.

If *LOAD* and *SHL* are low, the output of the NOR gate is high and flip-flop outputs return to their data inputs. This forces the data to be retained in each flip-flop as the positive clock edges arrive. In other words, the register is inactive when *LOAD* and *SHL* are low, and the contents are stored indefinitely.

When *LOAD* is low and *SHL* is high, the circuit acts like a shift-left register, as previously described. On the other hand, when *LOAD* is high and *SHL* is low, the circuit acts like a buffer register because the *X* bits set up the flip-flops for broadside loading. (Having *LOAD* and *SHL* simultaneously high is forbidden because it's impossible to do both operations on a single clock edge.)

By adding more flip-flops we can build a controlled shift register of any length. And with more gates, the shift-right operation can be included. As an example, the 74198 is a TTL 8-bit bidirectional shift register. It can broadside load, shift left, or shift right.

8-4 RIPPLE COUNTERS

A *counter* is a register capable of counting the number of clock pulses that have arrived at its clock input. In its simplest form it is the electronic equivalent of a binary odometer.

The Circuit

Figure 8-9*a* shows a counter built with *JK* flip-flops. Since the *J* and *K* inputs are returned to a high voltage, each flip-flop will toggle when its clock input receives a negative edge.

Here's how the counter works. Visualize the *Q* outputs as a binary word

$$Q = Q_3Q_2Q_1Q_0$$

Q_3 is the most significant bit (MSB), and Q_0 is the least significant bit (LSB). When *CLR* goes low; all flip-flops reset. This results in a digital word of

$$Q = 0000$$

When *CLR* returns to high, the counter is ready to go. Since the LSB flip-flop receives each clock pulse, Q_0 toggles once per negative clock edge, as shown in the timing diagram of Fig. 8-9*b*. The remaining flip-flops toggle less often because they receive their negative edges from the preceding flip-flops.

For instance, when Q_0 goes from 1 back to 0, the *Q* flip-flop receives a negative edge and toggles. Likewise

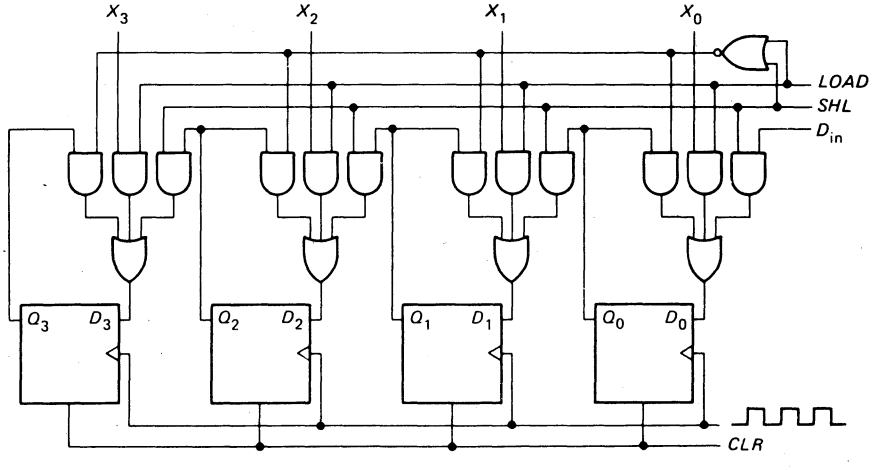


Fig. 8-8 Shift register with broadside load.

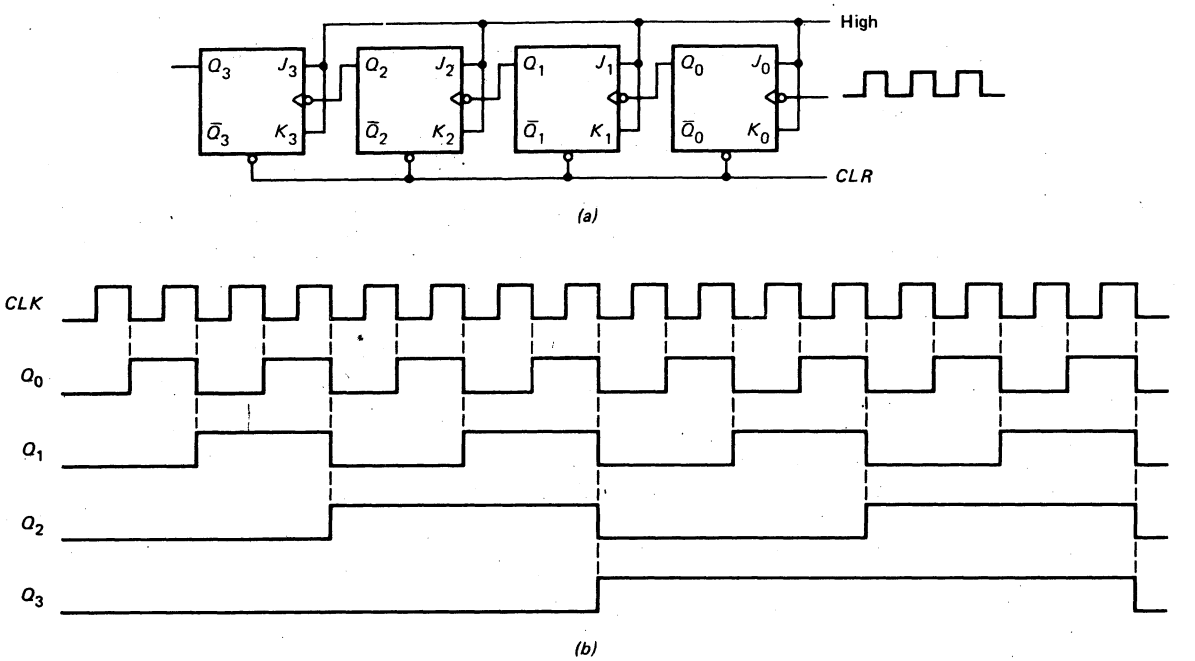


Fig. 8-9 (*a*) Ripple counter; (*b*) timing diagram.

when Q_1 changes from 1 back to 0, the Q_2 flip-flop gets a negative edge and toggles. And when Q_2 goes from 1 to 0, the Q_3 flip-flop toggles. In other words, whenever a flip-flop resets to 0, the next higher flip-flop toggles (see Fig. 8-9*b*).

What does this remind you of? Reset and carry! Each flip-flop acts like a wheel in a binary odometer; whenever it resets to 0, it sends a carry to the next higher flip-flop. Therefore, the counter of Fig. 8-9*a* is the electronic equivalent of a binary odometer.

Counting

If *CLR* goes low then high, the register contents of Fig. 8-9*a* become

$$\mathbf{Q} = 0000$$

When the first clock pulse hits the LSB flip-flop, Q_0 becomes a 1. So the first output word is

$$\mathbf{Q} = 0001$$

When the second clock pulse arrives, Q_0 resets and carries; therefore, the next output word is

$$\mathbf{Q} = 0010$$

The third clock pulse advances Q_0 to 1; this gives

$$\mathbf{Q} = 0011$$

The fourth clock pulse forces the Q_0 flip-flop to reset and carry. In turn, the Q_1 flip-flop resets and carries. The resulting output word is

$$\mathbf{Q} = 0100$$

The fifth clock pulse gives

$$\mathbf{Q} = 0101$$

The sixth gives

$$\mathbf{Q} = 0110$$

and the seventh gives

$$\mathbf{Q} = 0111$$

On the eighth clock pulse, Q_0 resets and carries, Q_1 resets and carries, Q_2 resets and carries, and Q_3 advances to 1. So the output word becomes

$$\mathbf{Q} = 1000$$

The ninth clock pulse gives

$$\mathbf{Q} = 1001$$

The tenth gives

$$\mathbf{Q} = 1010$$

and so on.

TABLE 8-1. RIPPLE COUNTER

Count	Q_3	Q_2	Q_1	Q_0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

The last word is

$$\mathbf{Q} = 1111$$

corresponding to the fifteenth clock pulse. The next clock pulse resets all flip-flops. Therefore, the counter resets to

$$\mathbf{Q} = 0000$$

and the cycle repeats.

Table 8-1 summarizes the operation of the counter. *Count* represents the number of clock pulses that have arrived. As you see, the counter output is the binary equivalent of the decimal count.

Frequency Division

Each flip-flop in Fig. 8-9*a* divides the clock frequency by a factor of 2. This is why a flip-flop is sometimes called a *divide-by-2 circuit*. Since each flip-flop divides the clock frequency by 2, n flip-flops divide the clock frequency by 2^n.

The timing diagram of Fig. 8-9*b* illustrates the divide-by-2 action. Q_0 is one-half the clock frequency, Q_1 is one-fourth the clock frequency, Q_2 is one-eighth the clock frequency, and Q_3 is one-sixteenth of the clock frequency. In other words,

1 flip-flop divides by 2
2 flip-flops divide by 4
3 flip-flops divide by 8
4 flip-flops divide by 16

and

n flip-flops divide by 2^n

Ripple Counter

The counter of Fig. 8-9*a* is known as a *ripple counter* because the carry moves through the flip-flops like a ripple on water. In other words, the Q_0 flip-flop must toggle before the Q_1 flip-flop, which in turn must toggle before the Q_2 flip-flop, which in turn must toggle before the Q_3 flip-flop. The worst case occurs when the stored word changes from 0111 to 1000, or from 1111 to 0000. In either case, the carry has to move all the way to the MSB flip-flop. Given a t_p of 10 ns per flip-flop, it takes 40 ns for the MSB to change.

By adding more flip-flops to the left end of Fig. 8-9*a* we can build a ripple counter of any length. Eight flip-flops give an 8-bit ripple counter, twelve flip-flops result in a 12-bit ripple counter, and so on.

Controlled Counter

A controlled counter counts clock pulses only when commanded to do so. Figure 8-10 shows how it's done. The *COUNT* signal can be low or high. Since it conditions the J and K inputs, *COUNT* controls the action of the counter, forcing it to either do nothing or to count clock pulses.

When *COUNT* is low, the J and K inputs are low; therefore, all flip-flops remain latched in spite of the clock pulses driving the counter.

On the other hand, when *COUNT* is high, the J and K inputs are high. In this case, the counter works as previously described; each negative clock edge increments the stored count by 1.

EXAMPLE 8-2

As mentioned earlier, the program and data are stored in the memory before a computer run. The program is a list of instructions telling the computer how to process the data.

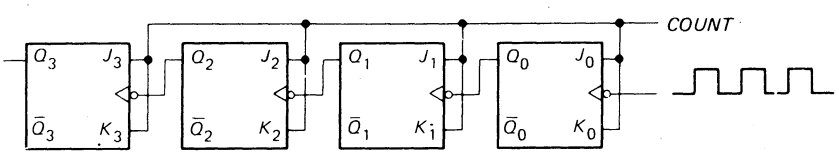


Fig. 8-10 Controlled ripple counter.

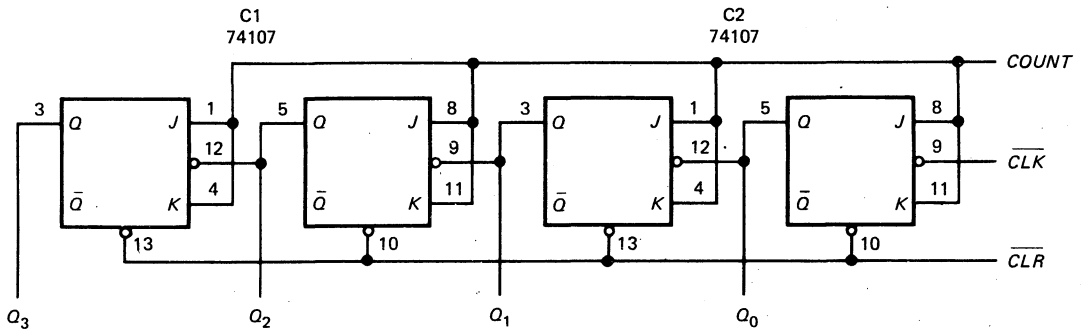


Fig. 8-11 SAP-1 program counter.

Every microcomputer has a *program counter* to keep track of the instruction being executed.

Figure 8-11 shows part of the program counter used in SAP-1. What does it do?

SOLUTION

To begin with, let's find out why the $\overline{CLR}$ and $\overline{CLK}$ signals are shown as complements. Signals are often available in complemented and uncomplemented form. The switch debouncer of Fig. 7-4*a* has two outputs, *CLR* and $\overline{CLR}$. In SAP-1 the *CLR* signal goes to any circuit that uses an active high clear and the $\overline{CLR}$ signal to any circuit with an active low clear. This is why $\overline{CLR}$ goes to the counter of Fig. 8-11; it has an active low clear. A similar idea applies to the clock signal.

The 74107 is a dual *JK* master-slave flip-flop. The SAP-1 program counter uses two 74107s. Although not shown, pin 14 ties to the 5-V supply, and pin 7 is the chip ground. Because master-slave flip-flops are used, a high $\overline{CLK}$ cocks the master and a low $\overline{CLK}$ triggers the slave.

Before a computer run, the operator pushes a clear button that sends a low $\overline{CLR}$ to the program counter. This resets its count to

$$\mathbf{Q} = 0000$$

When the operator releases the button, $\overline{CLR}$ goes high and the computer run begins.

After the first instruction has been fetched from the memory, *COUNT* goes high for one clock pulse and the count becomes

$$\mathbf{Q} = 0001$$

This count indicates that the first instruction has been fetched from the memory. (Later you will see how the computer executes the first instruction.)

After the first instruction has been executed, the computer fetches the second instruction in the memory. Once again, *COUNT* goes high for one clock pulse, producing a new count of

$$\mathbf{Q} = 0010$$

The program counter now indicates that the second instruction has been fetched from the memory.

Each time a new instruction is fetched from the memory, the program counter is incremented to produce the next higher count. In this way, the computer can keep track of which instruction it's working on.

8-5 SYNCHRONOUS COUNTERS

When the carry has to propagate through a chain of n flip-flops, the overall propagation delay time is nt_p. For this reason ripple counters are too slow for some applications. To get around the ripple-delay problem, we can use a *synchronous counter*.

The Circuit

Figure 8-12 shows one way to build a synchronous counter with positive-edge-triggered flip-flops. This time, clock pulses drive all flip-flops in parallel. Because of the simultaneous clocking, the correct binary word appears after one propagation delay time rather than four.

The least significant flip-flop has its J and K inputs tied to a high voltage; therefore, it responds to each positive clock edge. But the remaining flip-flops can respond to the positive clock edge only under certain conditions. As shown in Fig. 8-12, the Q_1 flip-flop toggles on the positive clock edge only when Q_0 is a 1. The Q_2 flip-flop toggles only when Q_1 and Q_0 are 1s. And the Q_3 flip-flop toggles only when Q_2, Q_1, and Q_0 are 1s. In other words, a flip-flop toggles on the next positive clock edge if all lower bits are 1s.

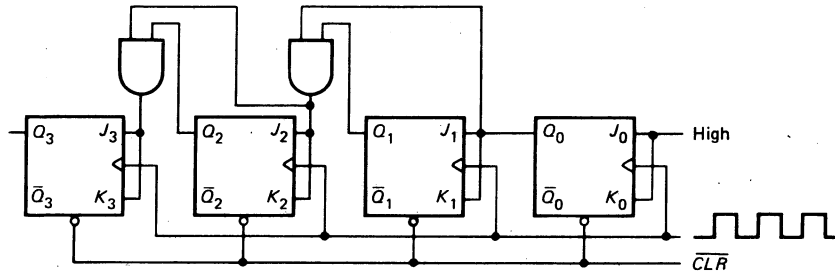


Fig. 8-12 Synchronous counter.

Here's the counting action. A low $\overline{CLR}$ resets the counter to

$$\mathbf{Q} = 0000$$

When the $\overline{CLR}$ line goes high, the counter is ready to go. The first positive clock edge sets Q_0 to get

$$\mathbf{Q} = 0001$$

Since Q_0 is now 1, the Q_1 flip-flop is conditioned to toggle on the next positive clock edge.

When the second positive clock edge arrives, Q_1 and Q_0 simultaneously toggle and the output word becomes

$$\mathbf{Q} = 0010$$

The third positive clock edge advances the count by 1:

$$\mathbf{Q} = 0011$$

Because Q_1 and Q_0 are now 1s, the Q_2, Q_1, and Q_0 flip-flops are conditioned to toggle on the next positive clock edge. When the fourth positive clock edge arrives, Q_2, Q_1, and Q_0 toggle simultaneously, and after one propagation delay time the output word becomes

$$\mathbf{Q} = 0100$$

The successive **Q** words are 0101, 0110, 0111, and so on up to 1111 (equivalent to decimal 15). The next positive clock edge resets the counter, and the cycle repeats.

By adding more flip-flops and gates we can build synchronous counters of any length. The advantage of a synchronous counter is its speed; it takes only one propagation delay time for the correct binary count to appear after the clock edge hits.

Controlled Counter

Figure 8-13 shows how to build a *controlled synchronous counter*. A low *COUNT* disables all flip-flops. When *COUNT* is high, the circuit becomes a synchronous counter; each positive clock edge advances the count by 1.

8-6 RING COUNTERS

Instead of counting with binary numbers, a *ring counter* uses words that have only a single high bit.

Circuit

Figure 8-14 is a *ring counter* built with *D* flip-flops. The Q_0 output sets up the D_1 input, the Q_1 output sets up the D_2 input, and so on. Therefore, a ring counter resembles a

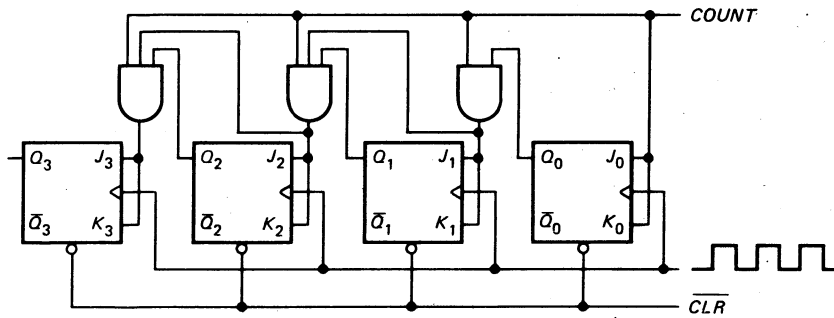


Fig. 8-13 Controlled synchronous counter.

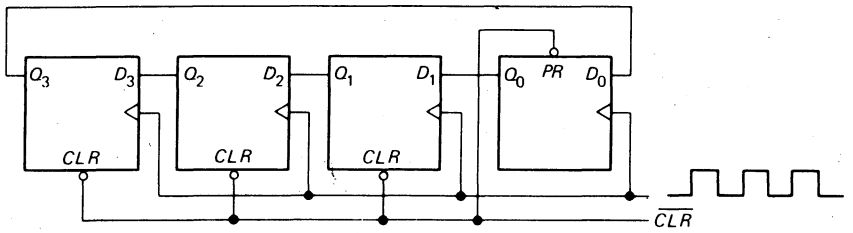


Fig. 8-14 Ring counter.

shift-left register because the bits are shifted left one position per positive clock edge. But the circuit differs because the final output is fed back to the D_0 input. This kind of action is called *rotate left;* bits are shifted left and fed back to the input.

When $\overline{CLR}$ goes low then back to high, the initial output word is

$$\mathbf{Q} = 0001$$

The first positive clock edge shifts the MSB into the LSB position; the other bits shift left one position. Therefore, the output word becomes

$$\mathbf{Q} = 0010$$

The second positive clock edge causes another rotate left and the output word changes to

$$\mathbf{Q} = 0100$$

After the third positive clock edge, the output word is

$$\mathbf{Q} = 1000$$

The fourth positive clock edge starts the cycle over because the rotate left produces

$$\mathbf{Q} = 0001$$

The stored 1 bit follows a circular path, moving left through the flip-flops until the final flip-flop sends it back to the first flip-flop. This is why the circuit is called a ring counter.

More Bits

Add more flip-flops and you can build a ring counter of any length. With six flip-flops we get a 6-bit ring counter. Again, the $\overline{CLR}$ signal resets all flip-flops except the LSB flip-flop. Therefore, the successive ring words are

$$\mathbf{Q} = 000001 \quad (0)$$
$$\mathbf{Q} = 000010 \quad (1)$$
$$\mathbf{Q} = 000100 \quad (2)$$
$$\mathbf{Q} = 001000 \quad (3)$$
$$\mathbf{Q} = 010000 \quad (4)$$
$$\mathbf{Q} = 100000 \quad (5)$$

Each of the foregoing words has only 1 high bit. The initial word stands for decimal 0 and the final word for decimal 5. If a ring counter has n flip-flops, therefore, the final ring word represents decimal $n - 1$.

Applications

Ring counters cannot compete with ripple and synchronous counters when it comes to ordinary counting, but they are invaluable when it's necessary to control a sequence of operations. Because each ring word has only 1 high bit, you can activate one of several devices.

For instance, suppose the six small boxes (A to F) of Fig. 8-15 are digital circuits that can be turned on by a high Q bit. When $\overline{CLR}$ goes low, Q_0 goes high and activates device A. After $\overline{CLR}$ returns to high, successive clock pulses turn on each device for a short time. In other words, as the stored 1 bit shifts left, it turns on B to F in sequence, and then the cycle starts over.

Many digital circuits participate during a computer run. To fetch and execute instructions, a computer has to activate

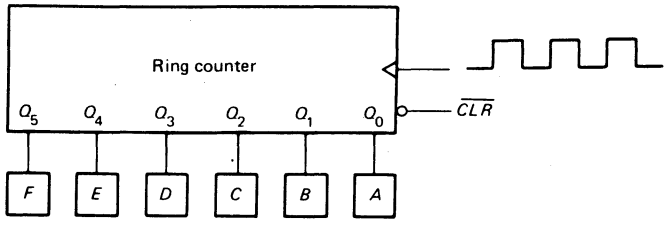
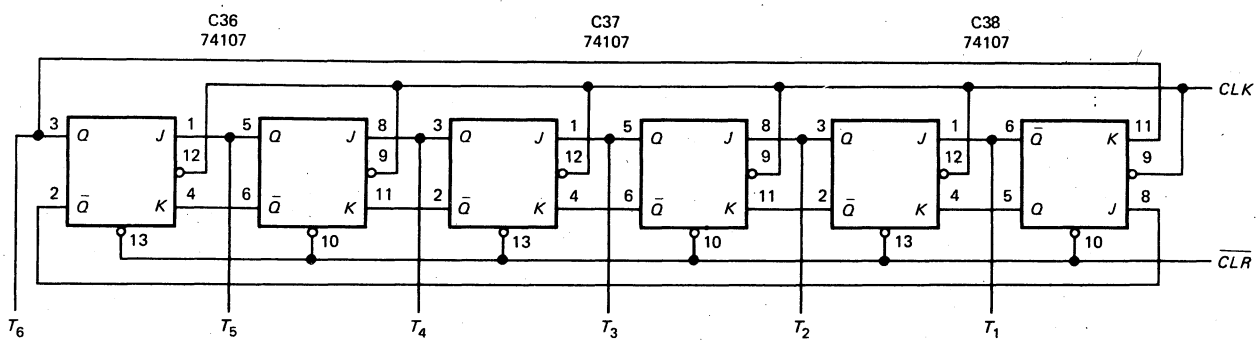


Fig. 8-15 Controlling a sequence of operations



Note: Pin 14 is connected to +5 V, and pin 7 is grounded.

Fig. 8-16 SAP-1 ring counter.

these circuits at precisely the right time and in the right sequence. This is where ring counters shine; they produce the ring words for timing different operations during a computer run.

EXAMPLE 8-3

Figure 8-16 shows the ring counter used in the SAP-1 computer. T_6 to T_1 are called *timing* signals because they control a sequence of digital operations. What does this ring counter do?

SOLUTION

The 74107 is a dual *JK* master-slave flip-flop, previously used in the SAP-1 program counter (Example 8-2). The flip-flops are connected in a rotate-left mode. Since the 74107 does not have a preset input, the Q_0 flip-flop is inverted so that its $\overline{Q}$ output drives the J input of the Q_1 flip-flop. In this way, a low $\overline{CLR}$ produces the initial timing word

$$T_6T_5T_4T_3T_2T_1 = 000001$$

In chunked form

$$\mathbf{T} = 000001$$

Because of the master-slave action, a complete clock pulse is needed to produce the next ring word. After $\overline{CLR}$ returns high, the successive clock pulses produce the timing words

$$\mathbf{T} = 000010$$
$$\mathbf{T} = 000100$$
$$\mathbf{T} = 001000$$
$$\mathbf{T} = 010000$$
$$\mathbf{T} = 100000$$

Then the cycle repeats.

EXAMPLE 8-4

The clock frequency in Fig. 8-16 is 1 kHz. $\overline{CLR}$ goes low then high. Show the timing diagram.

SOLUTION

Figure 8-17 is the timing diagram. Since the clock has a frequency of 1 kHz, it has a period of 1 ms. This is the amount of time between successive negative clock edges. Each negative clock edge produces the next ring word. When its turn comes, each timing signal goes high for 1 ms.

Notice that the *CLK* signal of Fig. 8-17 is the input to the ring counter of Fig. 8-16, whereas the complement $\overline{CLK}$ is the input to the program counter of Fig. 8-11. This half-cycle difference is deliberate. The reason is given in Chap. 10, which explains how the timing signals of Fig. 8-17 control circuits that fetch and execute each program instruction.

8-7 OTHER COUNTERS

The *modulus* of a counter is the number of output states it has. A 4-bit ripple counter has a modulus of 16 because it has 16 distinct states numbered from 0000 to 1111. By changing the design we can produce a counter with any desired modulus.

Mod-10 Counter

Figure 8-18*a* shows a way to build a modulus-10 (or mod-10) counter. The circuit counts from 0000 to 1001, as before. However, on the tenth clock pulse, the counter

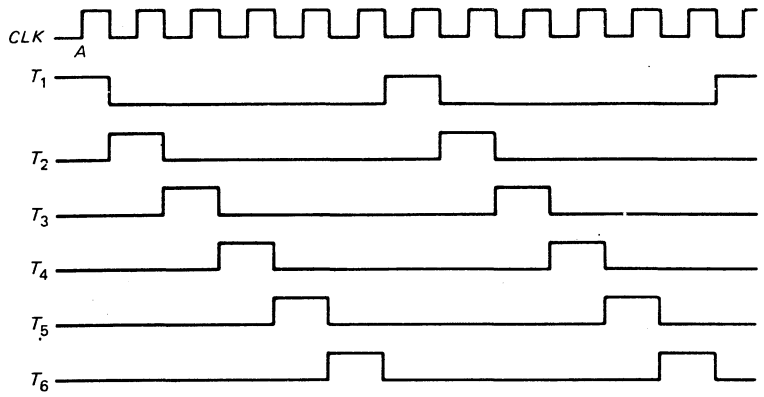


Fig. 8-17 SAP-1 clock and timing pulses.

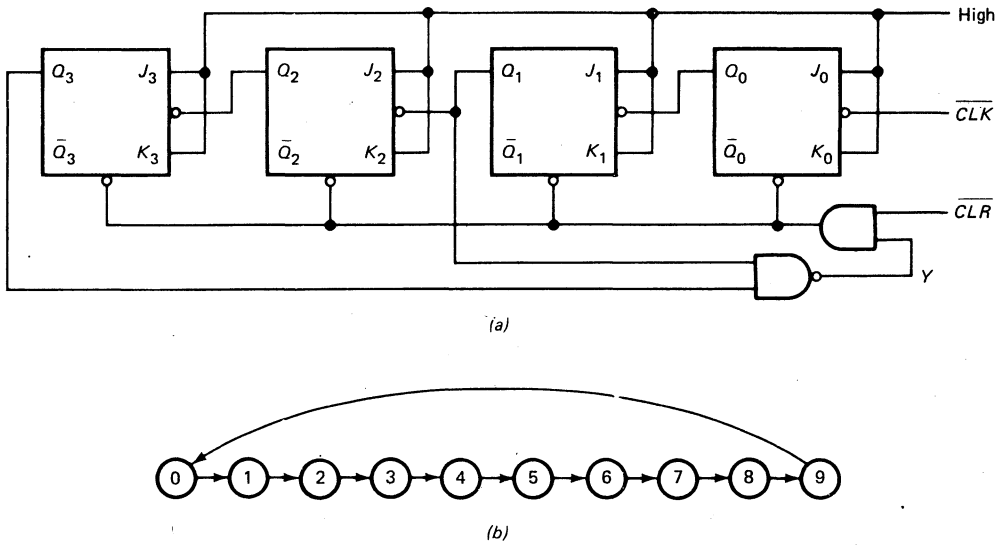


Fig. 8-18 Mod-10 counter.

generates its own clear signal and the count jumps back to 0000. In other words, the count sequence is

$$\begin{aligned}
\mathbf{Q} &= 0000 \quad (0)\\
\mathbf{Q} &= 0001 \quad (1)\\
\mathbf{Q} &= 0010 \quad (2)\\
\mathbf{Q} &= 0011 \quad (3)\\
\mathbf{Q} &= 0100 \quad (4)\\
\mathbf{Q} &= 0101 \quad (5)\\
\mathbf{Q} &= 0110 \quad (6)\\
\mathbf{Q} &= 0111 \quad (7)\\
\mathbf{Q} &= 1000 \quad (8)\\
\mathbf{Q} &= 1001 \quad (9)\\
\mathbf{Q} &= 0000 \quad (0)
\end{aligned}$$

As you see, the circuit skips states 10 to 15 (1010 through 1111). The counting sequence is summarized by the *state diagram* of Fig. 8-18*b*.

Why does the counter skip the states from 10 to 15? Because of the AND gate, the counter can be reset by a low $\overline{CLR}$ or a low Y. Initially, $\overline{CLR}$ goes low to produce

$$\mathbf{Q} = 0000$$

When $\overline{CLR}$ returns to high, the counter is ready for action. The output of the NAND gate is

$$Y = \overline{Q_3 Q_1}$$

This output is high for the first nine states (0000 to 1001). Nothing unusual happens when the circuit is counting from 0 to 9. On the tenth clock pulse, however, the **Q** word becomes

$$\mathbf{Q} = 1010$$

which means that Q_3 and Q_1 are high. Almost immediately, Y goes low, forcing the counter to reset to

$$\mathbf{Q} = 0000$$

Y then goes high, and the counter is ready to start over.

Since it takes 10 clock pulses to reset the counter, the output frequency of the Q_3 flip-flop is one-tenth of the clock frequency. This is why a mod-10 counter is also known as a *divide-by-10 circuit.*

A mod-10 counter like Fig. 8-18*a* is often called a *decade counter.* Because it counts from 0 to 9, it is a natural choice in BCD applications like frequency counters, digital voltmeters, and electronic wristwatches.

To get any other modulus, we can use the same basic idea. For instance, to get a mod-12 counter, we can drive the NAND gate of Fig. 8-18*a* with Q_3 and Q_2. Then the circuit counts from 0 to 11 (0000 to 1011). On the next clock pulse, Q_3 and Q_2 are high, which clears the counter. (What is the modulus if Q_3 and Q_0 drive the NAND gate?)

Down Counter

All the counters discussed so far have counted upward, toward higher numbers. Figure 8-19 shows a *down counter;* it counts from 1111 to 0000. Each flip-flop toggles when its clock input goes from 1 to 0. This is equivalent to an uncomplemented output going from 0 to 1. For instance, the Q_1 flip-flop toggles when $\overline{Q}_0$ goes from 1 to 0; this is equivalent to Q_0 going from 0 to 1.

A preset signal generated elsewhere is available in either uncomplemented or complemented form; *PRE* goes to all circuits with an active-high preset; $\overline{PRE}$ goes to all circuits with an active-low preset. Initially, the preset signal $\overline{PRE}$ goes low in Fig. 8-19, producing an output word of

$$\mathbf{Q} = 1111 \qquad (15)$$

When $\overline{PRE}$ goes high, the action starts. Notice that Q_0 toggles once per clock pulse. In the following discussion, a *positive toggle* means a change from 0 to 1, a *negative toggle* means a change from 1 to 0.

The first clock pulse produces a negative toggle in Q_0; nothing else happens:

$$\mathbf{Q} = 1110 \qquad (14)$$

The second clock pulse produces a positive toggle in Q_0, which produces a negative toggle in Q_1:

$$\mathbf{Q} = 1101 \qquad (13)$$

On the third clock pulse, Q_0 toggles negatively, and

$$\mathbf{Q} = 1100 \qquad (12)$$

On the fourth clock pulse, Q_0 toggles positively, Q_1 toggles positively, and Q_2 toggles negatively:

$$\mathbf{Q} = 1011 \qquad (11)$$

You should have the idea by now. The circuit is counting down, from 15 to 0. When it reaches 0,

$$\mathbf{Q} = 0000$$

On the next clock pulse, all flip-flops toggle positively to get

$$\mathbf{Q} = 1111$$

and the cycle repeats.

Up-Down Counter

Figure 8-20 shows how to build an *up-down counter.* The flip-flop outputs are connected to *steering* networks. An *UP* control signal produces either down counting or up counting. If the *UP* signal is low, $\overline{Q}_2$, $\overline{Q}_1$, and $\overline{Q}_0$ are transmitted to the clock inputs; this results in a down counter. On the other hand, when *UP* is high, Q_2, Q_1, and Q_0 drive the clock inputs and the circuit becomes an up counter.

Presettable Counter

In a *presettable counter,* the count starts at a number greater than zero. Figure 8-21*a* shows a presettable counter; the count begins with $P_3P_2P\ P_0$, a number between 0000 and 1111.

To start the analysis, look at the *LOAD* control line. When it is low, all NAND gates have high outputs; therefore,

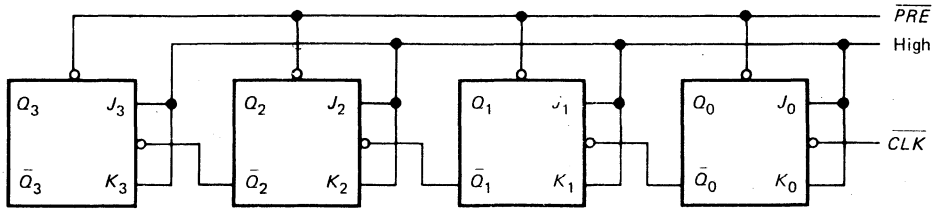


Fig. 8-19 Down counter.

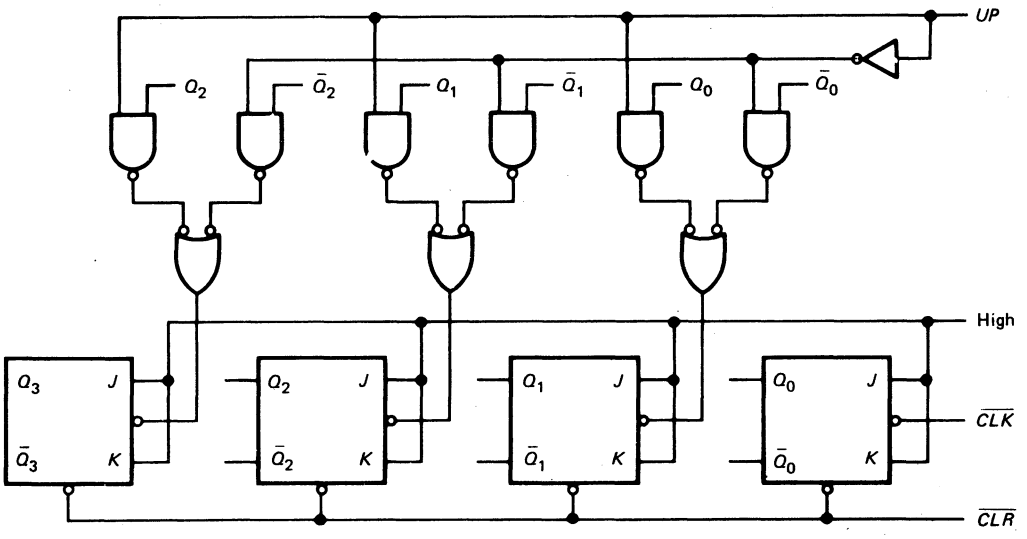


Fig. 8-20 Up-down counter.

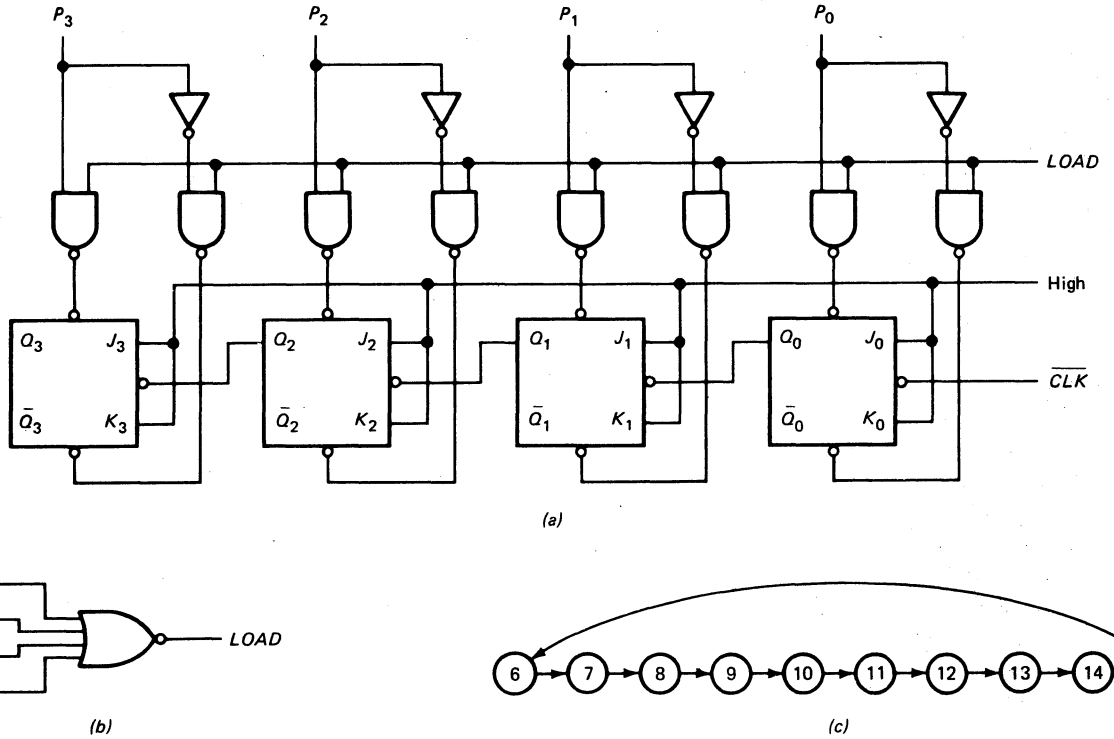


Fig. 8-21 Presettable counter.

the preset and clear inputs of all flip-flops are inactive. In this case, the circuit counts upward, as previously described. The data inputs P_3 to P_0 have no effect because the NAND gates are disabled.

When the *LOAD* line is high, the data inputs and their complements pass through the NAND gates and preset the counter to $P_3P_2P_1P_0$. As an example, suppose the preset input is

$$P_3P_2P_1P_0 = 0110$$

Because of the two left NAND gates, the low P_3 produces a high preset and a low clear for the Q_3 flip-flop; this clears

Q_3 to a 0. By a similar argument, the high P_2 sets Q_2, the high P_1 sets Q_1, and the low P_0 clears Q_0. Therefore, the counter is preset to

$$\mathbf{Q} = 0110$$

When *LOAD* returns to low, the circuit reverts to a counter. Successive clock pulses produce

$$\mathbf{Q} = 0111$$
$$\mathbf{Q} = 1000$$
$$\mathbf{Q} = 1001$$

up to a maximum count of

$$\mathbf{Q} = 1111$$

The next clock pulse resets the counter to

$$\mathbf{Q} = 0000$$

In summary,

1. When *LOAD* is low, the circuit counts.
2. When *LOAD* is high, the counter presets to $P_3P_2P_1P_0$.

Programmable Modulus

The most important use of a presettable counter is *programming a modulus*. Here's the idea. Let's add the NOR gate of Fig. 8-21*b* to the presettable counter of Fig. 8-21*a*. Then the Q outputs drive the NOR gate, and the NOR gate controls the *LOAD* line of the presettable counter. Because a NOR gate recognizes a word with all 0s and disregards all others, *LOAD* is high for $\mathbf{Q} = 0000$ and low for all other words. This means that the circuit presets when $\mathbf{Q} = 0000$ and counts when $\mathbf{Q}$ is 0001 to 1111.

If the preset input is 0110, successive clock pulses produce 0111, 1000, 1001, . . . , reaching a maximum value of

$$\mathbf{Q} = 1111$$

The next clock pulse resets the count to

$$\mathbf{Q} = 0000$$

Almost immediately, however, the NOR-gate outputs goes high, and the data inputs preset the counter to

$$\mathbf{Q} = 0110$$

In other words, the counter effectively skips states 0 to 5, illustrated by the state diagram of Fig. 8-21*c*.

Figure 8-21*c* shows 10 distinct states; by presetting 0110, we have programmed the counter to become a mod-10 counter. If we change the preset input, we get a different modulus. In general,

$$M = N - P \tag{8-1}$$

where M = modulus of preset counter
N = natural modulus
P = preset count

The natural modulus equals 2^n where n is the number of flip-flops in the counter. So four flip-flops give a natural modulus of 16, eight give a natural modulus of 256, and so on.

As an example, if you preset 82 into a preset counter with eight flip-flops, the modulus is

$$M = 256 - 82 = 174$$

In other words, this preset counter is equivalent to a divide-by-174 circuit.

TTL Counters

Table 8-2 lists some TTL counters. The 7490 is an industry standard, a widely used decade counter. This ripple counter has two sections, a divide-by-2 and a divide-by-5. This allows you to divide by 2, to divide by 5, or to cascade both sections to divide by 10.

The 7492 is a mod-12 ripple counter, organized in two sections by divide-by-2 and divide-by-6. This allows you to divide by 2, divide by 6, or cascade to divide by 12. The 7493 is a mod-16 ripple counter, with two sections of divide-by-2 and divide-by-8.

The 74160 and 74161 are presettable synchronous counters, the first being a decade counter and the second a divide-by-16 counter. Finally, the 74190 and 74191 are up-down presettable counters.

This is a sample of basic TTL counters; others are listed in Appendix 2.

TABLE 8-2. TTL COUNTERS

Number	Type
7490	Decade
7492	Divide-by-12
7493	Divide-by-16
74160	Presettable decade
74161	Presettable divide-by-16
74190	Up-down presettable decade
74191	Up-down presettable divide-by-16

8-8 THREE-STATE REGISTERS

The *three-state switch*, a development of the early 1970s, has greatly simplified computer wiring and design because it's ideal for *bus-organized computers* (the common type nowadays).

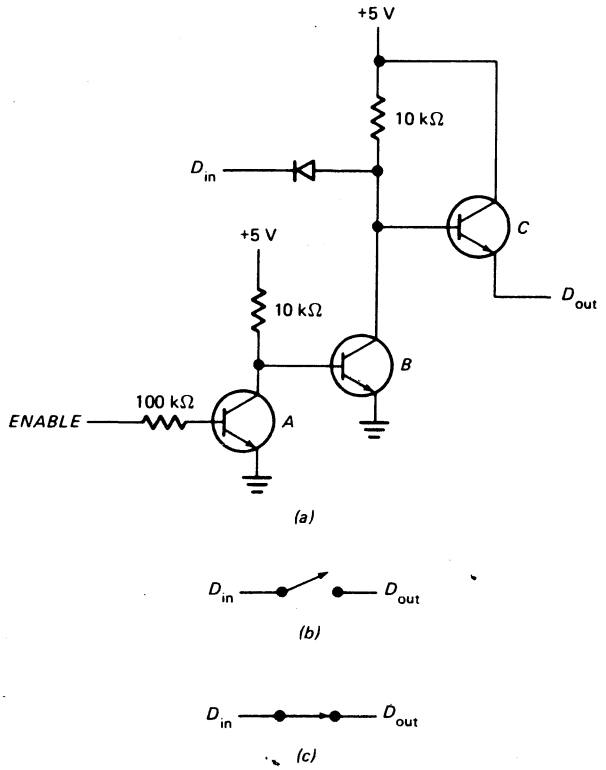


Fig. 8-22 (*a*) Three-state switch; (*b*) floating or high-impedance state; (*c*) output equals input.

Three-State Switch

Figure 8-22*a* is an example of a three-state switch. The *ENABLE* input can be low or high. When it's low, transistor A cuts off and transistor B saturates. This pulls the base of transistor C down to ground, opening its base-emitter diode. As a result, D_{out} floats. This floating state is equivalent to an *open* switch (Fig. 8-22*b*).

On the other hand, when *ENABLE* is high, transistor A saturates and transistor B cuts off. Now, the transistor C acts like an emitter follower, and the overall circuit is equivalent to a *closed* switch (Fig. 8-22*c*). In this case,

$$D_{out} = D_{in}$$

This means that D_{out} is low or high, the same as D_{in}.

Table 8-3 summarizes the action. When *ENABLE* is low, D_{in} is a don't care and D_{out} is open or floating. When *ENABLE* is high, the circuit acts like a noninverting buffer because D_{out} equals D_{in}.

TABLE 8-3. NORMALLY OPEN

ENABLE	D_{in}	D_{out}
0	X	Open
1	0	0
1	1	1

Commercial three-state switches are much more complicated than Fig. 8-22*a* (a totem-pole output and other enhancements are added). But simple as it is, Fig. 8-22*a* captures the key idea of a three-state switch; the output can be in any of three states: low, high, or floating (sometimes called the *high-impedance state* because the Thevenin impedance is high).

Three-state switches are also known as *Tri-state switches*. (Tri-state is a trademark name used by National Semiconductor, the originator of three-state TTL logic.)

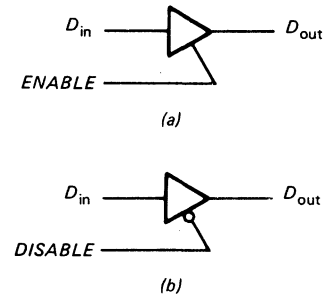


Fig. 8-23 (*a*) Normally open switch; (*b*) normally closed switch.

Normally Open Switch

Figure 8-23*a* is the symbol for a three-state noninverting buffer. When you see this symbol, remember the action: a low *ENABLE* means that the output is floating; a high *ENABLE* means that the output is 0 or 1, the same as the input. Think of this switch as *normally open;* to close it, you have to apply a high *ENABLE*.

In the 7400 series, the 74126 is a quad three-state normally open switch. This means four switches like Fig. 8-23*a* in one package. The SAP-1 computer uses five 74126s.

Normally Closed Switch

Figure 8-23*b* is different. This is the symbol for a *normally closed switch* because the control input *DISABLE* is active low. In other words, the switch is closed when *DISABLE* is low, and open when *DISABLE* is high. Table 8-4 summarizes the operation.

The 74125 is a quad three-state normally closed switch (four switches like Fig. 8-23*b* in one package).

TABLE 8-4. NORMALLY CLOSED

DISABLE	D_{in}	D_{out}
0	0	0
0	1	1
1	X	Open

Three-State Buffer Register

The main application of three-state switches is to convert the two-state output of a register to a three-state output. For instance, Fig. 8-24 shows a three-state buffer register, so called because of the three-state switches on the output lines. When *ENABLE* is low, the *Y* outputs float. But when *ENABLE* is high, the *Y* outputs equal the *Q* outputs; therefore,

$$Y = Q$$

You already know how the rest of the circuit works; it's the controlled buffer register discussed earlier. When *LOAD* is low, the contents of the register are unchanged. When *LOAD* is high, the next positive clock edge loads $X_3X_2X_1X_0$ into the register.

8-9 BUS-ORGANIZED COMPUTERS

A *bus* is a group of wires that transmit a binary word. In Fig. 8-25, vertical wires W_3, W_2, W_1, and W_0 are a bus; these wires are a common transmission path between the three-state registers. The input data bits for register A come from the **W** bus; at the same time, the three-state output of register A connects back to the **W** bus. Similarly, the other registers have their inputs and outputs connected to the **W** bus.

In Fig. 8-25 all control signals are in uncomplemented form; this means that the registers have active high inputs. In other words, a load input (L_A to L_D) must be high to set up for loading, and an enable signal (E_A to E_D) must be high to connect an output to the bus.

Register Transfers

The beauty of bus organization is the ease of transferring a word from one register to another. To begin with, the same clock signal drives all registers, but nothing happens until you apply high control inputs. In other words, as long as all *LOAD* and *ENABLE* inputs are low, the registers are isolated from the bus.

To transfer a word from one register to another, make the appropriate control inputs high. For instance here's how to transfer the contents of register A to the register D. Make E_A and L_D high; then the contents of register A appear on the bus and register D is set up for loading. When the next positive clock edge arrives, word **A** is stored in register D.

Here is another example. Suppose the following words are stored in the registers:

A = 0011
B = 0110
C = 1001
D = 1100

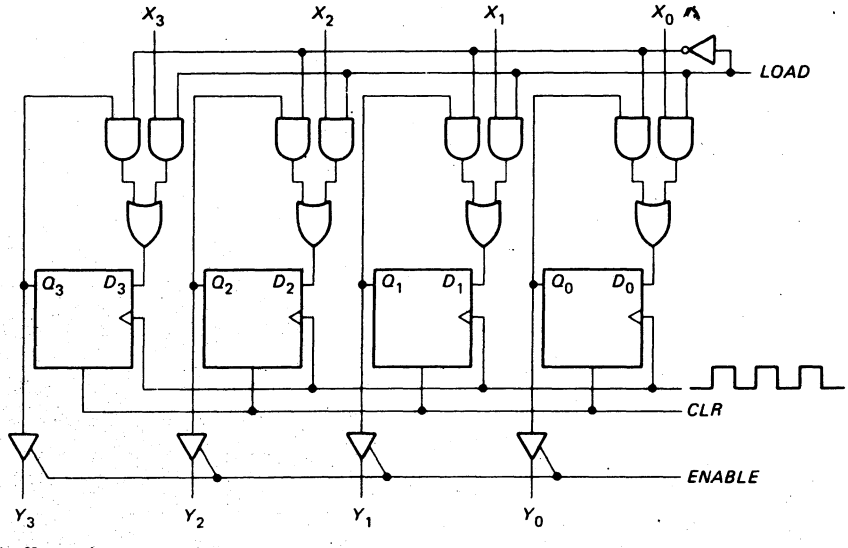


Fig. 8-24 Three-state buffer register.

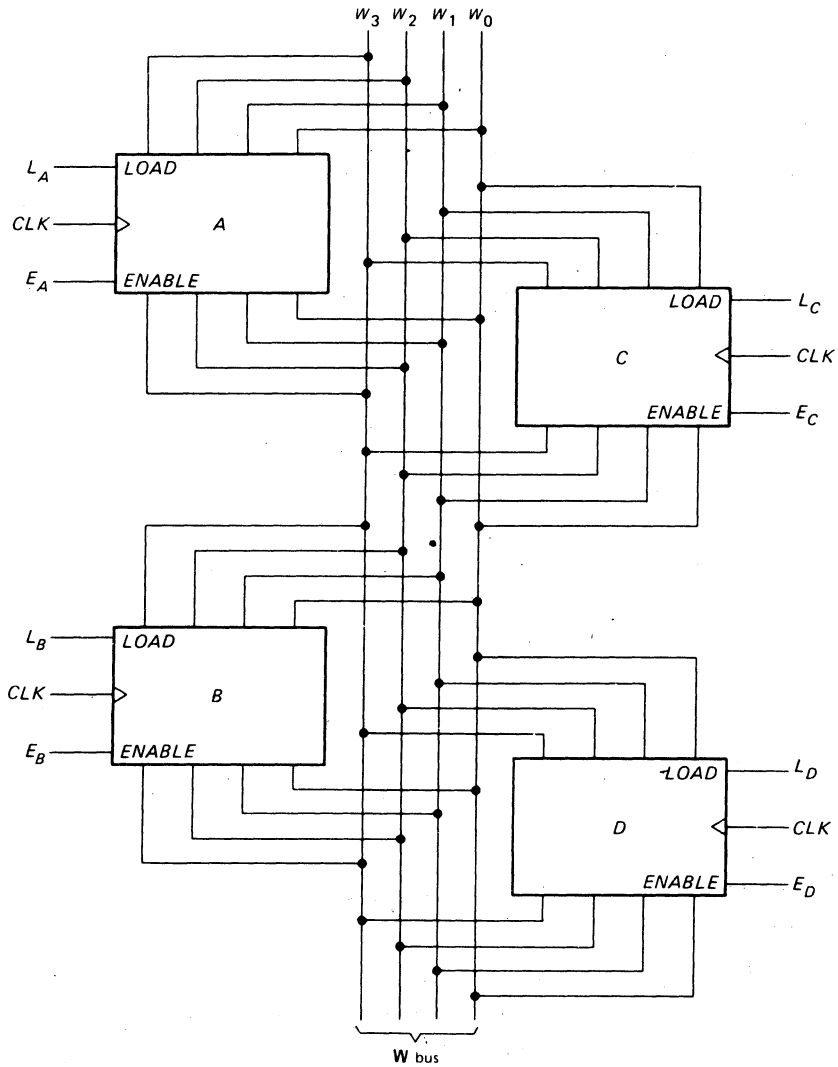


Fig. 8-25 Registers connected to bus.

To transfer word **C** into register B, make E_C and L_B high. The high E_C closes the three-state switches of register C, placing word **C** on the bus. The high L_B sets up register B for loading. When the next positive clock edge arrives, word **C** is stored in register B, and the new words are

A = 0011
B = 1001
C = 1001
D = 1100

The whole point of bus organization (connecting the registers to a common word path) is to simplify the wiring and operation of computers. As you will see in Chap. 10, SAP-1 is a bus-organized computer of incredible simplicity made possible by the three-state switch.

Simplified Drawings

Figure 8-25 shows a 4-bit bus. The same idea applies to any number of bits. For example, a 16-bit bus has 16 wires, each carrying 1 bit of a word. By connecting the inputs and outputs of 16-bit registers to this bus, we can transfer 16-bit words from one register to another.

Drawings get very messy unless we simplify the appearance of the bus. Figure 8-26 shows an abbreviated form of Fig. 8-25. The solid arrows represents words going into and out of registers. The solid bar represents the W bus.

EXAMPLE 8-5

Figure 8-27 shows part of the SAP-1 computer. Describe the circuitry.

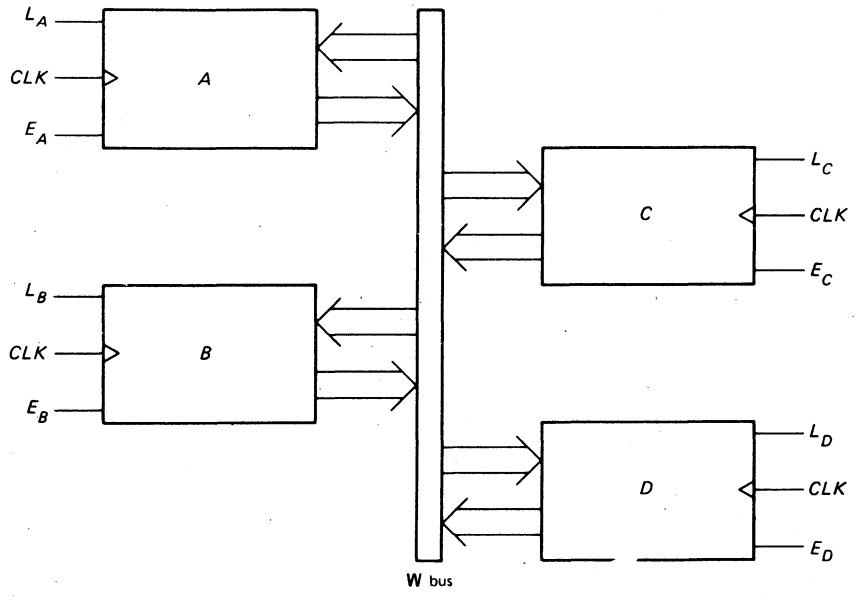


Fig. 8-26 Simplified bus diagram.

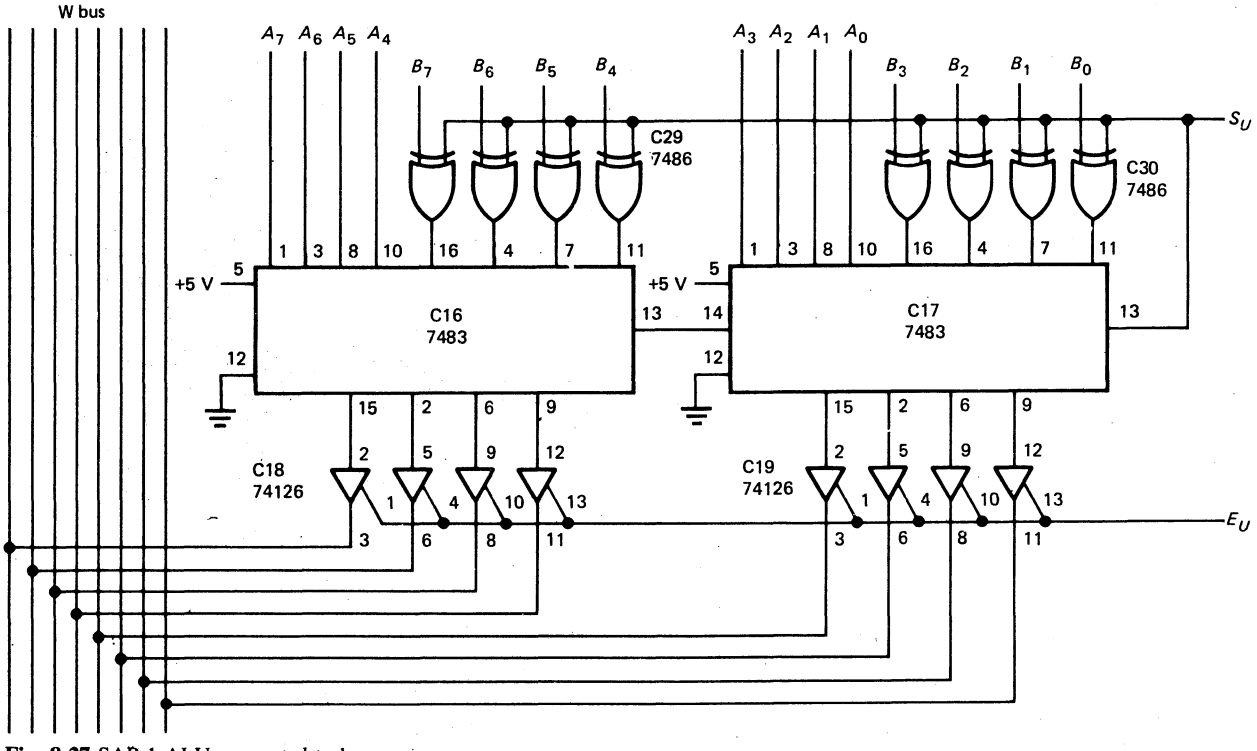


Fig. 8-27 SAP-1 ALU connected to bus.

SOLUTION

As discussed in Sec. 6-8, the 7483 is a 4-bit adder. The two 7483s of Fig. 8-27 are the ALU of the SAP-1 computer. The inputs to this ALU are the words

$$\mathbf{A} = A_7A_6A_5A_4A_3A_2A_1A_0$$
$$\mathbf{B} = B_7B_6B_5B_4B_3B_2B_1B_0$$

A pair of 7486s allow us to complement the **B** input for subtraction.

The sum (S_U low) or difference (S_U high) appears at the output (pins 15, 2, 6, 9 of C16 and pins 15, 2, 6, 9 of C17). Three-state switches (C18 and C19) connect the ALU output to the W bus when E_U is high. If E_U is low, the 74126s are open and the ALU output is isolated from the bus.

EXAMPLE 8-6

Figure 8-28 shows the *instruction register* (C8 and C9) of the SAP-1 computer. What does this 8-bit register do?

SOLUTION

Example 8-1 introduced the 74LS173. As you may recall, pins 9 and 10 are tied together and control the *LOAD* function. Because of the bubble, a low $\overline{L}_I$ is needed to set up the registers for loading. When $\overline{L}_I$ is low, the next positive clock edge loads the data on the bus into the instruction register.

The output of the instruction register is split; the upper nibble $I_7I_6I_5I_4$ goes to the *instruction decoder*, a circuit that will be discussed in Chap. 10. The lower nibble out of the instruction register goes back to the W bus.

The 74LS173 is a 4-bit three-state buffer register; it has internal three-state switches controlled by pins 1 and 2. The bubbles on pins 1 and 2 indicate active-low inputs; therefore, the output of C9 is connected to the bus when $\overline{E}_I$ is low and disconnected when $\overline{E}_I$ is high.

Notice that pins 1 and 2 of C8 are grounded; this means that the upper nibble is always a two-state output. In other words, the 74LS173 can be used as an ordinary two-state register by grounding pins 1 and 2. (This was done in Example 8-1, where we used two 74LS173s for the output register to drive an 8-bit LED display.)

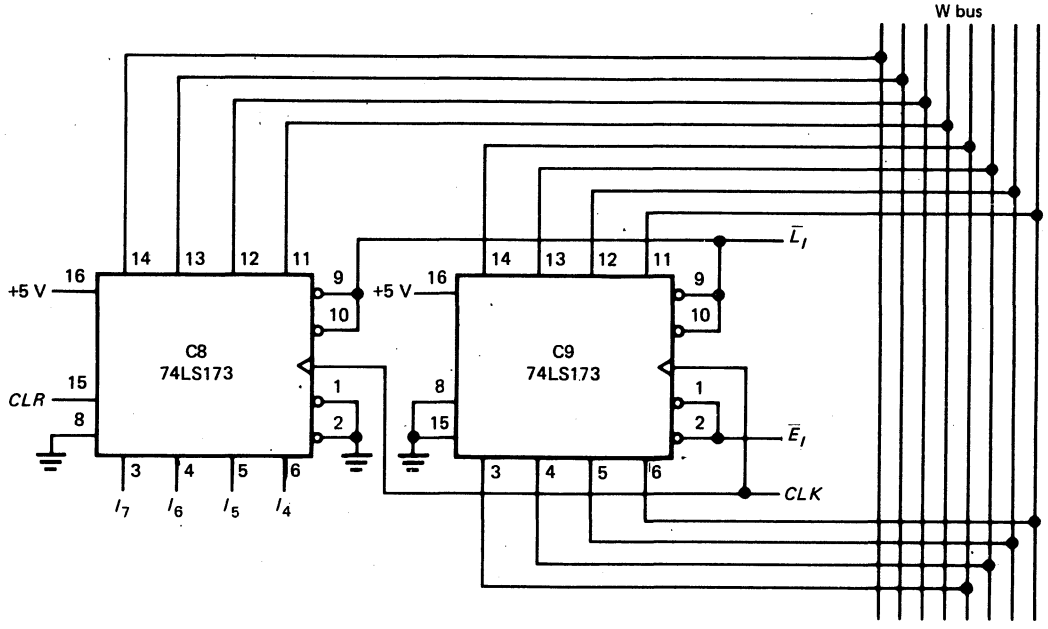


Fig. 8-28 SAP-1 instruction register.

GLOSSARY

buffer register A register that temporarily stores a word during data processing.

bus A group of wires used as a common word path by everal registers.

modulus The number of stable states a counter has.

parallel entry Loading all bits of a word in parallel during one clock pulse. Also called broadside loading.

presettable counter A counter that allows you to preset a

number from which the count begins. Sometimes called a programmable counter.

register A group of memory elements that store a word.

ring counter A counter producing words with 1 high bit, which shifts one position per clock pulse.

ripple counter A counter with cascaded flip-flops. This means that the carry has to propagate in series through the flip-flops.

serial entry Loading a word into a shift register 1 bit per clock pulse

shift register A register that can shift the stored bits one position to the left or right.

synchronous counter A counter in which the clock drives each flip-flop to eliminate the ripple delay.

three-state switch A noninverting buffer that can be closed or opened by a control signal. Also called a Tri-state switch.

SELF-TESTING REVIEW

Read each of the following and provide the missing words. Answers appear at the beginning of the next question.

1. When the *LOAD* input of a buffer register is active, the input word is stored on the next positive _____ edge. If *LOAD* then becomes inactive, the input word can change without effecting the _____ word.
2. (*clock, stored*) A shift register moves the _____ left or right. Serial loading means storing a word in a shift register by entering _____ bit per clock pulse. With parallel or broadside loading, it takes only one _____ pulse to load the input word.
3. (*bits, 1, clock*) One flip-flop divides the clock frequency by a factor of _____. Two flip-flops divide by 4, three flip-flops by 8, and four flip-flops by _____. In general, n flip-flops divide by 2^n.
4. (*2, 16*) In a ripple counter, the carry has to propagate through all the flip-flops to reach the MSB flip-flop. The overall propagation delay time is _____. A controlled counter counts _____ pulses only when the *COUNT* signal is active. The clock signal drives each flip-flop of a _____ counter.
5. (nt_p, *clock, synchronous*) Instead of counting with binary numbers, a ring counter uses words that have a single high _____. A ring counter is ideal for timing a sequence of digital operations.
6. (*bit*) The modulus of a counter is the number of stable output _____ it has. A mod-10 counter can divide the clock frequency by a factor of _____.
7. (*states, 10*) An up-down counter can count up or down. A presettable counter starts the count from a _____ number. This allows us to program the _____. If the modulus is M, a presettable counter is equivalent to a divide-by-M circuit.
8. (*preset, modulus*) A three-state switch has an output that is either low, high, or _____. Two types are available; normally open and normally closed. The main use of three-state switches is to convert the _____ output of a register to a three-state output.
9. (*floating, two-state*) A bus is a group of wires used by three-state registers as a common word path. Bus-organized computers, the common type nowadays, have several registers connected to one or more buses. Instructions and data travel along these buses as they move from one register to another.

PROBLEMS

8-1. Figure 8-29 shows an output register. Before time A the data word to be loaded is

$$\mathbf{X} = 1000\ 1101$$

and the LED display is

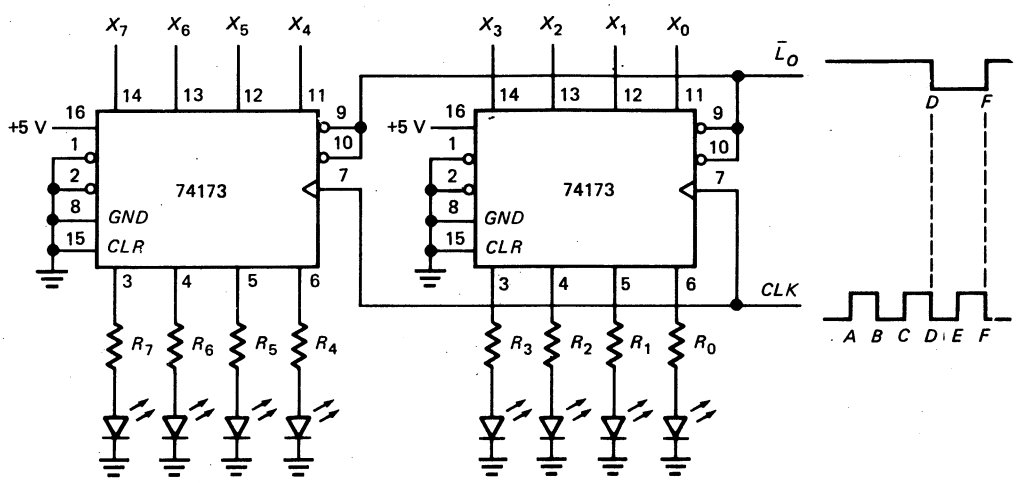
$$\mathbf{Q} = 0001\ 0111$$

a. What is the LED display at time D?
b. What is the LED display at time F?

8-2. The data sheet of a 74173 gives these values:

$t_{setup} = 17$ ns	(L_O input)
$t_{setup} = 10$ ns	(Data)
$t_{hold} = 2$ ns	(L_O input)
$t_{hold} = 10$ ns	(Data)

a. In Fig. 8-29, how far ahead of point E must the X bits be applied to ensure accurate loading?
b. Suppose the clock has a frequency of 1 MHz



Note: All resistors are 1 kΩ.

Fig. 8-29

and the X bits are applied at the point D. Is the setup time sufficient for the data inputs?

c. How long must you wait after point E before removing the X bits or letting them change?

8-3. Each output pin of a 74173 can source up to 5.2 mA. In Fig. 8-29 suppose the high output voltage is 3.5 V and the LED drop is 1.5 V. To get more light out of the LEDs, we want to reduce the current-limiting resistors. What is the minimum allowable resistance?

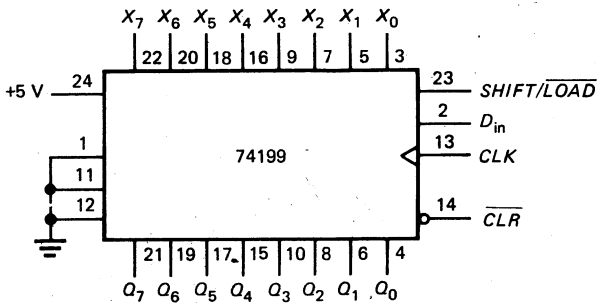


Fig. 8-30

8-4. A 74199 is an 8-bit shift-left register with a single control signal, as shown in Fig. 8-30. When *SHIFT/*$\overline{LOAD}$ is low, the circuit loads the **X** word on the next positive clock edge. When *SHIFT/*$\overline{LOAD}$ is high, the register shifts the bits to the left.

a. To clear the register, should $\overline{CLR}$ be low or high? When you are ready to run, what should $\overline{CLR}$ be?

b. Is the **X** word loaded on the positive or negative edge of the clock?

c. If **X** = 0100 1011, D_{in} = 0, and *SHIFT/*$\overline{LOAD}$ = 0, what does the **Q** output word equal after two positive clock edges?

d. If **X** = 0100 1011, D_{in} = 0, and *SHIFT/*$\overline{LOAD}$ = 1, what does the **Q** output word equal after two positive clock edges?

8-5. The clock frequency is 2 MHz. How long will it take to serially load the shift register of Fig. 8-30?

8-6. In Fig. 8-30, **Q** = 0001 0110. If *SHIFT/*$\overline{LOAD}$ is high and D_{in} is high, what does **Q** equal after three clock pulses?

8-7. Data from a satellite is received in serial form (1 bit after another). If this data is coming at a 5-MHz rate and if the clock frequency is 5 MHz, how long will it take to serially load a word in a 32-bit shift register?

8-8. A ripple counter has 16 flip-flops, each with a propagation delay time of 25 ns. If the count is

Q = 0111 1111 1111 1111

how long after the next active clock edge before

Q = 1000 0000 0000 0000

8-9. What is the maximum decimal count for the counter of the preceding problem?

8-10. When pins 1 and 12 of a 7490 are tied together as shown in Fig. 8-31, the divide-by-2 and divide-by-5 sections are cascaded to get a mod-10 counter. Pin 14 is the input and pin 11 is the output of each 7490. As a result, each 7490 acts like a divide-by-10 circuit and the overall circuit divides by 1,000

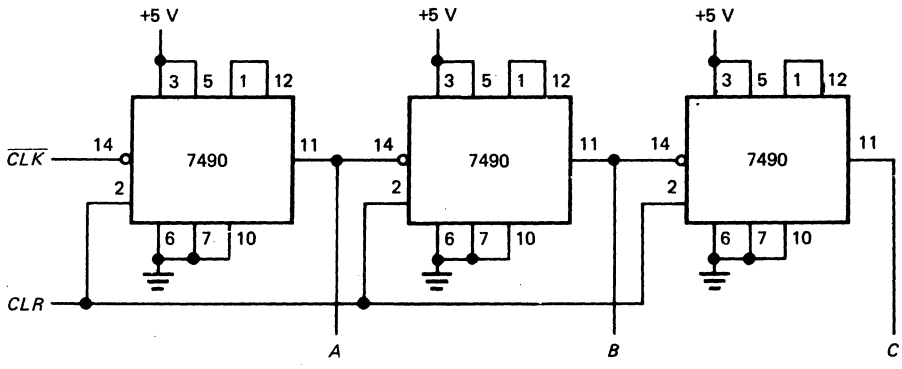


Fig. 8-31

If the clock has a frequency of 5 MHz, what is the frequency of *A*? Of *B*? Of *C*?

8-11. The clock signal driving a 6-bit ring counter has a frequency of 1 MHz. How long is each timing bit high? How long does it take to cycle through all the ring words?

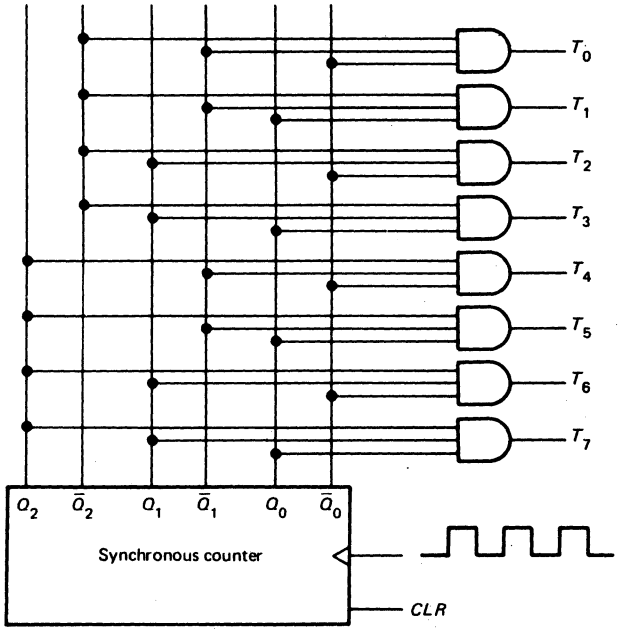


Fig. 8-32

8-12. Figure 8-32 shows another way to produce ring words. After the circuit is cleared,

$$\mathbf{Q} = Q_2Q_1Q_0 = 000$$

Since the AND gates are a 1-of-8 decoder, the first timing word is

$$\mathbf{T} = 0000\ 0001$$

What does **T** equal for each of the following:

a. **Q** = 001
b. **Q** = 010
c. **Q** = 101
d. **Q** = 111

8-13. If the clock frequency is 5 MHz in Fig. 8-32, how long does it take to produce all the ring words? How long is each timing bit high?

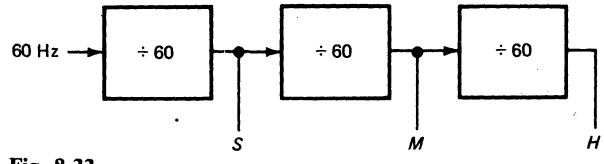


Fig. 8-33

8-14. In a digital clock, the 60-Hz line frequency is divided down to lower frequencies, as shown in Fig. 8-33. What are the frequency and period of the *S* output? Of the *M* output? Of the *H* output?

8-15. You have an unlimited number of the following ICs to work with: 7490, 7492, and 7493. Which of these would you use to build the divide-by-60 circuits of Fig. 8-33?

8-16. A presettable counter has eight flip-flops. If the preset number is 125, what is the modulus?

8-17. Given a presettable 8-bit counter, what number would you preset to get a divide-by-120 circuit?

8-18. In Fig. 8-34, we want to transfer the contents of register D to register C. Which are the *ENABLE* and *LOAD* inputs you should make high?

8-19. Look at Fig. 8-35 and answer each of these questions.

a. To add the inputs and put the answer on the bus, what should S_U and E_U be?
b. To subtract the inputs and put the answer on the bus, what should S_U and E_U be?
c. To isolate the ALU from the bus, what should E_U be?

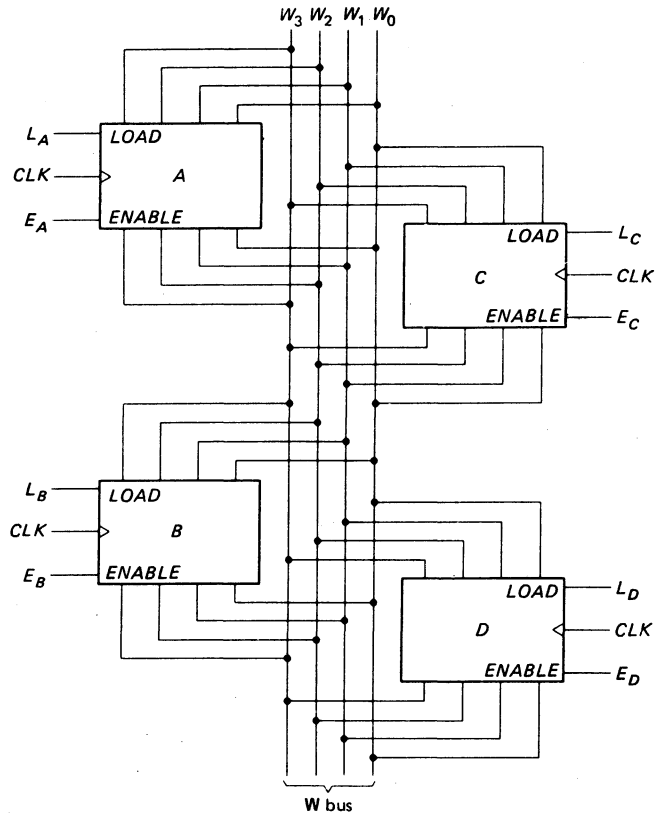


Fig. 8-34

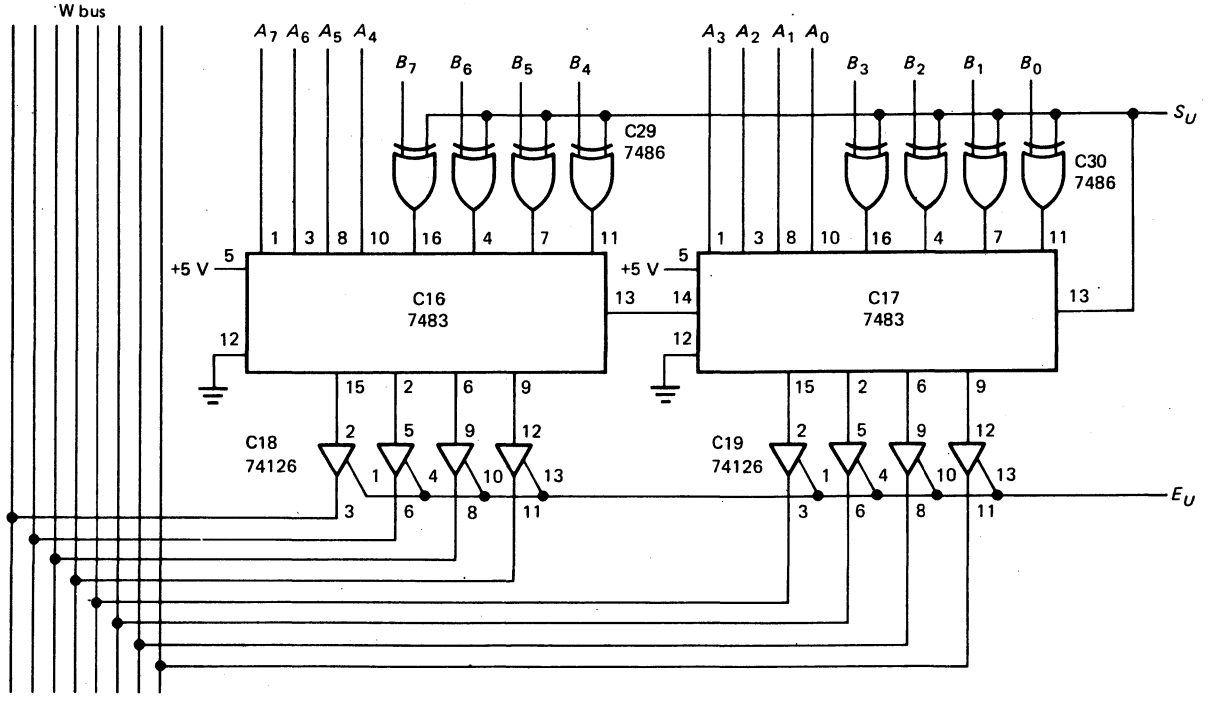


Fig. 8-35

Memories

9

The *memory* of a computer is where the program and data are stored before the calculations begin. During a computer run, the control section may store partial answers in the memory, similar to the way we use paper to record our work. The memory is therefore one of the most active parts of a computer, storing not only the program and data but processed data as well.

The memory is equivalent to thousands of registers, each storing a binary word. The latest generation of computers relies on semiconductor memories because they are less expensive and easier to work with than core memories. A typical microcomputer has a semiconductor memory with up to 65,536 memory locations, each capable of storing 1 byte of information.

9-1 ROMS

A *read-only memory* (ROM) is the simplest kind of memory. It is equivalent to a group of registers, each permanently storing a word. By applying control signals, we can *read* the word in any memory location. ("Read" means to make the contents of the memory location appear at the output terminals of the ROM.)

Diode ROM

Figure 9-1 shows one way to build a ROM. Each horizontal row is a register or memory location. The R_0 register

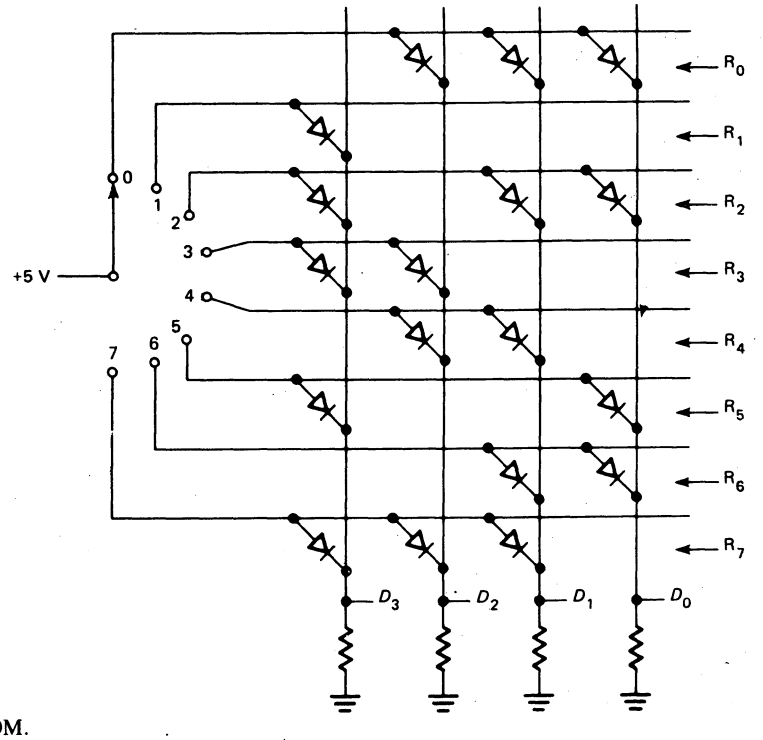


Fig. 9-1 Simple diode ROM.

TABLE 9-1. DIODE ROM

Register	Address	Word
R_0	0	0111
R_1	1	1000
R_2	2	1011
R_3	3	1100
R_4	4	0110
R_5	5	1001
R_6	6	0011
R_7	7	1110

contains three diodes, the R_1 register has one diode, and so on. The output of the ROM is the word

$$\mathbf{D} = D_3D_2D_1D_0$$

In switch position 0, a high voltage turns on the diodes in the R_0 register; all other diodes are off. This means that a high output appears at D_2, D_1, and D_0. Therefore, the word stored at memory location 0 is

$$\mathbf{D} = 0111$$

What happens if the switch is moved to position 1? The diode in the R_1 register conducts, forcing D_3 to go high. Because all other diodes are off, the output from the ROM becomes

$$\mathbf{D} = 1000$$

So the contents of memory location 1 are 1000.

As you move the switch to other positions, you will read the contents of the other memory locations. Table 9-1 shows these contents, which you can check by analyzing Fig. 9-1.

With discrete circuits we can change the contents of a memory location by adding or removing diodes. With integrated circuits, the manufacturer stores the words at the time of fabrication. In either case, the words are permanently stored once the diodes are wired in place.

Addresses

The *address* and *contents* of a memory location are two different things. As shown in Table 9-1, the address of a memory location is the same as the subscript of the register storing the word. This is why register 0 has an address of 0 and contents of 0111; register 1 has an address of 1 and contents of 1000; register 2 has an address of 2 and contents of 1011; and so on.

The idea of addresses applies to ROMs of any size. For example, a ROM with 256 memory locations has decimal addresses running from 0 to 255. A ROM with 1,024 memory locations has decimal addresses from 0 to 1,023.

On-Chip Decoding

Rather than switch-select the memory location, as shown in Fig. 9-1, IC manufacturers use *on-chip decoding*. Figure 9-2 gives you the idea. The three input pins (A_2, A_1, and A_0) supply the binary address of the stored word. Then a 1-of-8 decoder produces a high output to one of the registers.

For instance, if

$$\mathbf{ADDRESS} = A_2A_1A_0 = 100$$

the 1-of-8 decoder applies a high voltage to the R_4 register, and the ROM output is

$$\mathbf{D} = 0110$$

If you change the address word to

$$\mathbf{ADDRESS} = 110$$

you will read the contents of memory location 6, which is

$$\mathbf{D} = 0011$$

The circuit of Fig. 9-2 is a 32-bit ROM organized as 8 words of 4 bits each. It has three address (input) lines and four data (output) lines. This is a very small ROM compared with commercially available ROMs.

Number of Address Lines

With on-chip decoding, n address lines can select 2^n memory locations. For instance, we need 3 address lines in Fig. 9-2 to access 8 memory locations. Similarly, 4 address lines can access 16 memory locations, 8 address lines can access 256 memory locations, and so on.

9-2 PROMS AND EPROMS

With a ROM, you have to send a list of data to be stored in the different memory locations to the manufacturer, who then produces a *mask* (a photographic template of the circuit) used in mass production of your ROMs. In fabricating ROMs the manufacturer may use bipolar transistors or MOSFETs. But the idea is still basically the same; the transistors or MOSFETs act like the diodes of Fig. 9-2.

Programmable

A *programmable* ROM (PROM) is different. It allows the user to store the data. An instrument called a *PROM*

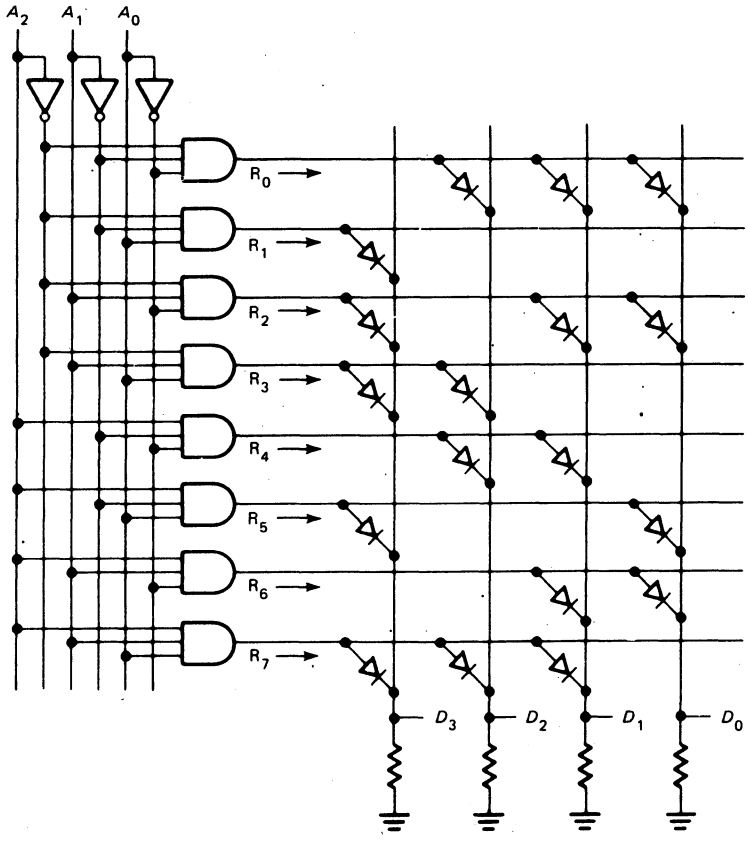


Fig. 9-2 ROM with on-chip decoding.

programmer does the storing by "burning in." (Fusible links at the bit locations can be burned open by high currents.) With a PROM programmer, the user can burn in the program and data. Once this has been done, the programming is permanent. In other words, the stored contents cannot be erased.

Erasable

The *erasable PROM* (EPROM) uses MOSFETs. Data is stored with a PROM programmer. Later, data can be erased with ultraviolet light. The light passes through a window in the IC package to the chip, where it releases stored charges. The effect is to wipe out the stored contents. In other words, the EPROM is ultraviolet-light-erasable and electrically reprogrammable.

The EPROM is helpful in design and development. It allows the user to erase and store until the program and data are perfected. Then the program and data can be sent to an IC manufacturer who produces a ROM mask for mass production.

Manufactured Devices

With large-scale integration, manufacturers can fabricate ROMs, PROMs, and EPROMs that store thousands of words. For instance, the 8355 is a 16,384-bit ROM organized as 2,048 words of 8 bits each. It has 11 address lines and 8 data lines.

As another example, the 2764 is 65,536-bit EPROM organized as 8,192 words of 8 bits each. It has 13 address lines and 8 data lines.

Access Time

The *access time* of a memory is the time it takes to read a stored word after applying address bits. Since bipolar transistors are faster than MOSFETs, bipolar memories have faster access times than MOS memories. For instance, the 3636 is a bipolar PROM with an access time of 80 ns; the 2716 is a MOS EPROM with an access time of 450 ns. You have to pay for the speed; a bipolar memory is more

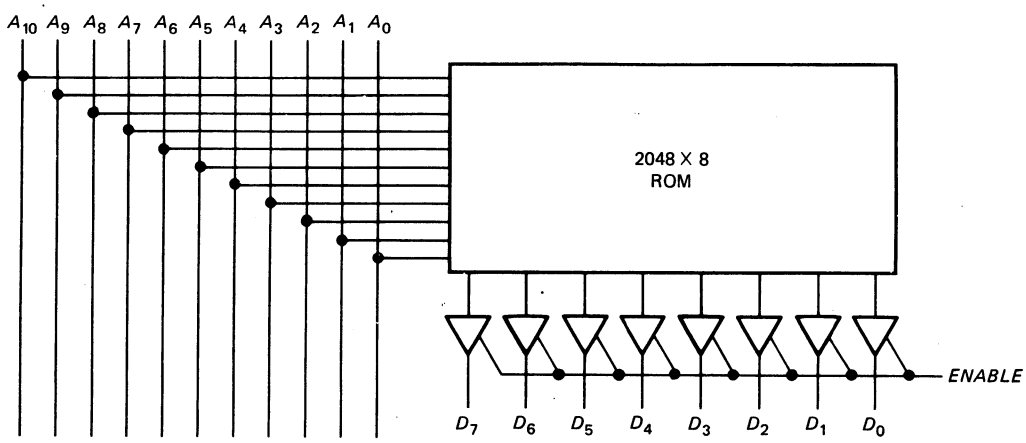


Fig. 9-3. Three-state ROM.

expensive than a MOS memory, so it's up to the designer to decide which type to use in a specific application.

Three-State Memories

By adding three-state switches to the data lines of a memory we can get a three-state output. As an example, Fig. 9-3 shows a 16,384-bit ROM organized as 2,048 words of 8 bits each. It has 11 address lines and 8 data lines. A low *ENABLE* opens all switches and floats the output lines. On the other hand, a high *ENABLE* allows the addressed word to reach the final output.

Most of the commercially available ROMs, PROMs, and EPROMs have three-state outputs. In other words, they have built-in three-state switches that allow you to connect or disconnect the output lines from a data bus. More will be said about this later.

Nonvolatile Memory

ROMs, PROMs, and EPROMs are *nonvolatile memories*. This means that they retain the stored data even when the power to the device is shut off. Not all memories are like this, as will be explained in Sec. 9-3.

EXAMPLE 9-1

A 16 × 8 ROM stores these words in its first four locations:

$$\mathbf{R}_0 = 1110\ 0010$$
$$\mathbf{R}_1 = 0101\ 0111$$
$$\mathbf{R}_2 = 0011\ 1100$$
$$\mathbf{R}_3 = 1011\ 1111$$

Express the stored contents in hexadecimal notation.

SOLUTION

In hexadecimal shorthand, the stored contents are

$$\mathbf{R}_0 = \text{E2H}$$
$$\mathbf{R}_1 = \text{57H}$$
$$\mathbf{R}_2 = \text{3CH}$$
$$\mathbf{R}_3 = \text{BFH}$$

9-3 RAMS

A *random-access memory* (RAM), also called a *read-write* memory, is equivalent to a group of addressable registers. After supplying an address, you can either read the stored contents of the memory location or write new contents into the memory location.

Core RAMs

The core RAM was the workhorse of earlier computers. It has the advantage of being nonvolatile; even though you shut off the power, a core RAM continues to store data. The disadvantage of core RAMs is that they are expensive and harder to work with than semiconductor memories.

Semiconductor RAMs

Semiconductor RAMs may be *static* or *dynamic*. The static RAM uses bipolar or MOS flip-flops; data is retained indefinitely as long as power is applied to the flip-flops. On the other hand, a dynamic RAM uses MOSFETs and capacitors that store data. Because the capacitor charge leaks off, the stored data must be *refreshed* (recharged) every few milliseconds. In either case, the RAMs are volatile; turn off the power and you lose the stored data.

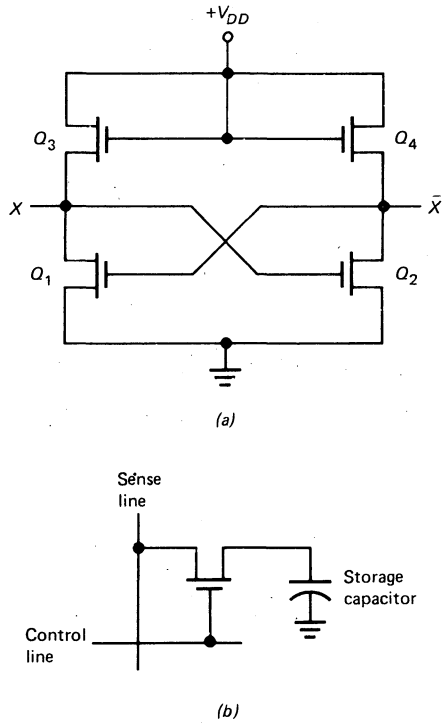


Fig. 9-4 (*a*) Static cell; (*b*) dynamic cell.

Static RAM

Figure 9-4*a* shows one of the flip-flops used in a static, MOS RAM. Q_1 and Q_2 act like switches. Q_3 and Q_4 are active loads, meaning that they behave like resistors. The circuit action is similar to the transistor latch discussed in Sec. 7-1. Either Q_1 conducts and Q_2 is cut off or vice versa. A static RAM will contain thousands of flip-flops like this, one for each stored bit. As long as power is applied, the flip-flop remains latched and can store the bit indefinitely.

Dynamic RAM

Figure 9-4*b* shows one of the memory elements (called *cells*) in a dynamic RAM. When the *sense* and *control* lines go high, the MOSFET conducts and charges the capacitor. When the sense and control lines go low, the MOSFET opens and the capacitor retains its charge. In this way, it can store 1 bit. A dynamic RAM may contain thousands of memory cells like Fig. 9-4*b*. Since only a single MOSFET and capacitor are needed, the dynamic RAM contains more memory cells than a comparable static RAM. In other words, a dynamic RAM has more memory locations than a static RAM of the same physical size.

The disadvantage of the dynamic RAM is the need to refresh the capacitor charge every few milliseconds. This complicates the design problem because more circuitry is needed. In short, it's much simpler to work with static RAMs than dynamic RAMs. The remainder of this book emphasizes static RAMs.

Three-State RAMs

Many of the commercially available RAMs, either static or dynamic, have three-state outputs. In other words, the manufacturer includes three-state switches on the chip so that you can connect or disconnect the output lines of the RAM from a data bus.

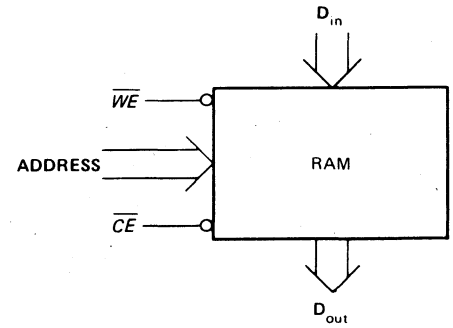


Fig. 9-5 Static RAM with inverted control inputs.

Figure 9-5 shows a static RAM and typical input signals. The **ADDRESS** bits select the memory location; control signals $\overline{WE}$ and $\overline{CE}$ select a write, read, or do nothing operation. $\overline{WE}$ is known as the *write-enable signal,* and $\overline{CE}$ is called the *chip-enable signal*. Notice that the control inputs are active low.

Table 9-2 summarizes the operation of the static RAM. Here's what happens. A low $\overline{CE}$ and low $\overline{WE}$ produce a write operation. This means that the input data $\mathbf{D}_{in}$ is stored in the addressed memory location. The three-state output data lines are floating during this write operation.

When $\overline{CE}$ is low and $\overline{WE}$ is high, we get a read operation. The contents of the addressed memory location appear on the data output lines because the internal three-state switches are closed at this time.

The final possibility is $\overline{CE}$ high. This is a holding pattern where nothing happens. Internal data at all memory locations is frozen or unchanged. Notice that the output data lines are floating.

TABLE 9-2. STATIC RAM

$\overline{CE}$	$\overline{WE}$	Operation	Output
0	0	Write	Floating
0	1	Read	Connected
1	X	Hold	Floating

Bubble Memories

A *bubble memory* sandwiches a thin film of magnetic material between two permanent bias magnets. Logical 1s and 0s are represented by magnetic bubbles in this thin film. The details of how a bubble memory works are too complicated to go into here. What is worth knowing is that bubble memories are nonvolatile and capable of storing huge amounts of data. For instance, the INTEL 7110 is a bubble memory that can store approximately 1 million bits. One disadvantage is they have slow access times.

EXAMPLE 9-2

Figure 9-6 shows the pin configuration of a 74189, a Schottky TTL static RAM with three-state outputs. This 64-bit RAM is organized as 16 words of 4 bits each. It has an access time of 35 ns. What are the different pin functions?

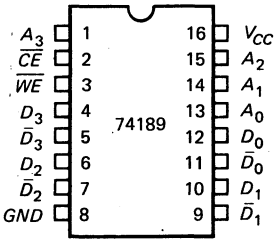


Fig. 9-6 Pinout for 74189.

SOLUTION

To begin with, 4 address bits can access $2^4 = 16$ words. This is why the 74189 needs 4 address bits to select the desired memory location.

The **ADDRESS** bits go to pin 1 (A_3), pin 15 (A_2), pin 14 (A_1), and pin 13 (A_0). The data inputs are pin 4 (D_3), pin 6 (D_2), pin 10 (D_1), and pin 12 (D_0). Because of the TTL design, the data is stored as the complement of the input bits. This is why the data outputs are pin 5 ($\overline{D}_3$), pin 7 ($\overline{D}_2$), pin 9 ($\overline{D}_1$), and pin 11 ($\overline{D}_0$).

The chip enable is pin 2, and the write enable is pin 3. These control signals work as previously described. $\overline{CE}$ and $\overline{WE}$ must be low for a write operation; $\overline{CE}$ must be low and $\overline{WE}$ high for a read, and $\overline{CE}$ must be high to do nothing.

Pin 16 gets the supply voltage, which is +5 V, and pin 8 is grounded.

9-4 A SMALL TTL MEMORY

Figure 9-7 shows a modified version of the SAP-1 memory. Two 74189s (see Appendix 3) are used to get a 16 × 8 memory. This means that we can store 16 words of 8 bits each. The bubbles on the output data pins (pins 5, 7, 9, 11) remind us that the stored data bits are the complements of the input data bits.

Addressing the Memory

The address bits come from an address-switch register (A_3, A_2, A_1, A_0). By setting the switches we can input any address from 0000 to 1111. As noted at the bottom of Fig. 9-7, an up address switch is equal to a 1. Therefore, the address with all switches up is 1111.

Setting Up Data

The data inputs come from the two other switch registers. The upper input nibble is D_7, D_6, D_5, and D_4. The lower input nibble is D_3, D_2, D_1, and D_0. By setting the data switches we can input any data word from 0000 0000 to 1111 1111, equivalent to 00H to FFH. The note at the bottom of Fig. 9-7 indicates that an up data switch produces an input 0 or an output 1. In other words, a data switch must be up to store a 1.

Programming the Memory

To *program the memory* (this means to store instruction and data words), the RUN-PROG switch must be in the PROG position. This grounds pin 2 ($\overline{CE}$) of each 74189. When the READ-WRITE switch is thrown to WRITE, pin 3 ($\overline{WE}$) is grounded and the complement of the input data word it written into the addressed memory location.

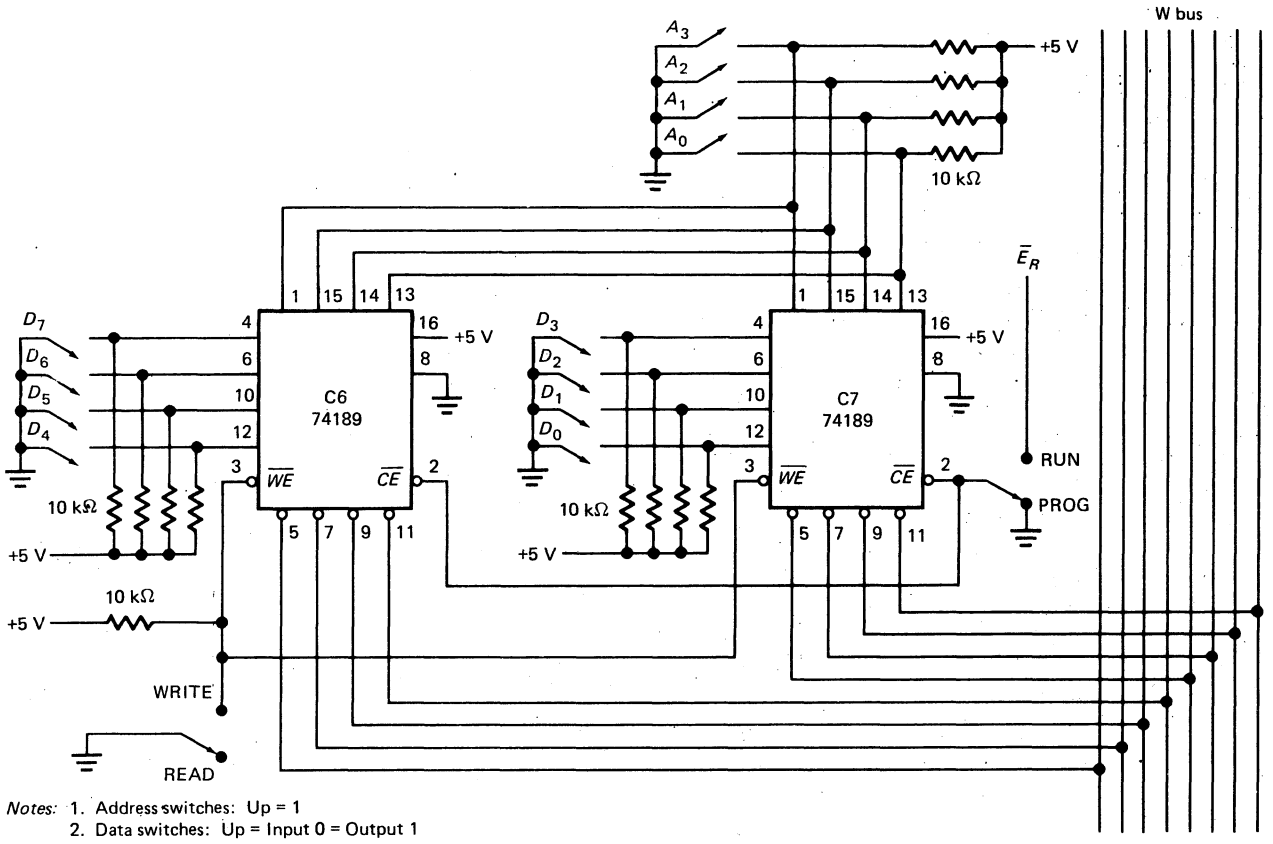
For instance, suppose we want to store the following words:

Address	Data
0000	0000 1111
0001	0010 1110
0010	0001 1101
0011	1110 1000

Begin by placing the RUN-PROG switch in the PROG position. To store the first data word at address 0000, set the switches as follows:

Address	Data
DDDD	DDDD UUUU

where D stands for down and U for up. When the READ-WRITE switch is thrown to WRITE, 0000 1111 is written into memory location 0000. The READ-WRITE switch is then returned to READ in preparation for the next WRITE operation.



Notes: 1. Address switches: Up = 1
2. Data switches: Up = Input 0 = Output 1

Fig. 9-7 Modified SAP-1 read-write memory.

To load the second word at address 0001, set the address and data switches as follows:

Address	Data
DDDU	DDUD UUUD

When the READ-WRITE switch is thrown to WRITE, the data word 0010 1110 is stored at memory location 0001.

Continuing like this, we can program the memory with the remaining words.

The SAP-1 memory is slightly different from Fig. 9-7 and will be discussed in Chap. 10. What we have discussed here, however, gives you an example of how a program and data can be entered into a memory before a computer run.

9-5 HEXADECIMAL ADDRESSES

During a computer run, the CPU sends binary addresses to the memory, where read or write operations occur. These address words may contain 16 or more bits. There's no need for us to get bogged down with long strings of binary numbers. We can chunk those 0s and 1s into neat strings of hexadecimal numbers. Using hexadecimal shorthand is standard in microprocessor work.

Typical microcomputers have an address bus with 16 address lines. The words on this bus have the binary format of

ADDRESS = XXXX XXXX XXXX XXXX

For convenience, we can chunk this into its equivalent hexadecimal form. For instance, instead of writing

ADDRESS = 0101 1110 0111 1100

we can write

ADDRESS = 5E7CH

The 16 address lines can access 2^{16} memory locations, equivalent to 65,536 words. The hexadecimal addresses are from 0000H to FFFFH. In microcomputers using 8-bit microprocessors, 1 byte is stored in each memory location. Figure 9-8 illustrates how to visualize such a memory. The first memory location has an address of 0000H, the second memory location an address of 0001H, the third an address

of 0002H, and so on. Moving toward higher memory, we eventually reach FFFDH, FFFEH, and FFFFH.

Notice that 1 byte is stored in each memory location. This is standard for first-generation microcomputers because they use 8-bit microprocessors like the Z80 and 6502. In other words, typical microcomputers have a maximum memory of 64K (1K = 1,024 bytes).The memory may be less than this, of course. For instance, a TRS80 microcomputer from Radio Shack comes with 16K of ROM and as little as 4K of RAM. It can be upgraded to a maximum of 16K of ROM and 48K of RAM, for a total memory of 64K.

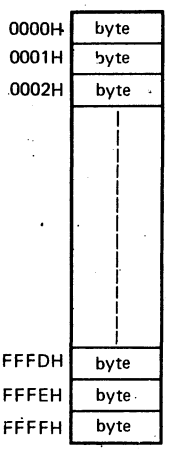


Fig. 9-8 Memory layout.

GLOSSARY

access time The time it takes to read the contents of a memory location after it has been addressed.

address A way of specifying the location of data in memory, similar to a house address.

dynamic memory A memory that relies on a MOSFET switch to charge a capacitor. This memory is highly volatile because not only must the power be kept on, but the capacitor charge must also be refreshed every few milliseconds.

EPROM Erasable programmable read-only memory, a device that is ultraviolet-erasable and electrically reprogrammable.

nonvolatile A type of memory in which the stored data is not lost when the power is turned off.

PROM Programmable read-only memory. With a PROM programmer, you can burn in your own programs and data.

RAM Random-access memory. It is also called a read-write memory because you can read the contents of a memory location or write new contents into it.

ROM Read-only memory. (ROM rhymes with Mom.) This device provides nonvolatile storage of programs and data. You can access any memory location by supplying its address.

static RAM A volatile memory using bipolar or MOSFET flip-flops. It is easy to work with. Refreshing data is unnecessary. You simply supply address and control bits for a read or write operation.

volatile A type of memory in which data stored in the memory is lost when the power is turned off.

SELF-TESTING REVIEW

Read each of the following and provide the missing words. Answers appear at the beginning of the next question.

1. The memory of a computer is where the _____ and _____ are stored before the calculations begin. During a computer run, partial answers may also be stored in the _____.
2. (*program, data, memory*) A read-only memory or _____ is equivalent to a group of memory locations, each permanently storing a word. The _____ is the only one who can store programs and data in a ROM.
3. (*ROM, manufacturer*) The _____ and contents of a memory location are two different things. Because the address is in binary form, the manufacturer uses on-chip decoding to access the memory location. With on-chip decoding, n address lines can access _____ memory locations.
4. (*address, 2^n*) The PROM allows users to store their own programs and data. An instrument called a PROM _____ does the storing or burning in. Once this is done, the programming is permanent.
5. (*programmer*) The _____ is ultraviolet-light-erasable and electrically programmable. This allows the user to erase and store until programs and data are perfected.
6. (*EPROM*) The _____ time of a memory is the

time it takes to read the contents of a memory location. Bipolar memories are faster than _____ memories but more expensive.

7. (*access, MOS*) ROMs, PROMs, and EPROMs are _____ memories. This means that they retain stored data even though the power is turned off. Core RAMs are also _____, but they are becoming obsolete.
8. (*nonvolatile, nonvolatile*) Semiconductor RAM memories may be static or _____. Both are volatile. The first type uses bipolar or MOS flip-flops, which means that data is stored as long as power is applied. The second type uses MOSFETs and capacitors to store data, which must be _____ every few milliseconds.
9. (*dynamic, refreshed*) The memory cell of a dynamic RAM is simpler and smaller than the memory cell of a _____ RAM. Because of this, the dynamic RAM can contains more memory cells than a _____ RAM of the same chip size.
10. (*static, static*) The _____ bits of a static RAM select the memory location. The write enable ($\overline{WE}$) and chip enable ($\overline{CE}$) select a write, read, or do-nothing. When $\overline{WE}$ and $\overline{CE}$ are both low, you get a _____ operation. When $\overline{WE}$ is high and $\overline{CE}$ is low, you get a _____ operation. $\overline{CE}$ high is the inactive state.
11. (*address, write, read*) During a computer run, the CPU sends binary addresses to the _____, where read or write operations occur. Typical microcomputers have an address bus with _____ bits.
12. (*memory, 16*) An address bus with 16 bits can access a maximum of 65,536 memory locations. The hexadecimal addresses of these memory locations are from 0000H to FFFFH. First-generation microcomputers store 1 byte in each memory location, which implies a maximum memory of 64K.

PROBLEMS

9-1. How many memory locations can 14 address bits access?

9-2. The 2708 is an 8,192-bit EPROM organized as a 1,024 × 8 memory. How many address pins does it have?

9-3. The 2732 is a 4,096 × 8 EPROM. How many address lines does it have?

9-4. An 8156 is a 2,048-bit static RAM with 256 words of 8 bits each. How many address lines does this RAM have?

9-5. Use U (up) and D (down) to program the TTL memory of Fig. 9-9 with the following data:

Address	Data
0000	1000 1001
0001	0111 1100
0010	0011 0110
0011	0010 0011
0100	0001 0111
0101	0101 1111
0110	1110 1101
0111	1111 1000

Show your answer by converting each 0 to a D and each 1 to a U.

9-6. The following data is to be programmed into the TTL memory of Fig. 9-9:

Address	Data
0H	EEH
1H	5CH
2H	26H
3H	6AH
4H	FDH
5H	15H
6H	94H
7H	C3H

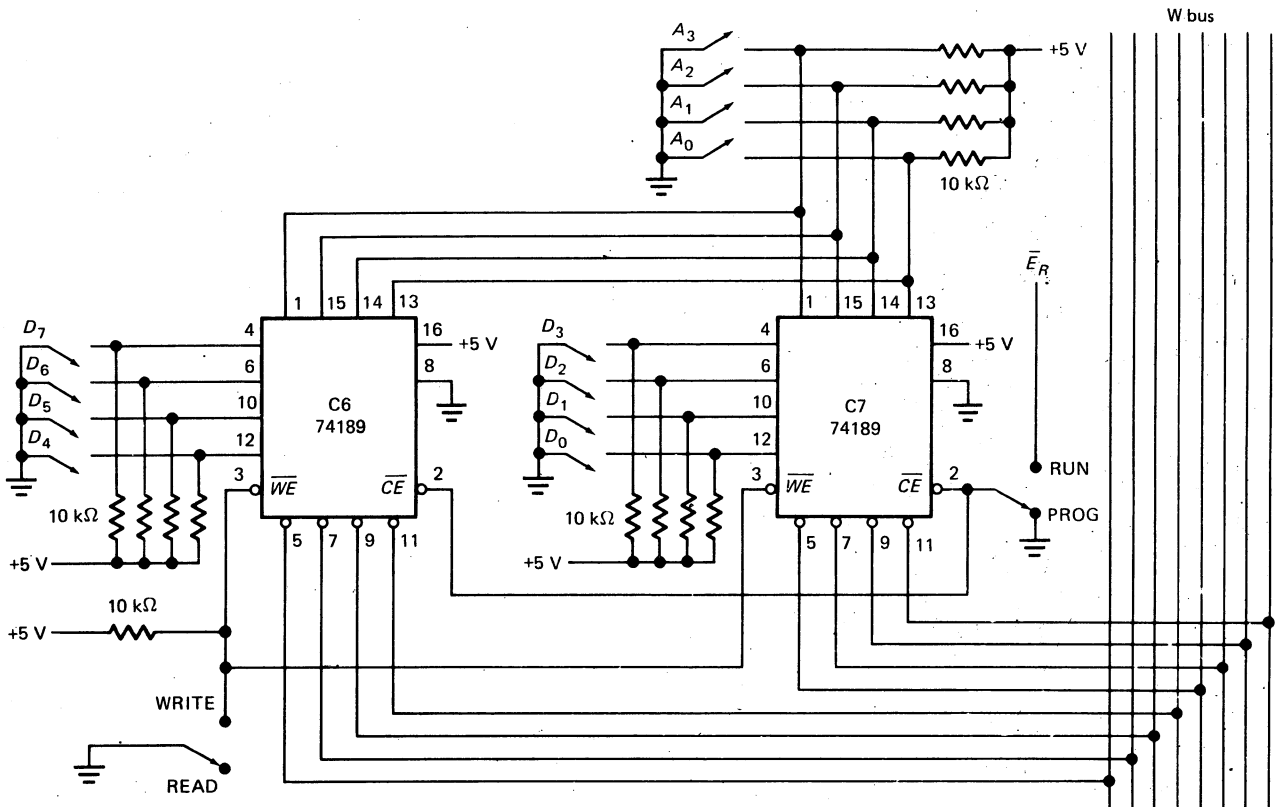
Convert these hexadecimal addresses and contents to ups (U) and downs (D) as described in Sec. 9-4.

9-7. Address 2000H contains the byte 3FH. What is the decimal equivalent of 3FH?

9-8. In a 32K memory, the hexadecimal addresses are from 0000H to 7FFFH. What is the decimal equivalent of the highest address?

9-9. What is the highest address in a 48K memory? Express the answer in hexadecimal and decimal form.

9-10. A byte is stored at hexadecimal location 6F9EH. What is the decimal address? (Use Appendix 1.)



Notes: 1. Address switches: Up = 1
2. Data switches: Up = Input 0 = Output 1

Fig. 9-9

9-11. Here is some data stored in a memory:

Address	Data
8E00H	2FH
8E01H	D4H
8E02H	CFH
8E03H	6EH
8E04H	53H
8E05H	7AH

a. What is the decimal equivalent of each stored byte? (Use Appendix 1.)
b. What is the decimal equivalent of the highest address?

9-12. Suppose there are four different memories with the following capacities:

Memory A = 16K
Memory B = 32K
Memory C = 48K
Memory D = 64K

All memories start with hexadecimal address 0000H.

a. How many bytes can memory C store? Express the answer in decimal.
b. What is the highest decimal address in memory A?
c. We want to store a byte at address C300H. Which memory must we use?
d. What is the highest hexadecimal address for each memory?

SAP-1

10

The SAP (Simple-As-Possible) computer has been designed for you, the beginner. The main purpose of SAP is to introduce all the crucial ideas behind computer operation without burying you in unnecessary detail. But even a simple computer like SAP covers many advanced concepts. To avoid bombarding you with too much all at once, we will examine three different generations of the SAP computer.

SAP-1 is the first stage in the evolution toward modern computers. Although primitive, SAP-1 is a big step for a beginner. So, dig into this chapter; master SAP-1, its architecture, its programming, and its circuits. Then you will be ready for SAP-2.

10-1 ARCHITECTURE

Figure 10-1 shows the *architecture* (structure) of SAP-1, a bus-organized computer. All register outputs to the W bus are three-state; this allows orderly transfer of data. All other register outputs are two-state; these outputs continuously drive the boxes they are connected to.

The layout of Fig. 10-1 emphasizes the registers used in SAP-1. For this reason, no attempt has been made to keep all control circuits in one block called the control unit, all input-output circuits in another block called the I/O unit, etc.

Many of the registers of Fig. 10-1 are already familiar from earlier examples and discussions. What follows is a brief description of each box; detailed explanations come later.

Program Counter

The program is stored at the beginning of the memory with the first instruction at binary address 0000, the second instruction at address 0001, the third at address 0010, and so on. The *program counter,* which is part of the control unit, counts from 0000 to 1111. Its job is to send to the memory the address of the next instruction to be fetched and executed. It does this as follows.

The program counter is reset to 0000 before each computer run. When the computer run begins, the program counter sends address 0000 to the memory. The program counter is then incremented to get 0001. After the first instruction is fetched and executed, the program counter sends address 0001 to the memory. Again the program counter is incremented. After the second instruction is fetched and executed, the program counter sends address 0010 to the memory. In this way, the program counter is keeping track of the next instruction to be fetched and executed.

The program counter is like someone pointing a finger at a list of instructions, saying do this first, do this second, do this third, etc. This is why the program counter is sometimes called a *pointer;* it points to an address in memory where something important is being stored.

Input and MAR

Below the program counter is the *input* and *MAR* block. It includes the address and data switch registers discussed in Sec. 9-4. These switch registers, which are part of the input unit, allow you to send 4 address bits and 8 data bits to the RAM. As you recall, instruction and data words are written into the RAM before a computer run.

The *memory address register* (MAR) is part of the SAP-1 memory. During a computer run, the address in the program counter is latched into the MAR. A bit later, the MAR applies this 4-bit address to the RAM, where a read operation is performed.

The RAM

The *RAM* is a 16 × 8 static TTL RAM. As discussed in Sec. 9-4, you can program the RAM by means of the address and data switch registers. This allows you to store a program and data in the memory before a computer run.

During a computer run, the RAM receives 4-bit addresses from the MAR and a read operation is performed. In this way, the instruction or data word stored in the RAM is placed on the W bus for use in some other part of the computer.

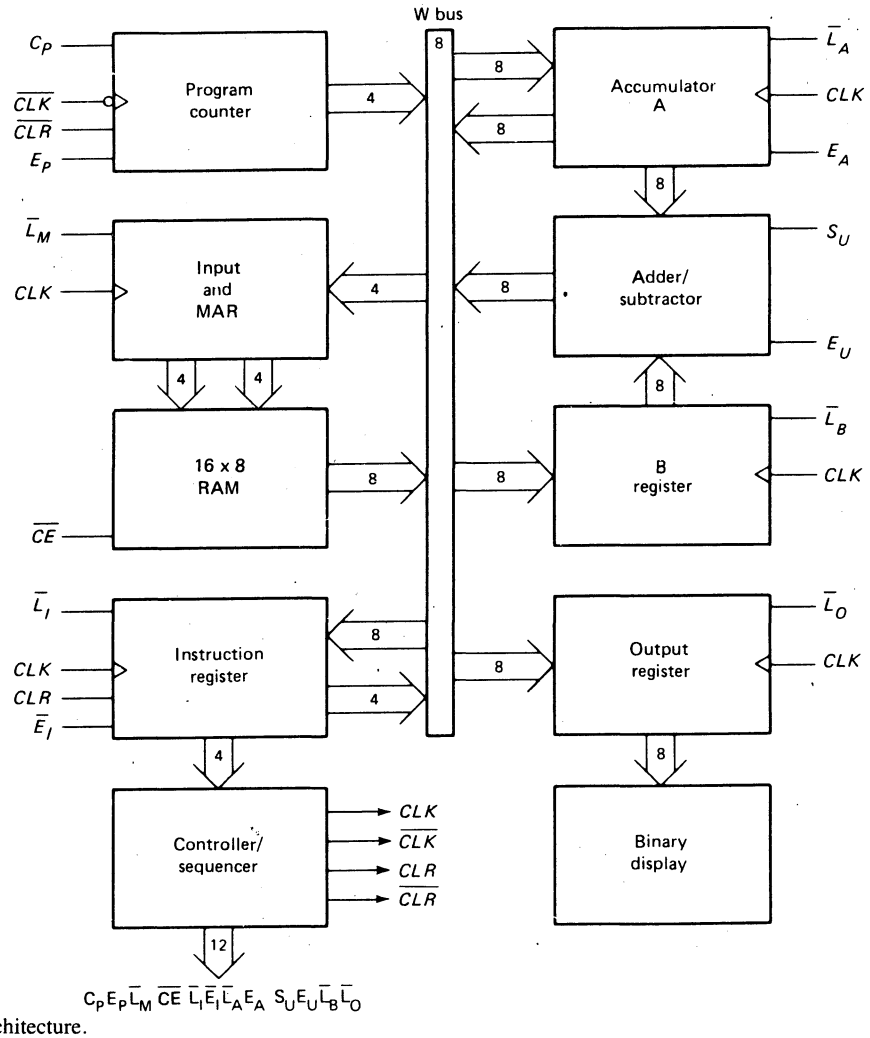


Fig. 10-1 SAP-1 architecture.

Instruction Register

The *instruction register* is part of the control unit. To fetch an instruction from the memory the computer does a memory read operation. This places the contents of the addressed memory location on the W bus. At the same time, the instruction register is set up for loading on the next positive clock edge.

The contents of the instruction register are split into two nibbles. The upper nibble is a two-state output that goes directly to the block labeled "Controller-sequencer." The lower nibble is a three-state output that is read onto the W bus when needed.

Controller-Sequencer

The lower left block contains the *controller-sequencer*. Before each computer run, a $\overline{CLR}$ signal is sent to the program counter and a *CLR* signal to the instruction register. This resets the program counter to 0000 and wipes out the last instruction in the instruction register.

A clock signal *CLK* is sent to all buffer registers; this synchronizes the operation of the computer, ensuring that things happen when they are supposed to happen. In other words, all register transfers occur on the positive edge of a common *CLK* signal. Notice that a $\overline{CLK}$ signal also goes to the program counter.

The 12 bits that come out of the controller-sequencer form a word controlling the rest of the computer (like a supervisor telling others what to do.) The 12 wires carrying the control word are called the *control bus*.

The control word has the format of

$$\mathbf{CON} = C_P E_P \overline{L}_M \overline{CE}\ \ \overline{L}_I \overline{E}_I \overline{L}_A E_A\ \ S_U E_U \overline{L}_B \overline{L}_O$$

This word determines how the registers will react to the next positive *CLK* edge. For instance, a high E_P and a low

L_M mean that the contents of the program counter are latched into the MAR on the next positive clock edge. As another example, a low $\overline{CE}$ and a low $\overline{L}_A$ mean that the addressed RAM word will be transferred to the accumulator on the next positive clock edge. Later, we will examine the timing diagrams to see exactly when and how these data transfers take place.

Accumulator

The *accumulator* (A) is a buffer register that stores intermediate answers during a computer run. In Fig. 10-1 the accumulator has two outputs. The two-state output goes directly to the adder-subtracter. The three-state output goes to the W bus. Therefore, the 8-bit accumulator word continuously drives the adder-subtracter; the same word appears on the W bus when E_A is high.

The Adder-Subtracter

SAP-1 uses a 2's-complement *adder-subtracter*. When S_U is low in Fig. 10-1, the sum out of the adder-subtracter is

$$\mathbf{S = A + B}$$

When S_U is high, the difference appears:

$$\mathbf{A = A + B'}$$

(Recall that the 2's complement is equivalent to a decimal sign change.)

The adder-subtracter is *asynchronous* (unclocked); this means that its contents can change as soon as the input words change. When E_U is high, these contents appear on the W bus.

B Register

The *B register* is another buffer register. It is used in arithmetic operations. A low $\overline{L}_B$ and positive clock edge load the word on the W bus into the B register. The two-state output of the B register drives the adder-subtracter, supplying the number to be added or subtracted from the contents of the accumulator.

Output Register

Example 8-1 discussed the output register. At the end of a computer run, the accumulator contains the answer to the problem being solved. At this point, we need to transfer the answer to the outside world. This is where the *output register* is used. When E_A is high and $\overline{L}_0$ is low, the next positive clock edge loads the accumulator word into the output register.

The output register is often called an *output port* because processed data can leave the computer through this register. In microcomputers the output ports are connected to *interface circuits* that drive peripheral devices like printers, cathode-ray tubes, teletypewriters, and so forth. (An interface circuit prepares the data to drive each device.)

Binary Display

The *binary display* is a row of eight light-emitting diodes (LEDs). Because each LED connects to one flip-flop of the output port, the binary display shows us the contents of the output port. Therefore, after we've transferred an answer from the accumulator to the output port, we can see the answer in binary form.

Summary

The SAP-1 control unit consists of the program counter, the instruction register, and the controller-sequencer that produces the control word, the clear signals, and the clock signals. The SAP-1 ALU consists of an accumulator, an adder-subtracter, and a B register. The SAP-1 memory has the MAR and a 16 × 4 RAM. The I/O unit includes the input programming switches, the output port, and the binary display.

10-2 INSTRUCTION SET

A computer is a useless pile of hardware until someone programs it. This means loading step-by-step instructions into the memory before the start of a computer run. Before you can program a computer, however, you must learn its *instruction set,* the basic operations it can perform. The SAP-1 instruction set follows.

LDA

As described in Chap. 9, the words in the memory can be symbolized by $\mathbf{R}_0$, $\mathbf{R}_1$, $\mathbf{R}_2$, etc. This means that $\mathbf{R}_0$ is stored at address 0H, $\mathbf{R}_1$ at address 1H, $\mathbf{R}_2$ at address 2H, and so on.

LDA stands for "load the accumulator." A complete LDA instruction includes the hexadecimal address of the data to be loaded. LDA 8H, for example, means "load the accumulator with the contents of memory location 8H." Therefore, given

$$\mathbf{R}_8 = 1111\ 0000$$

the execution of LDA 8H results in

$$\mathbf{A} = 1111\ 0000$$

Similarly, LDA AH means "load the accumulator with the contents of memory location AH," LDA FH means "load the accumulator with the contents of memory location FH," and so on.

ADD

ADD is another SAP-1 instruction. A complete ADD instruction includes the address of the word to be added. For instance, ADD 9H means "add the contents of memory location 9H to the accumulator contents"; the sum replaces the original contents of the accumulator.

Here's an example. Suppose decimal 2 is in the accumulator and decimal 3 is in memory location 9H. Then

$$\mathbf{A} = 0000\ 0010$$
$$\mathbf{R_9} = 0000\ 0011$$

During the execution of ADD 9H, the following things happen. First, $\mathbf{R_9}$ is loaded into the B register to get

$$\mathbf{B} = 0000\ 0011$$

and almost instantly the adder-subtracter forms the sum of **A** and **B**

$$\mathbf{SUM} = 0000\ 0101$$

Second, this sum is loaded into the accumulator to get

$$\mathbf{A} = 0000\ 0101$$

The foregoing routine is used for all ADD instructions; the addressed RAM word goes to the B register and the adder-subtracter output to the accumulator. This is why the execution of ADD 9H adds $\mathbf{R_9}$ to the accumulator contents, the execution of ADD FH adds $\mathbf{R_F}$ to the accumulator contents, and so on.

SUB

SUB is another SAP-1 instruction. A complete SUB instruction includes the address of the word to be subtracted. For example, SUB CH means "subtract the contents of memory location CH from the contents of the accumulator"; the difference out of the adder-subtracter then replaces the original contents of the accumulator.

For a concrete example, assume that decimal 7 is in the accumulator and decimal 3 is in memory location CH. Then

$$\mathbf{A} = 0000\ 0111$$
$$\mathbf{R_C} = 0000\ 0011$$

The execution of SUB CH takes place as follows. First, $\mathbf{R_C}$ is loaded into the B register to get

$$\mathbf{B} = 0000\ 0011$$

and almost instantly the adder-subtracter forms the difference of **A** and **B**:

$$\mathbf{DIFF} = 0000\ 0100$$

Second, this difference is loaded into the accumulator and

$$\mathbf{A} = 0000\ 0100$$

The foregoing routine applies to all SUB instructions; the addressed RAM word goes to the B register and the adder-subtracter output to the accumulator. This is why the execution of SUB CH subtracts $\mathbf{R_C}$ from the contents of the accumulator, the execution of SUB EH subtracts $\mathbf{R_E}$ from the accumulator, and so on.

OUT

The instruction OUT tells the SAP-1 computer to transfer the accumulator contents to the output port. After OUT has been executed, you can see the answer to the problem being solved.

OUT is complete by itself; that is, you do not have to include an address when using OUT because the instruction does not involve data in the memory.

HLT

HLT stands for halt. This instruction tells the computer to stop processing data. HLT marks the end of a program, similar to the way a period marks the end of a sentence. You must use a HLT instruction at the end of every SAP-1 program; otherwise, you get computer trash (meaningless answers caused by runaway processing).

HLT is complete by itself; you do not have to include a RAM word when using HLT because this instruction does not involve the memory.

Memory-Reference Instructions

LDA, ADD, and SUB are called *memory-reference instructions* because they use data stored in the memory. OUT and HLT, on the other hand, are not memory-reference instructions because they do not involve data stored in the memory.

Mnemonics

LDA, ADD, SUB, OUT, and HLT are the instruction set for SAP-1. Abbreviated instructions like these are called *mnemonics* (memory aids). Mnemonics are popular in computer work because they remind you of the operation that will take place when the instruction is executed. Table 10-1 summarizes the SAP-1 instruction set.

The 8080 and 8085

The 8080 was the first widely used microprocessor. It has 72 instructions. The 8085 is an enhanced version of the 8080 with essentially the same instruction set. To make SAP practical, the SAP instructions will be upward com-

TABLE 10-1. SAP-1 INSTRUCTION SET

Mnemonic	Operation
LDA	Load RAM data into accumulator
ADD	Add RAM data to accumulator
SUB	Subtract RAM data from accumulator
OUT	Load accumulator data into output register
HLT	Stop processing

patible with the 8080/8085 instruction set. In other words, the SAP-1 instructions LDA, ADD, SUB, OUT, and HLT are 8080/8085 instructions. Likewise, the SAP-2 and SAP-3 instructions will be part of the 8080/8085 instruction set. Learning SAP instructions is getting you ready for the 8080 and 8085, two very widely used microprocessors. Once you learn the 8080/8085 instruction set, you can branch out to other microprocessors.

EXAMPLE 10-1

Here's a SAP-1 program in mnemonic form:

Address	Mnemonics
0H	LDA 9H
1H	ADD AH
2H	ADD BH
3H	SUB CH
4H	OUT
5H	HLT

The data in higher memory is

Address	Data
6H	FFH
7H	FFH
8H	FFH
9H	01H
AH	02H
BH	03H
CH	04H
DH	FFH
EH	FFH
FH	FFH

What does each instruction do?

SOLUTION

The program is in the low memory, located at addresses 0H to 5H. The first instruction loads the accumulator with the contents of memory location 9H, and so the accumulator contents become

$$A = 01H$$

The second instruction adds the contents of memory location AH to the accumulator contents to get a new accumulator total of

$$A = 01H + 02H = 03H$$

Similarly, the third instruction add the contents of memory location BH

$$A = 03H + 03H = 06H$$

The SUB instruction subtracts the contents of memory location CH to get

$$A = 06H - 04H = 02H$$

The OUT instruction loads the accumulator contents into the output port: therefore, the binary display shows

0000 0010

The HLT instruction stops the data processing.

10-3 PROGRAMMING SAP-1

To load instruction and data words into the SAP-1 memory we have to use some kind of code that the computer can interpret. Table 10-2 shows the code used in SAP-1. The number 0000 stands for LDA, 0001 for ADD, 0010 for SUB, 1110 for OUT, and 1111 for HLT. Because this code tells the computer which operation to perform, it is called an *operation code* (op code).

As discussed earlier, the address and data switches of Fig. 9-7 allow you to program the SAP-1 memory. By design, these switches produce a 1 in the up position (U)

TABLE 10-2. SAP-1 OP CODE

Mnemonic	Op code
LDA	0000
ADD	0001
SUB	0010
OUT	1110
HLT	1111

and a 0 in the down position (D). When programming the data switches with an instruction, the op code goes into the upper nibble, and the *operand* (the rest of the instruction) into the lower nibble.

For instance, suppose we want to store the following instructions:

Address	Instruction
0H	LDA FH
1H	ADD EH
2H	HLT

First, convert each instruction to binary as follows:

LDA FH = 0000 1111
ADD EH = 0001 1110
HLT = 1111 XXXX

In the first instruction, 0000 is the op code for LDA, and 1111 is the binary equivalent of FH. In the second instruction, 0001 is the op code for ADD, and 1110 is the binary equivalent of EH. In the third instruction, 1111 is the op code for HLT, and XXXX are don't cares because the HLT is not a memory-reference instruction.

Next, set up the address and data switches as follows:

Address	Data
DDDD	DDDD UUUU
DDDU	DDDU UUUD
DDUD	UUUU XXXX

After each address and data word is set, you press the write button. Since D stores a binary 0 and U stores a binary 1, the first three memory locations now have these contents:

Address	Contents
0000	0000 1111
0001	0001 1110
0010	1111 XXXX

A final point. *Assembly language* involves working with mnemonics when writing a program. *Machine language* involves working with strings of 0s and 1s. The following examples bring out the distinction between the two languages.

EXAMPLE 10-2

Translate the program of Example 10-1 into SAP-1 machine language.

SOLUTION

Here is the program of Example 10-1:

Address	Instruction
0H	LDA 9H
1H	ADD AH
2H	ADD BH
3H	SUB CH
4H	OUT
5H	HLT

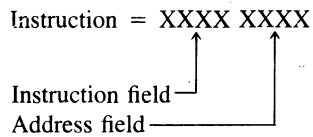
This program is in assembly language as it now stands. To get it into machine language, we translate it to 0s and 1s as follows:

Address	Instruction
0000	0000 1001
0001	0001 1010
0010	0001 1011
0011	0010 1100
0100	1110 XXXX
0101	1111 XXXX

Now the program is in machine language.

Any program like the foregoing that's written in machine language is called an *object program*. The original program with mnemonics is called a *source program*. In SAP-1 the operator translates the source program into an object program when programming the address and data switches.

A final point. The four MSBs of a SAP-1 machine-language instruction specify the operation, and the four LSBs give the address. Sometimes we refer to the MSBs as the *instruction field* and to the LSBs as the *address field*. Symbolically,



EXAMPLE 10-3

How would you program SAP-1 to solve this arithmetic problem?

$$16 + 20 + 24 - 32$$

The numbers are in decimal form.

SOLUTION

One way is to use the program of the preceding example, storing the data (16, 20, 24, 32) in memory locations 9H

to CH. With Appendix 1, you can convert the decimal data into hexadecimal data to get this assembly-language version:

Address	Contents
0H	LDA 9H
1H	ADD AH
2H	ADD BH
3H	SUB CH
4H	OUT
5H	HLT
6H	XX
7H	XX
8H	XX
9H	10H
AH	14H
BH	18H
CH	20H

The machine-language version is

Address	Contents
0000	0000 1001
0001	0001 1010
0010	0001 1011
0011	0010 1100
0100	1110 XXXX
0101	1111 XXXX
0110	XXXX XXXX
0111	XXXX XXXX
1000	XXXX XXXX
1001	0001 0000
1010	0001 0100
1011	0001 1000
1100	0010 0000

Notice that the program is stored ahead of the data. In other words, the program is in low memory and the data in high memory. This is essential in SAP-1 because the program counter points to address 0000 for the first instruction, 0001 for the second instruction, and so forth.

EXAMPLE 10-4

Chunk the program and data of the preceding example by converting to hexadecimal shorthand.

SOLUTION

Address	Contents
0H	09H
1H	1AH
2H	1BH
3H	2CH
4H	EXH
5H	FXH
6H	XXH
7H	XXH
8H	XXH
9H	10H
AH	14H
BH	18H
CH	20H

This version of the program and data is still considered machine language.

Incidentally, negative data is loaded in 2's-complement form. For example, −03H is entered as FDH.

10-4 FETCH CYCLE

The *control unit* is the key to a computer's automatic operation. The control unit generates the control words that fetch and execute each instruction. While each instruction is fetched and executed, the computer passes through different *timing states* (*T* states), periods during which register contents change. Let's find out more about these *T* states.

Ring Counter

Earlier, we discussed the SAP-1 ring counter (see Fig. 8-16 for the schematic diagram). Figure 10-2*a* symbolizes the ring counter, which has an output of

$$\mathbf{T} = T_6T_5T_4T_3T_2T_1$$

At the beginning of a computer run, the ring word is

$$\mathbf{T} = 000001$$

Successive clock pulses produce ring words of

$$\mathbf{T} = 000010$$
$$\mathbf{T} = 000100$$
$$\mathbf{T} = 001000$$
$$\mathbf{T} = 010000$$
$$\mathbf{T} = 100000$$

Then, the ring counter resets to 000001, and the cycle repeats. Each ring word represents one *T* state.

Figure 10-2*b* shows the timing pulses out of the ring counter. The initial state T_1 starts with a negative clock edge and ends with the next negative clock edge. During this *T* state, the T_1 bit out of the ring counter is high.

During the next state, T_2 is high; the following state has a high T_3; then a high T_4; and so on. As you can see, the

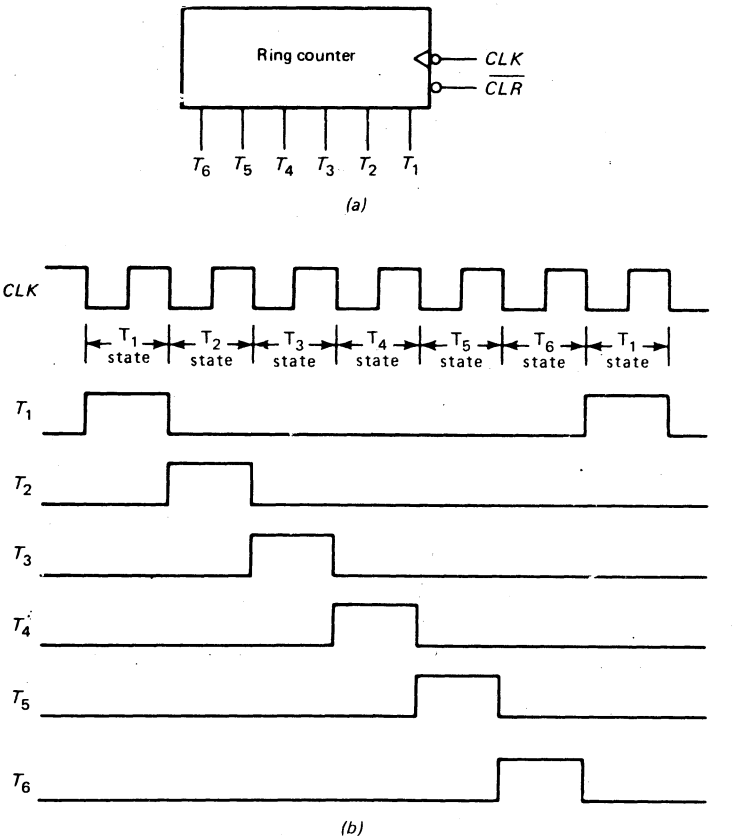


Fig. 10-2 Ring counter: (*a*) symbol; (*b*) clock and timing signals.

ring counter produces six T states. Each instruction is fetched and executed during these six T states.

Notice that a positive *CLK* edge occurs midway through each T state. The importance of this will be brought out later.

Address State

The T_1 state is called the *address state* because the address in the program counter (PC) is transferred to the memory address register (MAR) during this state. Figure 10-3*a* shows the computer sections that are active during this state (active parts are light; inactive parts are dark).

During the address state, E_P and $\overline{L}_M$ are active; all other control bits are inactive. This means that the controller-sequencer is sending out a control word of

$$\begin{aligned}\text{CON} &= C_P E_P \overline{L}_M \overline{CE} \quad \overline{L}_I \overline{E}_I \overline{L}_A E_A \quad S_U E_U \overline{L}_B \overline{L}_O \\ &= 0\ 1\ 0\ 1 \quad\ 1\ 1\ 1\ 0 \quad\ 0\ 0\ 1\ 1\end{aligned}$$

during this state

Increment State

Figure 10-3*b* shows the active parts of SAP-1 during the T_2 state. This state is called the *increment state* because the program counter is incremented. During the increment state, the controller-sequencer is producing a control word of

$$\begin{aligned}\text{CON} &= C_P E_P \overline{L}_M \overline{CE} \quad \overline{L}_I \overline{E}_I \overline{L}_A E_A \quad S_U E_U \overline{L}_B \overline{L}_O \\ &= 1\ 0\ 1\ 1 \quad\ 1\ 1\ 1\ 0 \quad\ 0\ 0\ 1\ 1\end{aligned}$$

As you see, the C_P bit is active.

Memory State

The T_3 state is called the *memory state* because the addressed RAM instruction is transferred from the memory to the instruction register. Figure 10-3*c* shows the active parts of SAP-1 during the memory state. The only active control bits during this state are $\overline{CE}$ and $\overline{L}_I$, and the word out of the controller-sequencer is

$$\begin{aligned}\text{CON} &= C_P E_P \overline{L}_M \overline{CE} \quad \overline{L}_I \overline{E}_I \overline{L}_A E_A \quad S_U E_U \overline{L}_B \overline{L}_O \\ &= 0\ 0\ 1\ 0 \quad\ 0\ 1\ 1\ 0 \quad\ 0\ 0\ 1\ 1\end{aligned}$$

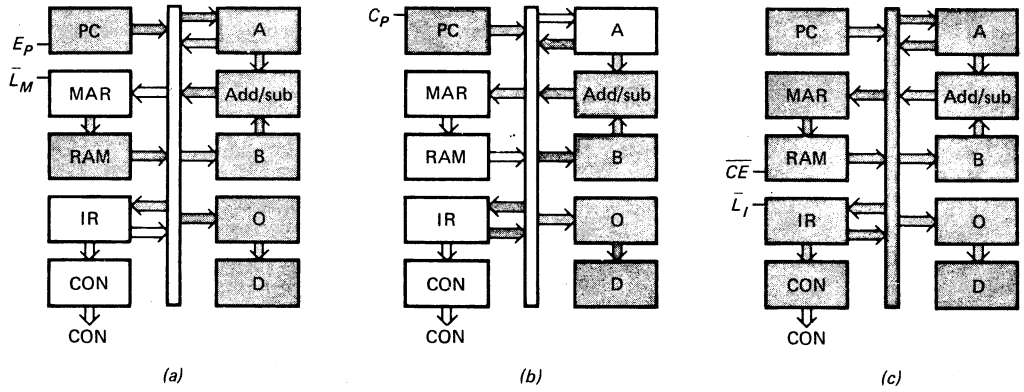


Fig. 10-3 Fetch cycle: (*a*) T_1 state; (*b*) T_2 state; (*c*) T_3 state.

Fetch Cycle

The address, increment, and memory states are called the *fetch cycle* of SAP-1. During the address state, E_P and $\overline{L}_M$ are active; this means that the program counter sets up the MAR via the W bus. As shown earlier in Fig. 10-2*b*, a positive clock edge occurs midway through the address state; this loads the MAR with the contents of the PC.

C_P is the only active control bit during the increment state. This sets up the program counter to count positive clock edges. Halfway through the increment state, a positive clock edge hits the program counter and advances the count by 1.

During the memory state, $\overline{CE}$ and $\overline{L}_I$ are active. Therefore, the addressed RAM word sets up the instruction register via the W bus. Midway through the memory state, a positive clock edge loads the instruction register with the addressed RAM word.

10-5 EXECUTION CYCLE

The next three states (T_4, T_5, and T_6) are the execution cycle of SAP-1. The register transfers during the execution cycle depend on the particular instruction being executed. For instance, LDA 9H requires different register transfers than ADD BH. What follows are the *control routines* for different SAP-1 instructions.

LDA Routine

For a concrete discussion, let's assume that the instruction register has been loaded with LDA 9H:

$$\mathbf{IR} = 0000\ 1001$$

During the T_4 state, the instruction field 0000 goes to the controller-sequencer, where it is decoded; the address field 1001 is loaded into the MAR. Figure 10-4*a* shows the active parts of SAP-1 during the T_4 state. Note that $\overline{E}_I$ and $\overline{L}_M$ are active; all other control bits are inactive.

During the T_5 state, $\overline{CE}$ and $\overline{L}_A$ go high. This means that the addressed data word in the RAM will be loaded into the accumulator on the next positive clock edge (see Fig. 10-4*b*).

T_6 is a no-operation state. During this third execution state, all registers are inactive (Fig. 10-4*c*). This means that the controller-sequencer is sending out a word whose bits are all inactive. *Nop* (pronounced *no op*) stands for "no operation." The T_6 state of the LDA routine is a nop.

Figure 10-5 shows the timing diagram for the fetch and LDA routines. During the T_1 state, E_P and $\overline{L}_M$ are active; the positive clock edge midway through this state will transfer the address in the program counter to the MAR. During the T_2 state, C_P is active and the program counter is incremented on the positive clock edge. During the T_3 state, $\overline{CE}$ and $\overline{L}_I$ are active; when the positive clock edge occurs, the addressed RAM word is transferred to the instruction register. The LDA execution starts with the T_4 state, where $\overline{L}_M$ and $\overline{E}_I$ are active; on the positive clock edge the address field in the instruction register is transferred to the MAR. During the T_5 state, $\overline{CE}$ and $\overline{L}_A$ are active; this means that the addressed RAM data word is transferred to the accumulator on the positive clock edge. As you know, the T_6 state of the LDA routine is a nop.

ADD Routine

Suppose at the end of the fetch cycle the instruction register contains ADD BH:

$$\mathbf{IR} = 0001\ 1011$$

During the T_4 state the instruction field goes to the controller-sequencer and the address field to the MAR (see Fig. 10-6*a*). During this state $\overline{E}_I$ and $\overline{L}_M$ are active.

Control bits $\overline{CE}$ and $\overline{L}_B$ are active during the T_5 state. This allows the addressed RAM word to set up the B

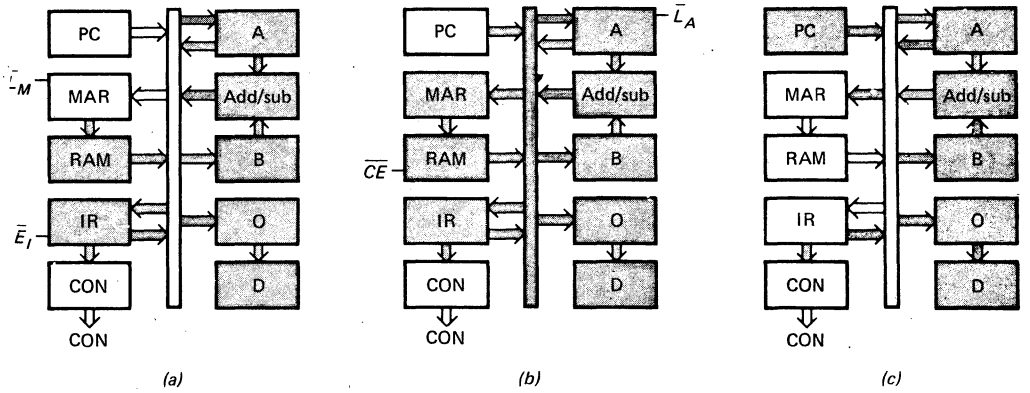


Fig. 10-4 LDA routine: (*a*) T_4 state; (*b*) T_5 state; (*c*) T_6 state.

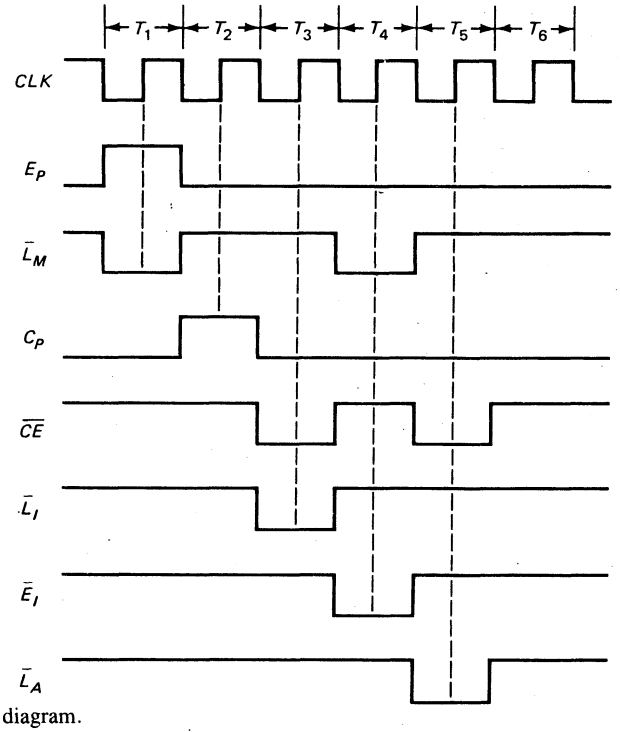


Fig. 10-5 Fetch and LDA timing diagram.

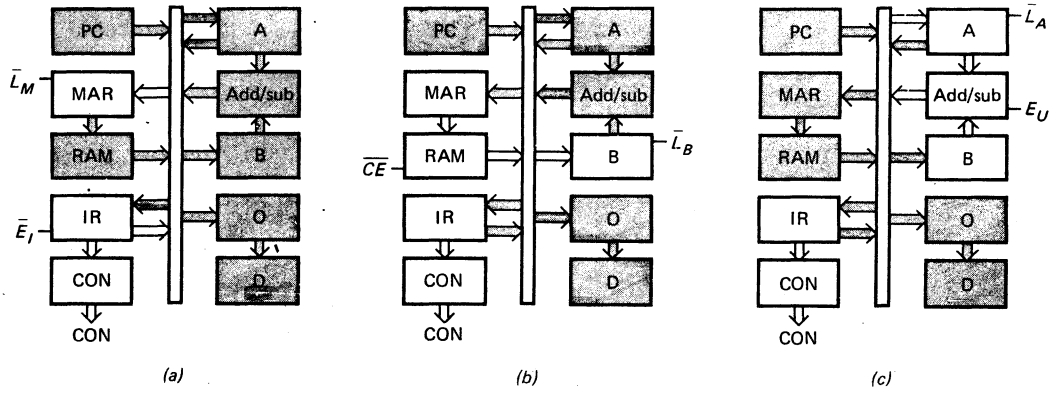


Fig. 10-6 ADD and SUB routines: (*a*) T_4 state; (*b*) T_5 state; (*c*) T_6 state.

register (Fig. 10-6*b*). As usual, loading takes place midway through the state when the positive clock edge hits the *CLK* input of the B register.

During the T_6 state, E_U and $\overline{L}_A$ are active; therefore, the adder-subtracter sets up the accumulator (Fig. 10-6*c*). Halfway through this state, the positive clock edge loads the sum into the accumulator.

Incidentally, setup time and propagation delay time prevent racing of the accumulator during this final execution state. When the positive clock edge hits in Fig. 10-6*c*, the accumulator contents change, forcing the adder-subtracter contents to change. The new contents return to the accumulator input, but the new contents don't get there until two propagation delays after the positive clock edge (one for the accumulator and one for the adder-subtracter). By then it's too late to set up the accumulator. This prevents accumulator racing (loading more than once on the same clock edge).

Figure 10-7 shows the timing diagram for the fetch and ADD routines. The fetch routine is the same as before: the T_1 state loads the PC address into the MAR; the T_2 state increments the program counter; the T_3 state sends the addressed instruction to the instruction register.

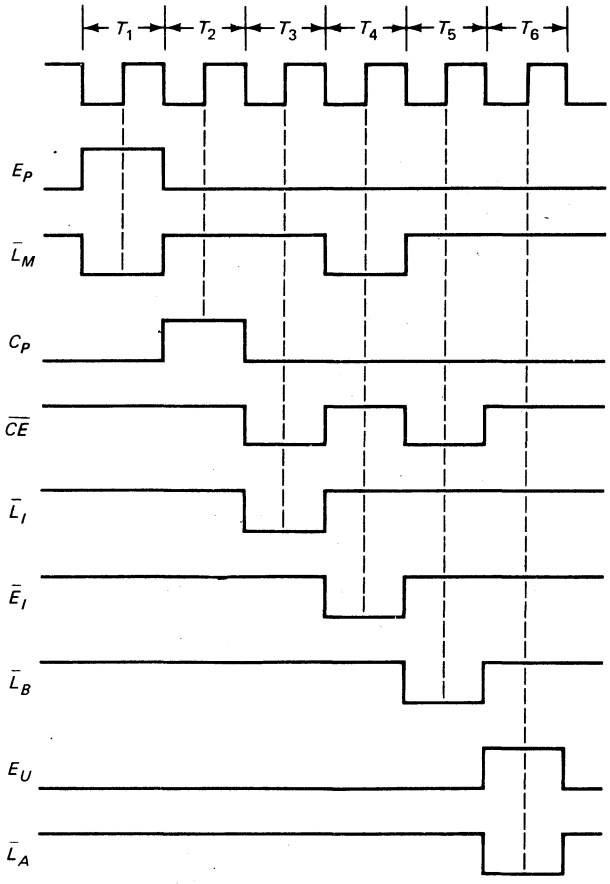


Fig. 10-7 Fetch and ADD timing diagram.

During the T_4 state, $\overline{E}_I$ and $\overline{L}_M$ are active; on the next positive clock edge, the address field in the instruction register goes to the MAR. During the T_5 state, $\overline{CE}$ and $\overline{L}_B$ are active; therefore, the addressed RAM word is loaded into the B register midway through the state. During the T_6 state, $\overline{E}_U$ and $\overline{L}_A$ are active; when the positive clock edge hits, the sum out of the adder-subtracter is stored in the accumulator.

SUB Routine

The SUB routine is similar to the ADD routine. Figure 10-6*a* and *b* show the active parts of SAP-1 during the T_4 and T_5 states. During the T_6 state, a high S_U is sent to the adder-subtracter of Fig. 10-6*c*. The timing diagram is almost identical to Fig. 10-7. Visualize S_U low during the T_1 to T_5 states and S_U high during the T_6 state.

OUT Routine

Suppose the instruction register contains the OUT instruction at the end of a fetch cycle. Then

$$\mathbf{IR} = 1110\ XXXX$$

The instruction field goes to the controller-sequencer for decoding. Then the controller-sequencer sends out the control word needed to load the accumulator contents into the output register.

Figure 10-8 shows the active sections of SAP-1 during the execution of an OUT instruction. Since E_A and $\overline{L}_O$ are active, the next positive clock edge loads the accumulator contents into the output register during the T_4 state. The T_5 and T_6 states are nops.

Figure 10-9 is the timing diagram for the fetch and OUT routines. Again, the fetch cycle is same: address state, increment state, and memory state. During the T_4 state, E_A and $\overline{L}_O$ are active; this transfers the accumulator word to the output register when the positive clock edge occurs.

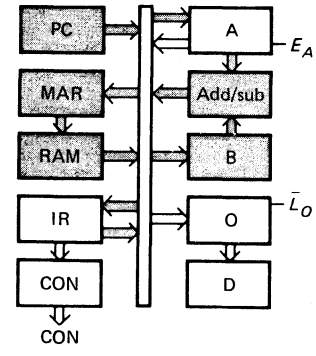


Fig. 10-8 T_4 state of OUT instruction.

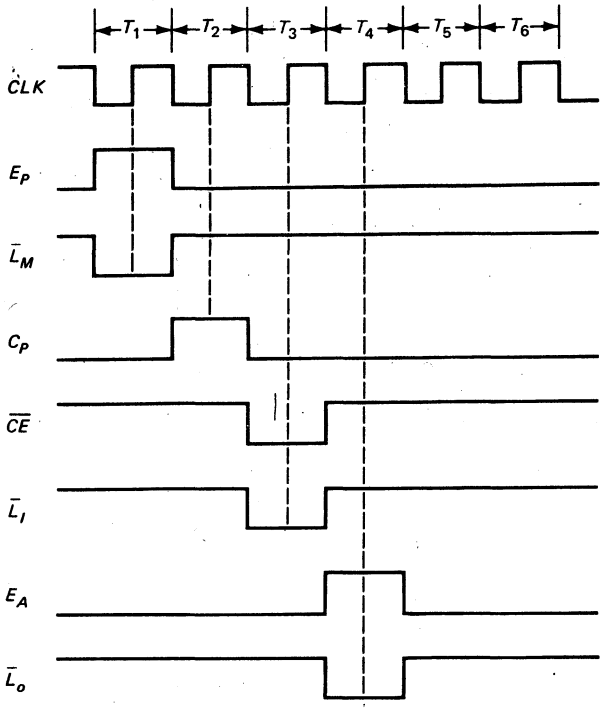


Fig. 10-9 Fetch and OUT timing diagram.

HLT

HLT does not require a control routine because no registers are involved in the execution of an HLT instruction. When the IR contains

IR = 1111 XXXX

the instruction field 1111 signals the controller-sequencer to stop processing data. The controller-sequencer stops the computer by turning off the clock (circuitry discussed later).

Machine Cycle and Instruction Cycle

SAP-1 has six *T* states (three fetch and three execute). These six states are called a *machine cycle* (see Fig. 10-10*a*). It takes one machine cycle to fetch and execute each instruction. The SAP-1 clock has a frequency of 1 kHz, equivalent to a period of 1 ms. Therefore, it takes 6 ms for a SAP-1 machine cycle.

SAP-2 is slightly different because some of its instructions take more than one machine cycle to fetch and execute. Figure 10-10*b* shows the timing for an instruction that requires two machine cycles. The first three *T* states are the fetch cycle; however, the execution cycle requires the next nine *T* states. This is because a two-machine-cycle instruction is more complicated and needs those extra *T* states to complete the execution.

The number of *T* states needed to fetch and execute an instruction is called the *instruction cycle*. In SAP-1 the instruction cycle equals the machine cycle. In SAP-2 and other microcomputers the instruction cycle may equal two or more machine cycles, as shown in Fig. 10-10*b*.

The instruction cycles for the 8080 and 8085 take from one to five machine cycles (more on this later).

EXAMPLE 10-5

The 8080/8085 programming manual says that it takes thirteen *T* states to fetch and execute the LDA instruction.

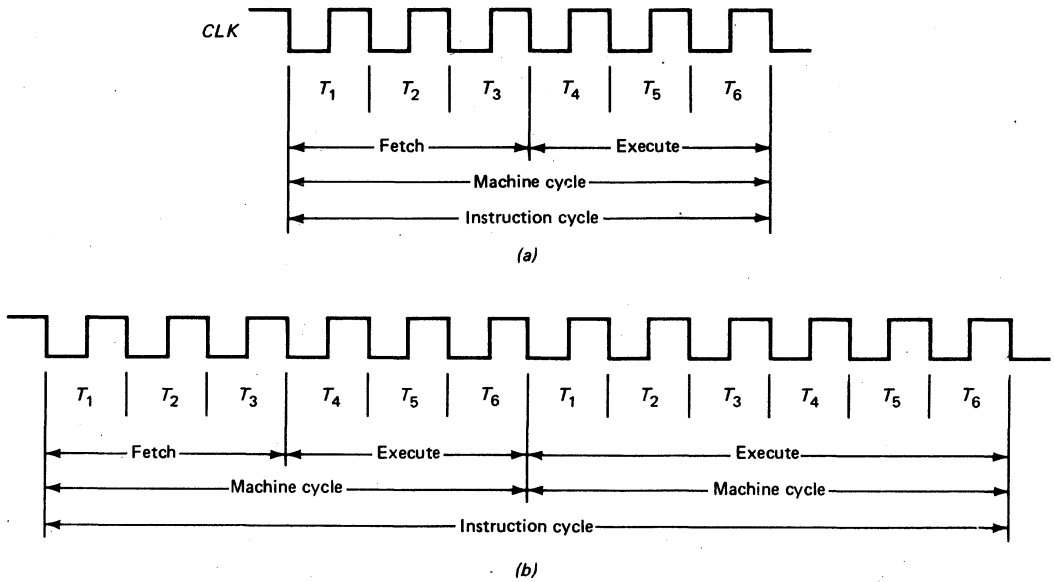


Fig. 10-10 (*a*) SAP-1 instruction cycle; (*b*) instruction cycle with two machine cycles.

If the system clock has a frequency of 2.5 MHz, how long is an instruction cycle?

SOLUTION

The period of the clock is

$$T = \frac{1}{f} = \frac{1}{2.5\,\text{MHz}} = 400\,\text{ns}$$

Therefore, each T state lasts 400 ns. Since it takes thirteen T states to fetch and execute the LDA instruction, the instruction cycle lasts for

$$13 \times 400\,\text{ns} = 5{,}200\,\text{ns} = 5.2\,\mu\text{s}$$

EXAMPLE 10-6

Figure 10-11 shows the six T states of SAP-1. The positive clock edge occurs halfway through each state. Why is this important?

SOLUTION

SAP-1 is a *bus-organized computer* (the common type nowadays). This allows its registers to communicate via the W bus. But reliable loading of a register takes place only when the setup and hold times are satisfied. Waiting half a cycle before loading the register satisfies the setup time; waiting half a cycle after loading satisfies the hold time. This is why the positive clock edge is designed to strike the registers halfway through each T state (Fig. 10-11).

There's another reason for waiting half a cycle before loading a register. When the *ENABLE* input of the sending register goes active, the contents of this register are suddenly dumped on the W bus. Stray capacitance and lead inductance prevent the bus lines from reaching their correct voltage levels immediately. In other words, we get transients on the W bus and have to wait for them to die out to ensure valid data at the time of loading. The half-cycle delay before clocking allows the data to settle before loading.

10-6 THE SAP-1 MICROPROGRAM

We will soon be analyzing the schematic diagram of the SAP-1 computer, but first we need to summarize the execution of SAP-1 instructions in a neat table called a *microprogram*.

Microinstructions

The controller-sequencer sends out control words, one during each T state or clock cycle. These words are like directions telling the rest of the computer what to do. Because it produces a small step in the data processing, each control word is called a *microinstruction*. When looking at the SAP-1 block diagram (Fig. 10-1), we can visualize a steady stream of microinstructions flowing out of the controller-sequencer to the other SAP-1 circuits.

Macroinstructions

The instructions we have been programming with (LDA, ADD, SUB, . . .) are sometimes called *macroinstructions* to distinguish them from microinstructions. Each SAP-1 macroinstruction is made up of three microinstructions. For example, the LDA macroinstruction consists of the microinstructions in Table 10-3. To simplify the appearance of these microinstructions, we can use hexadecimal chunking as shown in Table 10-4.

Table 10-5 shows the SAP-1 microprogram, a listing of each macroinstruction and the microinstructions needed to carry it out. This table summarizes the execute routines for the SAP-1 instructions. A similar table can be used with more advanced instruction sets.

10-7 THE SAP-1 SCHEMATIC DIAGRAM

In this section we examine the complete schematic diagram for SAP-1. Figures 10-12 to 10-15 show all the chips, wires, and signals. You should refer to these figures throughout the following discussion. Appendix 3 gives additional details for some of the more complicated chips.

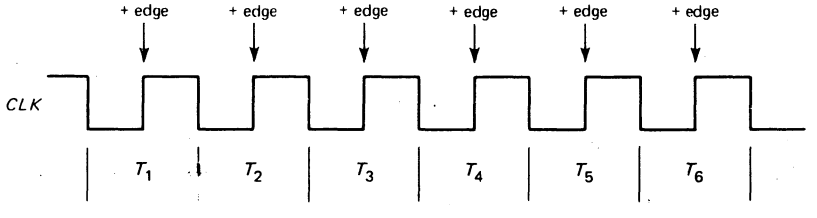


Fig. 10-11 Positive clock edges occur midway through T states.

TABLE 10-3

Macro	State	$C_P E_P \overline{L}_M \overline{CE}$	$\overline{L}_I \overline{E}_I \overline{L}_A E_A$	$S_U E_U \overline{L}_B \overline{L}_O$	Active
LDA	T_4	0 0 0 1	1 0 1 0	0 0 1 1	$\overline{L}_M$, $\overline{E}_I$
	T_5	0 0 1 0	1 1 0 0	0 0 1 1	$\overline{CE}$, $\overline{L}_A$
	T_6	0 0 1 1	1 1 1 0	0 0 1 1	None

TABLE 10-4

Macro	State	CON	Active
LDA	T_4	1A3H	$\overline{L}_M$, $\overline{E}_I$
	T_5	2C3H	$\overline{CE}$, $\overline{L}_A$
	T_6	3E3H	None

TABLE 10-5. SAP-1 MICROPROGRAM†

Macro	State	CON	Active
LDA	T_4	1A3H	$\overline{L}_M$, $\overline{E}_I$
	T_5	2C3H	$\overline{CE}$, $\overline{L}_A$
	T_6	3E3H	None
ADD	T_4	1A3H	$\overline{L}_M$, $\overline{E}_I$
	T_5	2E1H	$\overline{CE}$, $\overline{L}_B$
	T_6	3C7H	$\overline{L}_A$, E_U
SUB	T_4	1A3H	$\overline{L}_M$, $\overline{E}_I$
	T_5	2E1H	$\overline{CE}$, $\overline{L}_B$
	T_6	3CFH	$\overline{L}_A$, S_U, E_U
OUT	T_4	3F2H	E_A, $\overline{L}_O$
	T_5	3E3H	None
	T_6	3E3H	None

† **CON** = $C_P E_P \overline{L}_M \overline{CE}$ $\overline{L}_I \overline{E}_I \overline{L}_A E_A$ $S_U E_U \overline{L}_B \overline{L}_O$.

Program Counter

Chips C1, C2, and C3 of Fig. 10-12 are the *program counter*. Chip C1, a 74LS107, is a dual *JK* master-slave flip-flop, that produces the upper 2 address bits. Chip C2, another 74LS107, produces the lower 2 address bits. Chip C3 is a 74LS126, a quad three-state normally open switch; it gives the program counter a three-state output.

At the start of a computer run, a low $\overline{CLR}$ resets the program counter to 0000. During the T_1 state, a high E_P places the address on the W bus. During the T_2 state, a high C_P is applied to the program counter; midway through this state, the negative $\overline{CLK}$ edge (equivalent to positive *CLK* edge) increments the program counter.

The program counter is inactive during the T_3 to T_6 states.

MAR

Chip C4, a 74LS173, is a 4-bit buffer register; it serves as the MAR. Notice that pins 1 and 2 are grounded; this converts the three-state output to a two-state output. In other words, the output of the MAR is not connected to the W bus, and so there's no need to use the three-state output.

2-to-1 Multiplexer

Chip C5 is a 74LS157, a 2-to-1 nibble *multiplexer*. The left nibble (pins 14, 11, 5, 2) comes from the address switch register (S_1). The right nibble (pins 13, 10, 6, 3) comes from the MAR. The RUN-PROG switch (S_2) selects the nibble to reach to the output of C5. When S_2 is in the PROG position, the nibble out of the address switch register is selected. On the other hand, when S_2 is the RUN position, the output of the MAR is selected.

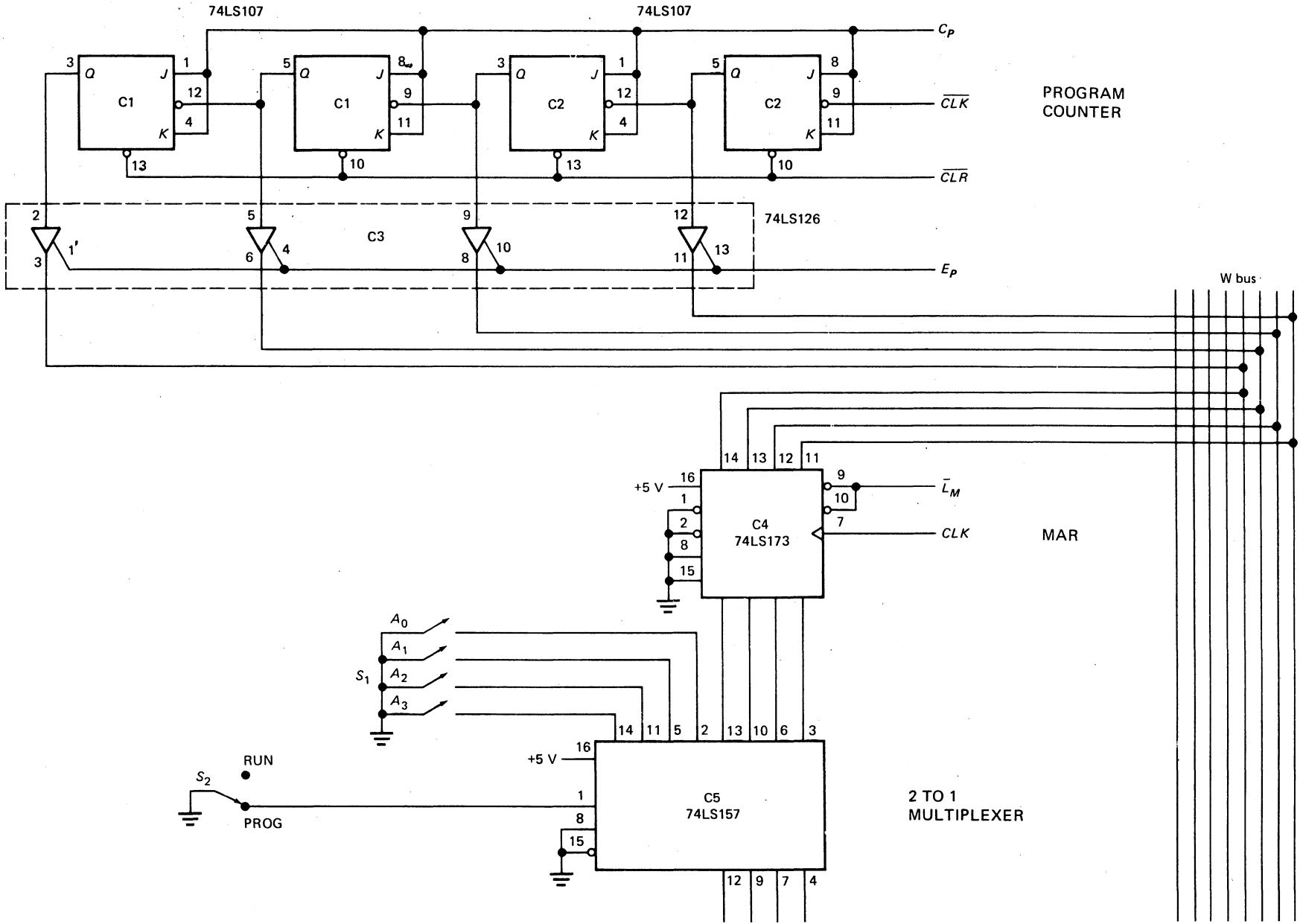
16 × 8 RAM

Chips C6 and C7 are 74189s. Each chip is a 16 × 4 static RAM. Together, they give us a 16 × 8 *read-write memory*. S_3 is the data switch register (8 bits), and S_4 is the read-write switch (a push-button switch). To program the memory, S_2 is put in the PROG position; this takes the $\overline{CE}$ input low (pin 2). The address and data switches are then set to the correct address and data words. A momentary push of the read-write switch takes $\overline{WE}$ low (pin 3) and loads the memory.

After the program and data are in memory, the RUN-PROG switch (S_2) is put in the RUN position in preparation for the computer run.

Instruction Register

Chips C8 and C9 are 74LS173s. Each chip is a 4-bit three-state buffer register. The two chips are the *instruction register*. Grounding pins 1 and 2 of C8 converts the three-state output to a two-state output, $I_7 I_6 I_5 I_4$. This nibble goes to the instruction decoder in the controller-sequencer. Signal $\overline{E}_I$ controls the output of C9, the lower nibble in the instruction register. When $\overline{E}_I$ is low, this nibble is placed on the W bus.


74LS107
74LS107
C1
C1
C2
C2
C_P
$\overline{CLK}$
$\overline{CLR}$
PROGRAM COUNTER
C3
74LS126
E_P
W bus
+5 V
C4
74LS173
$\overline{L}_M$
CLK
MAR
A_0
A_1
A_2
A_3
S_1
RUN
S_2
PROG
+5 V
C5
74LS157
2 TO 1
MULTIPLEXER

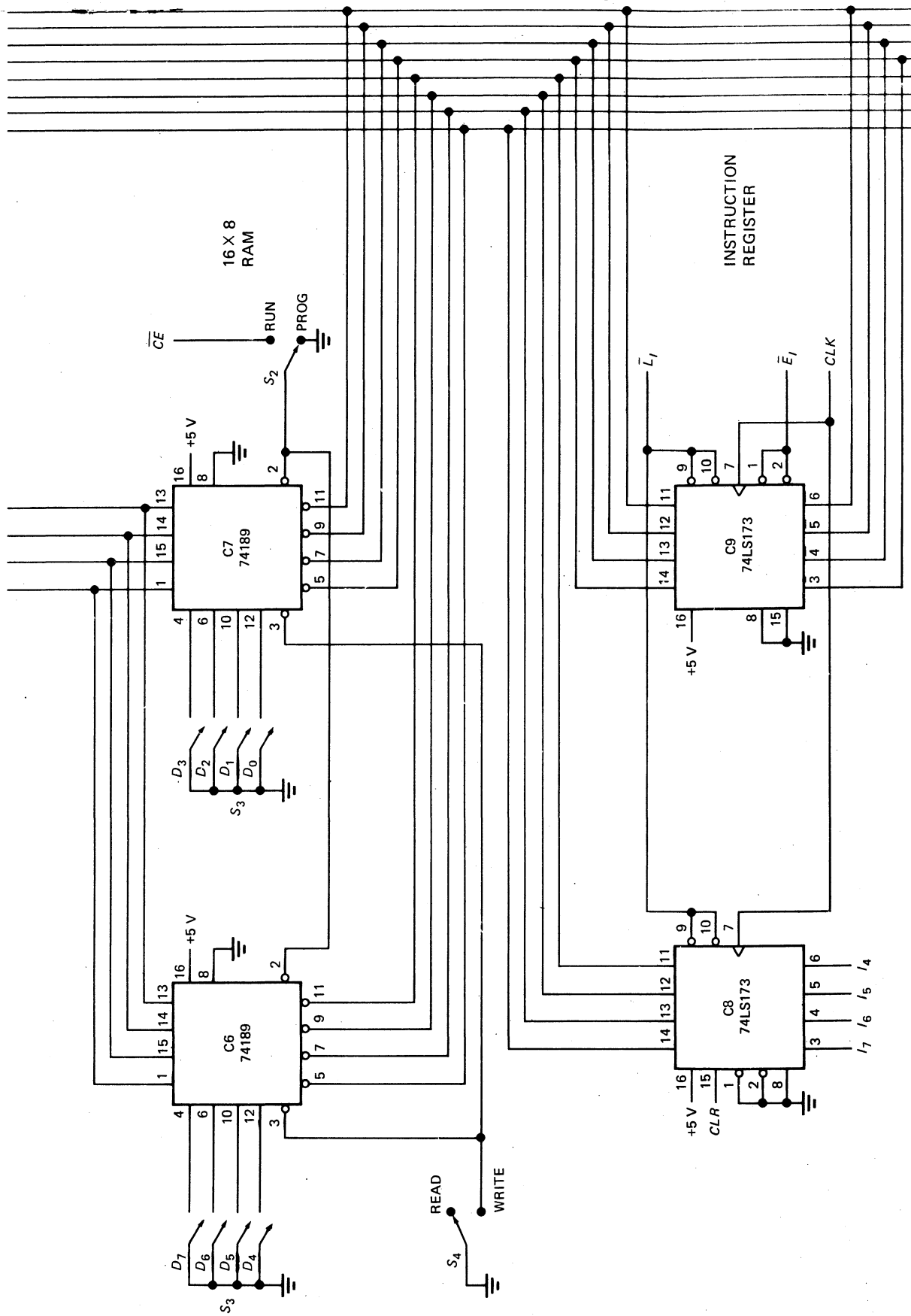
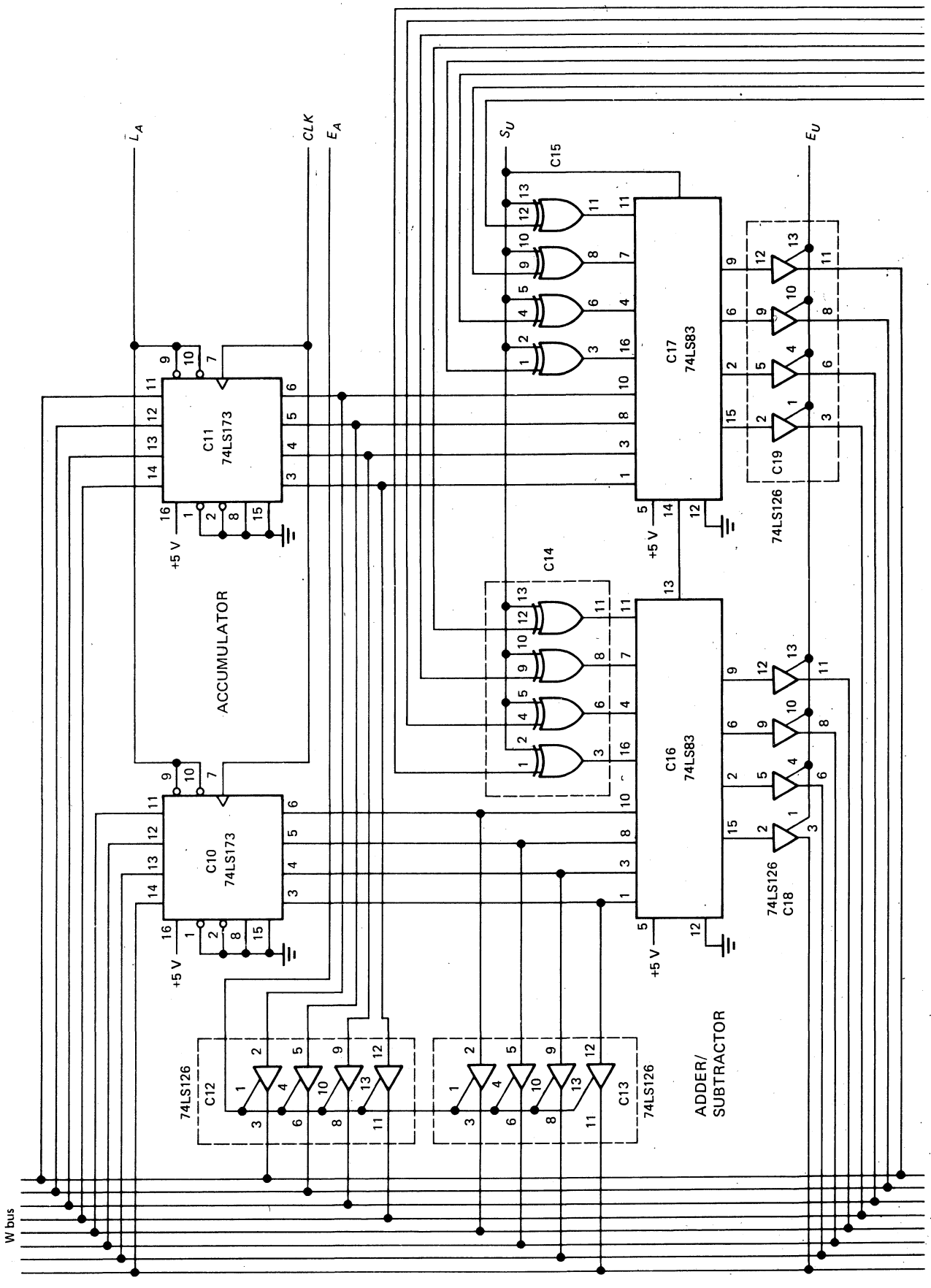


Fig. 10-12 SAP-1 program counter, memory, and instruction register.


$\bar{L}_A$
CLK
E_A
S_U
E_U
C15
C11
74LS173
C17
74LS83
C19
74LS126
C14
ACCUMULATOR
C10
74LS173
C16
74LS83
74LS126
C18
+5 V
74LS126
C12
74LS126
C13
ADDER/
SUBTRACTOR
W bus

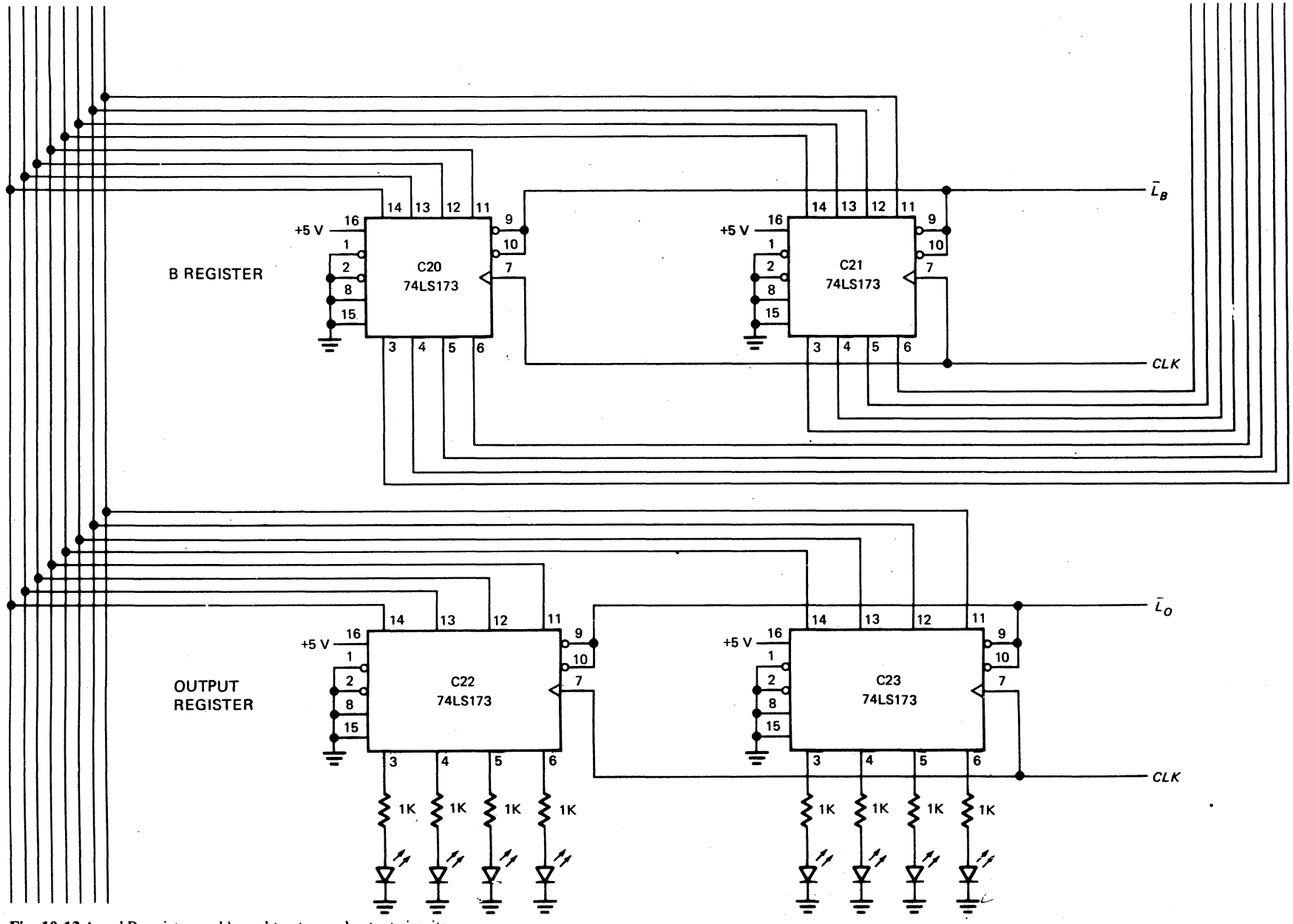


Fig. 10-13 A and B registers, adder-subtracter, and output circuits.

Accumulator

Chips C10 and C11, 74LS173s, are the *accumulator* (see Fig. 10-13). Pins 1 and 2 are grounded on both chips to produce a two-state output for the adder-subtracter. Chips C12 and C13 are 74LS126s; these three-state switches place the accumulator contents on the W bus when E_A is high.

Adder-subtracter

Chips C14 and C15 are 74LS86s. These EXCLUSIVE-OR gates are a controlled inverter. When S_U is low, the contents of the B register are transmitted. When S_U is high, the 1's complement is transmitted and a 1 is added to the LSB to form the 2's complement.

Chips C16 and C17 are 74LS83s. These 4-bit full adders combine to produce an 8-bit sum or difference. Chips C18 and C19, which are 74LS126s, convert this 8-bit answer into a three-state output for driving the W bus.

B Register and Output Register

Chips C20 and C21, which are 74LS173s, form the *B register*. It contains the data to be added or subtracted from the accumulator. Grounding pins 1 and 2 of both chips produces a two-state output for the adder-subtracter.

Chips C22 and C23 are 74LS173s and form the output register. It drives the binary display and lets us see the processed data.

Clear-Start Debouncer

In Fig. 10-14, the *clear-start debouncer* produces two outputs: *CLR* for the instruction register and $\overline{CLR}$ for the program counter and ring counter. *CLR* also goes to C29, the clock-start flip-flop. S_S is a push-button switch. When depressed, it goes to the CLEAR position, generating a high *CLR* and a low $\overline{CLR}$. When S_S is released, it returns to the START position, producing a low *CLR* and a high $\overline{CLR}$.

Notice that half of C24 is used for the clear-start debouncer and the other half for the single-step debouncer. Chip C24 is a 7400, a quad 2-input NAND gate.

Single-Step Debouncer

SAP-1 can run in either of two modes, manual or automatic. In the manual mode, you press and release S_6 to generate one clock pulse. When S_6 is depressed, *CLK* is high; when released, *CLK* is low. In other words, the *single-step debouncer* of Fig. 10-14 generates the *T* states one at a time as you press and release the button. This allows you to step through the different *T* states while troubleshooting or debugging. (Debugging means looking for errors in your program. You troubleshoot hardware and debug software.)

Manual-Auto Debouncer

Switch S_7 is a single-pole double-throw (SPDT) switch that can remain in either the MANUAL position or the AUTO position. When in MANUAL, the single-step button is active. When in AUTO, the computer runs automatically. Two of the NAND gates in C26 are used to debounce the MANUAL-AUTO switch. The other two NAND C26 gates are part of a NAND-NAND network that steers the single-step clock or the automatic clock to the final *CLK* and $\overline{CLK}$ outputs.

Clock Buffers

The output of pin 11, C26, drives the *clock buffers*. As you see in Fig. 10-14, two inverters are used to produce the final *CLK* output and one inverter to produce the $\overline{CLK}$ output. Unlike most of the other chips, C27 is standard TTL rather than a low-power Schottky (see SAP-1 Parts List, Appendix 4). Standard TTL is used because it can drive 20 low-power Schottky TTL loads, as indicated in Table 4-5.

If you check the data sheets of the 74LS107 and 74LS173 for input currents, you will be able to count the following low-power Schottky (LS) TTL loads on the clock and clear signals:

$$\begin{aligned} CLK &= 19 \text{ LS loads} \\ \overline{CLK} &= 2 \text{ LS loads} \\ CLR &= 1 \text{ LS load} \\ \overline{CLR} &= 20 \text{ LS loads} \end{aligned}$$

This means that the *CLK* and $\overline{CLK}$ signals out of C27 (standard TTL) are adequate to drive the low-power Schottky TTL loads. Also, the *CLR* and $\overline{CLR}$ signals out of C24 (standard TTL) can drive their loads.

Clock Circuits and Power Supply

Chip C28 is a 555 timer. This IC produces a rectangular 2-kHz output with a 75 percent duty cycle. As previously discussed, a *start-the-clock flip-flop* (C29) divides the signal down to 1 kHz and at the same time produces a 50 percent duty cycle.

The *power supply* consists of a full-wave bridge rectifier working into a capacitor-input filter. The dc voltage across the 1,000-μF capacitor is approximately 20 V. Chip C30, an LM340T-5, is a voltage regulator that produces a stable output of +5 V.

Instruction Decoder

Chip C31, a hex inverter, produces complements of the op-code bits, $I_7I_6I_5I_4$ (see Fig. 10-15). Then chips C32, C33, and C34 decode the op code to produce five output signals: *LDA*, *ADD*, *SUB*, *OUT*, and $\overline{HLT}$. Remember:

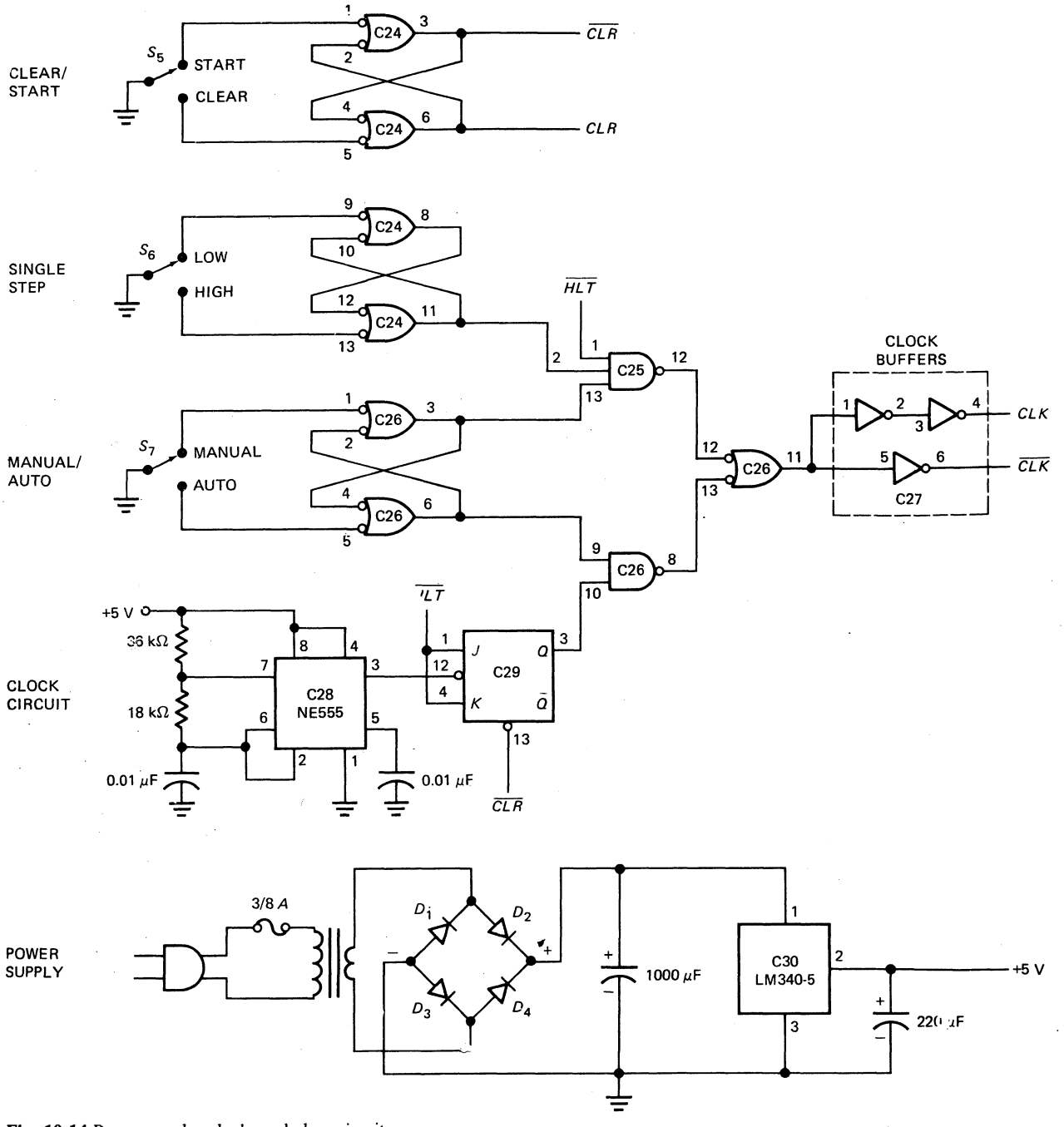


Fig. 10-14 Power supply, clock, and clear circuits.

only one of these is active at a time. ($\overline{HLT}$ is active low; all the others are active high.)

When the HLT instruction is in the *instruction register*, bits $I_7I_6I_5I_4$ are 1111 and $\overline{HLT}$ is low. This signal returns to C25 (single-step clock) and C29 (automatic clock). In either MANUAL or AUTO mode, the clock stops and the computer run ends.

Ring Counter

The ring counter, sometimes called a *state counter*, consists of three chips, C36, C37, and C38. Each of these chips is a 74LS107, a dual *JK* master-slave flip-flop. This counter is reset when the clear-start button (S_5) is pressed. The Q_0 flip-flop is inverted so that its $\overline{Q}$ output (pin 6, C38) drives

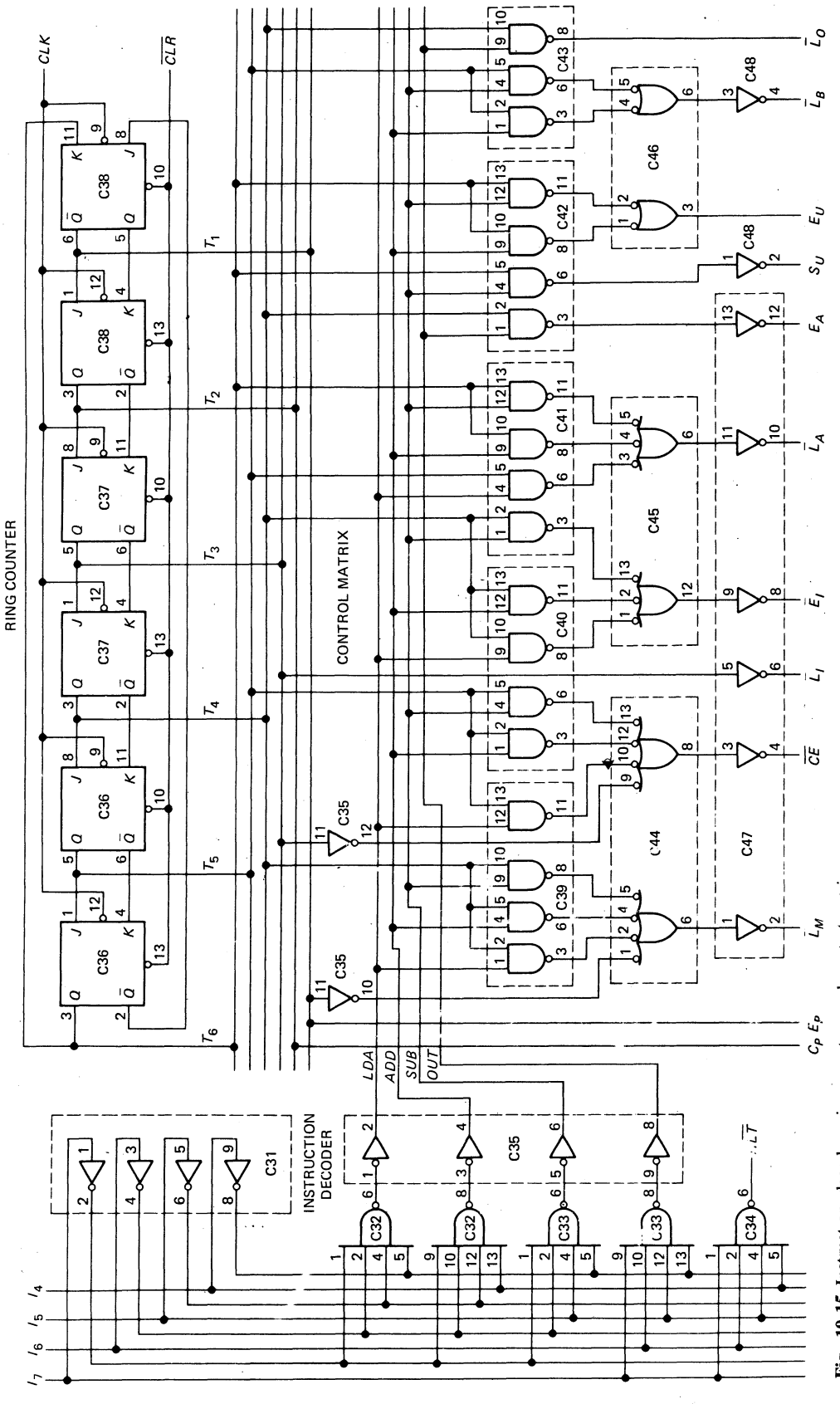


Fig. 10-15 Instruction decoder, ring counter, and control matrix.

the J input of the Q_1 flip-flop (pin 1, C38). Because of this, the T_1 output is initially high.

The *CLK* signal drives an active low input. This means that the negative edge of the *CLK* signal initiates each T state. Half a cycle later, the positive edge of the *CLK* signal produces register loading, as previously described.

Control Matrix

The *LDA*, *ADD*, *SUB*, and *OUT* signals from the instruction decoder drive the *control matrix*, C39 to C48. At the same time, the ring-counter signals, T_1 to T_6, are driving the matrix (a circuit receiving two groups of bits from different sources). The matrix produces **CON**, a 12-bit microinstruction that tells the rest of the computer what to do.

In Fig. 10-15, T_1 goes high, then T_2, then T_3, and so on. Analyze the control matrix and here is what you will find. A high T_1 produces a high E_P and a low $\overline{L}_M$ (address state); a high T_2 results in a high C_P (increment state); and a high T_3 produces a low $\overline{CE}$ and a low $\overline{L}_I$ (memory state). The first three T states, therefore, are always the fetch cycle in SAP-1. In chunked notation, the **CON** words for the fetch cycle are

State	CON	Active Bits
T_1	5E3H	E_P, $\overline{L}_M$
T_2	BE3H	C_P
T_3	263H	$\overline{CE}$, $\overline{L}_I$

During the execution states, T_4 through T_6 go high in succession. At the same time, only one of the decoded signals (*LDA* through *OUT*) is high. Because of this, the matrix automatically steers active bits to the correct output control lines.

For instance, when *LDA* is high, the only enabled 2-input NAND gates are the first, fourth, seventh, and tenth. When T_4 is high, it activates the first and seventh NAND gates, resulting in low $\overline{L}_M$ and low $\overline{E}_I$ (load MAR with address field). When T_5 is high, it activates the fourth and tenth NAND gates, producing a low $\overline{CE}$ and a low $\overline{L}_A$ (load RAM data into accumulator). When T_6 goes high, none of the control bits are active (nop).

You should analyze the action of the control matrix during the execution states of the remaining possibilities: high *ADD*, high *SUB*, and high *OUT*. Then you will agree the control matrix can generate the ADD, SUB, and OUT microinstructions shown in Table 10-5 (SAP-1 microprogram).

Operation

Before each computer run, the operator enters the program and data into the SAP-1 memory. With the program in low memory and the data in high memory, the operator presses and releases the clear button. The *CLK* and $\overline{CLK}$ signals drive the registers and counters. The microinstruction out of the controller-sequencer determines what happens on each positive *CLK* edge.

Each SAP-1 machine cycle begins with a fetch cycle. T_1 is the address state, T_2 is the increment state, and T_3 is the memory state. At the end of the fetch cycle, the instruction is stored in the instruction register. After the instruction field has been decoded, the control matrix automatically generates the correct execution routine. Upon completion of the execution cycle, the ring counter resets and the next machine cycle begins.

The data processing ends when a HLT instruction is loaded into the instruction register.

10-8 MICROPROGRAMMING

The control matrix of Fig. 10-15 is one way to generate the microinstructions needed for each execution cycle. With larger instruction sets, the control matrix becomes very complicated and requires hundreds or even thousands of gates. This is why *hardwired control* (matrix gates soldered together) forced designers to look for an alternative way to produce the control words that run a computer.

Microprogramming is the alternative. The basic idea is to store microinstructions in a ROM rather than produce them with a control matrix. This approach simplifies the problem of building a controller-sequencer.

Storing the Microprogram

By assigning addresses and including the fetch routine, we can come up with the SAP-1 microinstructions shown in Table 10-6. These microinstructions can be stored in a *control ROM* with the fetch routine at addresses 0H to 2H, the LDA routine at addresses 3H to 5H, the ADD routine at 6H to 8H, the SUB routine at 9H to BH, and the OUT routine at CH to EH.

To access any routine, we need to supply the correct addresses. For instance, to get the ADD routine, we need to supply addresses 6H, 7H, and 8H. To get the OUT routine, we supply addresses CH, DH, and EH. Therefore, accessing any routine requires three steps:

1. Knowing the starting address of the routine
2. Stepping through the routine addresses
3. Applying the addresses to the control ROM.

Address ROM

Figure 10-16 shows how to microprogram the SAP-1 computer. It has an *address* ROM, a *presettable* counter, and a *control* ROM. The address ROM contains the starting addresses of each routine in Table 10-6. In other words,

TABLE 10-6. SAP-1 CONTROL ROM

Address	Contents†	Routine	Active
0H	5E3H	Fetch	E_P, $\overline{L_M}$
1H	BE3H		C_P
2H	263H		$\overline{CE}$, $\overline{L_I}$
3H	1A3H	LDA	$\overline{L_M}$, $\overline{E_I}$
4H	2C3H		$\overline{CE}$, $\overline{L_A}$
5H	3E3H		None
6H	1A3H	ADD	$\overline{L_M}$, $\overline{E_I}$
7H	2E1H		$\overline{CE}$, $\overline{L_B}$
8H	3C7H		$\overline{L_A}$, E_U
9H	1A3H	SUB	$\overline{L_M}$, $\overline{E_I}$
AH	2E1H		$\overline{CE}$, $\overline{L_B}$
BH	3CFH		$\overline{L_A}$, S_U, E_U
CH	3F2H	OUT	E_A, $\overline{L_O}$
DH	3E3H		None
EH	3E3H		None
FH	X	X	Not used

† **CON** = $C_PE_P\overline{L}_M\overline{CE}$ $\overline{L}_I\overline{E}_I\overline{L}_AE_A$ $S_UE_U\overline{L}_B\overline{L}_O$.

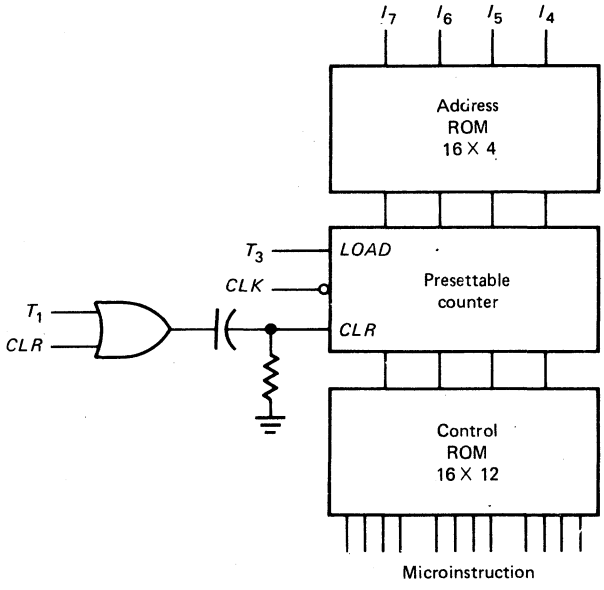


Fig. 10-16 Microprogrammed control of SAP-1.

the address ROM contains the data listed in Table 10-7. As shown, the starting address of the LDA routine is 0011, the starting address of the ADD routine is 0110, and so on.

When the op-code bits $I_7I_6I_5I_4$ drive the address ROM, the starting address is generated. For instance, if the ADD instruction is being executed, $I_7I_6I_5I_4$ is 0001. This is the input to the address ROM; the output of this ROM is 0110.

TABLE 10-7. ADDRESS ROM

Address	Contents	Routine
0000	0011	LDA
0001	0110	ADD
0010	1001	SUB
0011	XXXX	None
0100	XXXX	None
0101	XXXX	None
0110	XXXX	None
0111	XXXX	None
1000	XXXX	None
1001	XXXX	None
1010	XXXX	None
1011	XXXX	None
1100	XXXX	None
1101	XXXX	None
1110	1100	OUT
1111	XXXX	None

Presettable Counter

When T_3 is high, the load input of the *presettable counter* is high and the counter loads the starting address from the address ROM. During the other T states, the counter counts.

Initially, a high *CLR* signal from the clear-start debouncer is differentiated to get a narrow positive spike. This resets the counter. When the computer run begins, the counter output is 0000 during the T_1 state, 0001 during the T_2 state, and 0010 during the T_3 state. Every fetch cycle is the same because 0000, 0001, and 0010 come out of the counter during states T_1, T_2, and T_3.

The op code in the instruction register controls the execution cycle. If an ADD instruction has been fetched, the $I_7I_6I_5I_4$ bits are 0001. These op-code bits drive the address ROM, producing an output of 0110 (Table 10-7). This starting address is the input to the presettable counter. When T_3 is high, the next negative clock edge loads 0110 into the presettable counter. The counter is now preset, and counting can resume at the starting address of the ADD routine. The counter output is 0110 during the T_4 state, 0111 during the T_5 state, and 1000 during the T_6 state.

When the T_1 state begins, the leading edge of the T_1 signal is differentiated to produce a narrow positive spike which resets the counter to 0000, the starting address of the fetch routine. A new machine cycle then begins.

Control ROM

The control ROM stores the SAP-1 microinstructions. During the fetch cycle, it receives addresses 0000, 0001, and 0010. Therefore, its outputs are

5E3H
BE3H
263H

These microinstructions, listed in Table 10-6, produce the address state, increment state, and memory state.

If an ADD instruction is being executed, the control ROM receives addresses 0110, 0111, and 1000 during the execution cycle. Its outputs are

1A3H
2E1H
3C7H

These microinstructions carry out the addition as previously discussed.

For another example, suppose the OUT instruction is being executed. Then the op code is 1110 and the starting address is 1100 (Table 10-7). During the execution cycle, the counter output is 1100, 1101, and 1110. The output of the control ROM is 3F2H, 3E3H, and 3E3H (Table 10-6). This routine transfers the accumulator contents to the output port.

Variable Machine Cycle

The microinstruction 3E3H in Table 10-6 is a nop. It occurs once in the LDA routine and twice in the OUT routine. These nops are used in SAP-1 to get a *fixed machine cycle* for all instructions. In other words, each machine cycle takes exactly six T states, no matter what the instruction. In some computers a fixed machine cycle is an advantage. But when speed is important, the nops are a waste of time and can be eliminated.

One way to speed up the operation of SAP-1 is to skip any T state with a nop. By redesigning the circuit of Fig. 10-16 we can eliminate the nop states. This will shorten the machine cycle of the LDA instruction to five states (T_1, T_2, T_3, T_4, and T_5). It also shortens the machine cycle of the OUT instruction to four T states (T_1, T_2, T_3, and T_4).

Figure 10-17 shows one way to get a *variable machine cycle*. With an LDA instruction, the action is the same as before during the T_1 to T_5 states. When the T_6 state begins, the control ROM produces an output of 3E3H (the nop microinstruction). The NAND gate detects this nop instantly and produces a low output signal $\overline{NOP}$. $\overline{NOP}$ is fed back to the ring counter through an AND gate, as shown in Fig. 10-18. This resets the ring counter to the T_1 state, and a new machine cycle begins. This reduces the machine cycle of the LDA instruction from six states to five.

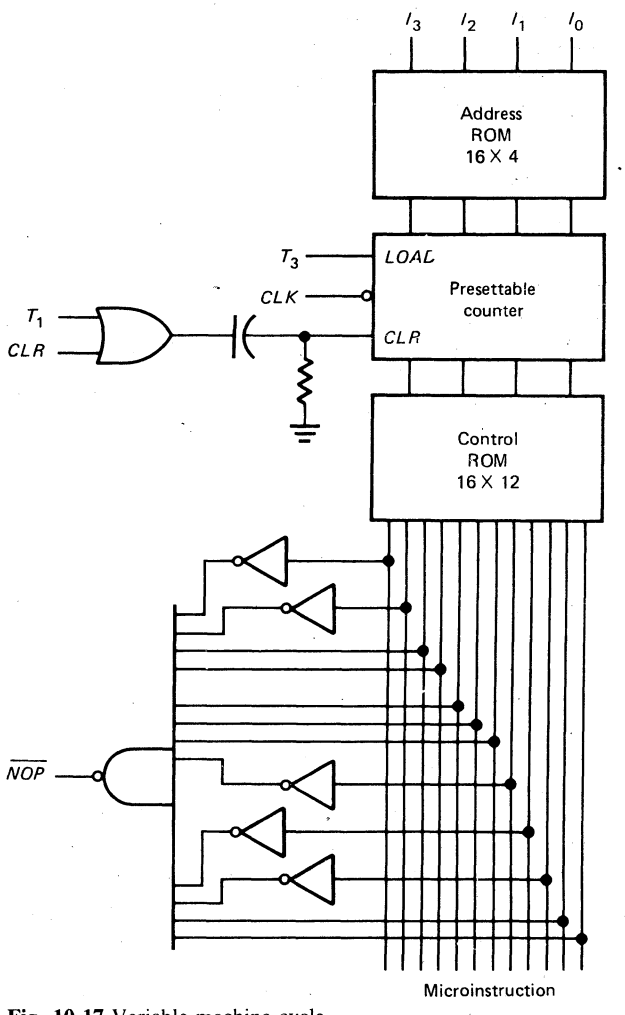


Fig. 10-17 Variable machine cycle.

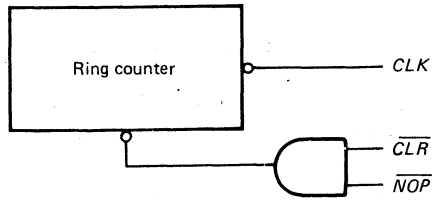


Fig. 10-18

With the OUT instruction, the first nop occurs in the T_5 state. In this case, just after the T_5 state begins, the control ROM produces an output of 3E3H, which is detected by the NAND gate. The low $\overline{NOP}$ signal then resets the ring counter to the T_1 state. In this way, we have reduced the machine cycle of the OUT instruction from six states to four.

Variable machine cycles are commonly used with microprocessors. In the 8085, for example, the machine cycles take from two to six T states because all unwanted nop states are ignored.

Advantages

One advantage of microprogramming is the elimination of the instruction decoder and control matrix; both of these become very complicated for larger instruction sets. In other words, it's a lot easier to store microinstructions in a ROM than it is to wire an instruction decoder and control matrix.

Furthermore, once you wire an instruction decoder and control matrix, the only way you can change the instruction set is by disconnecting and rewiring. This is not necessary with microprogrammed control; all you have to do is change the control ROM and the starting-address ROM. This is a big advantage if you are trying to upgrade equipment sold earlier.

Summary

In conclusion, most computers built nowadays use microprogrammed control instead of hardwired control. The microprogramming tables and circuits are more complicated than those for SAP-1, but the idea is the same. Microinstructions are stored in a control ROM and accessed by applying the address of the desired microinstruction.

GLOSSARY

address state The T_1 state. During this state, the address in the program counter is transferred to the MAR.

accumulator The place where answers to arithmetic and logic operations are accumulated. Sometimes called the A register.

assembly language The mnemonics used in writing a program.

B register An auxiliary register that stores the data to be added or subtracted from the accumulator.

fetch cycle The first part of the instruction cycle. During the fetch cycle, the address is sent to the memory, the program counter is incremented, and the instruction is transferred from the memory to the instruction register.

increment state The T_2 state. During this state, the program counter is incremented.

instruction cycle All the states needed to fetch and execute an instruction.

instruction register The register that receives the instruction from the memory.

instruction set The instructions a computer responds to.

LDA Mnemonic for load the accumulator.

machine cycle All the states generated by the ring counter.

machine language The strings of 0s and 1s used in a program.

macroinstruction One of the instructions in the instruction set.

MAR Memory address register. This register receives the address of the data to be accessed in memory. The MAR supplies this address to the memory.

memory-reference instruction An instruction that calls for a second memory operation to access data.

memory state The T_3 state. During this state, the instruction in the memory is transferred to the instruction register.

microinstruction A control word out of the controller-sequencer. The smallest step in the data processing.

nop No operation. A state during which nothing happens.

output register The register that receives processed data from the accumulator and drives the output display of SAP-1. Also called an output port.

object program A program written in machine language.

op code Operation code. That part of the instruction which tells the computer what operation to perform.

program counter A register that counts in binary. Its contents are the address of the next instruction to be fetched from the memory.

RAM Random-access memory. A better name is read-write memory. The RAM stores the program and data needed for a computer run.

source program A program written in mnemonics.

SELF-TESTING REVIEW

Read each of the following and provide the missing words. Answers appear at the beginning of the next question.

1. The _____ counter, which is part of the control unit, counts from 0000 to 1111. It sends to the memory the _____ of the next instruction.
2. (*program, address*) The MAR, or _____ register, latches the address from the program counter. A bit later, the MAR applies this address to the _____, where a read operation is performed.
3. (*memory-address, RAM*) The instruction register is

part of the control unit. The contents of the _____ register are split into two nibbles. The upper nibble goes to the _____.

4. (*instruction, controller-sequencer*) The controller-sequencer produces a 12-bit word that controls the rest of the computer. The 12 wires carrying this _____ word are called the control _____.
5. (*control, bus*) The _____ is a buffer register that stores sums or differences. Its two-state output goes to the adder-subtracter. The _____ produces the sum when S_U is low and the difference when S_U is high. The output register is sometimes called an output _____.
6. (*accumulator, adder-subtracter, port*) The SAP-1 _____ set is LDA, ADD, SUB, OUT, and HLT. LDA, ADD, and SUB are called _____ instructions because they use data stored in the memory.
7. (*instruction, memory-reference*) The 8080 was the first widely used microprocessor. The _____ is an enhanced version of the 8080 with essentially the same instruction set.
8. (*8085*) LDA, ADD, SUB, OUT, and HLT are coded as 4-bit strings of 0s and 1s. This code is called the _____ code. _____ language uses mnemonics when writing a program. _____ language uses strings of 0s and 1s.
9. (*op, Assembly, Machine*) SAP-1 has _____ *T* states, periods during which register contents change. The ring counter, or _____ counter, produces these *T* states. These six *T* states represent one machine cycle. In SAP-1 the instruction cycle has only one machine cycle. In microprocessors like the 8080 and the 8085, the _____ cycle may have from one to five machine cycles.
10. (*six, state, instruction*) The controller-sequencer sends out control words, one during each *T* state or clock cycle. Each control word is called a _____. Instructions like LDA, ADD, SUB, etc. are called _____. Each SAP-1 macroinstruction is made up of three _____.
11. (*microinstruction, macroinstructions, microinstructions*) With larger instruction sets, the control matrix becomes very complicated. This is why hardwired control is being replaced by _____. The basic idea is to store the _____ in a control ROM.
12. (*microprogramming, microinstructions*) SAP-1 uses a fixed machine cycle for all instructions. In other words, each machine cycle takes exactly six *T* states. Microprocessors like the 8085 have variable machine cycles because all unwanted nop states are eliminated.

PROBLEMS

10-1. Write a SAP-1 program using mnemonics (similar to Example 10-1) that will display the result of

$$5 + 4 - 6$$

Use addresses DH, EH, and FH for the data.

10-2. Convert the assembly language of Prob. 10-1 into SAP-1 machine language. Show the answer in binary form and in hexadecimal form.

10-3. Write an assembly-language program that performs this operation:

$$8 + 4 - 3 + 5 - 2$$

Use addresses BH to FH for the data.

10-4. Convert the program and data of Prob. 10-3 into machine language. Express the result in both binary and hexadecimal form.

10-5. Figure 10-19 shows the timing diagram for the ADD instruction. Draw the timing diagram for the SUB instruction.

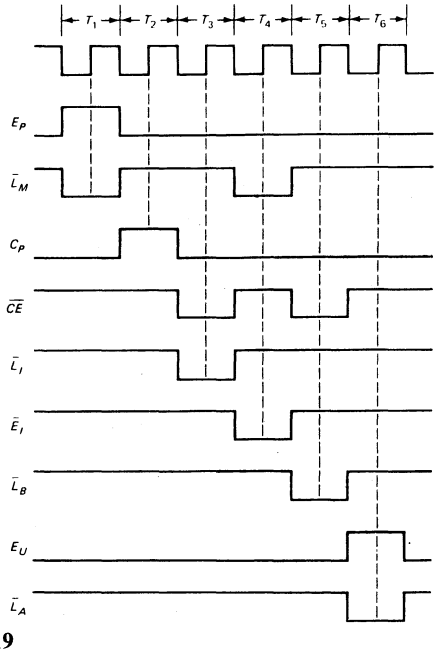
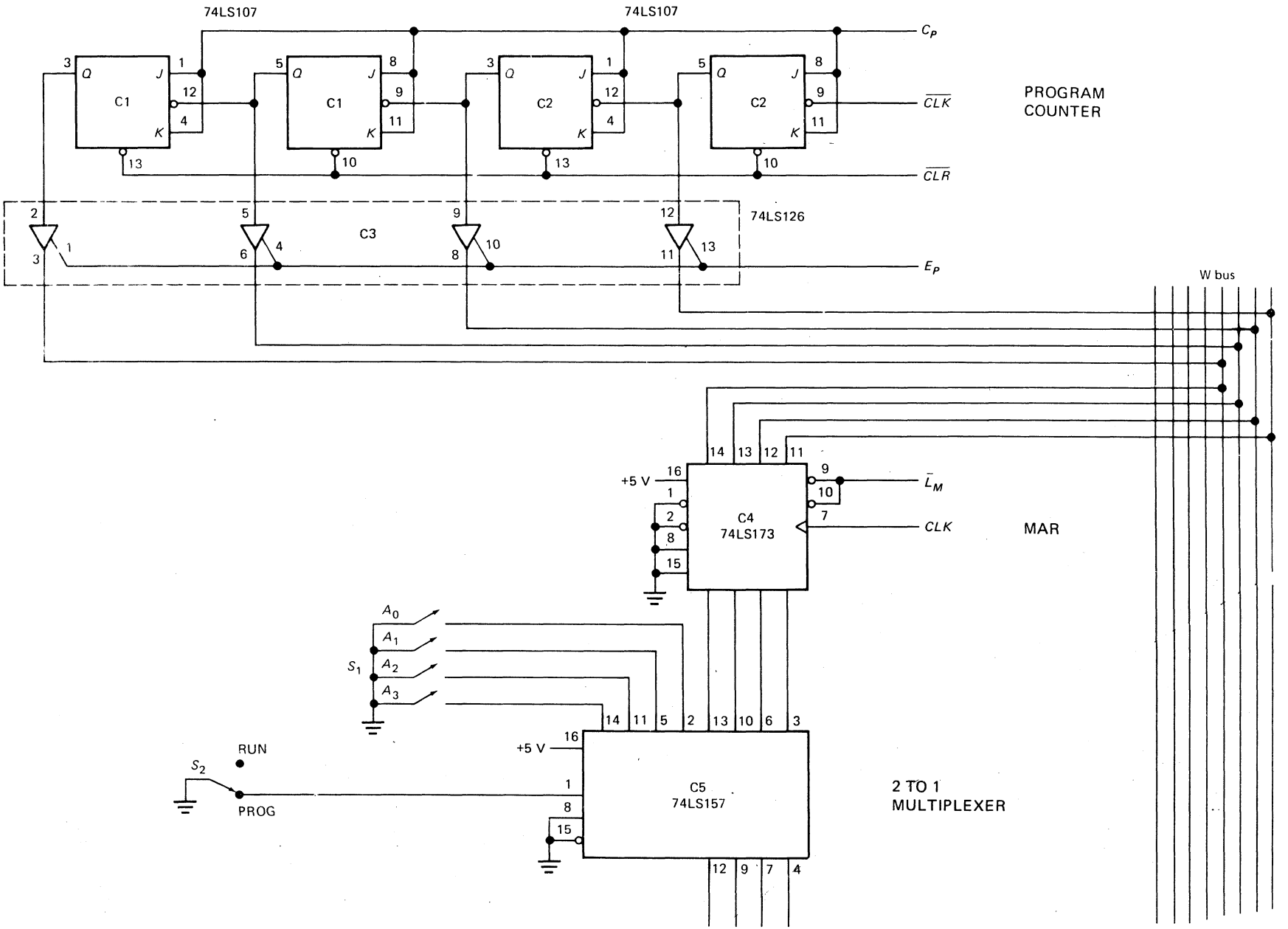


Fig. 10-19


74LS107
74LS107
C1
C1
C2
C2
C_P
$\overline{CLK}$
$\overline{CLR}$
PROGRAM COUNTER
74LS126
C3
E_P
W bus
C4
74LS173
+5 V
$\overline{L}_M$
CLK
MAR
A_0
A_1
A_2
A_3
S_1
RUN
PROG
S_2
C5
74LS157
+5 V
2 TO 1
MULTIPLEXER

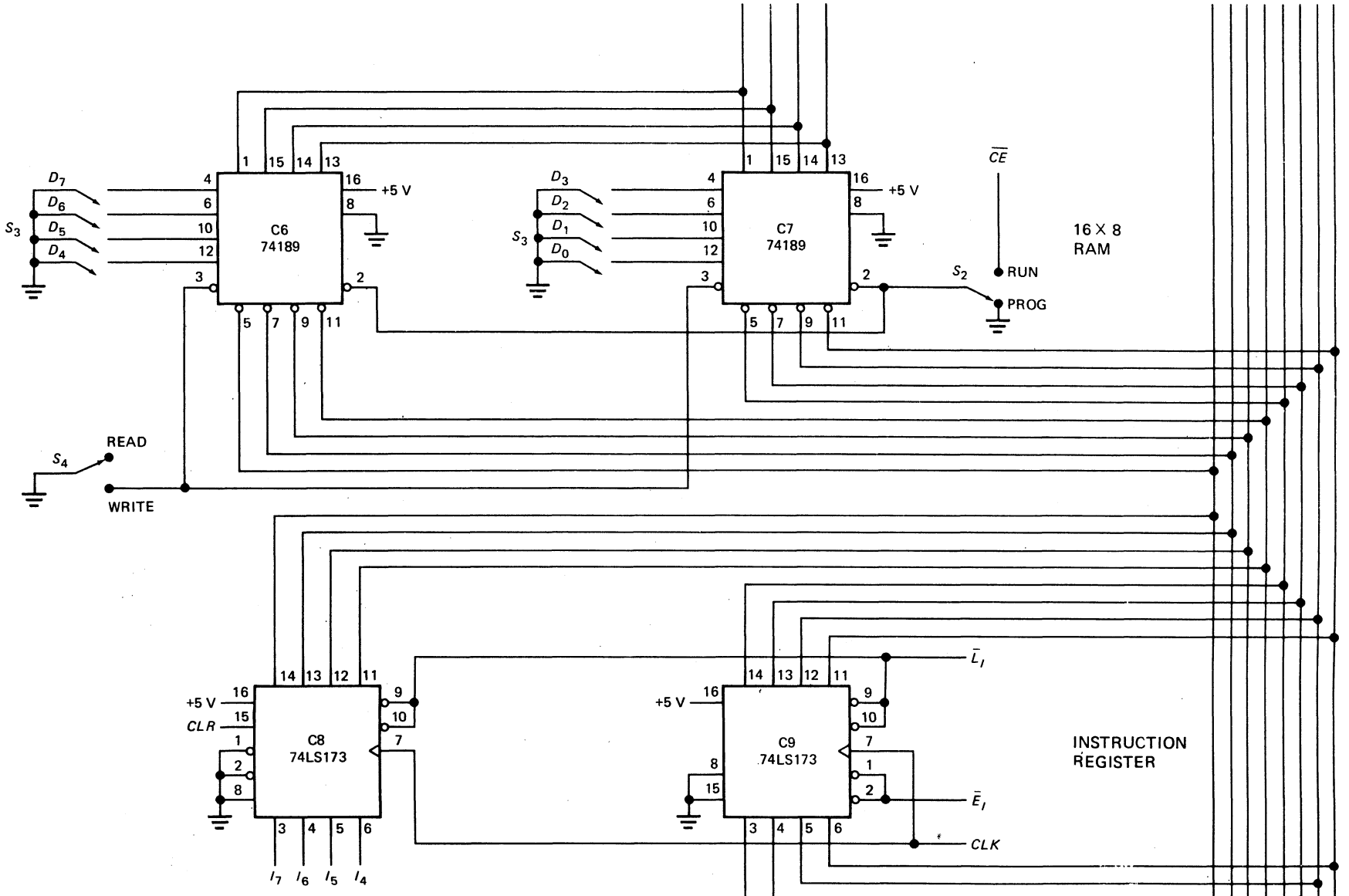
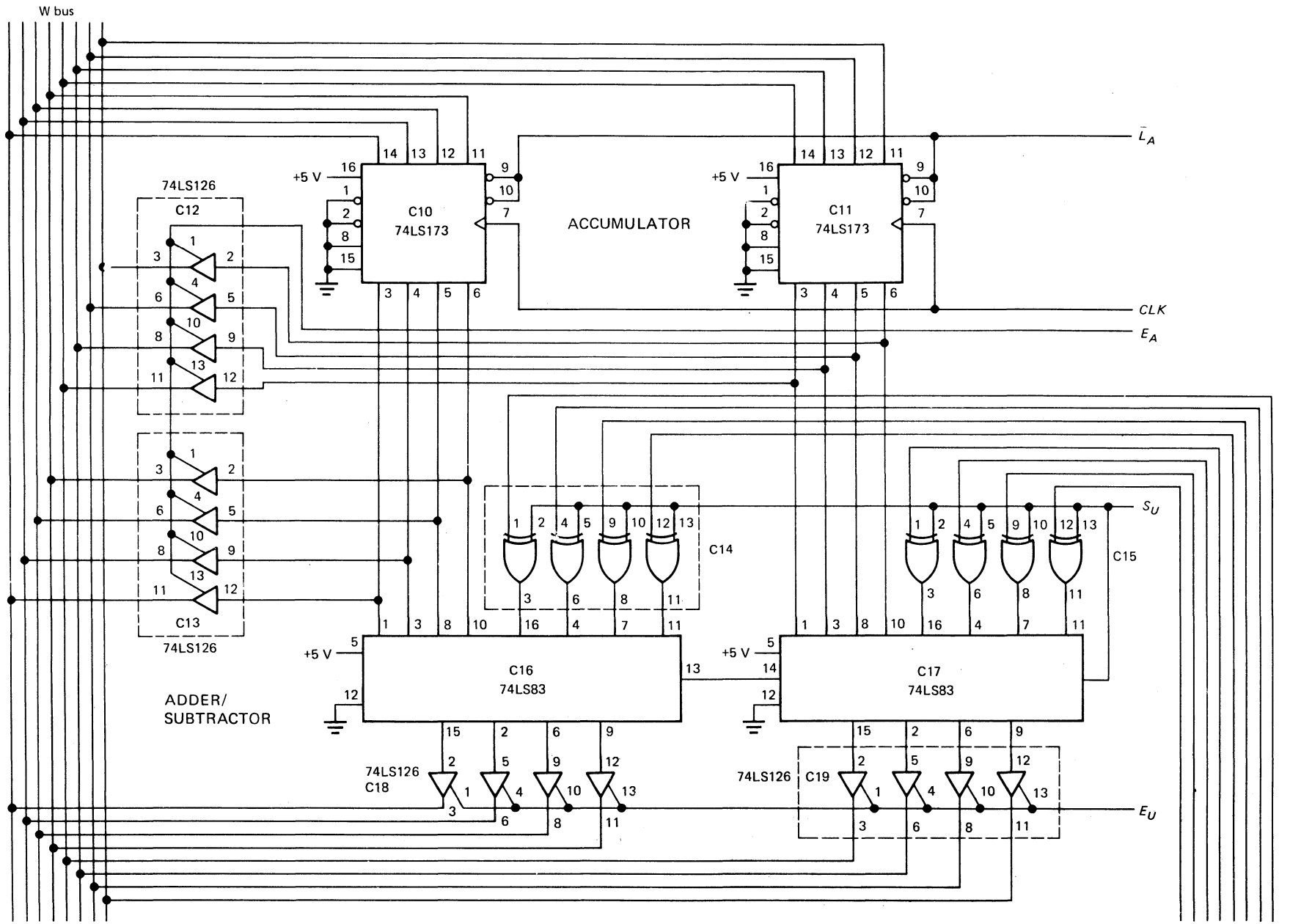

C6
74189
C7
74189
16 X 8
RAM
+5 V
D_7
D_6
D_5
D_4
D_3
D_2
D_1
D_0
S_3
$\overline{CE}$
S_2
RUN
PROG
READ
WRITE
S_4
C8
74LS173
C9
74LS173
CLR
$\overline{L}_I$
$\overline{E}_I$
CLK
I_7 I_6 I_5 I_4
INSTRUCTION
REGISTER

Fig. 10-20


W bus
74LS126
C12
C13
74LS126
ADDER/
SUBTRACTOR
+5 V
C10
74LS173
ACCUMULATOR
C11
74LS173
$\bar{L}_A$
CLK
E_A
C14
C15
S_U
C16
74LS83
C17
74LS83
74LS126
C18
74LS126
C19
E_U

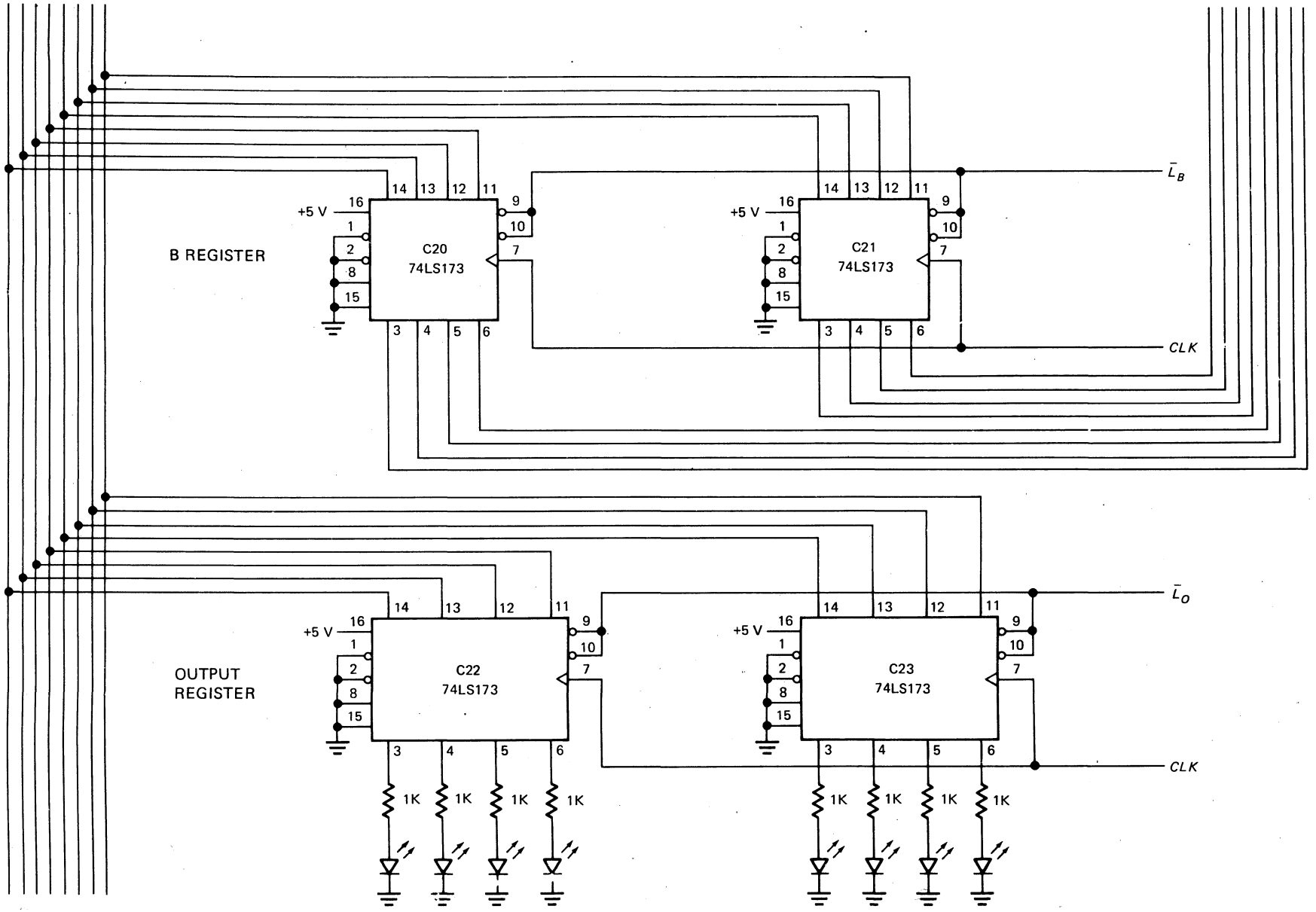

B REGISTER
C20
74LS173
C21
74LS173
+5 V
$\bar{L}_B$
CLK
OUTPUT
REGISTER
C22
74LS173
C23
74LS173
$\bar{L}_O$
1K

Fig. 10-21

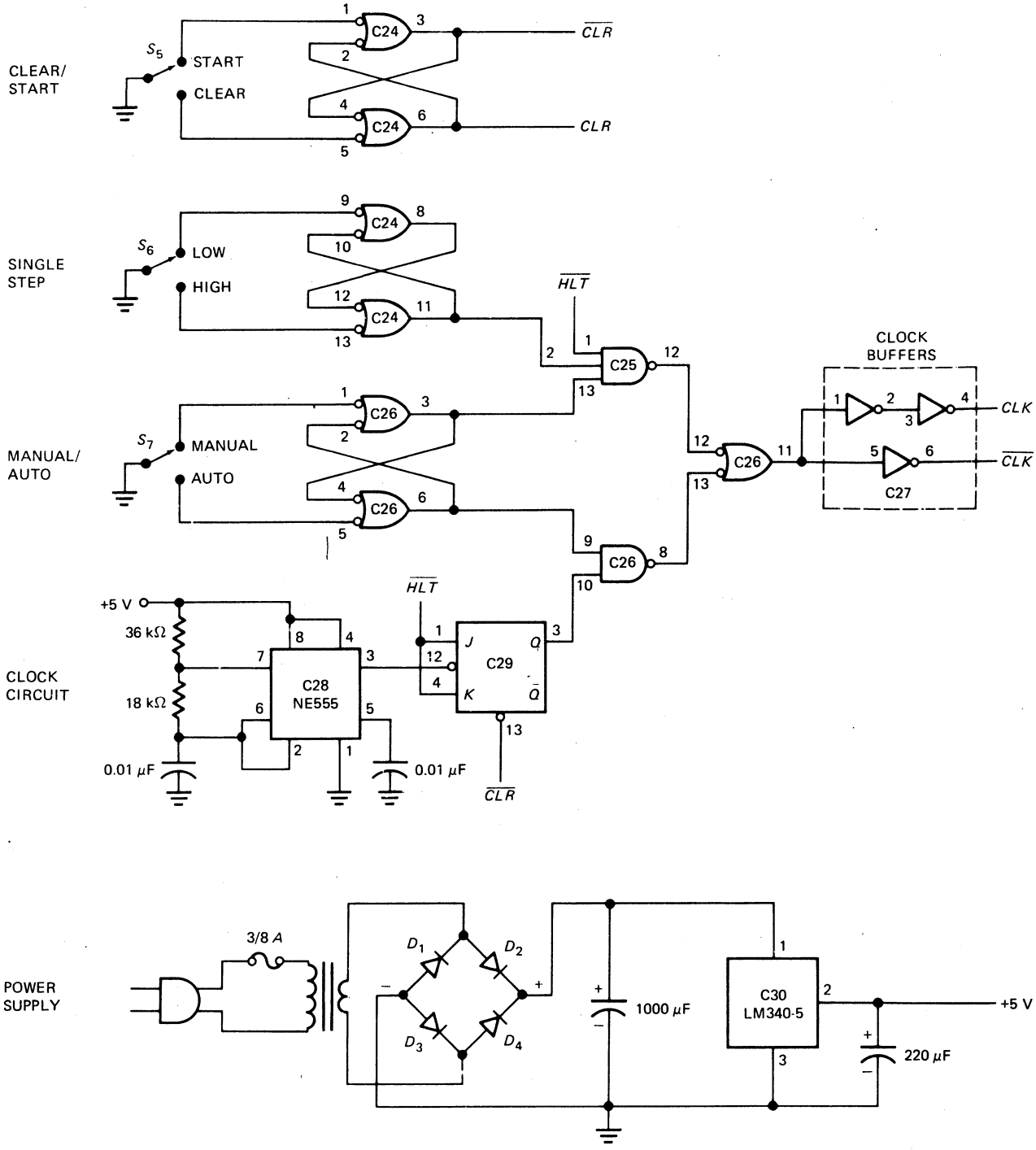

CLEAR/
START
S_5
START
CLEAR
C24
$\overline{CLR}$
CLR
SINGLE
STEP
S_6
LOW
HIGH
$\overline{HLT}$
C25
MANUAL/
AUTO
S_7
MANUAL
AUTO
C26
CLOCK
BUFFERS
C27
CLK
$\overline{CLK}$
CLOCK
CIRCUIT
+5 V
36 kΩ
18 kΩ
0.01 μF
C28
NE555
C29
J
K
Q
POWER
SUPPLY
3/8 A
D_1
D_2
D_3
D_4
1000 μF
C30
LM340-5
220 μF
+5 V

Fig. 10-22

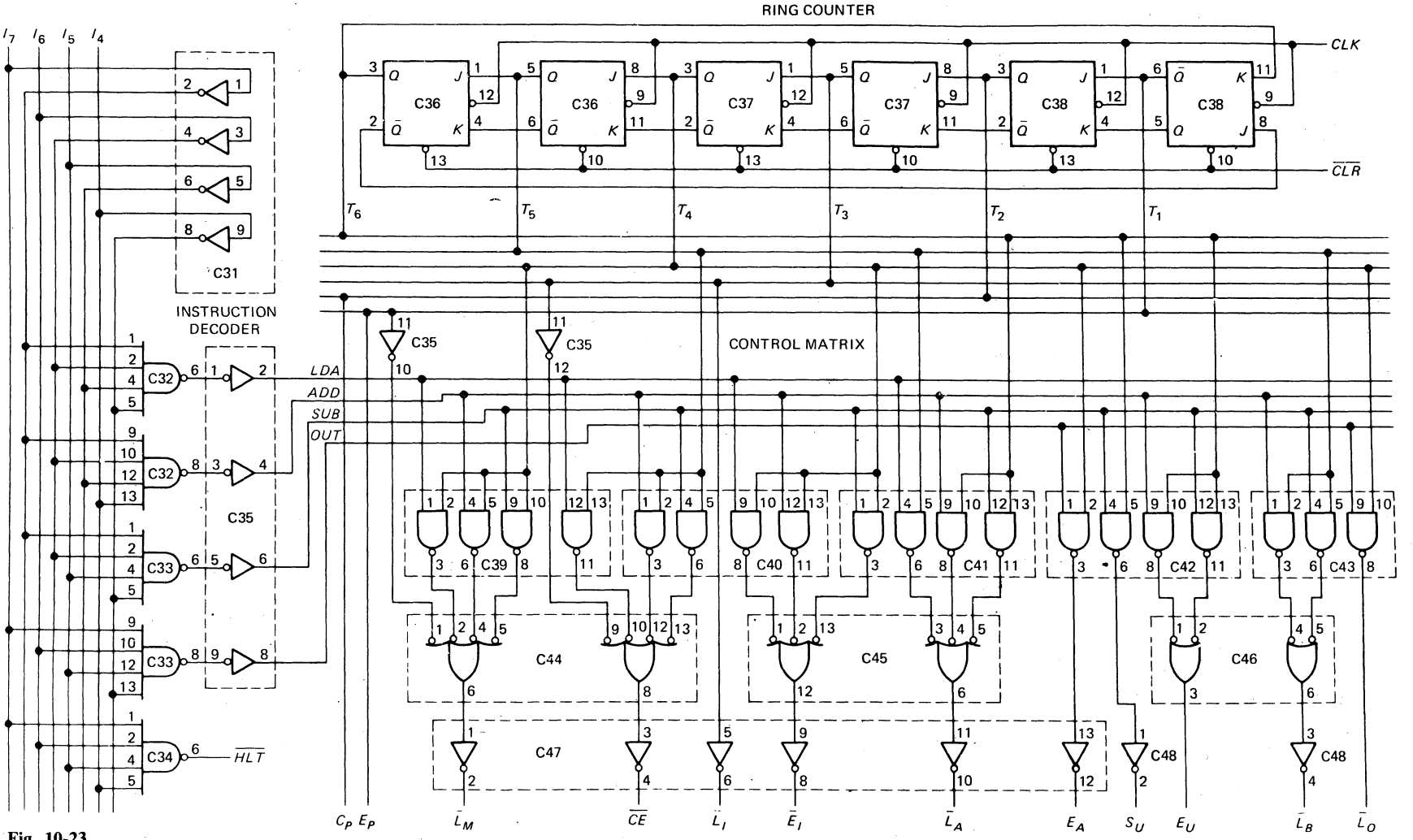

RING COUNTER
CLK
$\overline{CLR}$
C36
C37
C38
T_6
T_5
T_4
T_3
T_2
T_1
I_7 I_6 I_5 I_4
C31
INSTRUCTION DECODER
C32
C33
C34
C35
LDA
ADD
SUB
OUT
$\overline{HLT}$
CONTROL MATRIX
C39
C40
C41
C42
C43
C44
C45
C46
C47
C48
C_P E_P
$\overline{L_M}$
$\overline{CE}$
$\overline{L_I}$
$\overline{E_I}$
$\overline{L_A}$
E_A
S_U
E_U
$\overline{L_B}$
$\overline{L_O}$

Fig. 10-23

10-6. Suppose an 8085 uses a clock frequency of 3 MHz. The ADD instruction of an 8085 takes four T states to fetch and execute. How long is this?

10-7. What are the SAP-1 microinstructions for the LDA routine? For the SUB routine? Express the answers in binary and hexadecimal form.

10-8. Suppose we want to transfer the contents of the accumulator to the B register. This requires a new microinstruction. What is this microinstruction? Express your answer in hexadecimal and binary form.

10-9. Look at Fig. 10-20 and answer the following questions:

a. Are the contents of the program counter changed on the positive or negative edge of the $\overline{CLK}$ signal? At this instant, is the CLK signal on its rising or falling edge?

b. To increment the program counter, does C_P have to be low or high?

c. To clear the program counter, does $\overline{CLR}$ have to be low or high?

d. To place the contents of the program counter on the W bus, should E_P be low or high?

10-10. Refer to Fig. 10-21:

a. If $\overline{L}_A$ is high, what happens to the accumulator contents on the next positive clock edge?

b. If **A** = 0010 1100 and **B** = 1100 1110, what is on the W bus if E_A is high?

c. If **A** = 0000 1111, **B** = 0000 0001, and S_U = 1, what is on the W bus when E_U is high?

10-11. Answer the following questions for Fig. 10-22:

a. With S_5 in the CLEAR position, is the $\overline{CLR}$ output low or high?

b. With S_6 in the LOW position, is the output low or high for pin 11 C24?

c. To have a clock signal at pin 3 of C29, should $\overline{HLT}$ be low or high?

10-12. Refer to Fig. 10-23 to answer the following:

a. If $I_7I_6I_5I_4$ = 1110, only one of the output pins in C35 is high. Which pin is this?

b. $\overline{CLR}$ goes low. Which is the timing signal (T_1 to T_6) that goes high?

c. LDA and T_5 are high. Is the voltage low or high at pin 6, C45?

d. ADD and T_4 are high. Is the signal low or high at pin 12, C45?

SAP-2

11

SAP-1 is a computer because it stores a program and data before calculations begin; then it automatically carries out the program instructions without human intervention. And yet, SAP-1 is a primitive computing machine. It compares to a modern computer the way a Neanderthal human would compare to a modern person. Something is missing, something found in every modern computer.

SAP-2 is the next step in the evolution toward modern computers because it includes *jump* instructions. These new instructions force the computer to repeat or skip part of a program. As you will discover, jump instructions open up a whole new world of computing power.

11-1 BIDIRECTIONAL REGISTERS

To reduce the wiring capacitance of SAP-2, we will run only one set of wires between each register and the bus. Figure 11-1*a* shows the idea. The input and output pins are shorted; only one group of wires is connected to the bus.

Does this shorting the input and output pins ever cause trouble? No. During a computer run, either *LOAD* or *ENABLE* may be active, but not both at the same time. An active *LOAD* means that a binary word flows from the bus to the register input; during a load operation, the output lines are floating. On the other hand, an active *ENABLE* means that a binary word flows from the register to the bus; in this case, the input lines float.

The IC manufacturer can internally connect the input and output pins of a three-state register. This not only reduces the wiring capacitance; it also reduces the number of I/O pins. For instance, Fig. 11-1*b* has four I/O pins instead of eight.

Figure 11-1*c* is the symbol for a three-state register with internally connected input and output pins. The double-headed arrow reminds us that the path is *bidirectional*; data can move either way.

11-2 ARCHITECTURE

Figure 11-2 shows the architecture of SAP-2. All register outputs to the W bus are three-state; those not connected to the bus are two-state. As before, the controller-sequencer sends control signals (not shown) to each register. These control signals load, enable, or otherwise prepare the register for the next positive clock edge. A brief description of each box is given now.

Input Ports

SAP-2 has two input ports, numbered 1 and 2. A hexadecimal keyboard encoder is connected to port 1. It allows us to enter hexadecimal instructions and data through port 1. Notice that the hexadecimal keyboard encoder sends a *READY* signal to bit 0 of port 2. This signal indicates when the data in port 1 is valid.

Also notice the *SERIAL IN* signal going to pin 7 of port 2. A later example will show you how to convert serial input data to parallel data.

Program Counter

This time, the program counter has 16 bits; therefore, it can count from

$$\mathbf{PC} = 0000\ 0000\ 0000\ 0000$$

to

$$\mathbf{PC} = 1111\ 1111\ 1111\ 1111$$

This is equivalent to 0000H to FFFFH, or decimal 0 to 65,535.

A low $\overline{CLR}$ signal resets the PC before each computer run; so the data processing starts with the instruction stored in memory location 0000H.

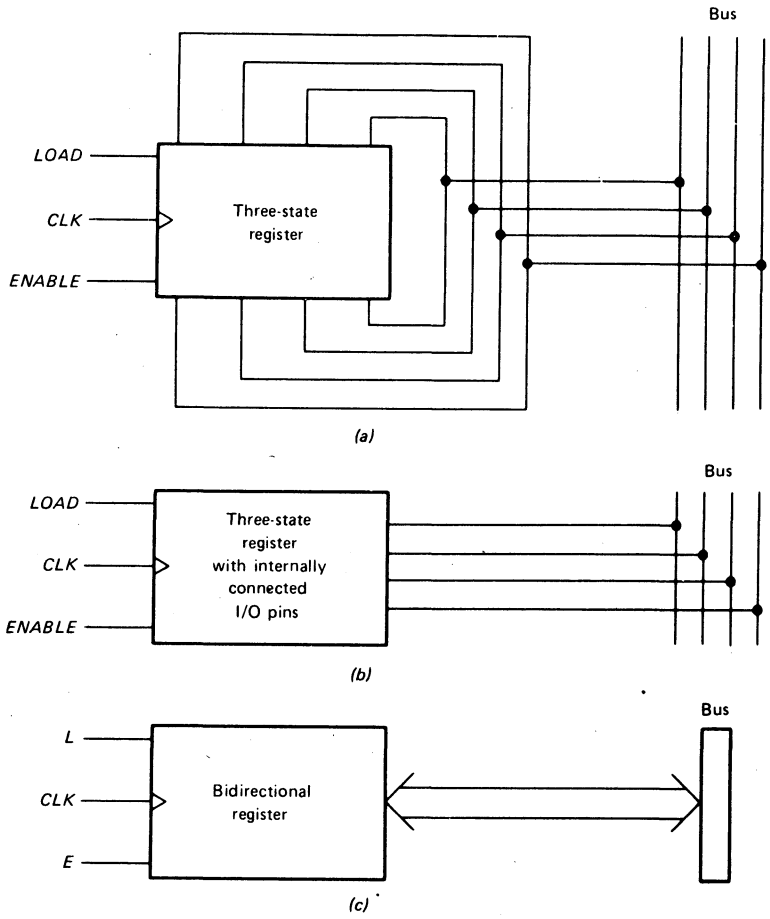


Fig. 11-1 Bidirectional register.

MAR and Memory

During the fetch cycle, the MAR receives 16-bit addresses from the program counter. The two-state MAR output then addresses the desired memory location. The memory has a 2K ROM with addresses of 0000H to 07FFH. This ROM contains a program called a *monitor* that initializes the computer on power-up, interprets the keyboard inputs, and so forth. The rest of the memory is a 62K RAM with addresses from 0800H to FFFFH.

Memory Data Register

The memory data register (MDR) is an 8-bit buffer register. Its output sets up the RAM. The memory data register receives data from the bus before a write operation, and it sends data to the bus after a read operation.

Instruction Register

Because SAP-2 has more instructions than SAP-1, we will use 8 bits for the op code rather than 4. An 8-bit op code can accommodate 256 instructions. SAP-2 has only 42 instructions, so there will be no problem coding them with 8 bits. Using an 8-bit op code also allows upward compatibility with the 8080/8085 instruction set because it is based on an 8-bit op code. As mentioned earlier, all SAP instructions are identical with 8080/8085 instructions.

Controller-Sequencer

The controller-sequencer produces the control words or microinstructions that coordinate and direct the rest of the computer. Because SAP-2 has a bigger instruction set, the controller-sequencer has more hardware. Although the CON word is bigger, the idea is the same: the control word or microinstruction determines how the registers react to the next positive clock edge.

Accumulator

The two-state output of the accumulator goes to the ALU the three-state output to the W bus. Therefore, the 8-bi word in the accumulator continuously drives the ALU, bu this same word appears on the bus only when E_A is active

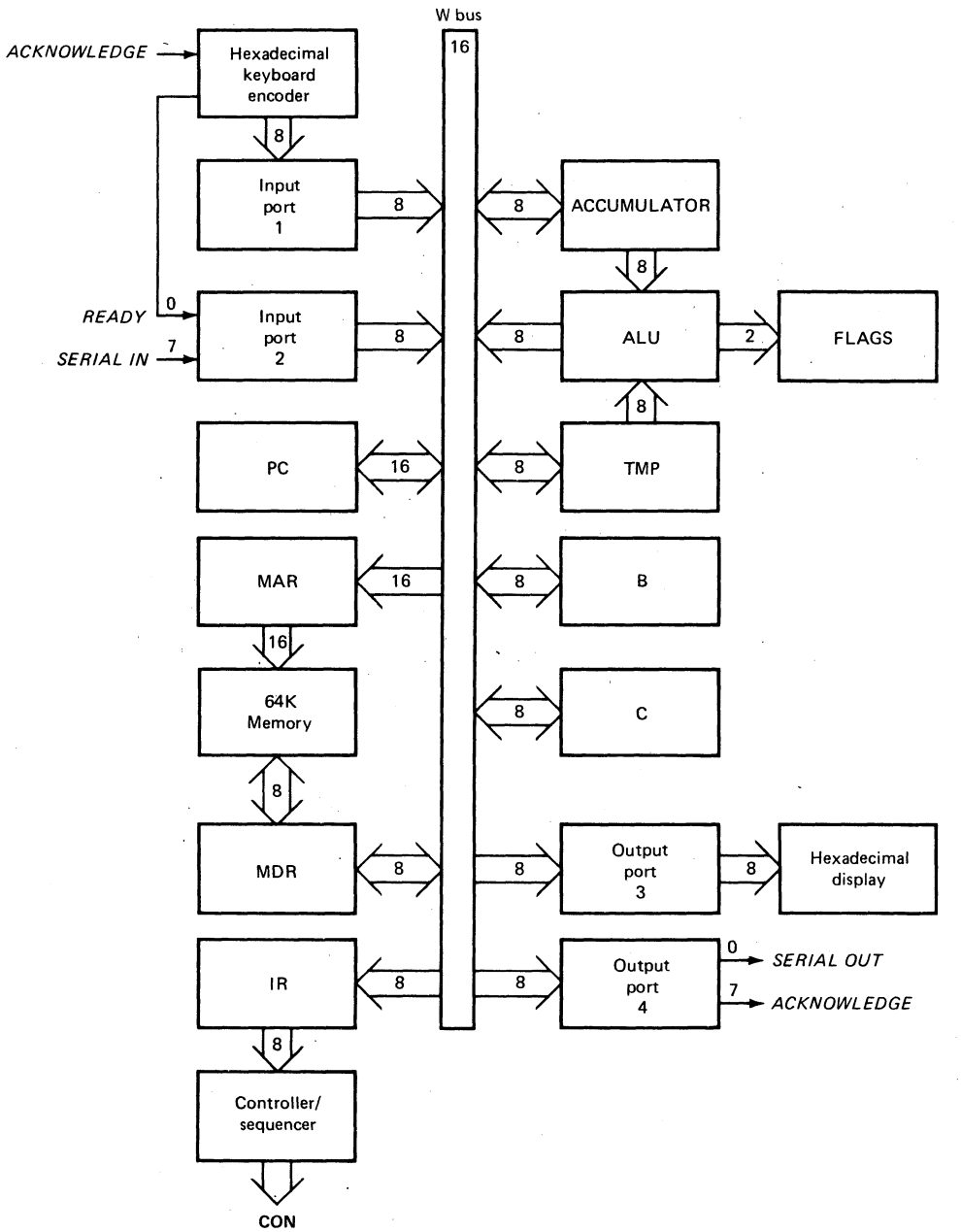


Fig. 11-2 SAP-2 block architecture.

ALU and Flags

Standard ALUs are commercially available as integrated circuits. These ALUs have 4 or more control bits that determine the arithmetic or logic operation performed on words **A** and **B.** The ALU used in SAP-2 includes arithmetic and logic operations.

In this book a *flag* is a flip-flop that keeps track of a changing condition during a computer run. The SAP-2 computer has two flags. The *sign flag* is set when the accumulator contents become negative during the execution of some instructions. The *zero flag* is set when the accumulator contents become zero.

TMP, B, and C Registers

Instead of using the B register to hold the data being added or subtracted from the accumulator, a *temporary* (TMP) register is used. This allows us more freedom in using the B register. Besides the TMP and B registers, SAP-2 includes a C register. This gives us more flexibility in moving data during a computer run.

Output Ports

SAP-2 has two output ports, numbered 3 and 4. The contents of the accumulator can be loaded into port 3, which drives a hexadecimal display. This allows us to see the processed data.

The contents of the accumulator can also be sent to port 4. Notice that pin 7 of port 4 sends an *ACKNOWLEDGE* signal to the hexadecimal encoder. This *ACKNOWLEDGE* signal and the *READY* signal are part of a concept called *handshaking*, to be discussed later.

Also notice the *SERIAL OUT* signal from pin 0 of port 4; one of the examples will show you how to convert parallel data in the accumulator into serial output data.

11-3 MEMORY-REFERENCE INSTRUCTIONS

The SAP-2 fetch cycle is the same as before. T_1 is the address state, T_2 is the increment state, and T_3 is the memory state. All SAP-2 instructions therefore use the memory during the fetch cycle because a program instruction is transferred from the memory to the instruction register.

During the execution cycle, however, the memory may or may not be used; it depends on the type of instruction that has been fetched. A memory-reference instruction (MRI) is one that uses the memory during the execution cycle.

The SAP-2 computer has an instruction set with 42 instructions. What follows is a description of the memory-reference instructions.

LDA and STA

LDA has the same meaning as before: *load the accumulator* with the addressed memory data. The only difference is that more memory locations can be accessed in SAP-2 because the addresses are from 0000H to FFFFH. For example, LDA 2000H means to load the accumulator with the contents of memory location 2000H.

To distinguish the different parts of an instruction, the mnemonic is sometimes called the *op code* and the rest of the instruction is known as the *operand*. With LDA 2000H, LDA is the op code and 2000H is the operand. Therefore, "op code" has a double meaning in microprocessor work; it may stand for the mnemonic or for the binary code used to represent the mnemonic. The intended meaning is clear from the context.

STA is a mnemonic for *store the accumulator*. Every STA instruction needs an address. STA 7FFFH means to store the accumulator contents at memory location 7FFFH. If

$$A = 8AH$$

the execution of STA 7FFFH stores 8AH at address 7FFFH.

MVI

MVI is the mnemonic for *move immediate*. It tells the computer to load a designated register with the byte that immediately follows the op code. For instance,

MVI A,37H

tells the computer to load the accumulator with 37H. After this instruction has been executed, the binary contents of the accumulator are

$$A = 0011\ 0111$$

You can use MVI with the A, B, and C registers. The formats for these instructions are

MVI A,byte
MVI B,byte
MVI C,byte

Op Codes

Table 11-1 shows the op codes for the SAP-2 instruction set. These are the 8080/8085 op codes. As you can see, 3A is the op code for LDA, 32 is the op code for STA, etc. Refer to this table in the remainder of this chapter.

EXAMPLE 11-1

Show the mnemonics for a program that loads the accumulator with 49H, the B register with 4AH, and the C register with 4BH; then have the program store the accumulator data at memory location 6285H.

SOLUTION

Here's one program that will work:

Mnemonics

```
MVI A,49H
MVI B,4AH
MVI C,4BH
STA 6285H
HLT
```

The first three instructions load 49H, 4AH, and 4BH into the A, B, and C registers. STA 6285H stores the accumulator contents at 6285H.

Note the use of HLT in this program. It has the same meaning as before: halt the data processing.

TABLE 11-1. SAP-2 OP CODES

Instruction	Op Code	Instruction	Op Code
ADD B	80	MOV B,A	47
ADD C	81	MOV B,C	41
ANA B	A0	MOV C,A	4F
ANA C	A1	MOV C,B	48
ANI byte	E6	MVI A,byte	3E
CALL address	CD	MVI B,byte	06
CMA	2F	MVI C,byte	0E
DCR A	3D	NOP	00
DCR B	05	ORA B	B0
DCR C	0D	ORA C	B1
HLT	76	ORI byte	F6
IN byte	DB	OUT byte	D3
INR A	3C	RAL	17
INR B	04	RAR	1F
INR C	0C	RET	C9
JM address	FA	STA address	32
JMP address	C3	SUB B	90
JNZ address	C2	SUB C	91
JZ address	CA	XRA B	A8
LDA address	3A	XRA C	A9
MOV A,B	78	XRI byte	EE
MOV A,C	79		

EXAMPLE 11-2

Translate the foregoing program into 8080/8085 machine language using the op codes of Table 11-1. Start with address 2000H.

SOLUTION

Address	Contents	Symbolic
2000H	3EH	MVI A,49H
2001H	49H	
2002H	06H	MVI B,4AH
2003H	4AH	
2004H	0EH	MVI C,4BH
2005H	4BH	
2006H	32H	STA 6285H
2007H	85H	
2008H	62H	
2009H	76H	HLT

There are a couple of new ideas in this machine-language program. With the

MVI A,49H

instruction, notice that the op code goes into the first address and the byte into the second address. This is true of all 2-byte instructions: op code into the first available memory location and byte into the next.

The instruction

STA 6285H

is a 3-byte instruction (1 byte for the op code and 2 for the address). The op code for STA is 32H. This byte goes into the first available memory location, which is 2006H. The address 6285H has 2 bytes. The lower byte 85H goes into the next memory location, and the upper byte 62H into the next location.

Why does the address get programmed with the lower byte first and the upper byte second? This is a peculiarity of the original 8080 design. To keep upward compatibility, the 8085 and some other microprocessors use the same scheme: lower byte into lower memory, upper byte into upper memory.

The last instruction HLT has an op code of 76H, stored in memory location 2009H.

In summary, the MVI instructions are 2-byte instructions, the STA is a 3-byte instruction, and the HLT is a 1-byte instruction.

11-4 REGISTER INSTRUCTIONS

Memory-reference instructions are relatively slow because they require more than one memory access during the instruction cycle. Furthermore, we often want to move data directly from one register to another without having to go through the memory. What follows are some of the SAP-2 register instructions, designed to move data from one register to another in the shortest possible time.

MOV

MOV is the mnemonic for *move*. It tells the computer to move data from one register to another. For instance,

MOV A,B

tells the computer to move the data in the B register to the accumulator. The operation is nondestructive, meaning that the data in B is copied but not erased. For example, if

A = 34H and **B** = 9DH

then the execution of MOV A,B results in

A = 9DH
B = 9DH

You can move data between the A, B, and C registers. The formats for all MOV instructions are

```
MOV A,B
MOV A,C
MOV B,A
MOV B,C
MOV C,A
MOV C,B
```

These instructions are the fastest in the SAP-2 instruction set, requiring only one machine cycle.

ADD and SUB

ADD stands for add the data in the designated register to the accumulator. For instance,

```
ADD B
```

means to add the contents of the B register to the accumulator. If

A = 04H and **B** = 02H

then the execution of ADD B results in

A = 06H

Similarly, SUB means subtract the data in the designated register from the accumulator. SUB C will subtract the contents of the C register from the accumulator.

The formats for the ADD and SUB instructions are

```
ADD B
ADD C
SUB B
SUB C
```

INR and DCR

Many times we want to increment or decrement the contents of one of the registers. INR is the mnemonic for *increment;* it tells the computer to increment the designated register. DCR is the mnemonic for *decrement,* and it instructs the computer to decrement the designated register. The formats for these instructions are

```
INR A
INR B
INR C
DCR A
DCR B
DCR C
```

As an example, if

B = 56H and **C** = 8AH

then the execution of INR B results in

B = 57H

and the execution of a DCR C produces

C = 89H

EXAMPLE 11-3

Show the mnemonics for adding decimal 23 and 45. The answer is to be stored at memory location 5600H. Also, the answer incremented by 1 is to be stored in the C register.

SOLUTION

As shown in Appendix 1, decimal 23 and 45 are equivalent to 17H and 2DH. Here is a program that will do the job:

Mnemonics

```
MVI A,17H
MVI B,2DH
ADD B
STA 5600H
INR A
MOV C,A
HLT
```

EXAMPLE 11-4

To *hand-assemble* a program means to translate a source program into a machine-language program by hand rather than machine. Hand-assemble the program of the preceding example starting at address 2000H.

SOLUTION

Address	Contents	Symbolic
2000H	3EH	MVI A,17H
2001H	17H	
2002H	06H	MVI B,2DH
2003H	2DH	
2004H	80H	ADD B
2005H	32H	STA 5600H
2006H	00H	
2007H	56H	
2008H	3CH	INR A
2009H	4FH	MOV C,A
200AH	76H	HLT

Notice that the ADD, INR, MOV, and HLT instructions are 1-byte instructions; the MVI instructions are 2-byte instructions, and the STA is a 3-byte instruction.

11-5 JUMP AND CALL INSTRUCTIONS

SAP-2 has three *jump* instructions; these can change the program sequence. In other words, instead of fetching the next instruction in the usual way, the computer may jump or *branch* to another part of the program.

JMP

To begin with, JMP is the mnemonic for jump; it tells the computer to get the next instruction from the designated memory location. Every JMP instruction includes an address that is loaded into the program counter. For instance,

JMP 3000H

tells the computer to get the next instruction from memory location 3000H.

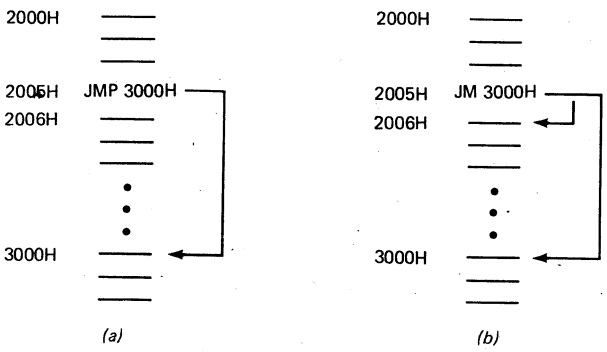


Fig. 11-3 (*a*) Unconditional jump; (*b*) conditional jump.

Here is what happens. Suppose JMP 3000H is stored at 2005H, as shown in Fig. 11-3*a*. At the end of the fetch cycle, the program counter contains

PC = 2006H

During the execution cycle, the JMP 3000H loads the program counter with the designated address:

PC = 3000H

When the next fetch cycle begins, the next instruction comes from 3000H rather than 2006H (see Fig. 11-3*a*).

JM

SAP-2 has two flags called the *sign flag* and the *zero flag*. During the execution of some instructions, these flags will be set or reset, depending on what happens to the accumulator contents. If the accumulator contents become negative, the sign flag will be set; otherwise, the sign flag is cleared. Symbolically,

$$S = \begin{cases} 0 & \text{if } A \geqslant 0 \\ 1 & \text{if } A < 0 \end{cases}$$

where S stands for sign flag. The sign flag will remain set or clear until another operation that affects the flag.

JM is a mnemonic for *jump if minus;* the computer will jump to a designated address if and only if the sign flag is set. As an example, suppose a JM 3000H is stored at 2005H. After this instruction has been fetched,

PC = 2006H

If $S = 1$, the execution of JM 3000H loads the program counter with

PC = 3000H

Since the program counter now points to 3000H, the next instruction will come from 3000H.

If the jump condition is not met ($S = 0$), the program counter is unchanged during the execution cycle. Therefore, when the next fetch cycle begins, the instruction is fetched from 2006H.

Figure 11-3*b* symbolizes the two possibilities for a JM instruction. If the minus condition is satisfied, the computer jumps to 3000H for the next instruction. If the minus condition is not satisfied, the program *falls through* to the next instruction.

JZ

The other flag affected by accumulator operations is the zero flag. During the execution of some instructions, the accumulator will become zero. To record this event, the zero flag is set; if the accumulator contents do not go to zero, the zero flag is reset. Symbolically,

$$Z = \begin{cases} 0 & \text{when } A \neq 0 \\ 1 & \text{when } A = 0 \end{cases}$$

JZ is the mnemonic for *jump if zero;* it tells the computer to jump to the designated address only if the zero flag is set. Suppose a JZ 3000H is stored at 2005H. If $Z = 1$ during the exection of JZ 3000H, the next instruction is fetched from 3000H. On the other hand, if $Z = 0$, the next instruction will come from 2006H.

JNZ

JNZ stands for *jump if not zero*. In this case, we get a jump when the zero flag is clear and no jump when it is set. Suppose a JNZ 7800H is stored at 2100H. If $Z = 0$, the next instruction will come from 7800H; however, if $Z = 1$, the program falls through to the instruction at 2101H.

JM, JZ, and JNZ are called *conditional jumps* because the program jump occurs only if certain conditions are satisfied. On the other hand, JMP is *unconditional;* once this instruction is fetched, the execution cycle always jumps the program to the specified address.

CALL and RET

A *subroutine* is a program stored in the memory for possible use in another program. Many microcomputers have subroutines for finding sines, cosines, tangents, logarithms, square roots, etc. These subroutines are part of the software supplied with the computer.

CALL is the mnemonic for *call the subroutine*. Every CALL instruction must include the starting address of the desired subroutine. For instance, if a square-root subroutine starts at address 5000H and a logarithm subroutine at 6000H, the execution of

CALL 5000H

will jump to the square-root subroutine. On the other hand, a

CALL 6000H

produces a jump to the logarithm subroutine.

RET stands for *return*. It is used at the end of every subroutine to tell the computer to go back to the original program. A RET instruction is to a subroutine as a HLT is to a program. Both tell the computer that something is finished. If you forget to use a RET at the end of a subroutine, the computer cannot get back to the original program and you will get computer trash.

When a CALL is executed in the SAP-2 computer, the contents of the program counter are automatically saved in memory locations FFFEH and FFFFH (the last two memory locations). The CALL address is then loaded into the program counter, so that execution begins with the fi instruction in the subroutine. After the subroutine is finishe the RET instruction causes the address in memory locatio FFFEH and FFFFH to be loaded back into the progra counter. This returns control to the original program.

Figure 11-4 shows the program flow during a subroutir The CALL 5000H sends the computer to the subrouti located at 5000H. After this subroutine has been complete the RET sends the computer back to the instruction followi the CALL.

CALL is unconditional, like JMP. Once a CALL h been fetched into the instruction register, the computer w jump to the starting address of the subroutine.

More on Flags

The sign or zero flag may be set or reset during certa instructions. Table 11-2 lists the SAP-2 instructions th can affect the flags. All these instructions use the acc mulator during the execution cycle. If the accumulator go negative or zero while one of these instructions is bei executed, the sign or zero flag will be set.

For instance, suppose the instruction is ADD C. Tl contents of the C register are added to the accumulat contents. If the accumulator contents become negative zero in the process, the sign or zero flag will be set.

A word about the INR and DCR instructions. Since the instructions use the accumulator to add or subtract 1 fro the designated register, they also affect the flags. F instance, to execute a DCR C, the contents of the C regist are decremented by sending these contents to the accum lator, subtracting 1, and sending the result back to the register. If the accumulator goes negative while the DC C is executed, the sign flag is set; if the accumulator go to zero, the zero flag is set.

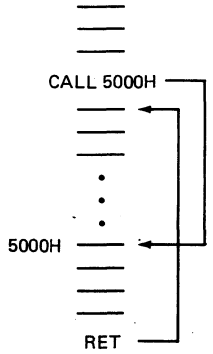


Fig. 11-4 CALL instruction.

TABLE 11-2. INSTRUCTIONS AFFECTING FLAGS

Instruction	Flags Affected
ADD	S, Z
SUB	S, Z
INR	S, Z
DCR	S, Z
ANA	S, Z
ORA	S, Z
XRA	S, Z
ANI	S, Z
ORI	S, Z
XRI	S, Z

EXAMPLE 11-5

Hand-assemble the following program starting at address 2000H:

```
MVI C,03H
DCR C
JZ 0009H
JMP 0002H
HLT
```

SOLUTION

Address	Contents	Symbolic
2000H	0EH	MVI C,03H
2001H	03H	
2002H	0DH	DCR C
2003H	CAH	JZ 2009H
2004H	09H	
2005H	20H	
2006H	C3H	JMP 2002H
2007H	02H	
2008H	20H	
2009H	76H	HLT

EXAMPLE 11-6

In the foregoing program, how many times is the DCR instruction executed?

SOLUTION

Figure 11-5 illustrates the program flow. Here is what happens. The MVI C,03H instruction loads the C register with 03H. DCR C reduces the contents to 02H. The contents are greater than zero; therefore, the zero flag is reset, and the JZ 2009H is ignored. The JMP 2002H returns the computer to the DCR C instruction.

The second time the DCR C is executed, the contents drop to 01H; the zero flag is still reset. JZ 2009H is again ignored, and the JMP 2002H returns the computer to DCR C.

The third DCR C reduces the contents to zero. This time the zero flag is set, and the JZ 2009H jumps the program to HLT instruction.

A *loop* is part of a program that is repeated. In this example, we have passed through the loop (DCR C and JZ 2009H) 3 times, as shown in Fig. 11-5. Note that the number of passes through the loop equals the number initially loaded into the C register. If we change the first instruction to

```
MVI C,07H
```

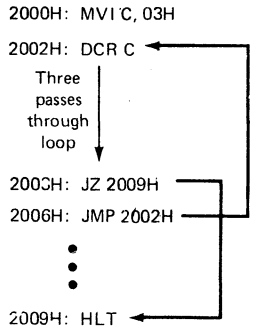


Fig. 11-5 Looping.

the computer will loop 7 times. Similarly, if we wanted to pass through the loop 200 times (equivalent to C8H), the first instruction would be

```
MVI C,C8H
```

The C register acts like a presettable down counter. This is why it is sometimes referred to as a *counter*.

The point to remember is this. We can set up a loop by using an MVI, DCR, JZ, and JMP in a program. The number loaded into the designated register (the counter) determines the number of passes through the loop. If we put new instructions inside the loop, these added instructions will be executed *X* times, the number preset into the counter.

EXAMPLE 11-7

When you buy a microcomputer, you often purchase software to do different jobs. One of the programs you can buy is an *assembler*. The assembler allows you to write programs in mnemonic form. Then the assembler converts these mnemonics into machine language. In other words, if you have an assembler, you no longer have to hand-assemble your programs; the computer does the work for you.

Show the assembly-language version of the program in Example 11-5. Include *labels* and *comments*.

SOLUTION

```
Label     Instruction     Comment
          MVI C,03H       ;Load counter with decimal 3
REPEAT:   DCR C           ;Decrement counter
          JZ END          ;Test for zero
          JMP REPEAT      ;Do it again
END:      HLT
```

When you write a program, it helps to include your own comments about what the instruction is supposed to do. These comments jog your memory if you have to read the program months later. The first comment reminds us that we are presetting the down counter with decimal 3, the second comment reminds us that we are decrementing the counter, the third comment tells us that we are testing for zero before jumping, and the fourth comment tells us that the program will loop back.

When the assembler converts your source program into an object program, it ignores everything after the semicolon. Why? Because that's the way the assembler program is written. The semicolon is a coded way to tell the computer that your personal comments follow. (Remember the ASCII code. 3BH is the ASCII for a semicolon. When the assembler encounters 3BH in your source programs, it knows comments follow.)

Labels are another programming aid used with jumps and calls. When we write an assembly-language program, we often have no idea what address to use in a jump or call instruction. By using a label instead of a numerical address we can write programs that make sense to us. The assembler will keep track of our labels and automatically assign the correct addresses to them. This is a great laborsaving feature of an assembler.

For instance, when the assembler converts the foregoing program to machine language, it will replace JZ by CA (op code of Table 11-1) and END by the address of the HLT instruction. Likewise, it will replace JMP by C3 (op code) and REPEAT by the address of the DCR C instruction. The assembler determines the addresses of the HLT and JMP by counting the number of bytes needed by all instructions and figuring out where the HLT and DCR C instructions will be in the final assembled program.

All you have to remember is that you can make up any label you want for jump and call instructions. The same label followed by a colon is placed in front of the instruction you are trying to jump to. When the assembler converts your program into machine language, the colon tells it a label is involved.

One more point about labels. With SAP-2, the labels can be from one to six characters, the first of which must be a letter. Labels are usually words or abbreviations, but numbers can be included. The following are examples of acceptable labels:

REPEAT
DELAY
RDKBD
A34
B12C3

The first two are words; the third is an abbreviation for read the keyboard. The last two are labels that include numbers. The restrictions on length (no more than six characters) and starting character (must be letter) are typical of commercially available assemblers.

EXAMPLE 11-8

Show a program that multiplies decimal 12 and 8.

SOLUTION

The hexadecimal equivalents of 12 and 8 are 0CH and 08H. Let us set up a loop that adds 12 to the accumulator during each pass. If the computer loops 8 times, the accumulator contents will equal 96 (decimal) at the end of the looping.

Here's one assembly-language program that will do the job:

Label	Mnemonic	Comment
	MVI A,00H	;Clear accumulator
	MVI B,0CH	;Load decimal 12 into B
	MVI C,08H	;Preset counter with 8
REPEAT:	ADD B	;Add decimal 12
	DCR C	;Decrement the counter
	JZ DONE	;Test for zero
	JMP REPEAT	;Do it again
DONE:	HLT	;Stop it

The comments tell most of the story. First, we clear the accumulator. Next, we load decimal 12 into the B register. Then the counter is preset to decimal 8. These first three instructions are part of the initialization before entering a loop.

The ADD B begins the loop by adding decimal 12 to accumulator. The DCR C reduces the count to 7. Since the zero flag is clear, JZ DONE is ignored the first time through and the program flow returns to the ADD B instruction.

You should be able to see what will happen. ADD B is inside the loop and will be executed 8 times. After eight passes through the loop, the zero flag is set; then the JZ DONE will take the program out of the loop to the HLT instruction.

Since 12 is added 8 times,

$$12 + 12 + 12 + 12 + 12 + 12 + 12 + 12 = 96$$

(Because decimal 96 is equivalent to hexadecimal 60, the accumulator contains 0110 0000.) Repeated addition like this is equivalent to multiplication. In other words, adding 12 eight times is identical to 12×8. Most microprocessors do not have multiplication hardware; they only have an adder-subtracter like the SAP computer. Therefore, with the typical microprocessor, you have to use some form of programmed multiplication such as repeated addition.

EXAMPLE 11-9

Modify the foregoing multiply program by using a JNZ instead of a JZ.

SOLUTION

Look at this:

Label	Mnemonic	Comment
	MVI A,00H	;Clear accumulator
	MVI B,0CH	;Load decimal 12 into B
	MVI.C,08H	;Preset counter with 8
REPEAT:	ADD B	;Add decimal 12
	DCR C	;Decrement the counter
	JNZ REPEAT	;Test for zero
	HLT	;Stop it

This is simpler. It eliminates one JMP instruction and one label. As long as the counter is greater than zero, the JNZ will force the computer to loop back to REPEAT. When the counter drops to zero, the program will fall through the JNZ to the HLT.

EXAMPLE 11-10

Hand-assemble the foregoing program starting at address 2000H.

SOLUTION

Address	Contents	Symbolic
2000H	3EH	MVI A,00H
2001H	00H	
2002H	06H	MVI B,0CH
2003H	0CH	
2004H	0EH	MVI, C,08H
2005H	08H	
2006H	80H	ADD B
2007H	0DH	DCR C
2008H	C2H	JNZ 2006H
2009H	06H	
200AH	20H	
200BH	76H	HLT

The first three instructions initialize the registers before the multiplication begins. If we change the initial values, we can multiply other numbers.

EXAMPLE 11-11

Change the multiplication part of the foregoing program into a subroutine located at starting address F006H.

SOLUTION

Address	Contents	Symbolic
F006H	80H	ADD B
F007H	0DH	DCR C
F008H	C2H	JNZ F006H
F009H	06H	
F00AH	F0H	
F00BH	C9H	RET

Here's what happened. The initializing instructions depend on the numbers we are multiplying, so they don't belong in the subroutine. The subroutine should contain only the multiplication part of the program.

In relocating the program we *mapped* (converted) addresses 2006H–200BH to F006H–F00BH. Also, the HLT was changed to a RET to get us back to the original program.

EXAMPLE 11-12

The multiply subroutine of the preceding example is used in the following program. What does the program do?

```
MVI A,00H
MVI B,10H
MVI C,0EH
CALL F006H
HLT
```

SOLUTION

Hexadecimal 10H is equivalent to decimal 16, and hexadecimal 0EH is equivalent to decimal 14. The first three instructions clear the accumulator, load the B register with decimal 16, and preset the counter to decimal 14. The CALL sends the computer to the multiply subroutine of the preceding example. When the RET is executed, the accumulator contents are E0H, which is equivalent to 224.

Incidentally, a *parameter* is a piece of data that the subroutine needs to work properly. The multiply subroutine located at F006H needs three parameters to work properly (*A*, *B*, and *C*). We pass these parameters to the multiply subroutine by clearing the accumulator, loading the B register with the multiplicand, and presetting the C register with the multiplier. In other words, we set $A = 00H$, $B = 10H$, and $C = 0EH$. Passing data to a subroutine in this way is called *register parameter passing*.

11-6 LOGIC INSTRUCTIONS

A microprocessor can do logic as well as arithmetic. What follows are the SAP-2 logic instructions. Again, they are a subset of the 8080/8085 instructions.

CMA

CMA stands for "complement the accumulator." The execution of a CMA inverts each bit in the accumulator, producing the 1's complement.

ANA

ANA means to AND *the accumulator* contents with the designated register. The result is stored in the accumulator. For instance,

ANA B

means to AND the contents of the accumulator with the contents of the B register. The ANDing is done on a bit-by-bit basis. For example, suppose the two registers contain

$$\mathbf{A} = 1100\ 1100 \qquad (11\text{-}1)$$

and

$$\mathbf{B} = 1111\ 0001 \qquad (11\text{-}2)$$

The execution of an ANA B results in

$$\mathbf{A} = 1100\ 0000$$

Notice that the ANDing is bitwise, as illustrated in Fig. 11-6. The ANDing is done on pairs of bits; A_7 is ANDed with B_7, A_6 with B_6, A_5 with B_5, and so on, with the result stored in the accumulator.

Two ANA instructions are available in SAP-2: ANA B and ANA C. Table 11-1 shows the op codes.

ORA

ORA is the mnemonic for OR *the accumulator* with the designated register. The two ORA instructions in SAP-2 are ORA B and ORA C. As an example, if the accumulator and B register contents are given by Eqs. 11-1 and 11-2, then executing ORA B gives

$$\mathbf{A} = 1111\ 1101$$

XRA

XRA means XOR *the accumulator* with the designated register. The SAP-2 instruction set contains XRA B and XRA C. If the accumulator and B contents are given by Eqs. 11-1 and 11-2, the execution of XRA B produces

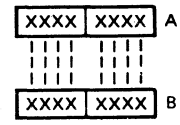


Fig. 11-6 Logic instructions are bitwise.

$$\mathbf{A} = 0011\ 1101$$

ANI

SAP-2 also has immediate logic instructions. ANI means AND *immediate*. It tells the computer to AND the accumulator contents with the byte that immediately follows the op code. For instance, if

$$\mathbf{A} = 0101\ 1110$$

the execution of ANI C7H will AND

0101 1110 with 1100 0111

to produce new accumulator contents of

$$\mathbf{A} = 0100\ 0110$$

ORI

ORI is the mnemonic for OR *immediate*. The accumulator contents are ORed with the byte that follows the op code. If

$$\mathbf{A} = 0011\ 1000$$

the execution of ORI 5AH will OR

0011 1000 with 0101 1010

to produce new accumulator contents of

0111 1010

XRI

XRI means XOR *immediate*. If

$$\mathbf{A} = 0001\ 1100$$

the execution of XRI D4H will XOR

0001 1100 with 1101 0100

to produce

$$\mathbf{A} = 1100\ 1000$$

11-7 OTHER INSTRUCTIONS

This section looks at the last of the SAP-2 instructions. Since these instructions don't fit any particular category, they are being collected here in a miscellaneous group.

NOP

NOP stands for *no operation*. During the execution of a NOP, all *T* states are do nothings. Therefore, no register changes occur during a NOP.

The NOP instruction is used to waste time. It takes four *T* states to fetch and execute the NOP instruction. By repeating a NOP a number of times, we can delay the data processing, which is useful in timing operations. For instance, if we put a NOP inside a loop and execute it 100 times, we create a time delay of 400 *T* states.

HLT

We have already used this. HLT stands for *halt*. It ends the data processing.

IN

IN is the mnemonic for *input*. It tells the computer to transfer data from the designated port to the accumulator. Since there are two input ports, you have to designate which one is being used. The format for an input operation is

IN byte

For instance,

IN 02H

means to transfer the data in port 2 to the accumulator.

OUT

OUT stands for *output*. When this instruction is executed, the accumulator word is loaded into the designated output port. The format for this instruction is

OUT byte

Since the output ports are numbered 3 and 4 (Fig. 11-2), you have to specify which port is to be used. For instance,

OUT 03H

will transfer the contents of the accumulator to port 3.

RAL

RAL is the mnemonic for *rotate the accumulator left*. This instruction will shift all bits to the left and move the MSB into the LSB position, as illustrated in Fig. 11-7*a*. As an example, suppose the contents of the accumulator are

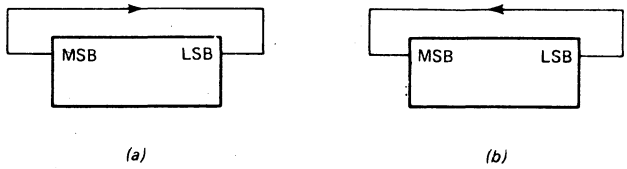


Fig. 11-7 Rotate instructions: (*a*) RAL; (*b*) RAR.

A = 1011 0100

Executing the RAL will produce

A = 0110 1001

As you see, all bits moved left, and the MSB went to the LSB position.

RAR

RAR stands for *rotate the accumulator right*. This time, the bits shift to the right, the LSB going to the MSB position, as shown in Fig. 11-7*b*. If

A = 1011 0100

the execution of a RAR will result in

A = 0101 1010

EXAMPLE 11-13

The bits in a byte are numbered 7 to 0 (MSB to LSB). Show a program that can input a byte from port 2 and determine if bit 0 is a 1 or a 0. If the bit is a 1, the program is to load the accumulator with an ASCII Y (yes). If the bit is a 0, the program should load the accumulator with an ASCII N (no). The yes or no answer is to be sent to output port 3.

SOLUTION

```
Label    Mnemonic     Comment
         IN 02H       ;Get byte from port 2
         ANI 01H      ;Isolate bit 0
         JNZ YES      ;Jump if bit 0 is a 1
         MVI A,4EH    ;Load N into accumulator
         JMP DONE     ;Skip next instruction
YES:     MVI A,59H    ;Load Y into accumulator
DONE:    OUT 03H      ;Send answer to port 3
         HLT
```

The IN 02H transfers the contents of input port 2 to the accumulator to get

$$A = A_7A_6A_5A_4A_3A_2A_1A_0$$

The immediate byte in ANI 01H is

0000 0001

This byte is called a *mask* because its 0s will mask or blank out the corresponding high bits in the accumulator. In other words, after the execution of ANI 01H the accumulator contents are

$$A = 0000\ 000A_0$$

If A_0 is 1, the JNZ YES will produce a jump to the MVI A,59H; this loads a 59H (the ASCII for Y) into the accumulator. If A_0 is 0, the program falls through to the MVI A,4EH. This loads the accumulator with the ASCII for N.

The OUT 03H loads the answer, either ASCII Y or N, into port 3. The hexadecimal display therefore shows either 59H or 4EH.

EXAMPLE 11-14

Instead of a parallel output at port 3, we want a serial output at port 4. Modify the foregoing program so that it converts the answer (59H or 4EH) into a serial output at bit 0, port 4.

SOLUTION

Label	Mnemonic	Comment
	IN 02H	
	ANI 01H	
	JNZ YES	
	MVI A,4EH	
	JMP DONE	
YES:	MVI A,59H	
DONE:	MVI C,08H	;Load counter with 8
AGAIN:	OUT 04H	;Send LSB to port 4
	RAR	;Position next bit
	DCR C	;Decrement count
	JNZ AGAIN	;Test count
	HLT	

In converting from parallel to serial data, the A_0 bit is sent first, then the A_1 bit, then the A_2 bit, and so on.

EXAMPLE 11-15

Handshaking is an interaction between a CPU and a peripheral device that takes place during an I/O data transfer.

In SAP-2 the handshaking takes place as follows. After you enter two digits (1 byte) into the hexadecimal encoder of Fig. 11-2, the data is loaded into port 1; at the same time, a *high READY* bit is sent to port 2.

Before accepting input data, the CPU checks the *READY* bit in port 2. If the *READY* bit is low, the CPU waits. If the *READY* bit is high, the CPU loads the data in port 1. After the data transfer is finished, the CPU sends a high *ACKNOWLEDGE* signal to the hexadecimal keyboard encoder; this resets the *READY* bit to 0. The *ACKNOWLEDGE* bit then is reset to low.

After you key in a new byte, the cycle starts over with new data going to the port 1 and a high *READY* bit to port 2.

The sequence of SAP-2 handshaking is

1. *READY* bit (bit 0, port 2) goes high.
2. Input the data in port 1 to the CPU.
3. *ACKNOWLEDGE* bit (bit 7, port 4) goes high to reset *READY* bit.
4. Reset the *ACKNOWLEDGE* bit.

Write a program that inputs a byte of data from port 1 using handshaking. Store the byte in the B register.

SOLUTION

Label	Mnemonic	Comment
STATUS:	IN 02H	;Input byte from port 2
	ANI 01H	;Isolate *READY* bit
	JZ STATUS	;Jump back if not ready
	IN 01H	;Transfer data in port 1
	MOV B,A	;Transfer from A to B
	MVI A,80H	;Set *ACKNOWLEDGE* bit
	OUT 04H	;Output high *ACKNOWLEDGE*
	MVI A,00H	;Reset *ACKNOWLEDGE* bit
	OUT 04H	;Output low *ACKNOWLEDGE*
	HLT	

If the *READY* bit is low, the ANI 01H will force the accumulator contents to go to zero. The JZ STATUS therefore will loop back to IN 02H. This looping will continue until the *READY* bit is high, indicating valid data in port 1.

When the *READY* bit is high, the program falls through the JZ STATUS to the IN 01H. This transfers a byte from port 1 to the accumulator. The MOV sends the byte to the B register. The MVI A,80H sets the *ACKNOWLEDGE* bit

(bit 7). The OUT 04H sends this high *ACKNOWLEDGE* to the hexadecimal encoder where the internal hardware resets the *READY* bit. Then the *ACKNOWLEDGE* bit is reset in preparation for the next input cycle.

11-8 SAP-2 SUMMARY

This section summarizes the SAP-2 *T* states, flags, and addressing modes.

T States

The SAP-2 controller-sequencer is microprogrammed with a variable machine cycle. This means that some instructions take longer than others to execute. As you recall, the idea behind microprogramming is to store the control routines in a ROM and access them as needed.

Table 11-3 shows each instruction and the number of *T* states needed to execute it. For instance, it takes four *T* states to execute the ADD B instruction, seven to execute the ANI byte, eighteen to execute the CALL, and so on. Knowing the number of *T* states is important in timing applications.

Notice that the JM instruction has *T* states of 10/7. This means it takes 10 *T* states when a jump occurs but only 7 without the jump. The same idea applies to the other conditional jumps; 10 *T* states for a jump, 7 with no jump.

Flags

As you know, the accumulator goes negative or zero during the execution of some instructions. This affects the sign and zero flags. Figure 11-8 shows the circuits used in SAP-2 to set the flags.

When the accumulator contents are negative, the leading bit A_7 is a 1. This sign bit drives the lower AND gate. When the accumulator contents are zero, all bits are zero and the output of the XOR gate is a 1. This XOR output drives the upper AND gate. If gating signal L_F is high, the flags will be updated to reflect the sign and zero condition of the accumulator. This means the Z_{FLAG} will be high when the accumulator contents are zero; the S_{FLAG} will be high when the accumulator contents are negative.

Not all instructions affect the flags. As shown in Table 11-3, the instructions that update the flags are ADD, ANA, ANI, DCR, INR, ORA, ORI, SUB, XRA, and XRI. Why only these instructions? Because the L_F signal of Fig. 11-8 is high only when these instructions are executed. This is accomplished by microprogramming an L_F bit for each instruction. In other words, in the control ROM we store a high L_F bit for the foregoing instructions, and a low L_F bit for all others.

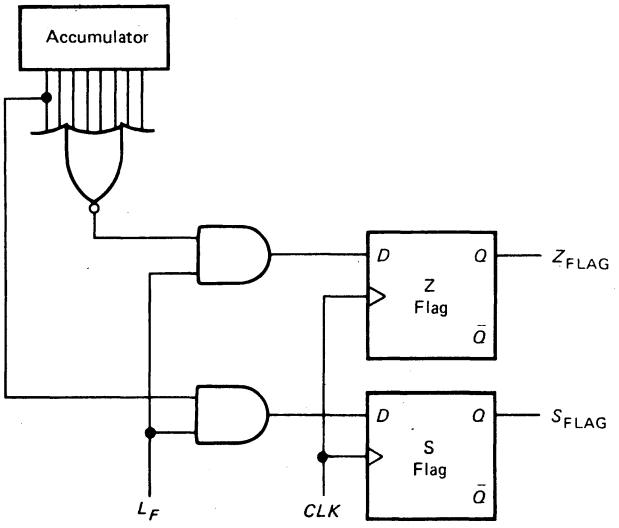


Fig. 11-8 Setting the flags.

Conditional Jumps

As mentioned earlier, the conditional jumps take ten *T* states when the jump occurs but only seven *T* states when no jump take place. Briefly, this is accomplished as follows During the execution cycle the address ROM sends the computer to the starting address of a conditional-jump microroutine. The initial microinstruction looks at the flags and judges whether or not to jump. If a jump is indicated, the microroutine continues; otherwise, it is aborted and the computer begins a new fetch cycle.

Addressing Modes

The SAP-2 instructions access data in different ways. It is the operand that tells us how the data is to be accessed. For instance, the first instructions discussed were

```
LDA address
STA address
```

These are examples of *direct* addressing because we specify the address where the data is to be found.

Immediate addressing is different. Instead of giving an address for the data, we give the data itself. For instance,

```
MVI A,byte
```

accesses the data to be loaded into the accumulator by using the byte in memory that immediately follows the op code. Table 11-3 shows the other immediate instructions.

An instruction like

```
MOV A,B
```

TABLE 11-3. SAP-2 INSTRUCTION SET

Instruction	Op Code	*T* States	Flags	Addressing	Bytes
ADD B	80	4	S, Z	Register	1
ADD C	81	4	S, Z	Register	1
ANA B	A0	4	S, Z	Register	1
ANA C	A1	4	S, Z	Register	1
ANI byte	E6	7	S, Z	Immediate	2
CALL address	CD	18	None	Immediate	3
CMA	2F	4	None	Implied	1
DCR A	3D	4	S, Z	Register	1
DCR B	05	4	S, Z	Register	1
DCR C	0D	4	S, Z	Register	1
HLT	76	5	None	—	1
IN byte	DB	10	None	Direct	2
INR A	3C	4	S, Z	Register	1
INR B	04	4	S, Z	Register	1
INR C	0C	4	S, Z	Register	1
JM address	FA	10/7	None	Immediate	3
JMP address	C3	10	None	Immediate	3
JNZ address	C2	10/7	None	Immediate	3
JZ address	CA	10/7	None	Immediate	3
LDA address	3A	13	None	Direct	3
MOV A,B	78	4	None	Register	1
MOV A,C	79	4	None	Register	1
MOV B,A	47	4	None	Register	1
MOV B,C	41	4	None	Register	1
MOV C,A	4F	4	None	Register	1
MOV C,B	48	4	None	Register	1
MVI A,byte	3E	7	None	Immediate	2
MVI B,byte	06	7	None	Immediate	2
MVI C,byte	0E	7	None	Immediate	2
NOP	00	4	None	—	1
ORA B	B0	4	S, Z	Register	1
ORA C	B1	4	S, Z	Register	1
ORI byte	F6	7	S, Z	Immediate	2
OUT byte	D3	10	None	Direct	2
RAL	17	4	None	Implied	1
RAR	1F	4	None	Implied	1
RET	C9	10	None	Implied	1
STA address	32	13	None	Direct	3
SUB B	90	4	S, Z	Register	1
SUB C	91	4	S, Z	Register	1
XRA B	A8	4	S, Z	Register	1
XRA C	A9	4	S, Z	Register	1
XRI byte	EE	7	S, Z	Immediate	2

is an example of *register addressing*. The data to be loaded is stored in a CPU register rather than in the memory. Register addressing has the advantage of speed because fewer *T* states are needed for this type of instruction.

Implied addressing means that the location of the data is contained within the op code itself. For instance,

RAL

tells us to rotate the accumulator bits left. The data is in the accumulator; this is why no operand is needed with implied addressing.

Bytes

Each instruction occupies a number of bytes in the memory. SAP-2 instructions are either 1, 2, or 3 bytes long. Table 11-3 shows the length of each instruction. As you see, ADD instructions are 1-byte instructions, ANI instructions are 2-byte instructions, CALLs are 3-byte instructions, and so forth.

EXAMPLE 11-16

SAP-2 has a clock frequency of 1 MHz. This means that each *T* state has a duration of 1 μs. How long does it take to execute the following SAP-2 subroutine?

Label	Mnemonic	Comment
	MVI C,46H	;Preset count to decimal 70
AGAIN:	DCR C	;Count down
	JNZ AGAIN	;Test count
	NOP	;Delay
	RET	

SOLUTION

The MVI is executed once to initialize the count. The DCR is executed 70 times. The JNZ jumps back 69 times and falls through once. With the number of *T* states given in Table 11-3, we can calculate the total execution time of the subroutine as follows:

MVI:	1 × 7 × 1 μs = 7 μs	
DCR:	70 × 4 × 1 μs = 280	
JNZ:	69 × 10 × 1 μs = 690	(jump)
JNZ:	1 × 7 × 1 μs = 7	(no jump)
NOP:	1 × 4 × 1 μs = 4	
RET:	1 × 10 × 1 μs = 10	
	998 μs ≈ 1 ms	

As you see, the total time needed to execute the subroutine is approximately 1 ms.

A subroutine like this can produce a time delay of 1 ms whenever it is called. There are many applications where you need a delay.

According to Table 11-3, the instructions in the foregoing subroutine have the following byte lengths:

Instruction	MVI	DCR	JNZ	NOP	RET
Bytes	2	1	3	1	1

The total byte length of the subroutine is 8. As part of the SAP-2 software, the foregoing subroutine can be assembled and relocated at addresses F010H to F017H. Hereafter, the execution of a CALL F010H will produce a time delay of 1 ms.

EXAMPLE 11-17

How much time delay does this SAP-2 subroutine produce?

Label	Mnemonic	Comment
	MVI B,0AH	;Preset B counter with decimal 10
LOOP1:	MVI C,47H	;Preset C counter with decimal 71
LOOP2:	DCR C	;Count down on C
	JNZ LOOP2	;Test for C count of zero
	DCR B	;Count down on B
	JNZ LOOP1	;Test for B count of zero
	RET	

SOLUTION

This subroutine has two loops, one inside the other. The inner loop consists of DCR C and JNZ LOOP2. This inner loop produces a time delay of

DCR C:	71 × 4 × 1 μs = 284 μs	
JNZ LOOP2:	70 × 10 × 1 μs = 700	(jump)
JNZ LOOP2:	1 × 7 × 1 μs = 7	(no jump)
	991 μs	

When the C count drops to zero, the program falls through the JNZ LOOP2. The B counter is decremented, and the JNZ LOOP1 sends the program back to the MVI C,47H. Then we enter LOOP2 for a second time. Because LOOP2 is inside LOOP1, LOOP2 will be executed 10 times and the overall time delay will be approximately 10 ms.

Here are the calculations for the overall subroutine:

MVI B,0AH:	1 × 7 × 1 μs = 7 μs	
MVI C,47H:	10 × 7 × 1 μs = 70	
LOOP2:	10 × 991 μs = 9,910	
DCR B:	10 × 4 × 1 μs = 40	
JNZ LOOP1:	9 × 10 × 1 μs = 90	(jump)
JNZ LOOP1:	1 × 7 × 1 μs = 7	(no jump)
RET:	1 × 10 × 1 μs = 10	
	10,134 μs ≈ 10 ms	

This SAP-2 subroutine has a byte length of

$$2 + 2 + 1 + 3 + 1 + 3 + 1 = 13$$

It can be assembled and located at addresses F020H to F02CH. From now on, a CALL F020H will produce a time delay of approximately 10 ms.

By changing the first instruction to

MVI B,64H

the B counter is preset with decimal 100. In this case, the inner loop is executed 100 times and the overall time delay is approximately 100 ms. This 100-ms subroutine can be located at addresses F030H to F03CH.

EXAMPLE 11-18

Here is a subroutine with three loops *nested* one inside the other. How much time delay does it produce?

```
Label    Mnemonic      Comment
         MVI A,0AH     ;Preset A counter with
                         decimal 10
LOOP1:   MVI B,64H     ;Preset B counter with
                         decimal 100
LOOP2:   MVI C,47H     ;Preset C counter with
                         decimal 71
LOOP3:   DCR C         ;Count down C
         JNZ LOOP3     ;Test C for zero
         DCR B         ;Count down B
         JNZ LOOP2     ;Test B for zero
         DCR A         ;Count down A
         JNZ LOOP1     ;Test A for zero
         RET
```

SOLUTION

LOOP3 still takes approximately 1 ms to get through. LOOP2 makes 100 passes through LOOP3, so it takes about 100 ms to complete LOOP2. LOOP1 makes 10 passes through LOOP2; therefore, it takes around 1 s to complete the overall subroutine.

What do we have? A 1-s subroutine. It will be located in F040H to F052H. To produce a 1-s time delay, we would use a CALL F040H.

By changing the initial instruction to

MVI A,64H

LOOP1 will make 100 passes through LOOP2, which makes 100 passes through LOOP3. The resulting subroutine can be located at F060H to F072H and will produce a time delay of 10 s.

Table 11-4 summarizes the SAP-2 time delays. With these subroutines, we can produce delays from 1 ms to 10 s.

TABLE 11-4. SAP-2 SUBROUTINES

Label	Starting Address	Delay	Registers Used
D1MS	F010H	1 ms	C
D10MS	F020H	10 ms	B, C
D100MS	F030H	100 ms	B, C
D1SEC	F040H	1 s	A, B, C
D10SEC	F060H	10 s	A, B, C

EXAMPLE 11-19

The traffic lights on a main road show green for 50 s, yellow for 6 s, and red for 30 s. Bits 1, 2, and 3 of port 4 are the control inputs to peripheral equipment that runs these traffic lights. Write a program that produces time delays of 50, 6, and 30 s for the traffic lights.

SOLUTION

```
Label    Mnemonic      Comment
AGAIN:   MVI A,32H     ;Preset counter with
                         decimal 50
         STA SAVE      ;Save accumulator
                         contents
         MVI A,02H     ;Set bit 1
         OUT 04H       ;Turn on green light
LOOPGR:  CALL D1SEC    ;Call 1-s subroutine
         LDA SAVE      ;Load current A count
         DCR A         ;Decrement A count
         STA SAVE      ;Save reduced A count
         JNZ LOOPGR    ;Test for zero
         MVI A,06H     ;Preset counter with
                         decimal 6
         STA SAVE
         MVI A,04H     ;Set bit 2
         OUT 04H       ;Turn on yellow light
LOOPYE:  CALL D1SEC
         LDA SAVE
         DCR A
         STA SAVE
         JNZ LOOPYE
         MVI A,1EH     ;Preset counter with
                         decimal 30
         STA SAVE
         MVI A,08H     ;Set bit 3
         OUT 04H       ;Turn on red light
LOOPRE:  CALL D1SEC
         LDA SAVE
         DCR A
         STA SAVE
         JNZ LOOPRE
         JMP AGAIN
SAVE:    Data
```

Let's go through the green part of the program; the yellow and red are similar. The green starts with MVI A,32H, which loads decimal 50 into the accumulator. The STA SAVE will store this initial value in a memory location called SAVE. The MVI A,02H sets bit 1 in the accumulator; then the OUT 04H transfers this high bit to port 4. Since this port controls the traffic lights, the green light comes on.

The CALL D1SEC produces a time delay of 1 s. The LDA SAVE loads the accumulator with decimal 50. The DCR A decrements the count to decimal 49. The STA SAVE stores this decimal 49. Then the JNZ LOOPGR takes the program back to the CALL D1SEC for another 1-s delay.

The CALL D1SEC is executed 50 times; therefore, the green light is on for 50 s. Then the program falls through the JNZ LOOPGR to the MVI A,06H. The yellow part of the program then begins and results in the yellow light being on for 6 s. Finally, the red part of the program is executed and the red light is on for 30 s. The JMP AGAIN repeats the whole process. In this way, the program is controlling the timing of the green, yellow, and red lights.

EXAMPLE 11-20

Middle C on a piano has a frequency of 261.63 Hz. Bit 5 of port 4 is connected to an amplifier which drives a loudspeaker. Write a program that sends middle C to the loudspeaker.

SOLUTION

To begin with, the period of middle C is

$$T = \frac{1}{f} = \frac{1}{261.63 \text{ Hz}} = 3{,}822 \ \mu\text{s}$$

What we are going to do is send to port 4 a signal like Fig. 11-9. This square wave is high for 1,911 μs and low for 1,911 μs. The overall period is 3,822 μs, and the frequency is 261.63 Hz. Because the signal is square rather than sinusoidal, it will sound distorted but it will be recognizable as middle C.

Here is a program that sends middle C to the loudspeaker.

Label	Mnemonic	Comment
LOOP1:	OUT 04H	;Send bit to speaker
	MVI C,86H	;Preset counter with decimal 134
LOOP2:	DCR C	;Count down
	JNZ LOOP2	;Test count
	CMA	;Reset bit 5
	NOP	;Fine tuning
	NOP	;Fine tuning
	JMP LOOP1	;Go back for next half cycle

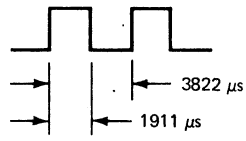


Fig. 11-9 Generating middle C note.

The OUT 04H sends a bit (either low or high) to the loudspeaker. The MVI presets the counter to decimal 134. Then comes LOOP2, the DCR and JNZ, which produces a time delay of 1,866 μs. The program then falls through to the CMA, which complements all bits in the accumulator. The two NOPs add a time delay of 8 μs. The JMP LOOP1 then takes the program back. When the OUT 04H is executed, bit 5 (complemented) goes to the loudspeaker. In this way the loudspeaker is driven into the opposite state. The execution time for both half cycles is 3,824 μs, close enough to middle C.

Here are the calculations for the time delay:

OUT 04H:	1 × 10 × 1 μs =	10 μs
MVI C,86H:	1 × 7 × 1 μs =	7
DCR C:	134 × 4 × 1 μs =	536
JNZ LOOP2:	133 × 10 × 1 μs =	1,330
JNZ LOOP2:	1 × 7 × 1 μs =	7
CMA:	1 × 4 × 1 μs =	4
2 NOPs:	2 × 4 × 1 μs =	8
JMP LOOP1:	1 × 10 × 1 μs =	10
		1,912 μs

This is the half-cycle time. The period is 3,824 μs.

EXAMPLE 11-21

Serial data is sometimes called a serial data stream because bits flow one after another. In Fig. 11-10 a serial data stream drives bit 7 of port 2 at a rate of approximately 600 bits per second. Write a program that inputs an 8-bit character in a serial data stream and stores it in memory location 2100H.

SOLUTION

Since approximately 600 bits are received each second, the period of each bit is

$$\frac{1}{600 \text{ Hz}} = 1{,}667 \ \mu\text{s}$$

The idea will be to input a bit from port 2, rotate the accumulator right, wait approximately 1,600 μs, then input another bit, rotate the accumulator right, and so on, until all bits have been received.

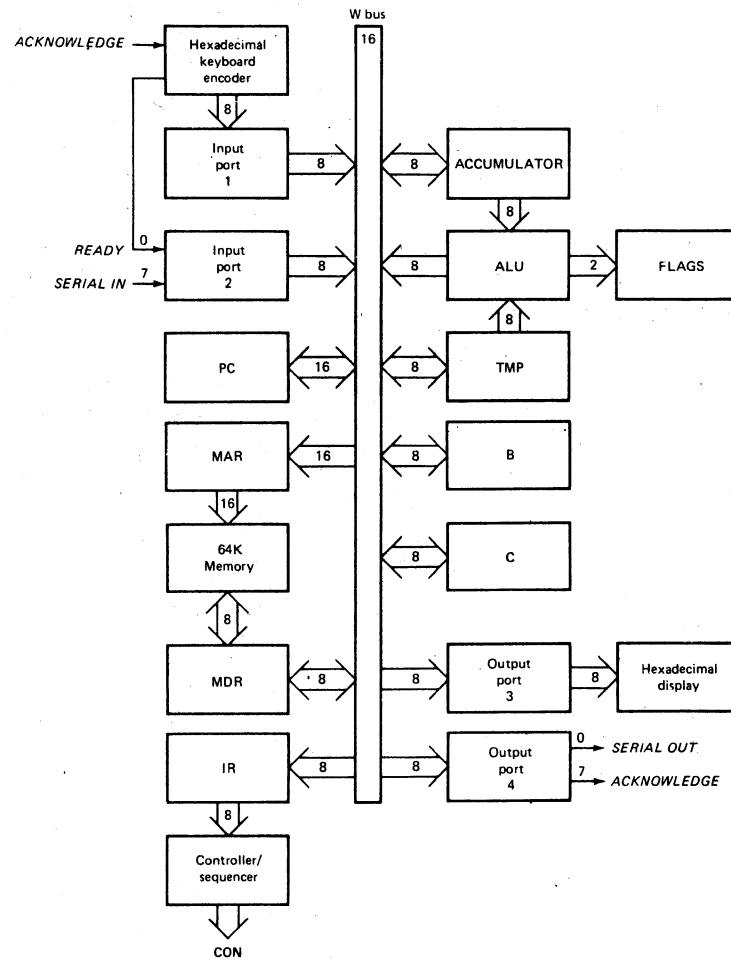


Fig. 11-10

Here is a program that will work:

Label	Mnemonic	Comment
	MVI B,00H	;Load zero into B register
	MVI C,07H	;Preset counter with decimal 7
BIT:	IN 02H	;Input data
	ANI 80H	;Isolate bit 7
	ORA B	;Update character
	RAR	;Move bits right
	MOV B,A	;Save bits in B
	MVI A,73H	;Begin a delay of 1,600 μs
DELAY:	DCR A	;Count down A
	JNZ DELAY	;Test A count for zero
	DCR C	;Count down C
	JNZ BIT	;Test C count for zero
	IN 02H	;Input last bit
	ANI 80H	;Isolate bit 7
	ORA B	
	STA 2100H	;Save character

The first instruction clears the B register. The second instruction loads decimal 7 into the C counter. The IN 02H brings in the data from port 2. The ANI mask isolates bit 7 because this is the *SERIAL IN* bit from port 2. The ORA B does nothing the first time through because B is full of 0s. The RAR moves the accumulator bits to the right. The MOV B,A stores the accumulator contents in the B register.

MVI A,73H presets the accumulator with decimal 115. Then comes a delay loop, DCR A and JNZ DELAY, that takes approximately 1,600 μs to complete.

The DCR C reduces the C count by 1, and the JNZ BIT tests the C count for zero. The program jumps back to the IN 02H to get the next bit from the serial data stream. The ANI mask isolates bit 7, which is then ORed with the contents of the B register; this combines the previous bit with the newly received bit. After another RAR, the two received bits are stored in the B register. Then comes another delay of approximately 1,600 μs.

The program continues to loop and each time a new bit is input from the serial data stream. After 7 bits have been

received, the program will fall through the JNZ BIT instruction.

The last four instructions do the following. The IN 02H brings in the eighth bit. The ANI isolates bit 7. The ORA B combines this new bit with the other seven bits in the B register. At this point, all received bits are in the accumulator. The STA 2100H then stores the byte in the accumulator at 2100H.

A concrete example will help. Suppose the 8 bits being received are 57H, the ASCII code for W. The LSB is received first, the MSB last. Here is how the contents of the B register appear after the execution of the ORA B:

A = 1000 0000	(First pass through loop)
A = 1100 0000	(Second pass)
A = 1110 0000	(Third pass)
A = 0111 0000	(Fourth pass)
A = 1011 1000	(Fifth pass)
A = 0101 1100	(Sixth pass)
A = 1010 1110	(Seventh pass)
A = 0101 0111	(Final contents)

Incidentally, the ASCII code only requires 7 bits; for this reason, the eighth bit (A_7) may be set to zero or used as a parity bit.

GLOSSARY

assembler A program that converts a source program into a machine-language program.

comment Personal notes in an assembly-language program that are not assembled. They refresh the programmer's memory at a later date.

conditional jump A jump that occurs only if certain conditions are satisfied.

direct addressing Addressing in which the instruction contains the address of the data to be operated on.

flag A flip-flop that keeps track of a changing condition during a computer run.

hand assembling Translating a source program into a machine-language program by hand rather than computer.

handshaking Interaction between a CPU and a peripheral device that takes place during an I/O operation. In SAP-2 it involves *READY* and *ACKNOWLEDGE* signals.

immediate addressing Addressing in which the data to be operated on is the byte immediately following the op code of the instruction.

implied addressing Addressing in which the location of the data is contained within the mnemonic.

label A name given to an instruction in an assembly-language program. To jump to this instruction, you can use the label rather than the address. The assembler will work out the correct address of the label and will use this address in the machine-language program.

mask A byte used with an ANI instruction to blank out certain bits.

register addressing Addressing in which the data is stored in a CPU register.

relocate To move a program or subroutine to another part of the memory. In doing this, the addresses of jump instructions must be converted to new addresses.

subroutine A program stored in higher memory that can be used repeatedly as part of a main program.

SELF-TESTING REVIEW

Read each of the following and provide the missing words. Answers appear at the beginning of the next question.

1. The controller-sequencer produces _____ words or microinstructions.
2. (*control*) A flag is a _____ that keeps track of a changing condition during a computer run. The sign flag is set when the accumulator contents go negative. The _____ flag is set when the accumulator contents go to zero.
3. (*flip-flop, zero*) In coding the LDA address and STA address instructions, the _____ byte of the address is stored in lower memory, the _____ byte in upper memory.
4. (*lower, upper*) The JMP instruction changes the program sequence by jumping to another part of the program. With the JM instruction, the jump is executed only if the sign flag is _____. With the JNZ instruction, the jump is executed only if the zero flag is _____.
5. (*set, clear*) Every subroutine must terminate with a _____ instruction. This returns the program to the instruction following the CALL. The CALL instruction is unconditional; it sends the computer to the starting address of a _____.
6. (*RET, subroutine*) An assembler allows you to write programs in mnemonic form. Then the assembler

converts these mnemonics into _____ language. The assembler ignores the _____ following a semicolon and assigns addresses to the labels. Labels can be up to six characters, the first of which must be a _____.

7. (*machine, comments, letter*) Repeated addition is one way to do _____. Programmed multiplication is used in most microprocessors because their ALUs can only add and subtract.
8. (*multiplication*) A parameter is a piece of data passed to a _____. When you call a subroutine, you often need to pass _____ for the subroutine to work properly.
9. (*subroutine, parameters*) A _____ is used to isolate a bit; it does this because the ANI sets all other bits to zero.
10. (*mask*) Handshaking is an interaction between a _____ and a peripheral device. In SAP-2 the _____ bit tells the CPU whether the input data is valid or not. After the data has been transferred into the computer, the CPU sends an _____ bit to the peripheral device.
11. (CPU, *READY, ACKNOWLEDGE*) The SAP-2 computer is microprogrammed with a _____ machine cycle. This means that some instructions take longer than others to execute.
12. (*variable*) The types of addressing covered up to now are direct, immediate, register, and implied.

PROBLEMS

11-1. Write a source program that loads the accumulator with decimal 100, the B register with decimal 150, and the C register with decimal 200.

11-2. Hand-assemble the source program of the preceding problem starting at address 2000H.

11-3. Write a source program that stores decimal 50 at memory location 4000H, decimal 51 at 4001H, and decimal 52 at 4002H.

11-4. Hand-assemble the source program in the preceding problem starting at address 2000H.

11-5. Write a source program that adds decimal 68 and 34, with the answer stored at memory location 5000H.

11-6. Hand-assemble the preceding program starting at address 2000H.

11-7. Here is a program:

Label	Mnemonic
LOOP:	MVI C,78H
	DCR C
	JNZ LOOP
	HLT

a. How many times (decimal) is the DCR C executed?
b. How many times does the program jump to LOOP?
c. How can you change the program to loop 210 times?

11-8. Which of the following are valid labels?
a. G100
b. UPDATE
c. 5TIMES
d. 678RED
e. T
f. REPEAT

11-9. Write a program that multiplies decimal 25 and 7 and stores the answer at 2000H. (Use the multiply subroutine located at F006H.)

11-10. Write a program that inputs a byte from port 1 and determines if the decimal equivalent is even or odd. If the byte is even, the program is to send an ASCII E to port 3; if odd, an ASCII O.

11-11. Modify the foregoing program so that it sends the answer in serial form to bit 0 of port 4.

11-12. Write a program that inputs a byte from port 1 using handshaking. Store the byte at address 4000H.

11-13. Hand assemble the foregoing program starting at address 2000H.

11-14. Write a subroutine that produces a time delay of approximately 500 μs.

11-15. Hand-assemble the preceding program starting at address 2000H.

11-16. Write a subroutine that produces a time delay of approximately 35 ms using a SAP-2 subroutine. Hand-assemble this subroutine and locate it at starting address E000H.

11-17. Write a subroutine that produces a time delay of 50 ms. (Use a SAP-2 subroutine.) Hand-assemble the program at starting address E100H.

11-18. Write a subroutine that produces a delay of 1 min. (Use CALL F060H.)

11-19. Hand-assemble the preceding subroutine at starting addresses F080H.

11-20. The C note one octave above middle C has a frequency of 523.25 Hz. Write a program that sends this note to bit 4 of port 4.

11-21. Hand-assemble the foregoing program starting at address 2000H.

SAP-3

12

The SAP-3 computer is an 8-bit microcomputer that is upward-compatible with the 8085 microprocessor. In this chapter, the emphasis is on the SAP-3 instruction set. This instruction set includes all the SAP-2 instructions of the preceding chapter plus new instructions to be discussed.

Appendix 5 shows the op codes, *T* states, flags, and so forth, for the SAP-3 instructions. In the remainder of this chapter, refer to Appendix 5 as needed.

12-1 PROGRAMMING MODEL

All you need to know about SAP-3 hardware is the programming model of Fig. 12-1. This is a diagram showing the CPU registers needed by a programmer.

Some of the CPU registers are familiar from SAP-2. For instance, the program counter (PC) is a 16-bit register that can count from 0000H to FFFFH or decimal 0 to 65,535. As you know, the program counter sends out the address of the next instruction to be fetched. This address is latched into the MAR.

CPU registers A, B, and C are the same as in SAP-2. These 8-bit registers are used in arithmetic and logic operations. Since the accumulator is only 8 bits wide, the range of unsigned numbers is 0 to 255; the range of signed 2's-complement numbers is -128 to $+127$.

SAP-3 has additional CPU registers (D, E, H, and L) for more efficient data processing. These 8-bit registers can be loaded with MOV and MVI instructions, the same as the A, B, and C registers. Also notice the F register, which stores flag bits S, Z, and others.

Finally, there is the *stack pointer* (SP), a 16-bit register. This new register controls a portion of memory known as the *stack*. The stack and the stack pointer are discussed later in this chapter.

Figure 12-1 shows all the CPU registers needed to understand the SAP-3 instruction set. With this programming model we can discuss the SAP-3 instruction set, which is upward-compatible with the 8080 and 8085. At the end of this chapter, you will know almost all of the 8080/8085 instruction set.

12-2 MOV AND MVI

The MOV and MVI instructions work the same as in SAP-2. The only difference is more registers to choose from. The format of any move instruction is

MOV reg1, reg2

where reg1 = A, B, C, D, E, H, or L
reg2 = A, B, D, D, E, H, or L

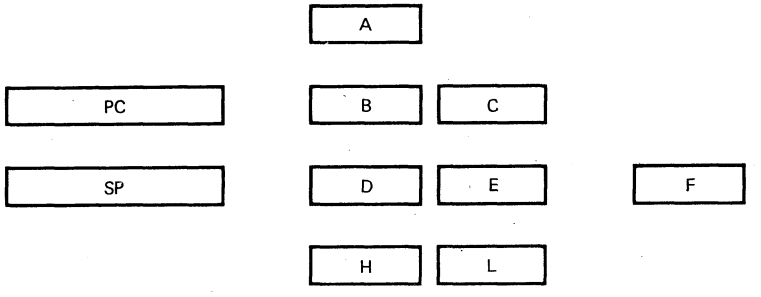


Fig. 12-1 SAP-3 programming model.

The MOV instructions send the data in reg2 to reg1. Symbolically,

$$\text{reg1} \leftarrow \text{reg2}$$

where the arrow indicates that the data in register 2 is copied nondestructively into register 1. At the end of the execution

$$\text{reg1} = \text{reg2}$$

For instance,

MOV L,A

copies A into L, so that

L = A

Similarly,

MOV E,H

gives

E = H

The immediate moves have the format of

MVI reg,byte

where reg = A, B, C, D, E, H, or L. Therefore, the execution of

MVI D,0EH

will result in

D = 0EH

Likewise,

MVI L,FFH

produces

L = FFH

What is the advantage of more CPU registers? As you may recall, MOV and MVI instructions use fewer *T* states than memory-reference instructions (MRIs). The extra CPU registers mean that we can use more MOV and MVI instructions and fewer MRIs. Because of this, SAP-3 programs can run faster than SAP-2 programs; furthermore, having more CPU registers for temporary storage simplifies program writing.

12-3 ARITHMETIC INSTRUCTIONS

Since the accumulator is only 8 bits wide, its contents can represent unsigned numbers from 0 to 255 or signed 2's complement numbers from −128 to +127. Whether signed or unsigned binary numbers are used, the programmer needs to detect *overflows*, sums or differences that lie outside the normal range of the accumulator. This is where the *carry* flag comes in.

Carry Flag

As shown in Fig. 6-7, a 4-bit adder-subtracter produces a sum $S_3S_2S_1S_0$ and a carry. In SAP-1, two 74LS83s (equivalent to eight full adders) produce an 8-bit sum and a carry. In this simple computer, the carry is disregarded. SAP-3, however, takes the carry into account.

Figure 12-2*a* shows the logic circuit used for the SAP-3 adder-subtracter. When *SUB* is low, the circuit adds the **A** and **B** inputs. If a final carry is generated, *CARRY* will be high and *CY* will be high. If there is no final carry, *CY* is low.

On the other hand, when *SUB* is high, the circuit forms the 2's complement of **B**, which is then added to **A**. Because of the final XOR gate, a high *CARRY* out of the last full-adder produces a low *CY*. If no carry occurs, *CY* is high.

In summary,

$$CY = \begin{cases} CARRY & \text{for ADD instructions} \\ \overline{CARRY} & \text{for SUB instructions} \end{cases}$$

During an add operation, *CY* is called a carry. During a subtract operation, *CY* is referred to as a *borrow*.

The 8-bit sum $S_7S_6S_5S_4S_3S_2S_1S_0$ is stored in the accumulator of Fig. 12-2*b*. The carry (or borrow) is stored in a special flip-flop called the *carry flag*, designated CY in Fig. 12-2*b*. This flag acts like the next higher bit of the accumulator. That is,

$$CY = A_8$$

Carry-Flag Instructions

There are two instructions we can use to control the carry flag. The STC instruction will set the CY flag if it is not already set. (STC stands for *set carry*.) So, if

$$CY = 0$$

the execution of a STC instruction produces

$$CY = 1$$

The other carry-flag instruction is the CMC, which stands for *complement the carry*. When executed, a CMC com-

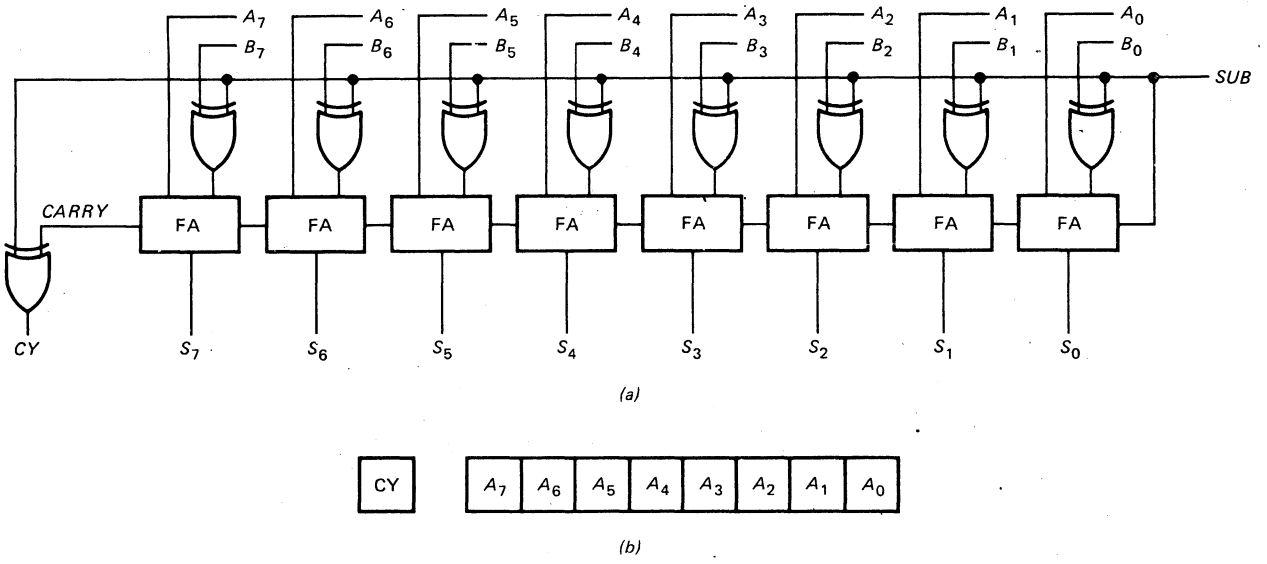


Fig. 12-2 (*a*) SAP-3 adder 2 subtracter; (*b*) carry flag and accumulator.

plements the value of CY. If $CY = 1$, CMC produces a CY of 0. On the other hand, if $CY = 0$, CMC results in a CY of 1.

If you want to reset the carry flag and its current status is unknown, you have to set it, then complement it. That is, execution of

```
STC
CMC
```

guarantees that the final value of CY will be 0 if the initial value of CY is unknown.

ADD Instructions

The format of the ADD instruction is

```
ADD reg
```

where reg = A, B, C, D, E, H, or L. This instruction adds the contents of the specified register to the accumulator contents. The sum is stored in the accumulator and the carry flag is set or reset, depending on whether there is a final carry or not.

For instance, suppose

A = 1111 0001 and **E** = 0000 1000

The instruction

```
ADD E
```

produces the binary addition

```
  1111 0001
+ 0000 1000
  1111 1001
```

There is no final carry; therefore, at the end of the instruction cycle,

$CY = 0$ and **A** = 1111 1001

As another example, suppose

A = 1111 1111 and **L** = 0000 0001

Then executing an ADD L produces

```
  1111 1111
+ 0000 0001
 1 0000 0000
```

At the end of the instruction cycle

$CY = 1$ and **A** = 0000 0000

ADC Instructions

The ADC instruction (add with carry) is formatted like this:

```
ADC reg
```

where reg = A, B, C, D, E, H, or L. This instruction adds the contents of the specified register plus the carry flag to the contents of the accumulator. Because it includes the CY flag, the ADC instruction allows us to add numbers outside the unsigned 0 to 255 range or the signed −128 to +127 range.

As an example, suppose

$$\mathbf{A} = 1000\ 0011$$
$$\mathbf{E} = 0001\ 0010$$

and $CY = 1$

The execution of

ADC E

produces the following addition:

$$\begin{array}{r} 1000\ 0011 \\ 0001\ 0010 \\ +\qquad\quad 1 \\ \hline 1001\ 0110 \end{array}$$

Therefore, the new accumulator and carry flag contents are

$$CY = 0 \qquad \mathbf{A} = 1001\ 0110$$

SUB Instructions

The SUB instruction is formatted as

SUB reg

where reg = A, B, C, D, E, H, or L. This instruction will subtract the contents of the specified register from the accumulator contents; the result is stored in the accumulator. If a final borrow occurs, the CY flag is set. If there is no borrow, the CY flag is reset. In other words, during subtraction the CY flag functions as a borrow flag.

For example, if

$$\mathbf{A} = 0000\ 1111 \quad \text{and} \quad \mathbf{C} = 0000\ 0001$$

then

SUB C

results in

$$\begin{array}{r} 0000\ 1111 \\ -\ 0000\ 0001 \\ \hline 0000\ 1110 \end{array}$$

Notice that there is no final borrow. In terms of 2's-complement addition, the foregoing subtraction appears like this:

$$\begin{array}{r} 0000\ 1111 \\ +\ 1111\ 1111 \\ \hline 1\ 0000\ 1110 \end{array}$$

The final *CARRY* is 1, but this is complemented during subtraction to get a *CY* of 0 (Fig. 12-2*a*). This is why the execution of SUB C produces

$$CY = 0 \qquad \mathbf{A} = 0000\ 1110$$

Here is another example. If

$$\mathbf{A} = 0000\ 1100 \quad \text{and} \quad \mathbf{C} = 0001\ 0010$$

then a SUB C produces

$$\begin{array}{r} 0000\ 1100 \\ -\ 0001\ 0010 \\ \hline 1\ 1111\ 1010 \end{array}$$

Notice the final borrow. This borrow occurs because the contents of the C register (decimal 18) are greater than the contents of the accumulator (decimal 12). In terms of 2's-complement arithmetic, the foregoing looks like

$$\begin{array}{r} 0000\ 1100 \\ +1110\ 1110 \\ \hline 0\ 1111\ 1010 \end{array}$$

In this case, *CARRY* is 0 and *CY* is 1. The final register and flag contents are

$$CY = 1 \qquad \text{and} \qquad \mathbf{A} = 1111\ 1010$$

SBB Instructions

SBB stands for *subtract with borrow*. This instruction goes one step further than the SUB. It subtracts the contents of a specified register and the CY flag from the accumulator contents. If

$$\mathbf{A} = 1111\ 1111$$
$$\mathbf{E} = 0000\ 0010$$

and $CY = 1$

the instruction SBB E starts by combining **E** and *CY* to get 0000 0011 and then subtracts this from the accumulator as follows:

$$\begin{array}{r} 1111\ 1111 \\ -\ 0000\ 0011 \\ \hline 1111\ 1100 \end{array}$$

The final contents are

$$CY = 0 \quad \text{and} \quad \mathbf{A} = 1111\ 1100$$

EXAMPLE 12-1

In unsigned binary, 8 bits can represent 0 to 255, whereas 16 bits can represent 0 to 65,535. Show a SAP-3 program that adds 700 and 900, with the final answer stored in the H and L registers.

SOLUTION

Double bytes can represent decimal 700 and 900 as follows:

$$700_{10} = 02BCH = 0000\ 0010\ 1011\ 1100_2$$
$$900_{10} = 0384H = 0000\ 0011\ 1000\ 0100_2$$

Here is how to add 700 and 900:

Label	Instruction	Comment
	MVI A,00H	;Clear the accumulator
	MVI B,02H	;Store upper byte (UB) of 700
	MVI C,BCH	;Store lower byte (LB) of 700
	MVI D,03H	;Store UB of 900
	MVI E,84H	;Store LB of 900
	ADD C	;Add LB of 700
	ADD E	;Add LB of 900
	MOV L,A	;Store partial sum
	MVI A,00H	;Clear the accumulator
	ADC B	;Add UB of 700 with carry
	ADD D	;Add UB of 900
	MOV H,A	;Store partial sum
	HLT	;Stop

The first five instructions initialize registers A through E. The ADD C and ADD E add the lower bytes BCH and 84H; this addition sets the carry flag because

$$\begin{array}{rr} \text{BCH} = & 1011\ 1100_2 \\ +\ 84\text{H} = & 1000\ 0100_2 \\ \hline 1\ 40\text{H} = & 1\ 0100\ 0000_2 \end{array}$$

The sum is stored in the L register and the final carry in the CY flag.

Next, the accumulator is cleared. The ADC B adds the upper byte plus the carry flag to get

$$\begin{array}{rr} 00\text{H} = & 0000\ 0000_2 \\ +\ 02\text{H} = & 0000\ 0010_2 \\ +\ 1\text{H} = & 1_2 \\ \hline 03\text{H} = & 0000\ 0011_2 \end{array}$$

Then the ADD D produces

$$\begin{array}{rr} 03\text{H} = & 0000\ 0011_2 \\ +\ 03\text{H} = & 0000\ 0011_2 \\ \hline 06\text{H} = & 0000\ 0110_2 \end{array}$$

The MOV H,A stores this upper sum in the H register.

So the program ends with the answer stored in the H and L registers as follows:

$$\mathbf{H} = 06\text{H} = 0000\ 0110_2$$
and
$$\mathbf{L} = 40\text{H} = 0100\ 0000_2$$

The complete answer is 0640H, which is equivalent to decimal 1,600.

12-4 INCREMENTS, DECREMENTS, AND ROTATES

This section is about increment, decrement, and rotate instructions. The increment and decrement are similar to those of SAP-2, but the rotates are different because of the carry flag.

Increment

The increment instruction appears as

INR reg

where reg = A, B, C, D, E, H, or L. It works as previously described. Therefore, given

$$\mathbf{L} = 0000\ 1111$$

the execution of INR L produces

$$\mathbf{L} = 0001\ 0000$$

The INR instruction has no effect on the carry flag, but, as before, it does affect the sign and zero flags. For instance, if

$$\mathbf{B} = 1111\ 1111$$

and the initial flags are

$$S = 1 \quad Z = 0 \quad CY = 0$$

then INR B produces

$$\mathbf{B} = 0000\ 0000$$
$$S = 0 \quad Z = 1 \quad CY = 0$$

As you see, the carry flag is unaffected even though the B register overflowed. At the same time, the zero flag has been set and the sign flag reset.

Decrement

The decrement is similar. It looks like

DCR reg

where reg = A, B, C, D, E, H, or L. If

E = 0111 0110

then a DCR E produces

E = 0111 0101

The DCR affects the sign and zero flags but not the carry flag. This is why the initial values may be

E = 0000 0000
$S = 0$ $Z = 1$ $CY = 0$

Executing a DCR E results in

E = 1111 1111
$S = 1$ $Z = 0$ $CY = 0$

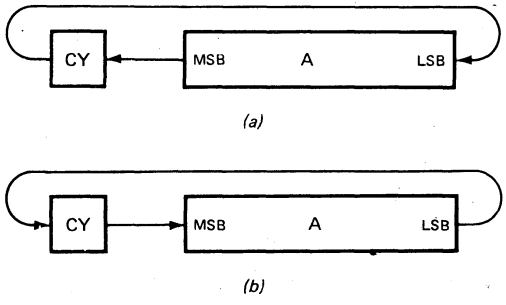


Fig. 12-3 (*a*) RAL; (*b*) RAR.

Rotate All Left

Figure 12-3*a* illustrates the RAL instruction used in SAP-3. The CY flag is included in the rotation of bits. RAL stands for rotate all left, which is a reminder that all bits including the CY flag are rotated to the left.

If the initial values are

$CY = 1$ **A** = 0111 0100

then executing a RAL instruction produces

$CY = 0$ **A** = 1110 1001

As you see, the original CY goes to the LSB position, and the original MSB goes to the CY flag.

Rotate All Right

The rotate-all-right instruction (RAR) rotates all bits including the CY flag to the right, as shown in Fig. 12-3*b*. If

$CY = 1$ **A** = 0111 0100

an RAR will result in

$CY = 0$ **A** = 1011 1010

This time, the original *CY* goes to the MSB position, and the original LSB goes into the CY flag.

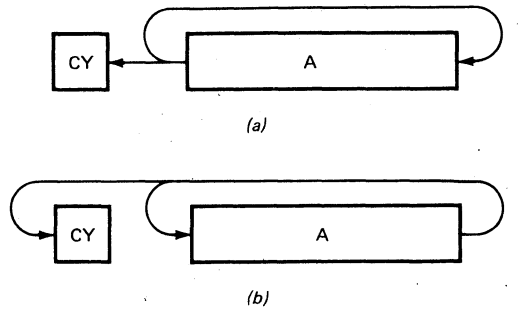


Fig. 12-4 (*a*) RLC; (*b*) RRC.

Rotate Left with Carry

Sometimes you don't want to treat the CY flag as an extension of the accumulator. In other words, you may not want to rotate all bits. Figure 12-4*a* illustrates the RLC instruction. The accumulator bits are rotated left, and the MSB is saved in the CY flag. For instance, given

$CY = 1$ **A** = 0111 0100

executing an RLC produces

$CY = 0$ **A** = 1110 1000

Rotate Right with Carry

Figure 12-4*b* shows how the RRC instruction rotates the bits. In this case, the accumulator bits are rotated right and the LSB is saved in the CY flag. So, given

$CY = 1$ **A** = 0111 0100

an RRC will result in

$CY = 0$ **A** = 0011 1010

Multiply and Divide by 2

Example 11-14 showed a program where the RAR instruction was used in converting from parallel to serial data. Parallel-to-serial conversion, and vice versa, is one of the main uses of rotate instructions.

There is another use for rotate instructions. Rotating has the effect of multiplying or dividing the accumulator contents by a factor of 2. Specifically, with the carry flag reset, an RAL has the effect of multiplying by 2, while the RAR divides by 2. This can be proved algebraically, but it's much easier to examine a few specific examples to see how it works.

Suppose

$CY = 0$ **A** = 0000 0111

Then an RAL produces

$CY = 0$ **A** = 0000 1110

The accumulator contents have changed from decimal 7 to decimal 14. The RAL has multiplied by 2.

Likewise, if

$CY = 0$ **A** = 0010 0001

then an RAL results in

$CY = 0$ **A** = 0100 0010

In this case, **A** has changed from decimal 33 to 66.

RAR instructions have the opposite effect; they divide by 2. If

$CY = 0$ **A** = 0001 1000

an RAR gives

$CY = 0$ **A** = 0000 1100

The decimal contents of the accumulator have changed from decimal 24 to 12.

Remember the basic idea. RAL instructions have the effect of multiplying by 2; RAR instructions divide by 2.

12-5 LOGIC INSTRUCTIONS

The SAP-3 logic instructions are almost the same as in SAP-2. For instance, three of the logic instructions are

ANA reg
ORA reg
XRA reg

where reg = A, B, C, D, E, H, or L. These instructions will AND, OR, or XOR the contents of the specified register with the contents of the accumulator on a bit-by-bit basis.

The only new logic instruction is the CMP, formatted as

CMP reg

where reg = A, B, C, D, E, H, or L. CMP compares the contents of the specified register with the contents of the accumulator. The zero flag indicates the outcome of this comparison as follows:

$$Z = \begin{cases} 1 & \text{if A = reg} \\ 0 & \text{if A} \neq \text{reg} \end{cases}$$

SAP-3 carries out a CMP as follows. The contents of the accumulator are copied in a temporary register. Then the contents of the specified register are subtracted from the contents of the temporary register. Since the ALU does the subtraction, the zero flag is affected. If the 2 bytes being compared are equal, the zero flag is set. If the bytes are unequal, the zero flag is reset. Because the temporary register is used, the accumulator contents are not changed by a CMP instruction.

For example, if

A = F8H
D = F8H
and $Z = 0$

executing a CMP D results in

A = F8H
D = F8H
and $Z = 1$

CMP has no effect on **A** and **D**; only the flag changes to indicate that **A** and **D** are equal. (If they were not equal, Z would be 0.)

CMP is a powerful instruction because it allows us to compare the accumulator contents with the data in a specified register. By following a CMP with a conditional zero jump, we can control loops in a new way. Later programs will show how this is done.

12-6 ARITHMETIC AND LOGIC IMMEDIATES

So far, we have introduced these arithmetic and logic instructions: ADD, ADC, SUB, SBB, ANA, ORA, XRA, and CMP. Each of these has the accumulator as an implied register; the data comes from a specified register (A, B, C, D, E, H, or L).

The immediate instructions from SAP-2 that carry over to SAP-3 are ANI, ORI, and XRI. As you know, each of these has the format of

ANI byte
ORI byte
XRI byte

where the immediate byte is ANDed, ORed, or XORed with the accumulator byte.

Besides the foregoing, SAP-3 has these immediate instructions:

ADI byte
ACI byte
SUI byte
SBI byte
CPI byte

The ADI adds the immediate byte to the accumulator byte. The ACI adds the immediate byte plus the CY flag to the accumulator byte. The SUI subtracts the immediate byte from the accumulator byte. The SBI subtracts immediate byte and the CY flag from the accumulator byte. The CPI compares the immediate byte with the accumulator byte; if the bytes are equal, the zero flag is set; if not, it is reset.

EXAMPLE 12-2

Show a program that subtracts 700 from 900 and stores the answer in the H and L registers.

SOLUTION

We need double bytes to represent 900 and 700 as follows:

900_{10} = 0384H = 0000 0011 1000 0100_2
700_{10} = 02BCH = 0000 0010 1011 1100_2

Here's the program for subtracting 700 from 900:

```
Label  Instruction      Comment
       MVI A, 84H   ;Load LB of 900
       SUI BCH      ;Subtract LB of 700
       MOV L,A      ;Save lower half answer
       MVI A, 03H   ;Load UB of 900
       SBI 02H      ;Subtract UB of 700 with borrow
       MOV H,A      ;Save upper half answer
```

The first two instructions subtract the lower bytes as follows:

```
  1000 0100
- 1011 1100
-----------
1 1100 1000
```

At this point,

$CY = 1$ **A** = C8H

The high *CY* flag indicates a borrow.

After saving C8H in the L register, the program loads the upper byte of 900 into the accumulator. The SBI is used instead of a SUI because of the borrow that occurred when subtracting the bytes. The execution of the SBI gives

```
  0000 0011
- 0000 0010
-          1
-----------
  0000 0000
```

This part of the answer is stored in the H register, so that the final contents are

H = 00H = $0000\,0000_2$
L = C8H = $1100\,1000_2$

12-7 JUMP INSTRUCTIONS

Here are the SAP-2 jump instructions that become part of the SAP-3 instruction set:

JMP address	(Unconditional jump)
JM address	(Jump if minus)
JZ address	(Jump if zero)
JNZ address	(Jump if not zero)

Here are some more SAP-3 jump instructions.

JP

JM stands for *jump if minus*. When the program encounters a JM address, it will jump to the specified address if the sign flag is set.

The JP instruction has the opposite effect. JP stands for *jump if positive* (including zero). This means that

JP address

produces a jump to the specified address if the sign flag is reset.

JC and JNC

The instruction

JC address

means to jump to the specified address if the carry flag is set. In short, JC stands for jump if carry. Similarly,

JNC address

means to jump to the specified address if the carry flag is not set. That is, jump if no carry.

Here is a program segment to illustrate JC and JNC:

```
Label      Instruction        Comment
           MVI A,FEH
REPEAT:    ADI 01H
           JNC REPEAT
           MVI A,C4H
           JC ESCAPE
             .
             .
             .
ESCAPE:    MOV L,A
```

The MVI loads the accumulator with FEH. The ADI adds 1 to get FFH. Since no carry takes place, the JNC takes the program back to the REPEAT point, where a second ADI is executed. This time the accumulator overflows to get contents of 00H with a carry. Since the CY flag is set, the program falls through the JNC. The accumulator is loaded with C4H. Then the JC produces a jump to the ESCAPE point, where the C4H is loaded into the L register.

JPE and JPO

Besides the sign, zero, and carry flag, SAP-3 has a *parity flag* designated P. During the execution of certain instructions (like ADD, INR, etc.), the ALU result is checked for parity. If the result has an even number of 1s, the parity flag is set; if an odd number of 1s, the flag is reset.

The instruction

JPE address

produces a jump to the specified address when the parity flag is set (even parity). On the other hand,

JPO address

results in a jump when the parity flag is reset (odd parity). For instance, given these flags,

$$S = 1 \quad Z = 0 \quad CY = 0 \quad P = 1$$

the program would jump if it encountered a JPE instruction; but it would fall through a JPO instruction.

Incidentally, we now have discussed all the flags in the SAP-3 computer. For upward compatibility with the 8085 microprocessor, these flags are stored in the F register, as shown in Fig. 12-5. For instance, if the contents of the F register are

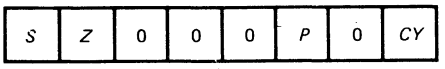


Fig. 12-5 F register stores flags.

$$\mathbf{F} = 0100\ 0101$$

then we know that the flags are

$$S = 0 \quad Z = 1 \quad P = 1 \quad CY = 1$$

EXAMPLE 12-3

What does the following program segment do?

SOLUTION

```
Label      Instruction        Comment
           MVI E,00H          ;Initialize counter
LOOP:      INR E              ;Increment counter
           MOV A,E            ;Load A with E
           CPI FFH            ;Compare to 255
           JNZ LOOP           ;Go back if not 255
```

The E register is being used as a counter. It starts at 0. The first time the INR and MOV are executed

$$\mathbf{A} = 01H$$

After executing the CPI, the zero flag is 0 because 01H and FFH are unequal. The JNZ then forces the program to return to the LOOP point.

The looping will continue until the INR and MOV have been executed 255 times to get

$$\mathbf{A} = FFH$$

On this pass through the loop, the CPI sets the zero flag because the accumulator byte and the immediate byte are equal. With the zero flag set for the first time, the program falls through the JNZ instruction.

Do you see the point? The computer will loop 255 times before it falls through the JNZ. One use of this program segment is to set up a time delay. Another use is to insert additional instructions inside the loop as follows:

```
Label    Instruction     Comment
         MVI E,00H
LOOP:      .
           .
           .
         INR E
         MOV A,E
         CPI FFH
         JNZ LOOP
```

The instructions at the beginning of the loop (symbolized by dots) will be executed 255 times. If you want to change the number of passes through the loop, modify the CPI instruction as required.

12-8 EXTENDED-REGISTER INSTRUCTIONS

Some SAP-3 instructions use pairs of CPU registers to process 16-bit data. In other words, during the execution of certain instructions, the CPU registers are cascaded, as shown in Fig. 12-6. The pairing is always as shown: B with C, D with E, and H with L. What follows are the SAP-3 instructions that use *register pairs*. Throughout these instructions, you will notice the letter X, which stands for extended register, a reminder that register pairs are involved.

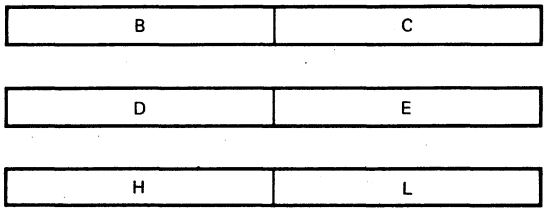


Fig. 12-6 Register pairs.

Load Extended Immediate

Since there are three register pairs (BC, DE, and HL), the LXI instruction can appear in any of these forms:

```
LXI B,dble
LXI D,dble
LXI H,dble
```

where B stands for BC
D stands for DE
H stands for HL
dble stands for double byte

The LXI instruction says to load the specified register pair with the double byte. For instance, if we execute

```
LXI B,90FFH
```

the B and C registers are loaded with the upper and lower bytes to get

B = 90H
C = FFH

Visualizing **B** and **C** paired off as shown in Fig. 12-6, we can write

BC = 90FFH

DAD Instructions

DAD stands for double-add. This instruction has three forms:

```
DAD B
DAD D
DAD H
```

where B stands for BC
D stands for DE
H stands for HL

The DAD instruction adds the contents of the specified register pair to the contents of the HL register pair; the result is then stored in the HL register pair. For instance, given

BC = F521H
HL = 0003H

the execution of a DAD B produces

HL = F524H

As you see, F521H and 0003H are added to get F524H. The result is stored in the HL register pair.

The DAD instruction affects the CY flag. If there is a carry out of the HL register pair, the CY flag is set; otherwise it is reset. As an example, if

DE = 0001H
HL = FFFFH

a DAD D will result in

HL = 0000H
CY = 1

Incidentally, a DAD H has the effect of adding the data in the HL register pair to itself. In other words, a DAD H doubles the value of HL. If

HL = 1234H

a DAD H results in

$$\mathbf{HL} = 2468\text{H}$$

INX and DCX

INX stands for *increment the extended register,* and DCX means *decrement the extended register.* The extended increment instructions are

```
INX B
INX D
INX H
```

where B stands for BC
D stands for DE
H stands for HL

The DCX instructions have a similar format: DCX B, DCX D, and DCX H.

The INX and DCX instructions have no effect on the flags. For instance, if

$$\mathbf{BC} = \text{FFFFH}$$
$$S = 1$$
$$Z = 0$$
$$P = 1$$
$$CY = 0$$

executing an INX B results in

$$\mathbf{BC} = 0000\text{H}$$
$$S = 1$$
$$Z = 0$$
$$P = 1$$
$$CY = 0$$

Notice that all flags are unaffected.

In summary, the extended register instructions are LXI, DAD, INX, and DCX. Of the three register pairs, the HL combination is special. The next section tells you why.

12-9 INDIRECT INSTRUCTIONS

As discussed in Chap. 10, the program counter is an *instruction pointer;* it points to the memory location where the next instruction is stored.

The HL register pair is different; it points to memory locations where data is stored. In other words, SAP-3 has several instructions where the HL register pair acts like a *data pointer.* The following discussion clarifies the idea.

Visualizing the HL Pointer

Figure 12-7*a* shows a 64K memory; it has 65,636 memory registers or memory locations where data is stored. The first memory location is M_{0000H}, the next is M_{0001H}, and so on. The memory location with address HL is M_{HL}.

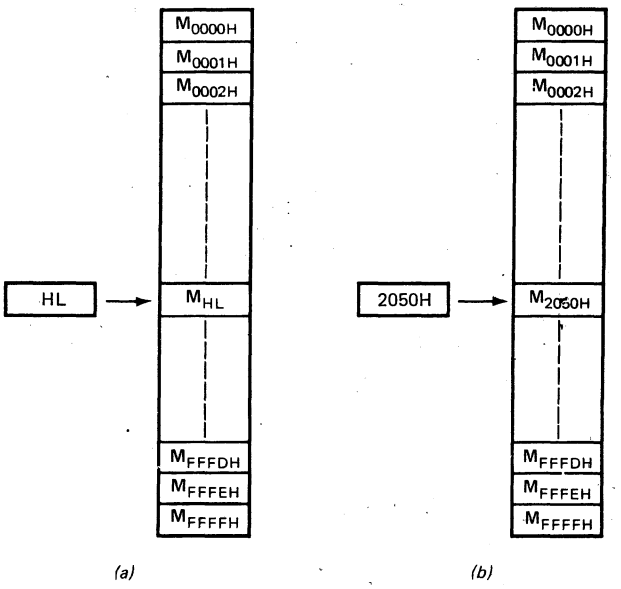


Fig. 12-7 (*a*) HL pointer; (*b*) pointing to 2050H.

With some SAP-3 instructions, the contents of the HL register pair are used as the address for data in memory. That is, the contents of the HL register pair are sent to the MAR, and then a memory read or write is performed. It's as though the HL register pair were pointing to the desired memory location, as shown in Fig. 12-7*a*.

For instance, suppose

$$\mathbf{HL} = 2050\text{H}$$

If HL is acting as a pointer, its contents (2050H) are sent to the MAR during one *T* state. During the next *T* state, the memory location whose address is 2050H undergoes a read or write operation. As shown in Fig. 12-7*b* the HL register pair points to the desired memory location.

Indirect Addressing

With direct addressing like LDA 5000H and STA 6000H, the programmer knows the address of the memory location because the instruction itself directly gives the address. With instructions that use the HL pointer, however, programmers do not know the address; all they know is that the address is stored in the HL register pair. Whenever an instruction uses the HL pointer, the addressing is called *indirect addressing.*

Indirect Read

One of the indirect instructions is

```
MOV reg,M
```

where reg = A, B, C, D, E, H, or L
M = M_{HL}

This instruction says to load the specified register with the data addressed by HL. After execution of this instruction, the designated register contains M_{HL}.

For instance, if

$$\mathbf{HL} = 3000H \quad \text{and} \quad \mathbf{M}_{3000H} = 87H$$

executing a

MOV C,M

produces

$$\mathbf{C} = 87H$$

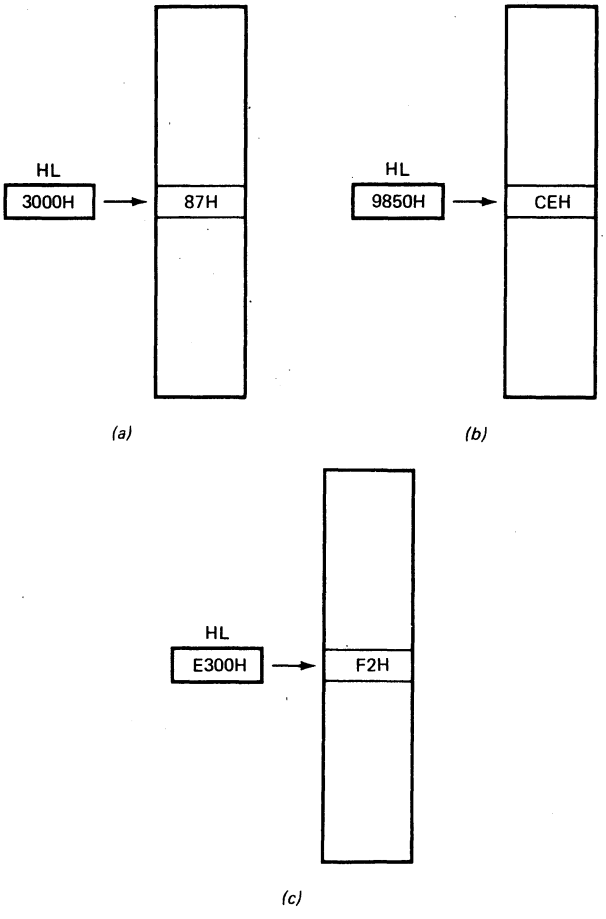


Fig. 12-8 Examples of indirect addressing.

Figure 12-8*a* shows how to visualize the MOV C,M. The HL pointer points to 87H, which is the data to be read into register C.

As another example, if

$$\mathbf{HL} = 9850H \quad \text{and} \quad \mathbf{M}_{9850H} = CEH$$

a MOV A,M results in

$$\mathbf{A} = CEH$$

Figure 12-8*b* illustrates the MOV A,M. The HL pointer points to CEH, which is the data to be loaded into the A register.

Indirect Write

Here is another indirect MOV instruction:

MOV M,reg

where M = M_{HL}
reg = A, B, C, D, E, H, or L

This says to load the memory location addressed by HL with the contents of the specified register. After execution of this instruction,

$$\mathbf{M}_{HL} = \text{reg}$$

As an example, if

$$\mathbf{HL} = E300H$$
$$\mathbf{B} = F2H$$

the execution of a MOV M,B produces

$$\mathbf{M}_{E300H} = F2H$$

Figure 12-8*c* illustrates the idea.

Indirect-Immediate Instructions

Sometimes we want to write immediate data into the memory location addressed by the HL pointer. The instruction to use in this case is

MVI M,byte

Here is an example. If **HL** = 3000H, executing a

MVI M,87H

produces

$$\mathbf{M}_{3000H} = 87H$$

Other Pointer Instructions

Here are more instructions using the HL pointer:

```
ADD M
ADC M
SUB M
SBB M
INR M
DCR M
ANA M
ORA M
XRA M
CMP M
```

In each of these, M is the memory location addressed by HL. Think of M as another register where data is stored. Each of the foregoing instructions operates on this data as previously described.

EXAMPLE 12-4

Suppose 256 bytes of data are stored in memory between addresses 2000H and 20FFH. Show a program that will copy these 256 bytes at addresses 3000H to 30FFH.

SOLUTION

```
Label   Instruction     Comment
        LXI H,1FFFH     ;Initialize pointer
LOOP:   INX H           ;Advance pointer
        MOV B,M         ;Read byte
        MOV A,H         ;Load 20H into accumulator
        ADI 10H         ;Add offset to get 30H
        MOV H,A         ;Offset pointer
        MOV M,B         ;Write byte in new location
        SUI 10H         ;Subtract offset
        MOV H,A         ;Restore H for next read
        MOV A,L         ;Prepare for compare
        CPI FFH         ;Check for 255
        JNZ LOOP        ;If not done, get next byte
        HLT             ;Stop
```

This looping program transfers each successive byte in the 2000H–20FFH area of memory into the 3000H–30FFH area of memory. Here are the details.

The LXI initializes the pointer with address 1FFFH. The first time into the loop, the INX will advance the HL pointer to 2000H. The MOV B,M then reads the first byte into the B register. The next three instructions

```
MOV A,H
ADI 10H
MOV H,A
```

offset the HL pointer to 3000H. Then the MOV M,B writes the first byte into location 3000H. The next two instructions, SUI and MOV, restore the HL pointer to 2000H. The MOV A,L puts 00H into the accumulator. Because the CPI FFH resets the zero flag, the JNZ forces the program to return to the LOOP entry point.

On the second pass through the loop, the computer will read the byte at 2001H and it will store this byte at 3001H. The looping will continue with successive bytes being moved from the 2000H–20FFH section of memory to the 3000H–30FFH area. Since the first byte is read from 2000H, the 256th byte is read from 20FFH. After this byte is stored at 30FFH, the pointer is restored to 20FFH. The MOV A,L then loads the accumulator to get

$$\mathbf{A} = \text{FFH}$$

This time, the CPI FFH will set the zero flag. Therefore, the program will fall through the JNZ to the HLT.

12-10 STACK INSTRUCTIONS

SAP-2 has a CALL instruction that sends the program to a subroutine. As you recall, before the jump takes place, the program counter is incremented and the address is saved at addresses FFFEH and FFFFH. The addresses FFFEH and FFFFH are set aside for the purpose of saving the return address. At the completion of a subroutine, the RET instruction loads the program counter with the return address, which allows the computer to get back to the main program.

The Stack

A *stack* is a portion of memory set aside primarily for saving return addresses. SAP-2 has a stack because addresses FFFEH and FFFFH are used exclusively for saving the return address of a subroutine call. Figure 12-9*a* shows how to visualize the SAP-2 stack.

SAP-3 is different. To begin with, the programmer decides where to locate the stack and how large to make it. As an example, Fig. 12-9*b* shows a stack between addresses 20E0H and 20FFH. This stack contains 32 memory locations for saving return addresses. Programmers can locate the stack anywhere they want in memory, but once they have set up the stack, they no longer use that portion of memory for program and data. Instead, the stack becomes a special space in memory, used for storing the return addresses of subroutine calls.

Stack Pointer

The instructions that read and write into the stack are called *stack instructions;* these include PUSH, POP, CALL, and

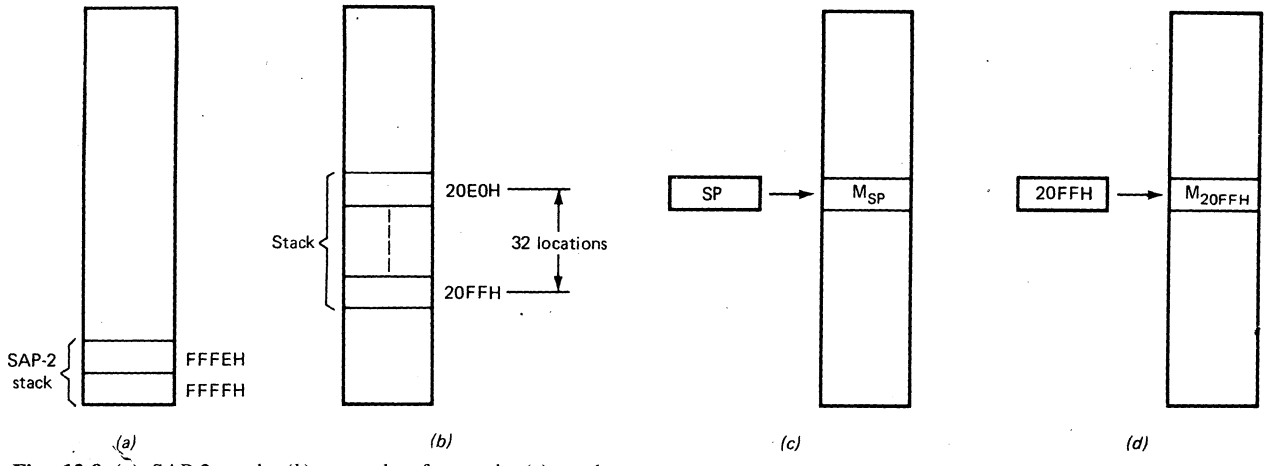


Fig. 12-9 (*a*) SAP-2 stack; (*b*) example of a stack; (*c*) stack pointer addresses the stack; (*d*) SP points to 20FFH.

others to be discussed. Stack instructions use indirect addressing because a 16-bit register called the *stack pointer* (SP) holds the address of the desired memory location. As shown in Fig. 12-9*c*, the stack pointer is similar to the HL pointer because the contents of the stack pointer indicate which memory location is to be accessed. For instance, if

$$\mathbf{SP} = 20FFH$$

the stack pointer points to memory location M_{20FFH} (see Fig. 12-9*d*). Depending on the stack instruction, a byte is then read from, or written into, this memory location.

To initialize the stack pointer, we can use the immediate load instruction

LXI SP,dble

For instance, if we execute

LXI SP,20FFH

the stack pointer is loaded with 20FFH.

PUSH Instructions

The contents of the accumulator and the flag register are known as the *program status word* (PSW). The format for this word is

$$\mathbf{PSW} = \mathbf{AF}$$

where **A** = contents of accumulator
F = contents of flag register

The accumulator contents are the high byte, and the flag contents the low byte. When calling subroutines, we usually have to save the program status word, so that the main program can resume after the subroutine is executed. We may also have to save the contents of the other registers.

PUSH instructions allow us to save data in a stack. Here are the four PUSH instructions:

PUSH B
PUSH D
PUSH H
PUSH PSW

where B stands for BC
D stands for DE
H stands for HL
PSW stands for program status word

When a PUSH instruction is executed, the following things happen:

1. The stack pointer is decremented to get a new value of SP − 1.
2. The high byte in the specified register pair is stored in M_{SP-1}.
3. The stack pointer is decremented again to get SP − 2.
4. The low byte in the specified register pair is stored in M_{SP-2}.

Here is an example. Suppose

BC = 5612H
SP = 2100H

When a PUSH B is executed,

1. The stack pointer is decremented to get 20FFH.
2. The high byte 56H is stored at 20FFH (Fig. 12-10*a*
3. The stack pointer is again decremented to get 20FE
4. The low byte 12H is stored at 20FEH (Fig. 12-10*b*

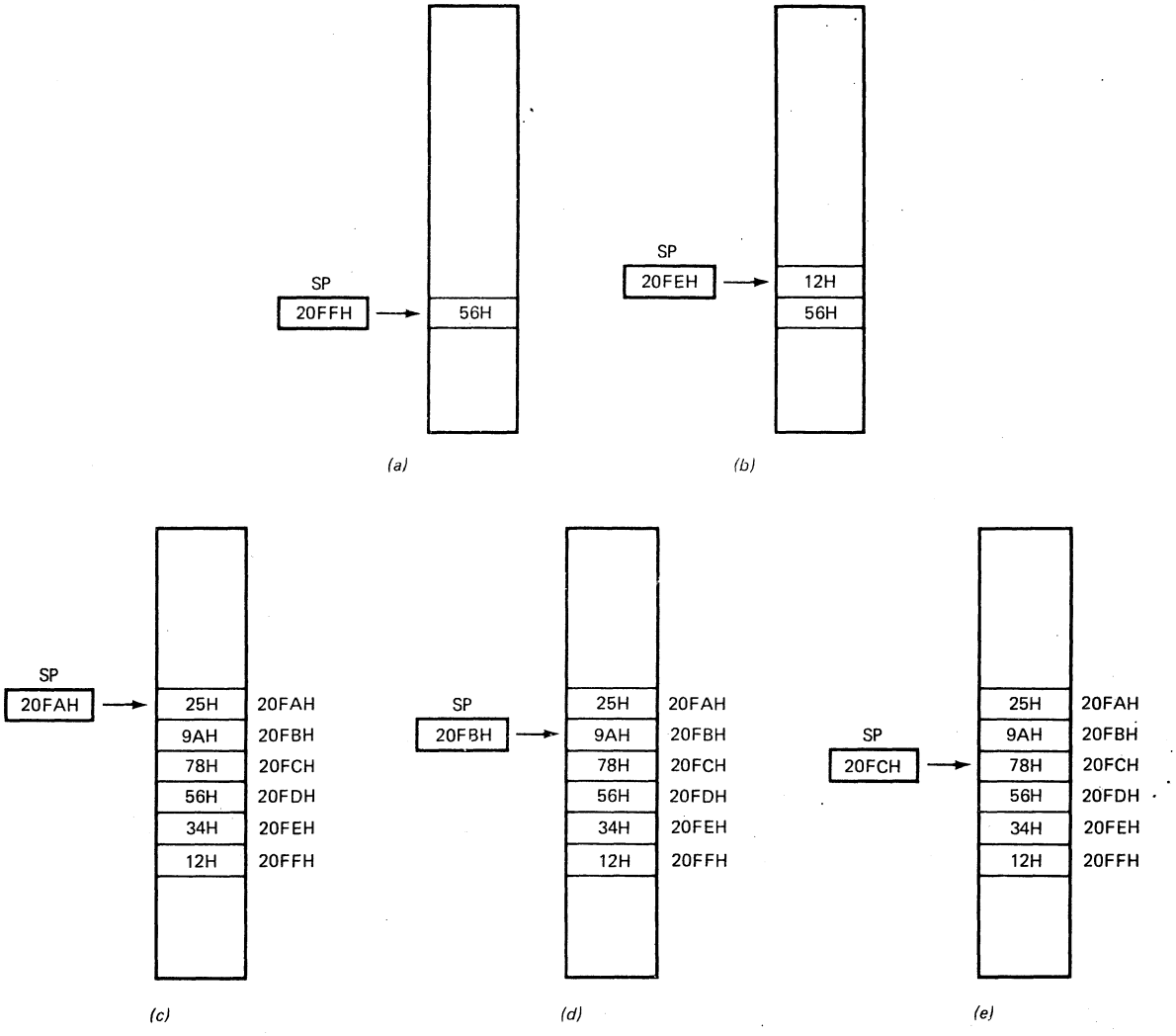


Fig. 12-10 Push operations: (*a*) high byte first; (*b*) low byte second; (*c*) 6 bytes pushed on stack; (*d*) popping a byte off the stack; (*e*) incrementing stack pointer.

Here's another example. Suppose

SP = 2100H
AF = 1234H
DE = 5678H
HL = 9A25H

then executing

```
PUSH PSW
PUSH D
PUSH H
```

loads the stack as shown in Fig. 12-10*c*. The first PUSH stores 12H at 20FFH and 34H at 20FEH. The next PUSH stores 56H at 20FDH and 78H at 20FCH. The last PUSH stores 9AH at 20FBH and 25H at 20FAH. Notice how the stack builds. Each new PUSH shoves data onto the stack.

POP Instructions

Here are four POP instructions:

```
POP B
POP D
POP H
POP PSW
```

where B stands for BC
D stands for DE
H stands for HL
PSW stands for program status word

When a POP is executed, the following happens:

1. The low byte is read from the memory location addressed by the stack pointer. This byte is stored in the lower half of the specified register pair.
2. The stack pointer is incremented.
3. The high byte is read and stored in the upper half of the specified register pair.
4. The stack pointer is incremented.

Here's an example. Suppose the stack is loaded as shown in Fig. 12-10*c* with the stack pointer at 20FAH. Then execution of POP B does the following:

1. Byte 25H is read from 20FAH (Fig. 12-10*c*) and stored in the C register.
2. The stack pointer is incremented to get 20FBH. Byte 9AH is read from 20FBH (Fig. 12-10*d*) and stored in the B register. The BC register pair now contains

$$\mathbf{BC} = 9A25H$$

3. The stack pointer is incremented to get 20FCH (Fig. 12-10*e*).

Each time we execute a POP, 2 bytes come off the stack. If we were to execute a POP PSW and a POP H in Fig. 12-10*e*, the final register contents would be

$$\mathbf{AF} = 5678H$$
$$\mathbf{HL} = 1234H$$

and the stack pointer would contain

$$\mathbf{SP} = 2100H$$

CALL and RET

The main purpose of the SAP-3 stack is to save return addresses automatically when using CALLs. When a

CALL address

is executed, the contents of the program counter are pushed onto the stack. Then the starting address of the subroutine is loaded into the program counter. In this way, the next instruction fetched is the first instruction of the subroutine. On completion of the subroutine, a RET instruction pops the return address off the stack into the program counter.

Here is an example:

Address	Instruction
2000H	LXI SP,2100H
2001H	
2002H	
2003H	CALL 8050H
2004H	
2005H	
2006H	MVI A,0EH
.	.
.	.
.	.
20FFH	HLT
.	
.	
.	
8050H	.
.	.
.	.
8059H	RET

To begin with, LXI and CALL instructions take 3 bytes each when assembled: 1 byte for the op code and 2 for the data. This is why the LXI instruction occupies 2000H to 2002H and the CALL occupies 2003H to 2005H.

The LXI loads the stack pointer with 2100H. During the execution of CALL 8050H, the address of the next instruction is saved in the stack. This address (2006H) is pushed onto the stack in the usual way; the stack pointer is decremented and the high byte 20H is stored; the stack pointer is decremented again, and the low byte 06H is stored (see Fig. 12-11*a*). The program counter is then loaded with 8050H, the starting address of the subroutine.

When the subroutine is completed, the RET instruction takes the computer back to the main program as follows. First, the low byte is popped from the stack into the lower half of the program counter; then the high byte is popped from the stack into the upper half of the program counter.

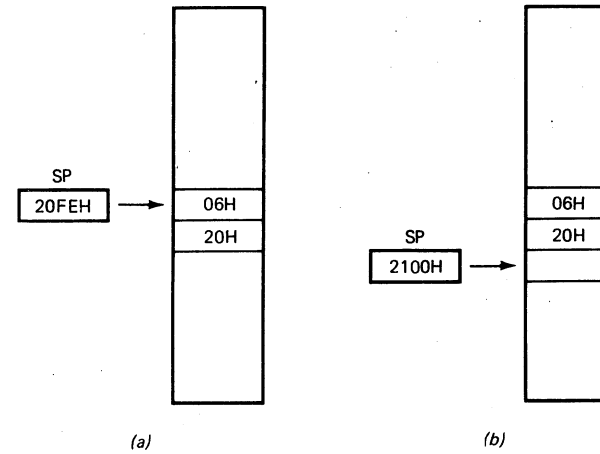


Fig. 12-11 (*a*) Saving a return address during a subroutine call; (*b*) popping the return address during a RET.

After the second increment, the stack pointer is back at 2100H, as shown in Fig. 12-11*b*.

The stack operation is automatic during CALL and RET instructions. All we have to do is initialize the setting of the stack pointer; this is purpose of the LXI SP,dble instruction. It sets the upper boundary of the stack. Then a CALL automatically pushes the return address onto the stack, and a RET automatically pops this return address off the stack.

Conditional Calls and Returns

Here is a list of the SAP-3 conditional calls:

```
CNZ address
CZ address
CNC address
CC address
CPO address
CPE address
CP address
CM address
```

They are similar to the conditional jumps discussed earlier. The CNZ branches to a subroutine only if the zero flag is reset, the CZ branches only if the zero flag is set, the CNC branches only if the carry flag is reset, and so forth.

The return from a subroutine may also be conditional. Here is a list of the conditional returns:

```
RNZ
RZ
RNC
RC
RPO
RPE
RP
RM
```

The RNZ will return only if the zero flag is reset, the RZ returns only when the zero flag is set, the RNC returns only if the carry flag is reset, and so on.

EXAMPLE 12-5

SAP-3 has a clock frequency of 1 MHz, the same as SAP-2. Write a program that provides a time delay of approximately 80 ms.

SOLUTION

```
Label   Mnemonic        Comment
        LXI SP,E000H    ;Initialize stack pointer
        MVI E,08H       ;Initialize counter
LOOP:   CALL F020H      ;Delay for 10 ms
        DCR E           ;Count down
        JNZ LOOP        ;Test for 8 passes
        HLT
```

You almost always use subroutines in complicated programs; this means that the stack will be used to save return addresses. For this reason, one of the first instructions in any program should be a LXI SP to initialize the stack pointer.

The 80-ms time delay program shown here starts with a LXI SP,E000H. This implies that the stack grows from address DFFFH toward lower memory. In other words, the stack pointer is decremented before the first push operation; this means that the stack begins at DFFFH.

The remainder of the program is straightforward. The E register is used as a counter. The program calls the 10-ms time delay 8 times. Therefore, the overall time delay is approximately 80 ms.

GLOSSARY

data pointer Another name for the HL register pair because some instructions use its contents to address the memory.

extended register A pair of CPU registers that act like a 16-bit register with certain instructions.

indirect addressing Addressing in which the address of data is contained in the HL register pair.

overflow A sum or difference that lies outside the normal range of the accumulator.

pop To read data from the stack.

push To save data in the stack.

stack A portion of memory reserved for return addresses and data.

stack pointer A 16-bit register that addresses the stack. The stack pointer must be initialized by an LXI instruction before calling subroutines.

SELF-TESTING REVIEW

Read each of the following and provide the missing words. Answers appear at the beginning of the next question.

1. An _____ is a sum or difference that lies outside the normal range of the accumulator. One way to detect an overflow is with the _____ flag.
2. (*overflow, carry*) To reset the carry flag, you may use an _____ followed by a CMC. STC stands for _____ the carry flag.
3. (*STC, set*) The ADC instruction adds the _____ flag and the contents of the specified register to the contents of the _____. SBB stands for subtract with _____.
4. (*carry, accumulator, borrow*) The RAL rotates all bits to the _____ with CY going to the LSB. RRC rotates the accumulator bits to the right with the LSB going to the carry flag.
5. (*left*) The CMP instruction compares the contents of the designated register with the contents of the accumulator. If the two are equal, the zero flag is _____. The CPI compares an immediate byte to the contents of the _____.
6. (*set, accumulator*) JM stands for jump if _____ . The program will branch to a new address if the _____ flag is set. JNZ means jump if not zero. With this instruction, the program branches only if the _____ flag is reset.
7. (*minus, sign, zero*) The LXI instruction is used to load register pairs. B is paired off with C, D with E, and H with _____. The HL register pair acts like a _____ pointer with some instructions. This type of addressing is called _____.
8. (*L, data, indirect*) The stack is a portion of memory reserved primarily for return addresses. The stack pointer is a 16-bit register that addresses the stack. It is necessary to initialize the stack pointer before calling any subroutines.

PROBLEMS

12-1. Write a program that adds decimal 345 and 753. (Use immediate bytes for the data.)

12-2. Write a program that subtracts decimal 456 from 983. (Use immediate data.)

12-3. Suppose that 1,024 bytes of data are stored between addresses 5000H and 53FFH. Write a program that copies these bytes at addresses 9000H to 93FFH.

12-4. Show a program that provides a delay of approximately 35 ms. If you use the SAP subroutines of Chap. 11, start your program with LXI SP,E000H.

12-5. Write a program that sends 1, 2, 3, . . . , 255 to port 22 with a time delay of 1 ms between OUT 22 instructions. (Use a LXI SP,E000H and a CALL F010H.)

12-6. Bytes arrive a port 21H at a rate of approximately 1 per millisecond. Write a program that inputs 256 bytes and stores them at addresses 8000H to 80FFH. (Use CALL F010H.)

12-7. Suppose that 512 bytes of data are stored at addresses 6000H to 61FFH and write a program that outputs these bytes to port 22H at a rate of approximately 100 bits per second. (Use CALL F020H.)

12-8. A peripheral device is sending serial data to bit 7 of port 21H at a rate of 1,000 bits per second. Write a program that converts any 8 bits in the serial data stream to an 8-bit parallel word, which is then sent to port 22H. (Use CALL F010H.)

12-9. Suppose that 256 bytes are stored at addresses 5000H to 50FFH and write a program that converts each of these bytes into a serial data stream at bit 0 of port 22H. Output the data at a rate of approximately 1,000 bits per second. (Use CALL F010H.)

The 8085

13

Intel Corporation introduced the 8080 in 1973. This 8-bit microprocessor set off the microcomputer explosion now taking place throughout the world. Although it was the most popular microprocessor of the early 70s, it had several disadvantages, such as needing two power supplies plus externally generated clock and control signals. In other words, the 8080 is not a CPU on a chip because the clock and controller are on separate chips.

Intel's 8085 is a 40-pin chip that is an enhanced version of the 8080. The 8085 has almost the same instruction set as the 8080, but it needs only one power supply (+5 V). Furthermore, the 8085 includes its own on-chip clock and control circuits. This means that the 8085 is truly a CPU on a chip. It is an ideal microprocessor to study because its principles are used in more advanced microprocessors.

Since the 8085 is only a CPU, it is necessary to connect memory and I/O chips to get a microcomputer or microprocessor-based system. As you will see later in this chapter, a minimum system can be built with three chips. One of these chips is the 8085.

The complete chip number is 8085A. A faster version of this basic chip is the 8085A-2. For simplicity, we will use the designation 8085 for either chip. Only when discussing specific differences will the complete numbers 8085A and 8085A-2 be used.

13-1 BLOCK DIAGRAM

Figure 13-1 shows the block diagram of the 8085. Refer to this block diagram throughout the following discussion. The drawing does not include the control signals driving each register. As you know, three-state registers need load and enable signals to communicate properly along a common bus. Therefore, even though they are not shown, control signals drive all the internal registers in Fig. 13-1.

Address, Data, and Control Buses

Near the top of the drawing is an 8-bit *internal data bus*. This carries instructions and data between the CPU registers.

The external buses are the ones we have to connect to other chips like memory, I/O, and so forth. Near the bottom left of the drawing is the external control bus ($\overline{RD}$, $\overline{WR}$, *ALE*, . . .). On the bottom right are the external *address* and *address-data* buses.

The upper 8 address bits are on a separate bus always used for address bits; this upper section of the address bus is designated A_{15}–A_8. The lower 8 bits are *multiplexed*. This means that the eight lower bus lines are used for address bits during some *T* states and for data bits during other *T* states. This is why the bus is labeled address-data bus, designated AD_7–AD_0. (Recall that SAP-1 and SAP-2 had a W bus; it was a multiplexed address-data bus.)

Why is multiplexing used in the 8085? Because at the time this chip was developed, the practical limit on the number of pins was 40. The only solution was to multiplex part of the address bus with the data bus.

Accumulator

The accumulator is connected to the 8-bit internal data bus. The bidirectional arrow between the accumulator and the bus indicates a three-state connection that allows the accumulator to send or receive data. The two-state output of the accumulator drives the ALU.

Temporary Register

The other input for the ALU comes from the *temporary register* (Temp. Reg. in Fig. 13-1). This 8-bit register stores the operands of arithmetic-logic operations. For instance, during an ADD C the contents of the C register are copied in the temporary register during one *T* state and added during another *T* state.

ALU and Flags

The ALU carries out the arithmetic and logic operations. As shown, the contents of the accumulator and the temporary register are the inputs to the ALU. The ALU result then is stored back in the accumulator.

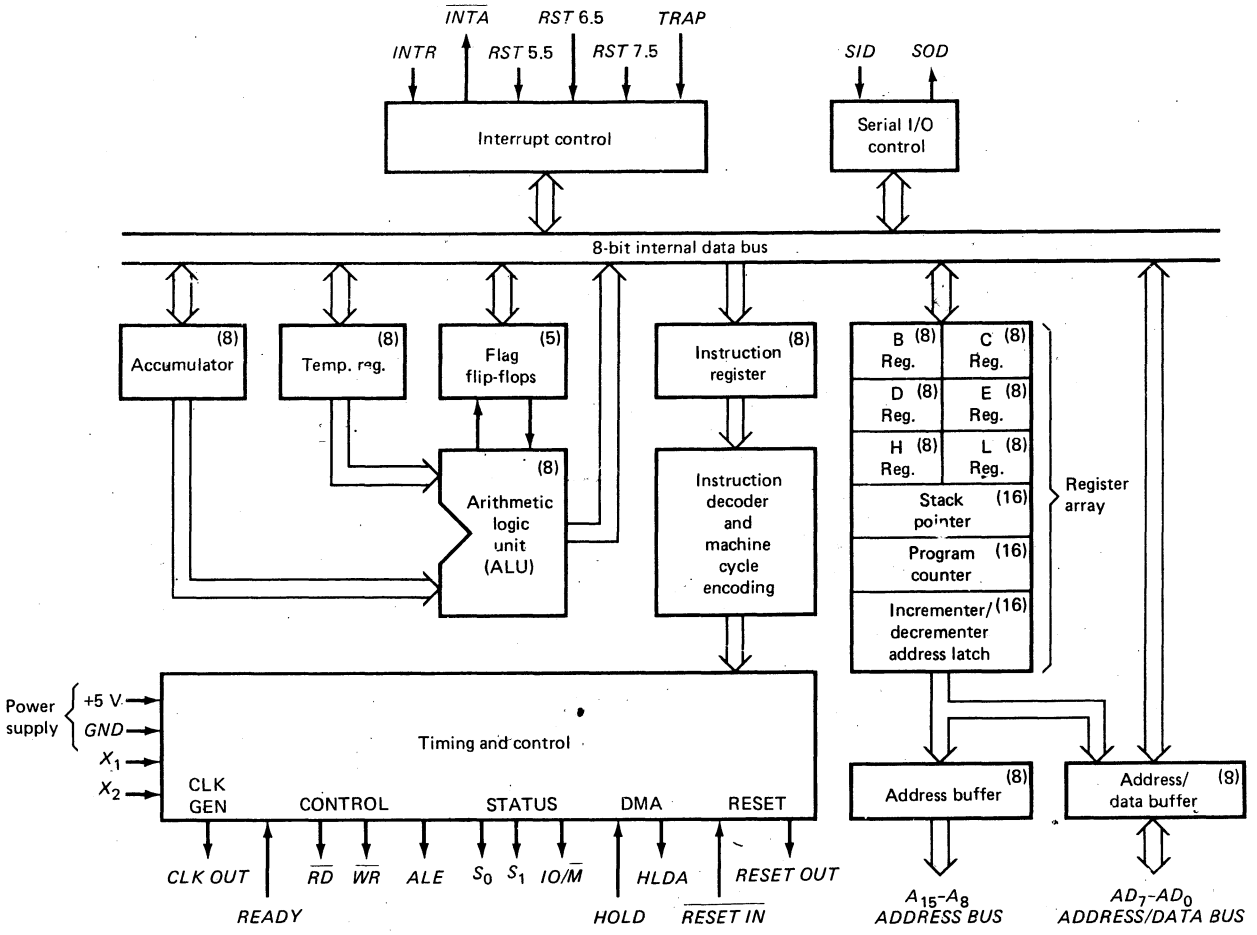


Fig. 13-1 8085 block diagram. (*Intel Corporation.*)

You already know about four flags: zero, sign, carry, and parity. The 8085 includes a fifth flag, called the *auxiliary carry flag*. It is used in BCD operations to be described later.

Instruction Register and Decoder

During the fetch cycle, the op code of an instruction is stored in the instruction register. This op code then drives the instruction decoder and machine-cycle encoder.

Timing and Control

The *timing and control section* of Fig. 13-1 includes an oscillator and a controller-sequencer. The oscillator generates the two-phase clock signals (*CLK* and $\overline{CLK}$) that synchronize all registers. The controller-sequencer also produces the control signals needed for internal and external control.

The controller-sequencer is microprogrammed; it has a ROM that stores all the microroutines needed for executing the instructions. After each instruction is fetched and stored in the instruction register, the op code is decoded to get the starting address of the desired microroutine. As each microinstruction is read out of the control ROM, control signals are sent to the internal and external data buses. The effect is to move data between registers, to perform arithmetic-logic operations, to input or output data, etc. The control ROM is sometimes called the *control store*.

CPU Registers

Notice the array of CPU registers (B, C, D, etc.). This register array is like a small on-chip RAM with addressable memory locations. Control signals select the register for a read or write operation. This means that the CPU can either load a register from the 8-bit internal data bus or output the register contents to this data bus.

Included in the register array are the stack pointer, program counter, and *incrementer-decrementer*. The incrementer-decrementer can add 1 or subtract 1 from the contents of the stack pointer or program counter.

Address Buffer and Address-Data Buffer

At the bottom right are two buffer registers called the *address buffer* and the *address-data buffer*. The contents of the stack pointer or program counter can be loaded into the address buffer and address-data buffer. The output of these buffers then drives the external address bus and address-data bus. Memory and I/O chips (not shown) are connected to these buses. In this way, the CPU can send the address of desired data to the memory or I/O chips.

The 8-bit internal data bus is also connected to the address-data buffer. The bidirectional arrow indicates a three-state connection that allows the address-data buffer to send or receive data from the 8-bit internal data bus.

Interrupt Control

Sometimes it is necessary to *interrupt* the execution of the main program to answer a request from an I/O device. For instance, an I/O device may send an interrupt signal to the interrupt control unit (top left of Fig. 13-1) to indicate that data is ready for input. The computer temporarily stops what it is doing, inputs the data, then returns to what it was doing.

The interrupt concept is analogous to your reading a book (main program), hearing the phone (interrupt), answering the phone (servicing the interrupt), then returning to your reading (main program).

Chapter 14 is about interrupts, how the 8085 handles them, what program instructions are used, and so on.

Serial I/O Control

Sometimes, I/O devices work with serial data rather than parallel. In this case, the serial data stream from an input device must be converted to 8-bit parallel data before the computer can use it. Likewise, the 8-bit data out of a computer must be converted to serial form before a serial output device can use it.

The *SID input* at the upper right of Fig. 13-1 is where serial input data enters the 8085. The *SOD output* is where the serial data leaves the 8085. Two new instructions known as SIM and RIM allow us to perform the serial-parallel conversions needed for serial I/O devices. More is said about these instructions in Chap. 14.

13-2 PINOUT DIAGRAM

Figure 13-2 is the pinout diagram for the 8085. To use the 8085 in a microprocessor-based system you need a general idea of what each pin does. What follows is a brief description of each pin.

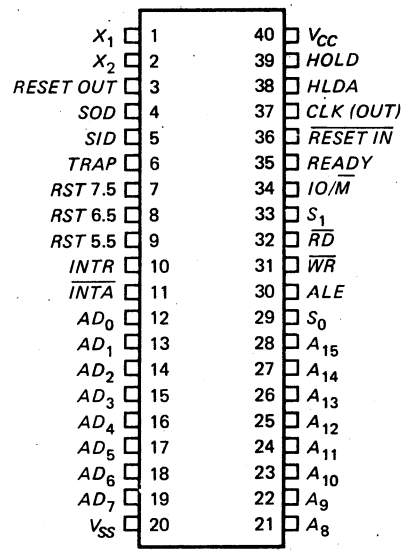


Fig. 13-2 8085 pinout.

Pins 1 and 2

Oscillators are crystal-, *LC*-, or *RC*-controlled. The 8085 has an on-chip oscillator with all the required circuitry except for the crystal, *LC* tank, or *RC* network that controls the frequency. This is the purpose of pins 1 and 2; you connect a crystal, *LC* circuit, or *RC* network to X_1 and X_2. In typical microcomputers a crystal is used for its frequency stability.

Pin 3

This pin carries the *RESET OUT* signal. When high, it indicates that the CPU is being reset; that is, the program counter, instruction register, and so on, are being reset to zero. The *RESET OUT* signal goes to peripheral chips. When you first power up, the whole system including the 8085 and peripheral chips is reset or initialized. After the *RESET OUT* goes low, the processing begins.

Pins 4 and 5

SOD stands for *serial out data*. Later, you will see a program that converts accumulator data into a serial data stream. This serial data comes out of pin 4, which can be connected to a serial output device.

SID stands for *serial in data*. Pin 5 is the input pin for serial data. Later, you will see instructions that convert serial input data to 8-bit form.

Pins 6 to 11

These pins are part of the interrupt control unit. The 8085 has five inputs for interrupt requests (analogous to the phone

ringing, someone knocking, the bathtub overflowing, etc.). As will be discussed in Chap. 14, a priority exists among the interrupt pins; some are more important than others. In order of their importance the five interrupt signals are designated *TRAP*, *RST 7.5*, *RST 6.5*, *RST 5.5*, and *INTR*. If two or more interrupts go high at the same time, the 8085 will service them in order of their importance (*TRAP* first, *RST 7.5* second, and so on).

Pins 6 to 10 are input pins for the interrupt signals. Pin 11, however, is an output pin with a signal called the interrupt acknowledge ($\overline{INTA}$). This particular signal is used in response to an *INTR* interrupt. More about this in Chap. 14.

Pins 12 to 28

Pins 12 to 19 carry the lower 8 address bits or the 8 data bits. As mentioned earlier, the lower half of the address bus is multiplexed with the data bus to keep the pin count at 40. Pin 20, labeled V_{SS}, is the system ground. Pins 21 to 28 are the rest of the address bus.

Pins 29 and 33

Pins 29 and 33 carry output signals known as status signals. Labeled S_0 and S_1, these status signals (and the $IO/\overline{M}$ signal) indicate whether an instruction fetch, memory read, memory write, or other operation is taking place.

Pin 30

The 8085 is a microprocessor (sometimes abbreviated μP). To work properly, it needs one or more memory chips connected to it. Each memory chip has its own MAR, usually called an *address latch*. This latch stores the incoming address from the address bus and address-data bus.

At what point in the machine cycle does a memory chip store the incoming address? This is where the *ALE* signal comes in. *ALE* stands for *address latch enable*. The *ALE* signal comes out of pin 30 and goes to peripheral chips such as memory chips. The falling edge of the *ALE* signal *strobes* (loads) the address on the address bus and address-data bus into the MAR or address latch of the memory chips.

Pins 31, 32, and 34

These three pins function together. They are connected to memory and I/O chips. To begin with, pin 34 carries the $IO/\overline{M}$ signal. A low $IO/\overline{M}$ indicates a memory operation, and a high $IO/\overline{M}$ means that an I/O instruction is being executed. In other words, a low $IO/\overline{M}$ signal enables the memory chips, and a high $IO/\overline{M}$ enables the I/O chips.

The $\overline{WR}$ and $\overline{RD}$ determine whether a write or a read is done. Since these signals are active low, a low $\overline{WR}$ means a write operation and a low $\overline{RD}$ means a read operation. (They are never both low at the same time.)

As an example, during a memory read, $IO/\overline{M}$ goes low, $\overline{WR}$ goes high, and $\overline{RD}$ goes low. When an OUT instruction is executed, $IO/\overline{M}$ goes high, $\overline{WR}$ goes low, and $\overline{RD}$ goes high.

Pin 35

Some peripheral devices are slow; they are unable to run at the same speed at the 8085. One way to slow down the 8085 is with the *READY* signal (pin 35).

Here is the idea. The 8085 addresses a peripheral device as the first step in sending or receiving data from that device. If the device is not ready, it will return a low *READY* bit to the 8085. The 8085 then generates a number of nop *T* states (called WAIT states). Eventually, when the peripheral device is ready, it will send a high *READY* signal to the 8085. Then the 8085 can complete the data transfer. (The action is a form of handshaking, described in Chap. 11.)

Pins 36 and 37

Pin 36 is an input carrying the $\overline{RESET\ IN}$ signal. This signal may come from an operator reset button or other source. When $\overline{RESET\ IN}$ is low, the CPU will reset the program counter, instruction register, and other circuits. It also sends a high *RESET OUT* to pin 3, as previously described. The CPU remains in reset until the $\overline{RESET\ IN}$ signal goes high. Then the data processing begins.

The *CLK* signal out of pin 37 is derived from the on-chip oscillator. *CLK* is the system clock; each cycle represents one *T* state. The *CLK* signal goes to peripheral chips and synchronizes their timing.

Pins 38 to 40

The IN instruction is the usual way to input data from peripheral devices. The accumulator is involved because it receives the input data. Similarly, the OUT instruction transfers data from the accumulator to output devices. In either case, going through the accumulator slows down I/O transfers.

The solution to speeding up memory-peripheral transfers is called *direct memory access* (DMA). In this approach, the 8085 turns over control of the buses to a DMA *controller*, a chip optimized for high-speed memory transfers. The *HOLD* and *HLDA* signals (pins 39 and 38) are used in DMA operations. With the DMA approach, large amounts of data can be transferred in a short time. Chapter 14 discusses this in more detail.

Pin 40 is last pin. It connects to a source of +5 V. The tolerance on the supply voltage is ±5 percent. The power dissipation is less than 1.5 W.

13-3 DRIVING THE X_1 AND X_2 INPUTS

The data sheet of an 8085A specifies these limits on the clock frequency:

$$f_{min} = 500 \text{ kHz} \qquad f_{max} = 3.125 \text{ MHz}$$

This means that the manufacturer guarantees correct operation only when the clock frequency is greater than 500 kHz and less than 3.125 MHz. (The 8085A-2 is a bit faster: 500 kHz to 5 MHz.)

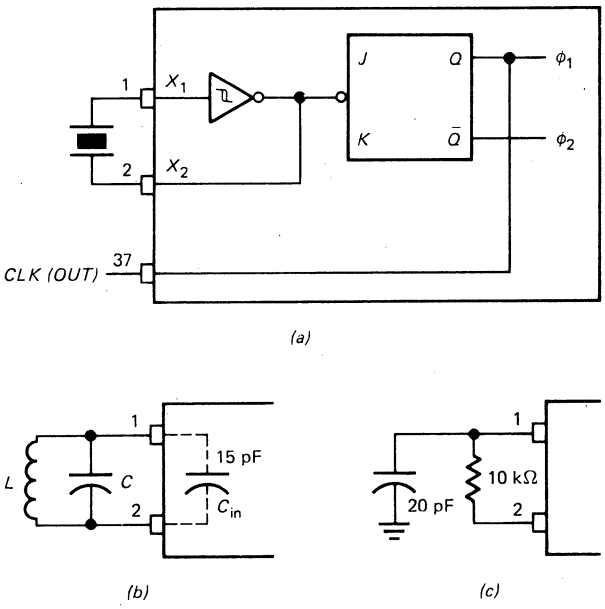


Fig. 13-3 (*a*) Crystal-driven; (*b*) *LC* tank; (*c*) *RC* network.

Divide-by-2

When a crystal is connected to the X_1 and X_2 inputs, as shown in Fig. 13-3*a*, it determines the frequency of the on-chip oscillator. The oscillator output from the Schmitt trigger drives a flip-flop that produces two clock signals, ϕ_1 and ϕ_2 (equivalent to *CLK* and $\overline{CLK}$). These signals drive the internal 8085 circuits. The *CLK* signal also goes to pin 37 to drive peripheral chips.

Because the flip-flop divides by 2, the clock frequency is half the driving frequency of the timing element. For instance, a 6-MHz crystal will produce a 3-MHz clock. In general, the frequency of the driving network connected to the X_1-X_2 inputs must be twice the desired clock frequency.

As mentioned earlier, the typical microcomputer uses a crystal because it provides a stable driving frequency. This ensures a fixed clock period, needed for accurate time delays.

LC Drive

You can use an *LC*-resonant tank instead of a crystal. This saves money and is all right in applications where a frequency tolerance of 10 percent is acceptable. Figure 13-3*b* shows how to connect the *LC* tank. The input capacitance C_{in} is approximately 15 pF, and the resonant frequency is given by

$$f = \frac{1}{2\pi\sqrt{L(C + C_{in})}}$$

The manufacturer recommends that *C* be greater than 30 pF to minimize frequency variations caused by C_{in} (it changes with temperature). Because of the divide-by-2 action described earlier, the clock frequency is half the resonant frequency.

RC Network

When an *RC* network is used to drive the X_1-X_2 terminals, the clock frequency has wide variations over the temperature and voltage range specified by the manufacturer. For this reason the manufacturer recommends the *RC* clock driver shown in Fig. 13-3*c*. This *RC* circuit results in driving frequency of approximately 3 MHz, which means a nominal clock frequency of 1.5 MHz. This way, the wide drifts in clock frequency will not exceed the 3.125-MHz limit of the 8085A.

13-4 NEW INSTRUCTIONS

The 8085 includes all the SAP-3 instructions. In addition, here are some new ones.

XCHG

The instruction

XCHG

will exchange the contents of the HL and DE register pairs. For instance, if

DE = 1234H
HL = 5678H

then execution of an XCHG results in

DE = 5678H
HL = 1234H

STAX

The STAX instruction has two forms:

STAX B
STAX D

where B stands for BC, and D for DE. This indirect instruction loads the contents of the accumulator into the memory location addressed by the specified register pair. In other words, the BC register pair or the DE register pair acts like a pointer.

As an example, if

A = FFH
BC = 2000H

execution of an STAX B will produce

$\mathbf{M}_{2000H}$ = FFH

In this case, the BC register pair acts like a pointer, as shown in Fig. 13-4*a*.

LDAX

LDAX is similar to STAX, except the addressed contents of memory are loaded into the accumulator. Either LDAX B or LDAX D may be used. If

DE = 3000H
$\mathbf{M}_{3000H}$ = 4CH

then execution of LDAX D results in

A = 4CH

This time, the DE register pair acts like a pointer (Fig. 13-4*b*).

LHLD

The format of this instruction is

LHLD address

This instruction will load the HL register pair with two successive bytes starting with the specified address. The low byte from the specified address goes into the L register; the high byte from the next address goes into the H register.

For instance, given

$\mathbf{M}_{1234H}$ = 00H and $\mathbf{M}_{1235H}$ = 50H

as shown in Fig. 13-4*c*, an LHLD 1234H produces

HL = 5000H

SHLD

This is similar to the LHLD, except that the 2 bytes in the HL register pair are stored at the specified address and the next higher address. If

HL = 47F9H

an SHLD 6200H results in

$\mathbf{M}_{6200H}$ = F9H and $\mathbf{M}_{6201H}$ = 47H

Figure 13-4*d* illustrates this example.

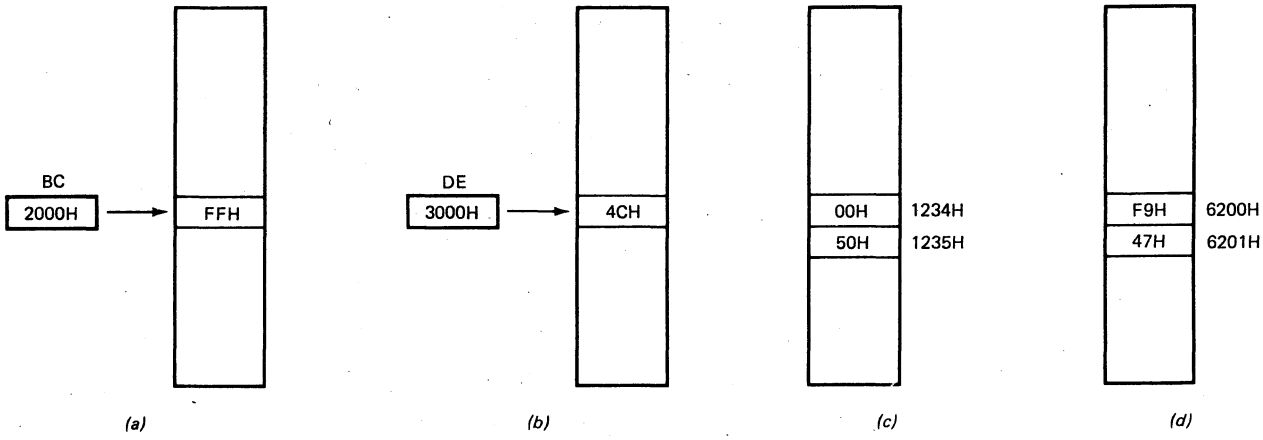
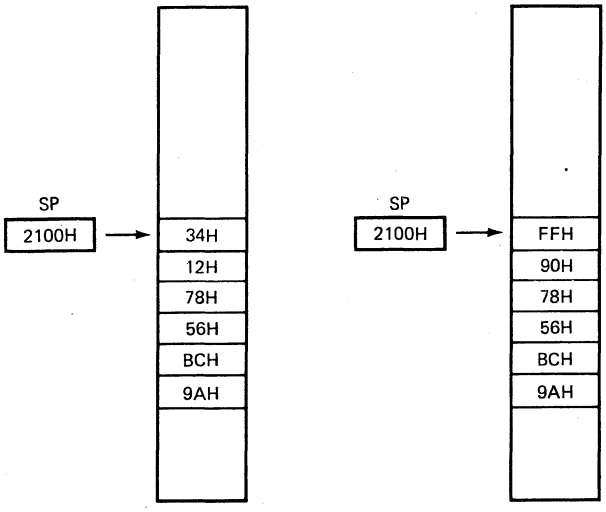


Fig. 13-4 (*a*) STAX; (*b*) LDAX; (*c*) LHLD; (*d*) SHLD.



(*a*) (*b*)

Fig. 13-5 XTHL.

PCHL

JMP is an unconditional jump instruction. Here is another unconditional jump:

PCHL

When this instruction is executed, the contents of the HL register pair are loaded into the program counter. For example, if

PC = 2000H and **HL** = 2090H

a PCHL results in

PC = 2090H

Therefore, the next instruction cycle will fetch the instruction at address 2090H.

SPHL

The SPHL instruction loads the contents of the HL register pair into the stack pointer. Given

SP = 2200H and **HL** = 2075H

execution of a SPHL produces

SP = 2075H

XTHL

The XTHL instruction exchanges the contents of the HL register pair with the 2 bytes at the top of the stack. Here is an example. Suppose the stack pointer is at 2100H, as shown in Fig. 13-5*a*. The 2 bytes at the top of the stack are 34H (low byte) and 12H (high byte). If

HL = 90FFH

executing an XTHL will result in

HL = 1234H

and the top of the stack will contain FFH and 90H, as shown in Fig. 13-5*b*. The XTHL preserves the contents of the stack pointer; in Fig. 13-5*b* the stack pointer contains 2100H after the XTHL is executed.

13-5 THE DAA INSTRUCTION

In some applications, BCD numbers are processed. Because BCD numbers are different from binary numbers, a problem arises when we try to add them. This section describes the problem and its solution.

Sums Greater than 9

Given decimal 539, each digit is encoded into its equivalent nibble:

```
 5   3   9
 ↓   ↓   ↓
0101 0011 1001
```

The largest possible nibble is 1001; combinations like 1010, 1011, 1100, 1101, 1110, and 1111 do not exist in the BCD code. In other words, nibbles 1010 to 1111 are forbidden in BCD processing. Because of this, we run into a problem when trying to add BCD numbers whose sum exceeds 9.

For instance, if we add 8 and 5 using binary addition, we get

```
  8     1000
+ 5   + 0101
 13     1101
```

The answer 1101 is okay in binary, but it is meaningless in BCD code. The BCD answer should be 0001 0011.

The problem arises because the BCD code uses only 10 of 16 possible nibbles. When the decimal sum of a BCD addition exceeds 9, we must somehow skip the six forbidden nibbles to restore the answer to BCD form.

Add 6 When Greater than 9

To skip forbidden nibbles, we can add 6 (0110) to the sum when it exceeds 9. For example, the BCD addition of 8 and 5 looks like this:

```
    1000
  + 0101
  ------
    1101      binary equivalent of 13
  + 0110      add 6 to return to BCD form
  ------
0001 0011     BCD equivalent of 13
```

After the first addition, the sum exceeds decimal 9. To correct this sum, we add 6.

As another example, here's the BCD addition of 9 and 8:

```
    1001
  + 1000
  ------
   1 0001     binary equivalent of 17
  + 0110      add 6 to return to BCD form
  ------
0001 0111     BCD equivalent of 17
```

The carry in this example is called an *auxiliary* carry AC to keep it distinct from the final carry CY discussed earlier. The accumulator contents are

$$\mathbf{A} = A_7A_6A_5A_4A_3A_2A_1A_0$$

In the 8085, the AC flag is set whenever there is a carry out of bit 3, whereas the CY flag is set whenever there is a carry out of bit 7.

Rules for BCD Addition

When the sum of two nibbles is 9 or less, the answer is right as is. When the sum of two nibbles is greater than 9, we have to add 0110 to return to BCD form. How do we know when the sum is greater than 9? Either the nibble will be a forbidden group (1010 through 1111), or there will be a carry to the next higher nibble.

Here are the rules for BCD addition:

1. If the sum of two nibbles is greater than 9, add 0110 to the sum to return to BCD form.
2. If the sum of two nibbles is 9 or less, leave it alone because it already is in BCD form.

Here's an example:

```
  98        1001 1000
+ 84      + 1000 0100
------    -----------
 182       1 0001 1100   binary sum
         + 0110 0110     correct nibbles for BCD form
           0001 1000 0010  answer in BCD form
```

The least significant nibbles produce a sum greater than 9; therefore, 0110 is added. The sum of the most significant nibbles also exceeds 9; so, 0110 must be added.

Here's another example:

```
  48        0100 1000
+ 39      + 0011 1001
------    -----------
  87        1000 0001   binary sum
          + 0000 0110   correct nibbles for BCD form
            1000 0111   answer in BCD form
```

In this case, the sum of the least significant nibbles exceeds 9; therefore, 0110 is added. The sum of the most significant nibbles is less than 9, so no correction is needed.

DAA

The ALU of the 8085 produces binary sums. If you are processing BCD numbers, you must use the DAA instruction to return these sums to BCD form (DAA stands for *decimal adjust accumulator*.) When a DAA is executed, the computer automatically checks for nibble sums greater than 9. If there are any, the computer adds 0110 as needed to return the answer to BCD form.

There are no instructions that allow you to test the auxiliary carry flag. The AC flag exists only to allow the DAA instruction to do its work. In other words, testing the AC flag is automatic, so you don't have to worry about whether it's set or not.

You should use the DAA instruction after any ALU operation involving BCD numbers. This will correct the accumulator contents before they are used by other instructions.

EXAMPLE 13-1

Here's a program segment:

```
MVI A,39H
ADI 97H
DAA
```

What does it do?

SOLUTION

The first two instructions represent

```
  0011 1001
+ 1001 0111
-----------
  1101 0000
```

At this point, the accumulator contents and flags are

$$A = 1101\ 0000$$
$$CY = 0$$
$$AC = 1$$

When the DAA is executed, the sum is corrected to get

$$\begin{array}{r} 1101\ 0000 \\ +\ 0110\ 0110 \\ \hline 1\ 0011\ 0110 \end{array}$$

Because of the final carry, the CY flag is set. Therefore, the accumulator contents and flags are

$$A = 0011\ 0110$$
$$CY = 1$$
$$AC = 1$$

13-6 THE MINIMUM SYSTEM

The 8085 is not a stand-alone device; you must connect memory and I/O chips to get a useful system. This section examines the *minimum system*, a connection of three chips that form a microprocessor-based system.

8085A

Figure 13-6 shows an 8085A connected to two other chips, the 8156 and the 8355. These chips are part of the MCS-85 family, a group of connectable chips that allow you to build systems.

The 8085A has an address bus (A_{15} to A_8) and an address-data bus (AD_7 to AD_0). During certain *T* states, the 16-bit contents of the program counter are placed on the address bus and address-data bus. During other *T* states, 8-bit data is placed on the address-data bus. When A_{13} is low, the 8156 is disabled.

ALE

In Fig. 13-6 notice the control signals coming out of the 8085A. *ALE*, the address latch enable, goes high at the beginning of each machine cycle. Midway through the first *T* state, *ALE* goes low. The falling edge of *ALE strobes* (loads) the address on the address bus and address-data bus into the memory chips.

CLK OUT, IO/$\overline{M}$, READY, and RESET OUT

CLK OUT is the system clock. The frequency is half the driving frequency of the crystal. Typically, a 6-MHz crystal is used, so that *CLK OUT* has a frequency of 3 MHz. The *IO/$\overline{M}$* signal is high during I/O operations and low during memory operations. The *READY* signal is an input to the 8085A; it indicates that a peripheral device is ready for a data transfer. The *RESET OUT* goes high when the 8085A is being reset or initialized.

$\overline{RD}$ and $\overline{WR}$

During I/O and memory operations, a low $\overline{RD}$ means that data will be read from an I/O device or memory chip. A low $\overline{WR}$, on the other hand, means that data will be sent to an I/O device or memory chip.

8156

The 8156 is a 2,048-bit static RAM organized as 256 words of 8 bits each. This means that it can store 256 bytes in a read-write memory. Furthermore, the 8156 has three I/O ports (not shown) that can be programmed to act as input or output ports. Chapter 15 goes into the details of the 8156 and how to use its I/O ports. Right now, all we are concerned with is the memory part of this chip.

The address-data pins of the 8156 (AD_7 to AD_0) are tied to the address-data bus of the 8085. The addresses or data on this bus vary from

0000 0000 to 1111 1111

equivalent to decimal 0 to 255.

8156 Control Signals

Look at the control signals driving the 8156. As mentioned earlier, the falling edge of the *ALE* signal is used to strobe an address into the MAR or address latch of the memory chips. When *ALE* goes from high to low, the address on the address-data bus is latched inside the 8156.

The chip-enable signal *CE* is connected to A_{13}; therefore, the 8156 is enabled only when A_{13} is high. During a memory operation, *IO/$\overline{M}$* is low; for a read, $\overline{RD}$ is low; for a write, $\overline{WR}$ is low. Finally, the *RESET* comes from the *RESET OUT* of the 8085. When high, *RESET* initializes the 8156.

8355

The 8355 is a 16,384-bit ROM organized as 2,048 words of 8 bits each. This means that it can store 2,048 bytes of instructions and fixed data. Address lines A_{10} to A_8 and AD_7 to AD_0 are connected to the 8085. These 11 lines can address 2,048 locations.

Again, notice how the ALE, CLK, IO/$\overline{M}$, $\overline{RD}$, and RESET pins are connected to the corresponding pins of the 8085. As before, the falling edge of the *ALE* signal loads the address into the 8355. During a memory operation, *IO/$\overline{M}$* is low and a low $\overline{RD}$ reads the data in the selected memory location.

The 8355 has two chip enables, $\overline{CE}_1$ and CE_2. In the minimum system of Fig. 13-6, CE_2 is made active high by

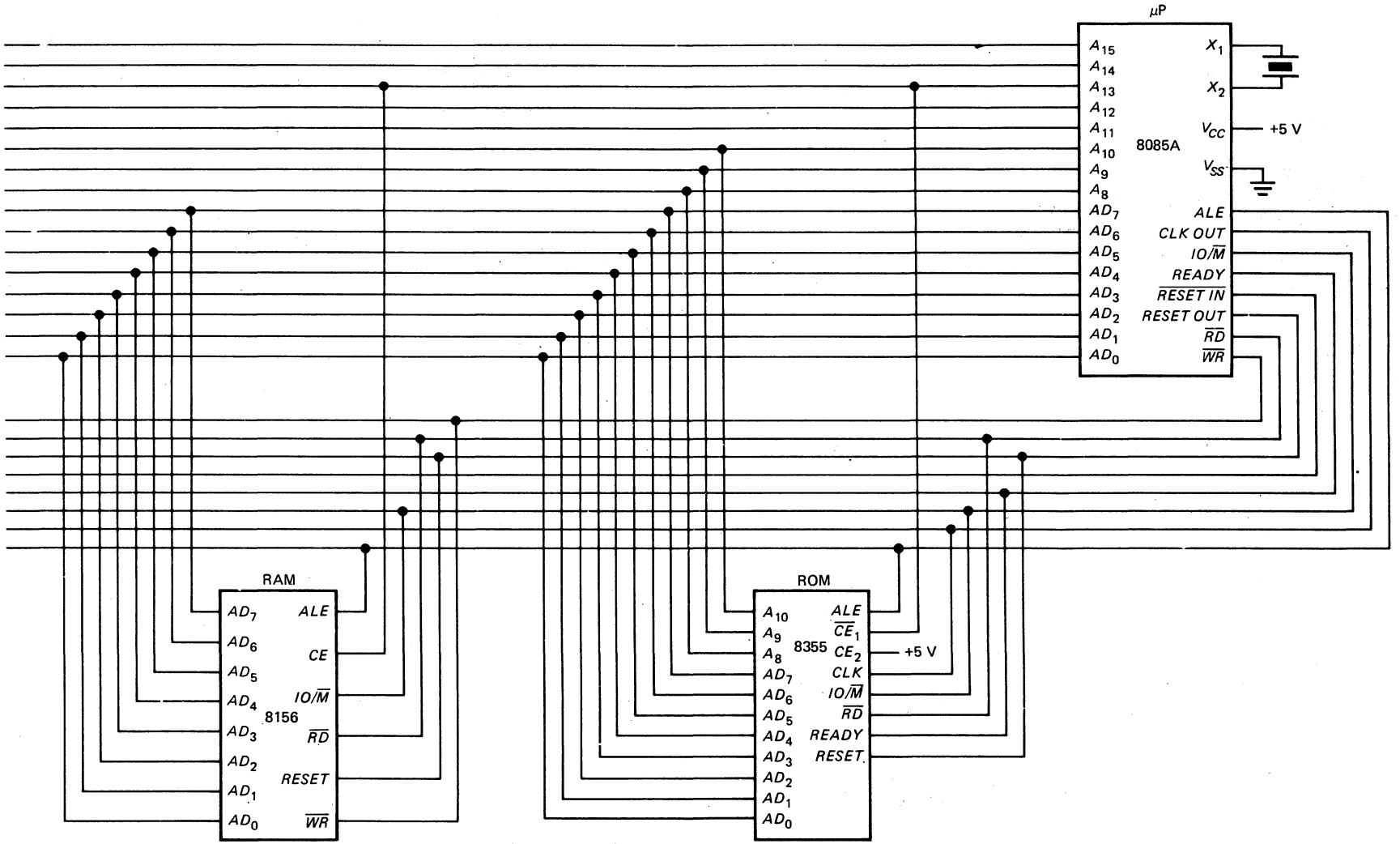


Fig. 13-6 Minimum system.

tying it to +5 V. On the other hand, $\overline{CE}_1$ is connected to A_{13}. Because $\overline{CE}_1$ is active low, the 8355 is enabled only when A_{13} is low.

The *READY* signal is low while the address is being loaded into the 8355. After the address has been latched, *READY* goes high, signaling the 8085 to proceed with the read operation.

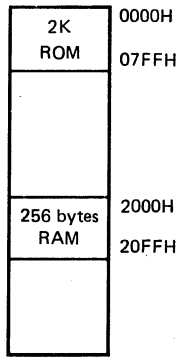


Fig. 13-7 ROM and RAM for minimum system.

Addressing the Memory Chips

When A_{13} is low, the ROM is enabled. The lowest addresses the 8355 responds to are

0000 0000 0000 0000
to 0000 0111 1111 1111

equivalent to 0000H to 07FFH. This is illustrated in Fig. 13-7, where you see the ROM occupying the first 2K of memory. Since the program counter starts at 0000H, the first instruction is fetched from ROM.

When A_{13} is high, the RAM is enabled. The lowest addresses the 8156 responds to are

0010 0000 0000 0000
to 0010 0000 1111 1111

This is equivalent to 2000H to 20FFH. In the minimum system of Fig. 13-6, therefore, the lowest area of RAM that we can address is 2000H to 20FFH, as shown in Fig. 13-7.

Folded Memory

The 16 address lines of the 8085 can address a total of 65,536 locations. In the minimum system of Fig. 13-6, we use only 2K of ROM and 256 bytes of RAM. The *unused* address lines are *don't cares;* they can be either 0s or 1s. Because some of the address lines are don't cares, the ROM and RAM sections of memory *fold back* (repeat).

For instance, the valid range of ROM addresses is

XX0X X000 0000 0000
to XX0X X111 1111 1111

where the Xs can be 0s or 1s. When the Xs are 0s, we get

0000 0000 0000 0000
to 0000 0111 1111 1111

or 0000H to 07FFH. This is the first ROM range shown in Fig. 13-8*a*.

If A_{11} is high and all other don't cares are low, the valid range of ROM addresses is

0000 1000 0000 0000
to 0000 1111 1111 1111

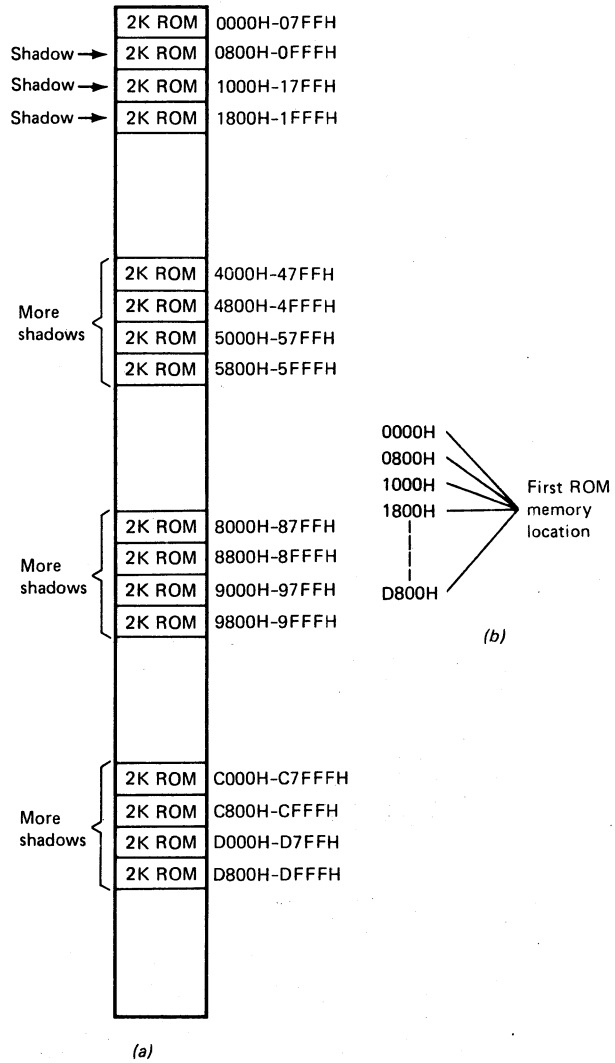


Fig. 13-8 Folded memory.

equivalent to 0800H to 0FFFH. The ROM will respond to these addresses because A_{13} is still low. The second range of addresses shown in Fig. 13-8*a* is called a *shadow*.

Similarly, if A_{12} is high and all other don't cares are low, the valid range of addresses becomes

0001 0000 0000 0000
to 0001 0111 1111 1111

This is from 1000H to 17FFH, as shown in Fig. 13-8*a*.

By checking the remaining don't-care combinations, we can find a total of 15 shadows, areas where the ROM folds back or repeats (see Fig. 13-8*a*). What this all amounts to is that several addresses can access the same memory location in the ROM. For example, addresses 0000H, 0800H, 1000H, and so on all point to the first memory location in the ROM (Fig. 13-8*b*). If any of these addresses appears on the address bus and address-data bus, the 8355 will access its first memory location.

The RAM also folds back. When A_{13} is high, the valid RAM address range is from

XX1X XXXX 0000 0000
to XX1X XXXX 1111 1111

which is equivalent to the following ranges:

2000H–20FFH
2100H–21FFH
2200H–22FFH
.
.
.
FF00H–FFFFH

This means that the first RAM range is from 2000H to 20FFH and all the rest are shadows.

Foldback (also known as foldover) causes no problems as long as we agree which ROM and RAM ranges will be used in programming. Usually, the lowest ROM and RAM areas are used; the shadows are not. From here on, the minimum system described in this book will use the lowest ROM and RAM ranges:

ROM 0000H–07FFH
RAM 2000H–20FFH

Typical Memory Map

By connecting more memory chips to the 8085 we can make the memory any size we want up to 64K. Almost all microprocessor-based systems have the memory laid out as shown in Fig. 13-9. The lowest part of memory is reserved for fixed instructions and data. In a microcomputer this area stores subroutines that operate the keyboard, the video display, and so forth. A ROM, PROM, or EPROM is used

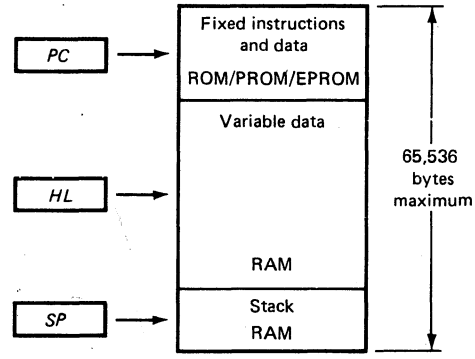


Fig. 13-9 Typical memory map.

for this section of memory with the program counter acting as a pointer.

Next comes the variable data area, which uses a RAM. In a microcomputer this is where the user programs are stored. Also, intermediate results of data processing may be saved in this section of the memory. During the execution of indirect instructions, the HL pointer controls this area.

The stack is usually in the highest part of memory, the stack pointer automatically indicating where data is to be pushed or popped.

13-7 FETCHING AND EXECUTING INSTRUCTIONS

Figures 13-10 and 13-11 show how each of the 8085 instructions are fetched and executed. These figures tell you what happens during each *T* state of a specific instruction. This section discusses a few of the instructions to give you an idea of how to follow an instruction all the way through its fetch and execute cycles.

Basic Timing

To begin with, the shaded *T* states of Figs. 13-10 and 13-11 indicate unused states. A glance at both figures shows that an instruction cycle may take from one to five machine cycles. Furthermore, the first machine cycle has either four or six *T* states; the remaining machine cycles have either two or three *T* states.

The NOP has the shortest instruction cycle: one machine cycle with four *T* states. The CALL instruction has the longest instruction cycle: five machine cycles with 18 *T* states.

Register Move

Let's take a look at the register-move instruction

MOV reg1,reg2

Refer to Fig. 13-10 throughout this discussion. During the first T state, the contents of the program counter are placed on the address bus and address-data bus (PC OUT). Midway through this T state, the *ALE* signal goes from high to low; this latches the address into the selected memory chip.

During the second T state the program counter is incremented (PC + 1 → PC). During the third T state, the op code for the MOV instruction is read out of the memory and sent to the instruction register (INSTR → IR). In the T_4 state, the contents of register 2 are copied in the temporary register. The T_5 and T_6 states are shaded; this means they are not used.

The word FEO in the T_1 state of the second machine cycle stands for *fetch-execute overlap*. This will be explained later. As far as the MOV instruction is concerned, nothing happens during this T state.

During the T_2 state of M_2 the contents of the temporary register are copied into register 1. This completes the instruction cycle of the register-move instruction.

Indirect Memory Read

An indirect read instruction takes longer because two memory operations are needed, one for the instruction fetch and one for the data. Consider the instruction

MOV reg,M

This will copy the addressed memory data into a designated register. In Fig. 13-10 the first three T states are the same fetch cycle as before, PC OUT, PC + 1 → PC, and INSTR → IR. The X in the fourth T state means that this T state is needed for instruction decoding and other internal operations that must take place before the execution cycle can begin.

In the second machine cycle M_2, the contents of the HL register are placed on the address bus and address-data bus during the T_1 state. Midway through this T state, the falling edge of the *ALE* signal strobes the address into the memory chips. Because of the memory access time, it takes two states (T_2 and T_3) to read the data and copy it into the selected register. This is why the arrow straddles states T_2 and T_3.

Notice that two machine cycles and seven T states are needed to fetch and execute a memory MOV instruction. With a 3-MHz clock, this means that it takes

$$7 \times 330\,\text{ns} = 2.31\,\mu\text{s}$$

to fetch and execute an indirect memory read instruction.

Indirect Memory Write

The next instruction we want to discuss is

MOV M,reg

As shown in Fig. 13-11, the first three T states are the fetch portion. During the fourth T state, the contents of the designated register are copied in the temporary register. T_5 and T_6 are skipped.

In the second machine cycle, the contents of the HL pointer are placed on the address bus and address-data bus during the T_1 state. Midway through this T state, the *ALE* signal latches the address into the memory chip.

During T_2 and T_3 of the second machine cycle, the contents of the temporary register are transferred to the addressed memory location.

Immediate MOV

The instruction

MVI reg,byte

begins with a fetch (T_1, T_2, and T_3) in Fig. 13-11. The T_4 state is for decoding.

In the second machine cycle, the contents of the program counter are placed on the address bus and address-data bus during T_1. During T_2, the PC is incremented. While this is happening, the memory is being accessed; then the immediate data is read and loaded into the designated register during T_3.

ADD

The add instruction

ADD reg

starts with the usual fetch. Then during the T_4 state of M_1, the contents of the selected register are loaded into the temporary register.

The second machine cycle begins with an FEO in the T_1 state (explained in detail later). As far as the ADD instruction is concerned, nothing happens during the T_1 state. During the T_2 state the contents of the accumulator and temporary register are added; the result is stored back in the accumulator.

JMP

The unconditional jump instruction is

JMP address

After the op code is fetched, decoding takes place during the fourth T state (see Fig. 13-10).

In the second machine cycle, the contents of the program counter are placed on the address bus and address-data bus; midway through the T_1 state, this address is latched into a memory chip. States T_2 and T_3 increment the program

Mnemonic	M_1						M_2		
	T_1	T_2	T_3	T_4	T_5	T_6	T_1	T_2	T_3
ACI byte	PC OUT	PC+1→PC	INSTR→IR	X			PC OUT	PC+1→PC	byte→TMP
ADC reg				reg→TMP			FEO	A+TMP+CY→A	
ADC M				X			HL OUT	M_{HL} → TMP	
ADD reg				reg→TMP			FEO	A+TMP→A	
ADD M				X			HL OUT	M_{HL} → TMP	
ADI byte				X			PC OUT	PC+1→PC	byte→TMP
ANA reg				reg→TMP			FEO	A·TMP→A	
ANA M				X			HL OUT	M_{HL} → TMP	
ANI byte				X			PC OUT	PC+1→PC	byte→TMP
CALL address				SP−1 → SP		X	PC OUT	PC+1→PC	low byte→Z
C cond address				If condition is true (T), then SP − 1 → SP.		X	PC OUT	T: PC+1→PC F: PC+2→PC	low byte→Z F: Start M_1
CMA				A→ALU COMPLEMENT			FEO	ALU→A	
CMC				CY→ALU COMPLEMENT			FEO	ALU→A	
CMP reg				A→ALU reg→TMP			FEO	A − TMP FLAGS	
CMP M				X			HL OUT	M_{HL} → TMP	
CPI byte				X			PC OUT	PC+1→PC	byte→TMP
DAA				X			FEO	ADJUST A FLAGS	
DAD RP				X			RP_L→A	L→TMP A+TMP→ALU	ALU→L CY
DCR reg				reg→TMP TMP+1→ALU			FEO	ALU→reg	
DCR M				X			HL OUT	M_{HL} → TMP TMP−1 → ALU	
DCX RP				RP−1 → RP		X			
DI				X			FEO RESET INTERRUPT FLIP–FLOP		
EI				X			FEO SET INTERRUPT FLIP–FLOP		
HLT				X			PC OUT	HALT MODE	
IN byte				X			PC OUT	PC+1→PC	byte→Z, W
INR reg				reg→TMP			FEO	ALU→reg	
INR M				X			HL OUT	M_{HL} → TMP TMP+1 → ALU	
INX RP				RP+1 → RP		X			
JMP address				X			PC OUT	PC+1→PC	low byte→Z
J cond address				X			PC OUT	T: PC+1→PC F: PC+2→PC	low byte→Z F: Start M_1
LDA address				X			PC OUT	PC+1→PC	low byte→Z
LDAX RP				X			RP OUT	M_{RP} → A	
LHLD address				X			PC OUT	PC+1→PC	low byte→Z
LXI RP, dble				X			PC OUT	PC+1→PC	low byte→RP_L
MOV reg1,reg2				reg2→TMP			FEO	TMP→reg1	
MOV reg,M	PC OUT	PC+1→PC	INSTR→IR	X			HL OUT	M_{HL} → reg	

Fig. 13-10 Fetch and execution.

M_3			M_4			M_5		
T_1	T_2	T_3	T_1	T_2	T_3	T_1	T_2	T_3
FEO	A+TMP+CY→A FLAGS							
FEO	A+TMP+CY→A FLAGS							
FEO	A+TMP→A FLAGS							
FEO	A+TMP→A FLAGS							
FEO	A·TMP→A FLAGS							
FEO	A·TMP→A FLAGS							
PC OUT	PC+1→PC	high byte→W	SP OUT	PC_H → stack SP−1→SP		SP OUT	PC_L → stack	
PC OUT	PC+1→PC	high byte→W	SP OUT	PC_H → stack SP−1→SP		SP OUT	PC_L → stack	
FEO	A−TMP FLAGS							
FEO	A−TMP FLAGS							
RP_H→A	H→TMP A+TMP+CY→ALU	ALU→H CY						
HL OUT	ALU →	M_{HL}						
WZ OUT	PORT →	A						
HL OUT	ALU →	M_{HL}						
PC OUT	PC+1→PC	high byte→W	FEO WZ OUT	WZ+1→PC				
PC OUT	PC+1→PC	high byte→W	FEO WZ OUT	WZ+1→PC				
PC OUT	PC+1→PC	high byte→W	WZ OUT	M_{WZ} →	A			
PC OUT	PC+1→PC	high byte→W	WZ OUT	M_{WZ} → L WZ+1→WZ		WZ OUT	M_{WZ} →	H
PC OUT	PC+1→PC	high byte→RP_H						

Mnemonic	M_1						M_2		
	T_1	T_2	T_3	T_4	T_5	T_6	T_1	T_2	T_3
MOV M,reg	PC OUT	PC+1→PC	INSTR→IR	reg→TMP			HL OUT	TMP →	→ M_{HL}
MVI reg, byte				X			PC OUT	PC+1→PC	byte→reg
MVI M,byte				X			PC OUT	PC+1→PC	byte→TMP
NOP				X					
ORA reg				reg→TMP			FEO	A OR TMP→A	
ORA M				X			HL OUT	M_{HL} →	→ TMP
ORI byte				X			PC OUT	PC+1→PC	byte→TMP
OUT byte				X			PC OUT	PC+1→PC	byte→Z, W
PCHL				HL →	→ PC	X			
POP RP				X			SP OUT	SP+1→SP	low byte→RP_L
POP PSW				X			SP OUT	SP+1→SP	low byte→FLAGS
PUSH RP				SP−1 →	→ SP	X	SP OUT	SP−1→SP	high byte→stack
PUSH PSW				SP−1 →	→ SP	X	SP OUT	SP−1→SP	A→stack
RAL				A, CY→ALU ROTATE			FEO	ALU→A, CY	
RAR				A, CY→ALU ROTATE			FEO	ALU→A, CY	
RET				X			SP OUT	SP+1→SP	low byte→PC_L
R cond				JUDGE CONDITION →		F: Start M_1	SP OUT	SP+1→SP	low byte→PC_L
RIM				X			FEO	Interrupt mask →	→ A
RLC				A→ALU ROTATE			FEO	A→ALU, CY	
RRC				A→ALU ROTATE			FEO	A→ALU, CY	
SBB reg				reg→TMP			FEO	A − TMP − CY→A	
SBB M				X			HL OUT	M_{HL} →	→ TMP
SBI byte				X			PC OUT	PC+1→PC	byte→TMP
SHLD address				X			PC OUT	PC+1→PC	low byte→Z
SIM				X			FEO	A →	→ Interrupt mask
SPHL				HL →	→ SP	X			
STA address				X			PC OUT	PC+1→PC	low byte→Z
STAX RP				X			RP OUT	A →	→ M_{RP}
STC				1→ALU			FEO	ALU→CY	
SUB reg				reg→TMP			FEO	A − TMP→A	
SUB M				X			HL OUT	M_{HL} →	→ TMP
SUI byte				X			PC OUT	PC+1→PC	byte→TMP
XCHG				X			FEO	HL↔DE	
XRA reg				reg→TMP			FEO	A ⊕ TMP→A	
XRA M				X			HL OUT	M_{HL} →	→ TMP
XRI byte	PC OUT	PC+1→PC	INSTR→IR	X			PC OUT	PC+1→PC	byte→TMP

Fig. 13-11 Fetch and execution.

M_3			M_4			M_5		
T_1	T_2	T_3	T_1	T_2	T_3	T_1	T_2	T_3
HL OUT	TMP →	M_{HL}						
FEO	A OR TMP→A							
FEO	A OR TMP→A							
WZ OUT	A →	PORT						
SP OUT	SP+1→SP	high byte→RP_H						
SP OUT	SP+1→SP	high byte→A						
SP OUT	low byte →	stack						
SP OUT	FLAGS →	stack						
SP OUT	SP+1→SP	high byte→PC_H						
SP OUT	SP+1→SP	high byte→PC_H						
FEO	A−TMP−CY→A							
FEO	A−TMP−CY→A							
PC OUT	PC+1→PC	high byte→W	WZ OUT	L → WZ+1→WZ	M_{WZ}	WZ OUT	H →	M_{WZ}
PC OUT	PC+1→PC	high byte→W	WZ OUT	A →	M_{WZ}			
FEO	A−TMP→A							
FEO	A−TMP→A							
FEO	A ⊕ TMP→A							

counter and transfer the lower byte into the Z register. (The 8085 has two internal registers labeled W and Z, used for temporary storage of data. These registers are invisible because the programmer never uses them.)

During the third machine cycle, the program counter again addresses the memory during T_1. During T_2, the PC is incremented. Then during T_3 the upper byte is transferred from the memory to the W register.

Notice that it takes three machine cycles and ten T states to complete the JMP instruction. After the last T state, a new instruction cycle begins as follows. The contents of the WZ register pair are placed on the address bus and address-data bus during the T_1 state of the next machine cycle. During the T_2 state the contents of the WZ pair are incremented, and the result is stored in the program counter. Hereafter, control returns to the program counter.

In other words, jump instructions are executed by loading the jump address in the WZ register pair. During the next fetch cycle, the WZ register pair takes over temporarily from the program counter. After the program counter has been loaded with the incremented WZ contents, the PC regains control.

Fetch-Execute Overlap

In Fig. 13-10 it takes six T states to fetch and execute an ADD instruction. A glance at Appendix 5 shows that the ADD instruction requires four T states. Why the difference?

The ADD instruction does not use the address bus and address-data bus during the second machine cycle. All that happens is that the contents of the accumulator and temporary register are added; the result is stored back in the accumulator. Since the address bus and address-data bus are not used during the second machine cycle of an ADD instruction, we can use the address bus and address-data bus for some other purpose.

One way to save processing time is by starting the next instruction cycle during the second machine cycle of the ADD instruction. This is called *fetch-execute overlap*. Figure 13-12 illustrates fetch-execute overlap for this program segment:

```
ADD B
MOV B,A
```

The first three T states of Fig. 13-12 fetch the op code for the ADD B instruction. During T_4, the contents of the B register are loaded into the temporary register.

During the second machine cycle of the ADD B, the instruction cycle of the MOV B,A begins. During T_1, the contents of the program counter are latched into the memory. During T_2, the ADD B is finished off by adding the contents of the accumulator and temporary register; the result is stored back in the accumulator. During this same T state, the program counter is incremented. During T_3, the MOV op code is loaded into the instruction register.

Fetch-execute overlap saves processing time because the final execution states of the ADD instruction are taking place during the fetch states of the next instruction. When figuring how long it takes to run a program, all ADD instructions are counted as four T states rather than six because the last two T states are counted in the next instruction.

All arithmetic, logic, and rotate instructions have a final machine cycle with a free address bus and address-data bus. Because of this, fetch-execute overlap is used with these instructions. In other words, the final machine cycle of these instructions is overlapped by the fetch portion of the next instruction cycle.

13-8 8085 TIMING DIAGRAMS

Figures 13-10 and 13-11 summarize the fetch and execution of all 8085 instructions. To deepen our understanding of how the 8085 works, let's look at the timing diagrams of a few instructions. This will illustrate how the external address, data, and control signals interract during an instruction cycle.†

Register Move

Figure 13-13 is the timing diagram of

```
MOV reg1,reg2
```

† Actually, the 8085 microprogram is more complicated than shown here; Figs. 13-10 and 13-11 are only approximations of what is going on inside the 8085.

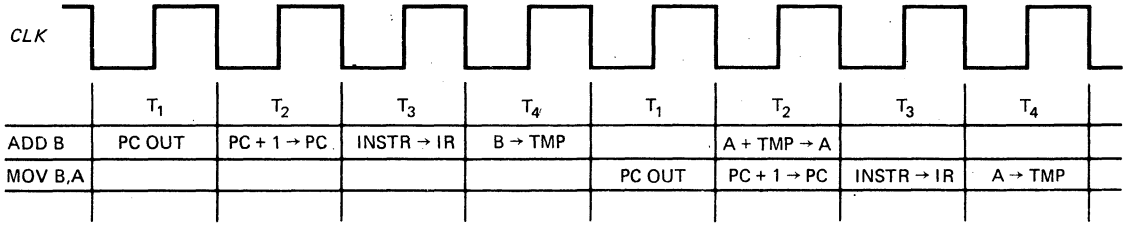


Fig. 13-12 ADD timing diagram.

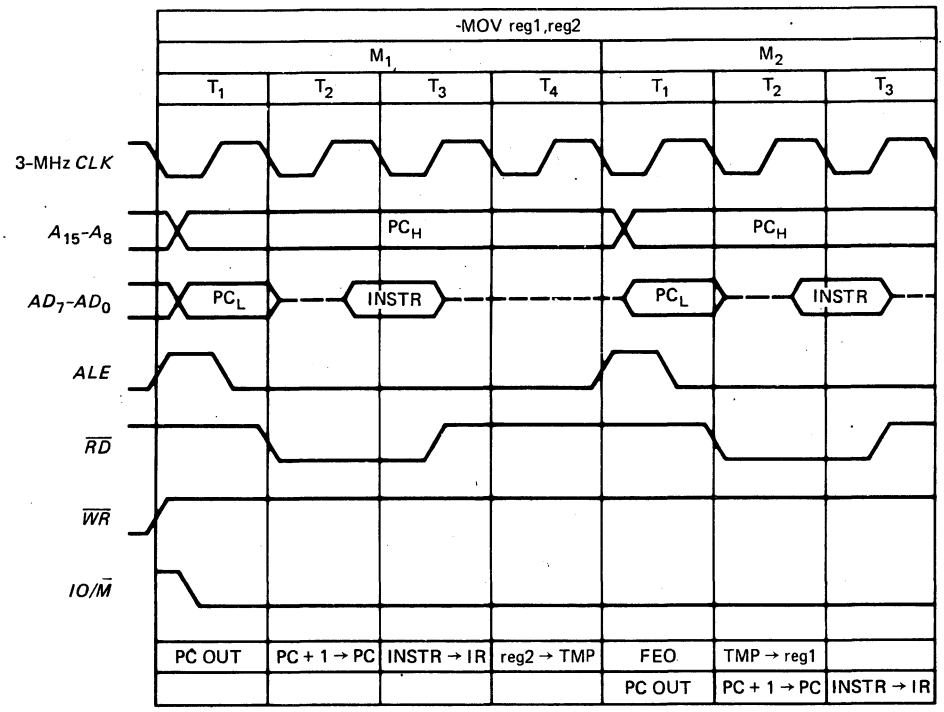


Fig. 13-13 Register MOV timing diagram.

for a 3-MHz clock. Each CLK cycle represents one T state. At the bottom of the diagram are the actions that take place: PC OUT, PC + 1 → PC, INSTR → IR, and so on. As indicated, a fetch-execute overlap (FEO) occurs during the second machine cycle.

During the T_1 state of the first machine cycle, the contents of the program counter are placed on the address bus (A_{15} to A_8) and address-data bus (AD_7 to AD_0). Since address bits may be low or high, it is customary to use the double-sided waveforms shown in Fig. 13-13; the high byte out of the program counter (PC_H) goes to the address bus, and the low byte (PC_L) to the address-data bus.

The *ALE* signal initially goes high; then midway through the T_1 state, *ALE* goes low. It is this falling edge that latches the address bits into the memory chips. Also note how $IO/\overline{M}$ goes low near the beginning of the T_1 state; this enables the peripheral chips for a memory operation rather than an I/O operation.

During the T_2 state, the program counter is incremented. The address disappears from the address-data bus (AD_7 to AD_0) at the beginning of the T_2 state. This is necessary because an instruction fetch is in progress and the address-data bus is needed. The dashed line on the AD_7–AD_0 waveform means that the data on the bus is invalid or meaningless at this time. Toward the end of the T_2 state, the op code (INSTR) appears on the address-data bus. The precise time when the INSTR appears depends on the memory access time, the length of the buses, and other factors.

At the beginning of the T_2 state, $\overline{RD}$ goes low and stays low until the middle of the T_3 state. During the T_3 state, the INSTR on the address-data bus is copied into the instruction register. During the T_4 state, the contents of register 2 are copied into the temporary register.

The second machine cycle represents a fetch-execute overlap. The new address in the program counter is placed on the address bus and address-data bus. Again, the falling edge of the *ALE* signal latches this address into the memory chips. In the T_2 state, the contents of the temporary register are copied into register 1. This completes the execution of the MOV instruction. During the same T state, the program counter is incremented. Toward the end of the T state, the next instruction appears on the address-data bus. Then the rest of the new instruction cycle is carried out.

Indirect Read

Figure 13-14 is the timing diagram of a

MOV reg,M

This indirect instruction will copy the contents of the addressed memory location into the designated register. The first machine cycle is an instruction fetch; the second is a memory read.

In the T_1 state of the second machine cycle, the contents of the HL register are placed on the address bus and address-data bus; the high byte goes to A_{15}–A_8; the low byte to

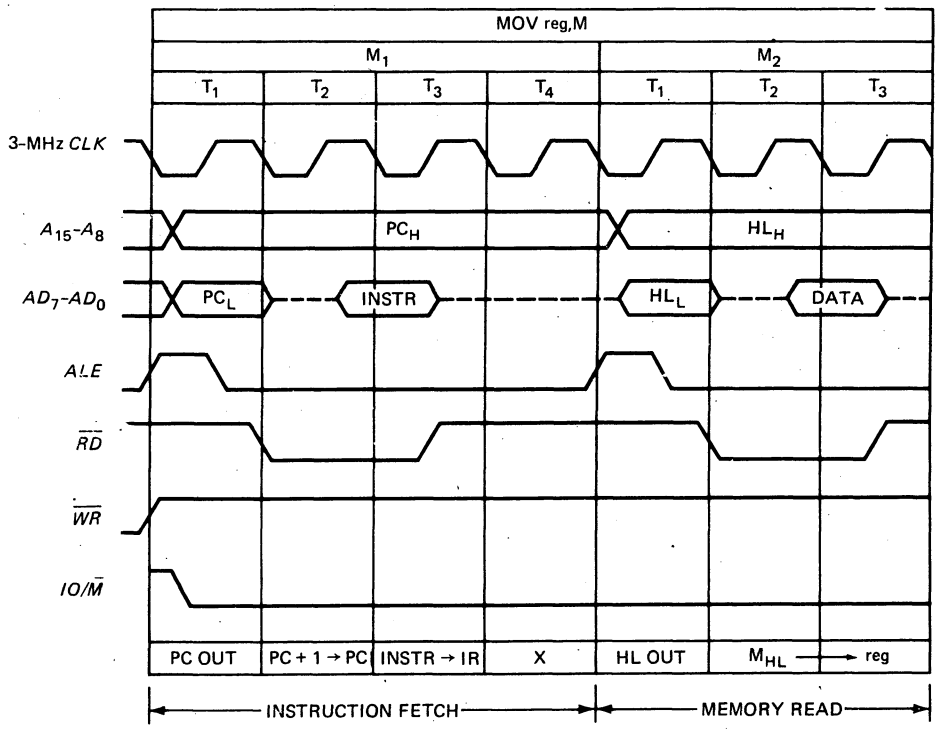


Fig. 13-14 Timing diagram for indirect read.

AD_7–AD_0. After ALE latches this HL address into the memory chips, a read operation takes. Notice that the data is read and transferred to the designated register during the T_2 and T_3 states of M_2.

Indirect Write

The indirect write instruction

MOV M,reg

loads the contents of a designated register into the memory location addressed by the HL pointer. Figure 13-15 shows the timing diagram. As with all instructions, the first three T states fetch and store the op code in the instruction register. During the T_4 state, the contents of the designated register are copied into the temporary register.

In the T_1 state of M_2, the high and low bytes of the HL register are placed on the address bus and address-data bus. After being latched in memory chips by the *ALE* signal, the low byte of the HL address disappears from the address-data bus. A bit later, the data from the temporary register is sent to the address-data bus. Since the $\overline{WR}$ signal is low, this data is written into the addressed memory location.

OUT Instruction

Our final example is the instruction

OUT byte

whose timing diagram is shown in Fig. 13-16. Three machine cycles are needed for this instruction. As indicated at the bottom of Fig. 13-16, the first machine cycle is an instruction fetch, the second is a memory read, and the third is an output write.

The action of the first machine cycle is familiar from preceding examples, so let us proceed directly to the second machine cycle. In the T_1 state of M_2, the incremented PC address is latched into the memory chips. Toward the end of the T_2 state, a byte appears on the address-data bus. During the T_3 state, this byte is copied in the W and Z registers (byte → Z,W). In symbols,

$$\mathbf{W} = \text{byte}$$
$$\mathbf{Z} = \text{byte}$$

This byte represents the port number.

In the T_1 state of the third machine cycle, the $IO/\overline{M}$ signal goes high. This enables the peripheral chips for an I/O operation rather than a memory operation. During this T state, the contents of the W register are placed on the address bus, and the contents of the Z register on the address-data bus (WZ OUT). Since the same byte is in both registers, the same port number (IO PORT) is being sent along the buses to the peripheral chips. (Chapter 15 explains why the port number is duplicated on the buses.)

During the T_2 state of M_3, the contents of the accumulator are placed on the address-data bus and the $\overline{WR}$ signal goes low. The selected peripheral chip then writes the accumulator data into the designated output port during the T_2 and T_3 states.

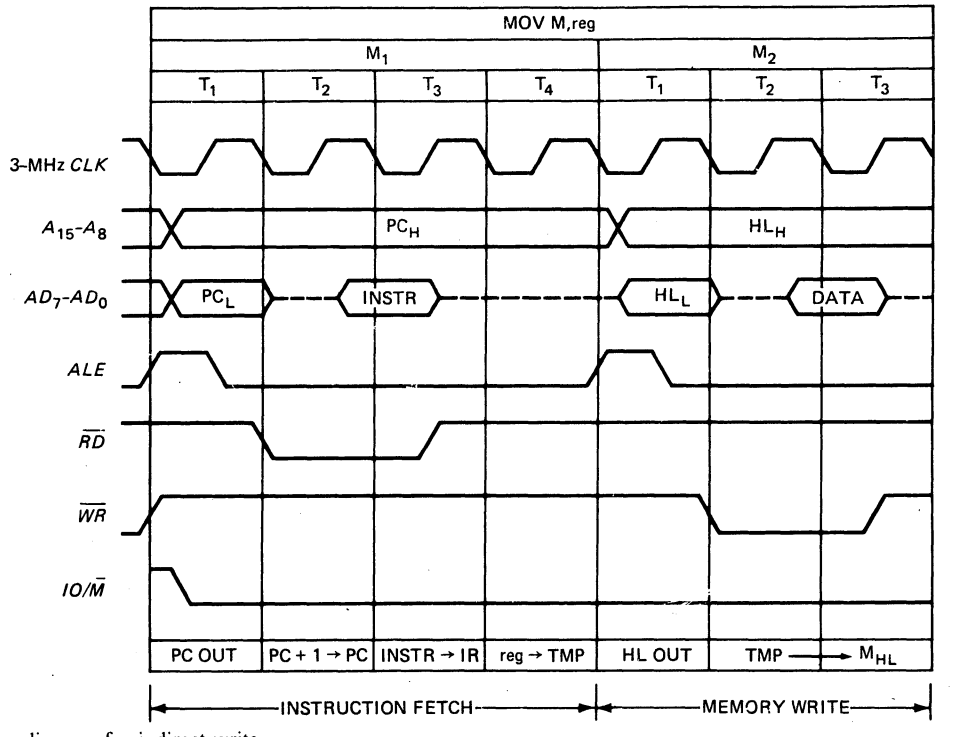


Fig. 13-15 Timing diagram for indirect write.

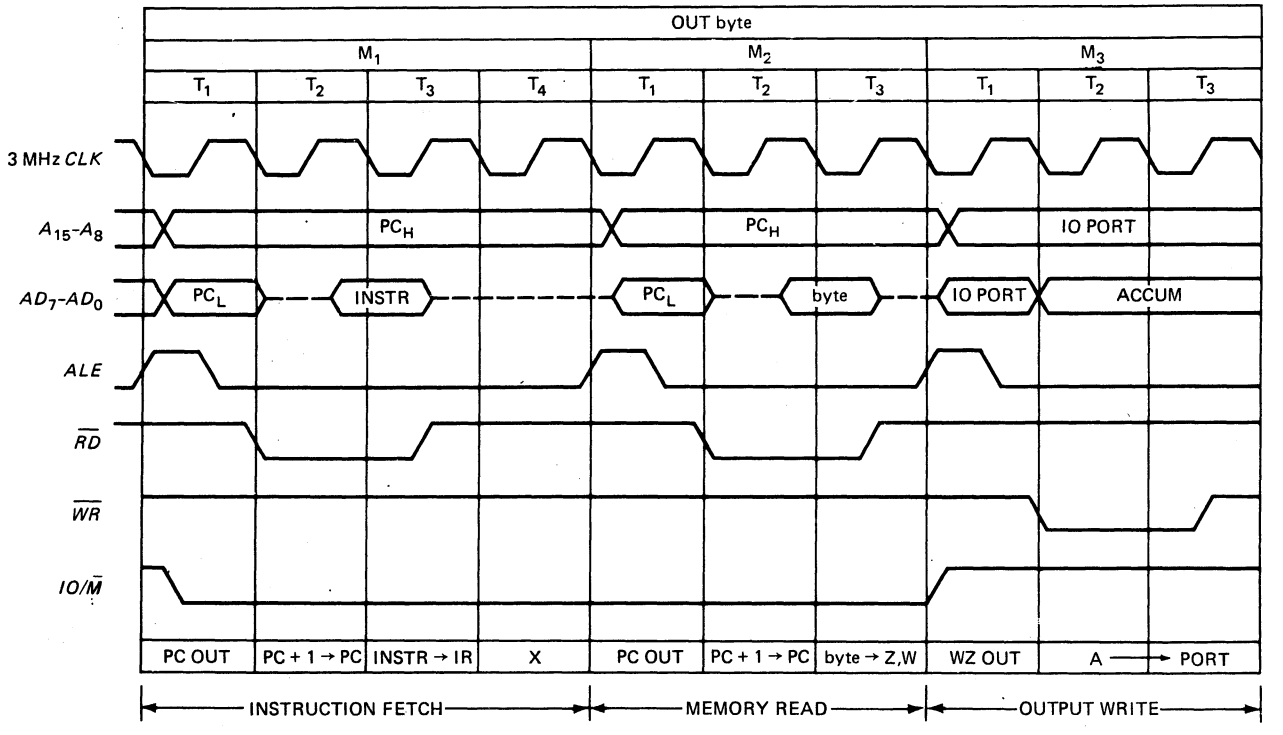


Fig. 13-16 Timing diagram for OUT instruction.

GLOSSARY

address latch The MAR inside a memory chip.
ALE signal Address latch enable signal. The falling edge of the *ALE* signal strobes the address on the address bus and address-data bus into the address latch of each memory chip.
auxiliary carry A carry from bit 3 to bit 4. The AC flag is set when there is an auxiliary carry.
control store The control ROM that stores the 8085 microinstructions.
direct memory access A mode in which the 8085 turns its address bus, address-data bus, and control bus over to a DMA controller. This allows the controller to transfer large amounts of the data directly from I/O to memory or vice-versa. Abbreviated DMA.
fetch-execute overlap Operation in which the fetch cycle of an instruction starts during the execution cycle of the preceding instruction. Fetch-execute overlap is used because it speeds up data processing.
folded memory Operation in which the ROM or RAM sections fold back, meaning that more than one address can access the same memory location. This is possible because unconnected address lines are don't cares and can be low or high without affecting memory operations.
incrementer-decrementer A circuit that adds or subtracts 1 from the contents of the program counter or stack pointer.
interrupt A signal from a peripheral device requesting an I/O transfer or other service.
strobe Timing control with ALE. Since the ALE frequency is different from the clock frequency, the word "strobe" is used with ALE. For instance, we say that the *ALE* signal strobes an address (rather than clocks an address).
WAIT *state* A holding pattern into which the 8085 can go by generating WAIT states (nops). It does this whenever its *READY* input is low.

SELF-TESTING REVIEW

Read each of the following and provide the missing words. Answers appear at the beginning of the next question.

1. The upper 8 address bits appear on the _____ bus. During some *T* states, the lower 8 address bits appear on the _____ bus; during other *T* states, data appears on this bus. The reason for multiplexing the lower part of the address bus with the data bus was to keep the _____ count at 40.
2. (*address, address-data, pin*) The controller-sequencer of the 8085 is microprogrammed; it has a ROM that stores all the _____ needed for executing the instructions. This control ROM is sometimes called the control store.
3. (*microroutines*) Each memory chip has its own MAR, usually called an address _____. The falling edge of the *ALE* signal strobes the _____ on the address bus and address-data bus into the address latch of each memory chip.
4. (*latch, address*) If a peripheral device is not ready for a data transfer, it will send a low *READY* bit to the 8085. The 8085 then generates _____ states until a high *READY* bit is received.
5. (*WAIT*) The way to speed up memory-peripheral data transfers is called _____ memory access, abbreviated _____. With this approach, data is moved directly from I/O to memory or vice-versa. The 8085 turns over control of its buses to a DMA _____.
6. (*direct, DMA, controller*) Usually, a _____ drives the X_1 and X_2 inputs; this produces an accurate clock period, needed for precise time delays. If a tolerance of 10 percent is acceptable, an _____ resonant circuit may be used. If very wide drifts in clock frequency are tolerable, an _____ network can be used.
7. (*crystal, LC, RC*) Whenever the data being processed is in BCD form, a _____ instruction should be executed after an ALU operation. The computer will check for nibble sums greater than 9 and will add 0110 as needed.
8. (*DAA*) The _____ system is a connection of three chips that form a microprocessor-based system: the 8085 microprocessor, the 8156 RAM, and the 8355 RAM. The memory chips include I/O ports, which can be programmed for either input or output operation.
9. (*minimum*) In a minimum system, the memory _____ back because some _____ lines are don't cares. The minimum system described in this book uses the lowest ROM and RAM ranges: 0000H–07FFH and 2000H–20FFH.
10. (*folds, address*) One way to save processing time is by starting the instruction cycle of the next instruction during the _____ cycle of a current instruction. This is called _____ overlap.
11. (*execution, fetch-execute*) A high $IO/\overline{M}$ signal indicates an I/O operation; a low $IO/\overline{M}$ signals a memory operation. Whether I/O or memory is used, a low $\overline{RD}$ reads or inputs data; a low $\overline{WR}$ writes or outputs data.

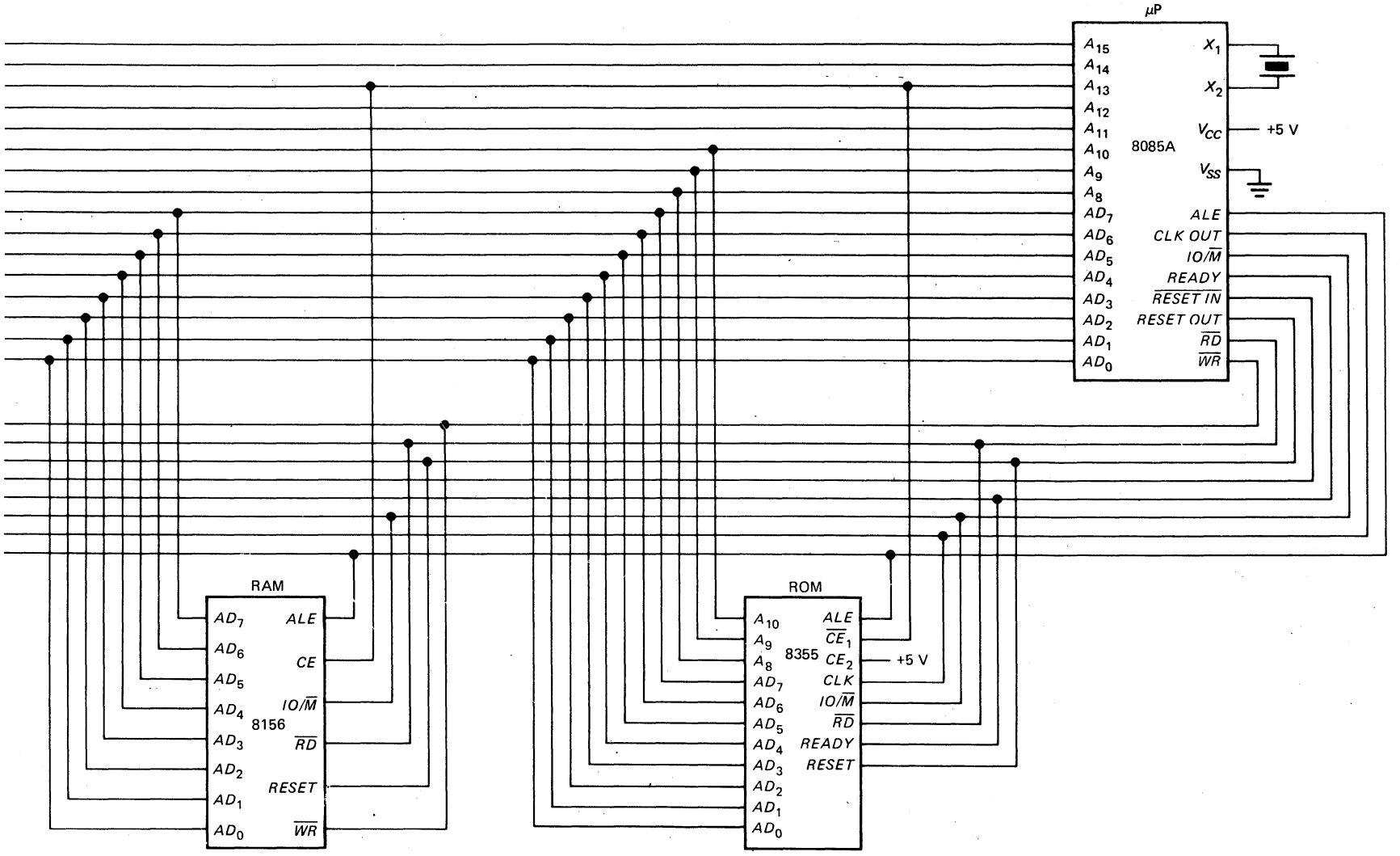

μP
A15
A14
A13
A12
A11
A10
A9
A8
AD7
AD6
AD5
AD4
AD3
AD2
AD1
AD0
8085A
X1
X2
VCC
+5 V
VSS
ALE
CLK OUT
IO/M
READY
RESET IN
RESET OUT
RD
WR
RAM
AD7
AD6
AD5
AD4
AD3
AD2
AD1
AD0
8156
ALE
CE
IO/M
RD
RESET
WR
ROM
A10
A9
A8
AD7
AD6
AD5
AD4
AD3
AD2
AD1
AD0
8355
ALE
CE1
CE2
+5 V
CLK
IO/M
RD
READY
RESET

Fig. 13-17

Mnemonic	M_1						M_2		
	T_1	T_2	T_3	T_4	T_5	T_6	T_1	T_2	T_3
ACI byte	PC OUT	PC+1→PC	INSTR→IR	X			PC OUT	PC+1→PC	byte→TMP
ADC reg				reg→TMP			FEO	A+TMP+CY→A	
ADC M				X			HL OUT	M_{HL} → TMP	
ADD reg				reg→TMP			FEO	A+TMP→A	
ADD M				X			HL OUT	M_{HL} → TMP	
ADI byte				X			PC OUT	PC+1→PC	byte→TMP
ANA reg				reg→TMP			FEO	A·TMP→A	
ANA M				X			HL OUT	M_{HL} → TMP	
ANI byte				X			PC OUT	PC+1→PC	byte→TMP
CALL address				SP−1 → SP		X	PC OUT	PC+1→PC	low byte→Z
C cond address				If condition is true (T), then SP−1→SP.		X	PC OUT	T: PC+1→PC F: PC+2→PC	low byte→Z F: Start M_1
CMA				A→ALU COMPLEMENT			FEO	ALU→A	
CMC				CY→ALU COMPLEMENT			FEO	ALU→A	
CMP reg				A→ALU reg→TMP			FEO	A−TMP FLAGS	
CMP M				X			HL OUT	M_{HL} → TMP	
CPI byte				X			PC OUT	PC+1→PC	byte→TMP
DAA				X			FEO	ADJUST A FLAGS	
DAD RP				X			RP_L→A	L→TMP A+TMP→ALU	ALU→L CY
DCR reg				reg→TMP TMP+1→ALU			FEO	ALU→reg	
DCR M				X			HL OUT	M_{HL} → TMP TMP−1 → ALU	
DCX RP				RP−1 → RP		X			
DI				X			FEO RESET INTERRUPT FLIP–FLOP		
EI				X			FEO SET INTERRUPT FLIP–FLOP		
HLT				X			PC OUT	HALT MODE	
IN byte				X			PC OUT	PC+1→PC	byte→Z, W
INR reg				reg→TMP			FEO	ALU→reg	
INR M				X			HL OUT	M_{HL} → TMP TMP+1 → ALU	
INX RP				RP+1 → RP		X			
JMP address				X			PC OUT	PC+1→PC	low byte→Z
J cond address				X			PC OUT	T: PC+1→PC F: PC+2→PC	low byte→Z F: Start M_1
LDA address				X			PC OUT	PC+1→PC	low byte→Z
LDAX RP				X			RP OUT	M_{RP} → A	
LHLD address				X			PC OUT	PC+1→PC	low byte→Z
LXI RP, dble				X			PC OUT	PC+1→PC	low byte→RP_L
MOV reg1,reg2				reg2→TMP			FEO	TMP→reg1	
MOV reg,M	PC OUT	PC+1→PC	INSTR→IR	X			HL OUT	M_{HL} → reg	

Fig. 13-18

M_3			M_4			M_5		
T_1	T_2	T_3	T_1	T_2	T_3	T_1	T_2	T_3
FEO	A+TMP+CY→A FLAGS							
FEO	A+TMP+CY→A FLAGS							
FEO	A+TMP→A FLAGS							
FEO	A+TMP→A FLAGS							
FEO	A·TMP→A FLAGS							
FEO	A·TMP→A FLAGS							
PC OUT	PC+1→PC	high byte→W	SP OUT	PC_H →	stack SP−1→SP	SP OUT	PC_L →	stack
PC OUT	PC+1→PC	high byte→W	SP OUT	PC_H →	stack SP−1→SP	SP OUT	PC_L →	stack
FEO	A−TMP FLAGS							
FEO	A−TMP FLAGS							
RP_H→A	H→TMP A+TMP+CY→ALU	ALU→H CY						
HL OUT	ALU →	M_{HL}						
WZ OUT	PORT →	A						
HL OUT	ALU →	M_{HL}						
PC OUT	PC+1→PC	high byte→W	FEO WZ OUT	WZ+1→PC				
PC OUT	PC+1→PC	high byte→W	FEO WZ OUT	WZ+1→PC				
PC OUT	PC+1→PC	high byte→W	WZ OUT	M_{WZ} →	A			
PC OUT	PC+1→PC	high byte→W	WZ OUT	M_{WZ} →	L WZ+1→WZ	WZ OUT	M_{WZ} →	H
PC OUT	PC+1→PC	high byte→RP_H						

PROBLEMS

13-1. An 8085A is crystal driven and is operating at its maximum clock frequency. What is the frequency of the crystal?

13-2. An 8085A is *LC*-driven with $L = 15\ \mu H$ and $C = 50$ pF. What is the clock frequency?

13-3. What instruction would you use to do the following?

a. Load the HL register with the bytes at 2000H and 2001H.

b. Load the program counter with the contents of the HL register pair.

c. Exchange the contents of the stack pointer with the contents of the HL register pair.

13-4. Add the following BCD numbers and correct the sum where necessary to get a BCD answer:

a. 0100 0100 + 0011 0001

b. 1001 1001 + 0110 0111

c. 0001 0011 0101 + 1000 0111 0100

13-5. In Fig. 13-17 (see page 235) a write operation into memory location 2056H is in progress:

a. Is A_{13} low or high?

b. Is $IO/\overline{M}$ low or high?

c. Is $\overline{RD}$ low or high?

d. Is $\overline{WR}$ low or high?

13-6. Memory location 0500H is being addressed in Fig. 13-17.

a. Is the RAM or ROM being addressed?

b. Is A_{13} low or high?

c. Is $IO/\overline{M}$ low or high?

d. Is $\overline{RD}$ low or high?

13-7. If the CE input to the 8156 is moved from A_{13} to A_{15}, which of the following address ranges does the RAM respond to?

a. 2000H–20FFH

b. 4000H–40FFH

c. 6000H–60FFH

d. 8000H–80FFH

13-8. In Fig. 13-18 (see pages 236 and 237) three machine cycles are needed to fetch and execute the ACI byte instruction. How many machine cycles are needed for each of the following instructions:

a. CMA

b. DCR M

c. JMP

d. LHLD

13-9. Addresses 2000H and 2001H contain ADI 74H. At the end of the T_2 state of the M_2 cycle, what does the program counter contain?

13-10. During the execution of OUT 22H, what is loaded into the W register? Into the Z register?

13-11. Addresses 2000H, 2001H, 2002H contain C3H, 90H, 30H.

a. What instruction is this?

b. How many machine cycles does the instruction require?

c. Which byte is loaded into W register?

d. Which byte is loaded into the Z register?

13-12. An 8085 is driven by a 6-MHz crystal. Write a program that produces a time delay of 0.5 s.

I/O Operations

14

There are three basic ways to get data into or out of a memory. They are called *programmed I/O*, *interrupt-driven I/O*, and *direct memory access* (DMA). Although programmed I/O is the slowest of the three, it is used in simpler microprocessor systems where speed is unimportant. As the system becomes more complex, the interrupt approach becomes necessary. In the most advanced systems, DMA is needed because it is the only way to transfer large amounts of data in a short time.

14-1 PROGRAMMED I/O

Programmed I/O uses instructions to get data into or out of a CPU. To correctly time the data transfers, programmed I/O relies either on clock timing (synchronous) or on handshaking (asynchronous). The latter type is used more often because it is simpler. This section gives two examples of asynchronous programmed I/O.

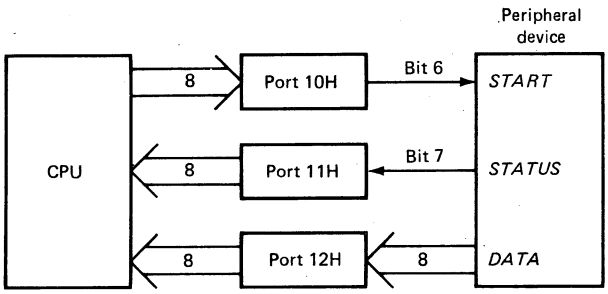


Fig. 14-1 Programmed input.

Programmed Input

Figure 14-1 shows a CPU connected to ports 10H, 11H, and 12H. Bit 6 of port 10H is connected to the START input of the peripheral device, and bit 7 of port 11H to the STATUS output. The device can send data to the CPU through input port 12H.

The basic idea is this. When the CPU is ready to input a word, it sends a high *START* bit to the peripheral device. When the device has the data ready for transfer, it sends a high *STATUS* bit to port 11H. Until the *STATUS* bit is high, the CPU waits. As soon as the *STATUS* bit goes high, the CPU inputs 1 byte of data.

Suppose we want to input 256 bytes and store them at addresses 2000H through 20FFH. Here is an example of programmed I/O written for an 8085 system:

Label	Mnemonic	Comment
	LXI H,2000H	;Initialize HL pointer
	MVI C,00H	;Initialize counter
LOOP:	MVI A,40H	;Set *START* bit
	OUT 10H	;Send high *START* bit
WAIT:	IN 11H	;Get *STATUS* bit
	ANI 80H	;Isolate *STATUS* bit
	JZ WAIT	;Wait if device not ready
	IN 12H	;Input data
	MOV M,A	;Store data
	INX H	;Update HL pointer
	MVI A,00H	;Reset *START* bit
	OUT 10H	;Send low *START* bit
	DCR C	;Count down
	JNZ LOOP	;Go back if not finished
	HLT	

The LXI H sets the HL pointer at 2000H, the location of the first byte to be stored. The MVI C clears the C register, which acts like a counter during this program.

The MVI A sets bit 6 to get accumulator contents of 40H. Then the OUT 10H latches the word 40H in port 10H. Bit 6 of port 10H is connected to the START input of the peripheral device; therefore, the device receives a high *START* bit. When ready for the data transfer, the device will send back a high *STATUS* bit.

The IN 11H and ANI 80H isolate the *STATUS* bit. As long as the *STATUS* bit is low, the program remains in the WAIT loop. After the *STATUS* bit goes high, the program falls through the JZ.

The IN 12H inputs the data from the device. The MOV M,A stores this data at 2000H. The INX H increments the HL pointer. The MVI A,00H and OUT 10H clear the *START* bit. Then the DCR C decrements the counter. At this point, the C register contains FFH. The JNZ LOOP then returns the program to the LOOP point, where the cycle starts over.

After 256 bytes of data have been stored at 2000H through 20FFH, the program falls through the JNZ LOOP to the HLT.

Notice in Fig. 14-1 that only 1 bit of port 10H is used; the other 7 bits are don't cares. Likewise in port 11H, only 1 bit is used. Ports 10H and 11H are necessary for the handshaking operations; port 12H is needed for the data transfer.

Output Example

Figure 14-2 shows a CPU connected to handshaking ports 10H and 11H. It is also connected to output port 12H. As before, bit 6 of port 10H is the *START* bit, and bit 7 of port 11H is the *STATUS* bit.

Here is the procedure for output operations. When the CPU is ready, it will latch the data into port 12H. Then, the CPU sends a high *START* bit to indicate that valid data is waiting for transfer. After the peripheral device has loaded the data, it returns a high *STATUS* bit.

Suppose we want to output 256 bytes from memory locations 2000H to 20FFH. Here is an example of programmed output:

Label	Mnemonic	Comment
	LXI H,2000H	;Initialize HL pointer
	MVI C,00H	;Initialize counter
LOOP:	MOV A,M	;Get next byte
	OUT 12H	;Latch data into port 12H
	MVI A,40H	;Set *START* bit
	OUT 10H	;Send high *START* bit
WAIT:	IN 11H	;Get *STATUS* bit
	ANI 80H	;Isolate *STATUS* bit
	JZ WAIT	;Wait for high *STATUS*
	INX H	;Update pointer
	MVI A,00H	;Reset *START* bit
	OUT 10H	;Send low *START* bit
	DCR C	;Count down
	JNZ LOOP	;Go back if count not zero
	HLT	

The LXI presets the HL pointer at 2000H, and the MVI C clears the counter. The MOV A,M loads the data into the accumulator, and the OUT 12H latches this data into port 12H.

The MVI A sets the *START* bit, and the OUT 10H sends this high *START* bit to port 10H. At this point, the program enters the WAIT loop. The IN 11H, ANI 80H, and JZ

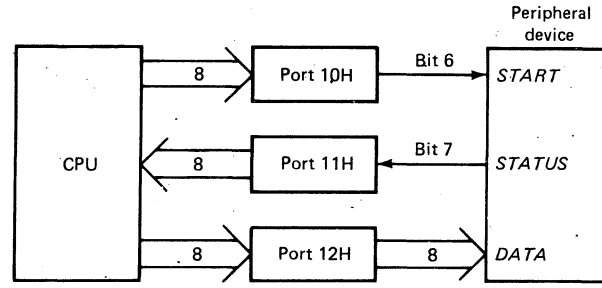


Fig. 14-2 Programmed output.

WAIT will be executed over and over until a high *STATUS* bit is returned by the peripheral device. This high *STATUS* indicates that the device has loaded the data and is ready for the next byte.

As before, the INX H increments the HL pointer; the MVI A,00H and OUT 10H clear the *START* bit; and the DCR C decrements the counter to get FFH. The JNZ LOOP returns the program to the MOV A,M, where the next cycle begins.

The program will loop until 256 bytes of data have been transferred from addresses 2000H to 20FFH to the peripheral device. Then the program falls through the JNZ to the HLT.

Incidentally, programmed I/O is sometimes referred to as *polled I/O*. In the examples given, we have used software to control the I/O transfers of a single peripheral device. By modifying the software, we can poll several peripheral devices and transfer data when each is ready.

EXAMPLE 14-1

An 8085 is receiving 100 bytes per second from a peripheral device. With the programmed input given earlier, what percent of the time is the CPU wasting? (Assume a 3-MHz clock.)

SOLUTION

The period is

$$T = \frac{1}{100 \text{ Hz}} = 0.01 \text{ s}$$

This means it takes the peripheral device 0.01 s to get each byte ready for transfer.

Let us work out the useful CPU time; this is the time needed to input and store a byte at its correct location; the useful time does not include the time needed for the *START* bit, nor does it include the time spent in the WAIT loop. The number of useful T states is

IN 12H	10
MOV M,A	7
INX H	6
DCR C	4
JNZ LOOP	10
	37

The useful CPU time is

$$37 \times 330\,\text{ns} = 12.21\,\mu\text{s} \approx 12\,\mu\text{s}$$

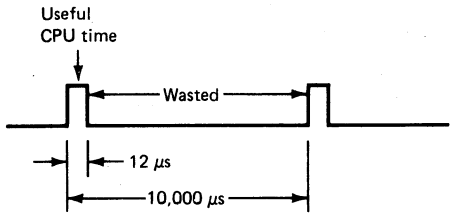


Fig. 14-3 Programmed I/O wastes most of cycle.

Figure 14-3 shows what is going on. The cycle time is 0.01 s, equivalent to 10,000 μs. The time needed to input and store a byte is only 12 μs, or 0.12 percent of the cycle time. In other words, almost 99.9 percent of the time is wasted because the CPU is waiting for the peripheral device to get the next byte ready for transfer.

In some applications, the wasted time is no problem because the CPU may have nothing better to do. But in other applications, having the CPU wait on the peripheral device is a major disadvantage.

14-2 RESTART INSTRUCTIONS

The 8085 has eight *restart* instructions: RST 0, RST 1, RST 2, RST 3, RST 4, RST 5, RST 6, and RST 7. These instructions were originally used in the interrupt system of the 8080. In the 8085, however, the restart instructions represent efficient ways to call frequently used subroutines.

What an RST Does

Table 14-1 shows the effect of each restart instruction. As indicated, an RST has the same effect as a CALL. For instance, the execution of

RST 0

will begin by pushing the contents of the program counter onto the stack. Then the program branches to address 0000H as shown in Fig. 14-4. The subroutine located between 0000H and 0007H is carried out with the RET returning the processing to the main program.

TABLE 14-1. RESTART INSTRUCTIONS

Instruction	Effect	Op Code	Vector Location
RST 0	CALL 0000H	C7	0000H
RST 1	CALL 0008H	CF	0008H
RST 2	CALL 0010H	D7	0010H
RST 3	CALL 0018H	DF	0018H
RST 4	CALL 0020H	E7	0020H
RST 5	CALL 0028H	EF	0028H
RST 6	CALL 0030H	F7	0030H
RST 7	CALL 0038H	FF	0038H

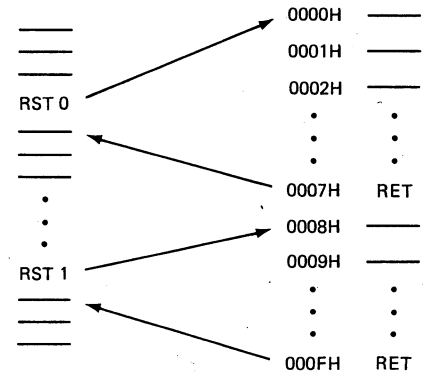


Fig. 14-4 Restarts are calls to vector locations.

If an RST 1 is encountered, the contents of the program counter are again pushed onto the stack, the program branches to 0008H and then returns to the main program.

A restart instruction is a special kind of call because it branches to a predetermined address. Table 14-1 includes the op code for the restart instructions. Notice that only 1 byte is required to code a restart instruction. The standard CALL uses three bytes; therefore, an RST instruction is an efficient way to call frequently used subroutines.

Vectored Calls

The word *vector* implies direction. The RST instructions are like vectors because they point to specific locations in memory. The starting address of each restart subroutine is called a *vector location*. RST 0 points to vector location 0000H, RST 1 points to vector location 0008H, and so on.

Notice that there are only 8 bytes from 0000H to 0007H, 0008H to 000FH, 0010H to 0017H, and so on. Most useful subroutines require a lot more than 8 bytes. For this reason, you rarely see subroutines stored in the restart locations. Instead, most programmers use the vector locations for the starting addresses of longer subroutines.

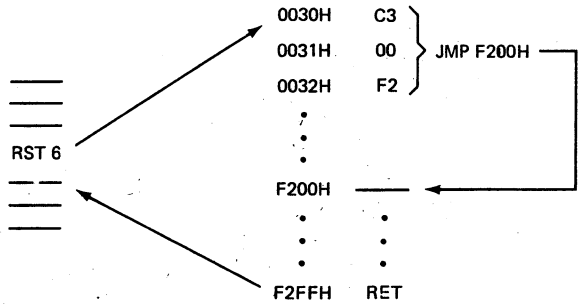


Fig. 14-5 Vector location contains JMP op code.

An Example

Suppose a subroutine with 256 bytes is stored at addresses F200H to F2FFH. If this subroutine is used a great deal, we can call it efficiently by using a RST 6 with a JMP F200H stored at address 0030H. The RST 6 takes the program to address 0030H; then the JMP F200H takes the program to the subroutine.

Figure 14-5 illustrates the program flow. When the program encounters the RST 6, it branches to address 0030H. Here it finds a JMP F200H. This produces a branch to the subroutine located at F200H. The long subroutine is then executed with the final instruction RET taking the processing back to the main program.

14-3 INTERRUPTS

Some pins on the 8085 allow peripheral equipment to interrupt the main program for I/O operations. When an interrupt occurs, the 8085 completes the instruction it is currently executing. Then it branches to a subroutine that *services* the peripheral device. Upon completion of the service subroutine, the CPU returns to the main program.

This type of I/O operation is called *interrupt-driven* I/O. It is more efficient than programmed I/O because the CPU does not wait for a high status signal. Instead, the CPU can process data while the peripheral device is getting ready for an I/O transfer.

Hardware Restarts

RST 0 to RST 7 are software restarts because they are instructions. Besides these software restarts, the 8085 has four *hardware restarts* designated TRAP (pin 6), RST 7.5 (pin 7), RST 6.5 (pin 8), and RST 5.5 (pin 9). When any of these pins is active, the internal circuits of the 8085 produce a hardware CALL to a predetermined vector location.

A hardware call like this is known as a *vectored interrupt* because the program branches to a vector location where the starting address of a service subroutine is stored. By connecting the hardware restart pins to peripheral devices, we can use interrupt-driven I/O instead of programmed I/O.

TABLE 14-2. RESTART LOCATIONS

Restart	Vector Location
RST 0	0000H
RST 1	0008H
RST 2	0010H
RST 3	0018H
RST 4	0020H
TRAP	0024H
RST 5	0028H
RST 5.5	002CH
RST 6	0030H
RST 6.5	0034H
RST 7	0038H
RST 7.5	003CH

Where are the vector locations for the hardware restarts? Exactly halfway between the software restart locations. As shown in Table 14-2, a TRAP restart sends the program to location 0024H, halfway between 0020H and 0028H. Similarly, the RST 5.5 sends the program to 002CH, which is halfway between 0028H and 0030H. RST 6.5 branches to 0034H, and RST 7.5 to 003CH.

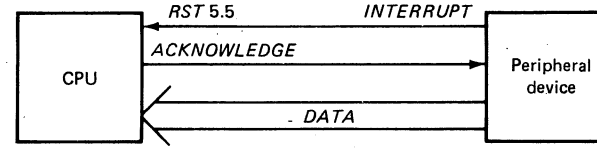


Fig. 14-6 Interrupt-driven I/O.

Interrupt-Driven I/O

Figure 14-6 illustrates the basic idea behind interrupt-driven I/O. When the peripheral device has a byte ready for data transfer, it sends a high bit to the RST 5.5 input. After saving the contents of the program counter in the stack, the CPU branches to vector location 002CH. Here it finds the starting address of the service subroutine; this subroutine inputs a byte from the peripheral device and stores it in memory.

After the byte has been stored, the CPU sends a high *ACKNOWLEDGE* bit to the peripheral device. This tells the peripheral device to get the next byte ready for transfer. Then the CPU returns to the main program where data processing continues. In this way, the CPU can process data in the main program rather than waiting for the peripheral device to get ready. When the peripheral device

TABLE 14-3. INTERRUPT PRIORITIES

Interrupt	Priority	Vector Location
TRAP	1	0024H
RST 7.5	2	003CH
RST 6.5	3	0034H
RST 5.5	4	002CH
INTR	5	None

has the next byte ready for transfer, it sends a high bit to the RST 5.5 input and the cycle repeats.

The advantage of interrupt-driven I/O is its efficiency. The CPU is no longer wasting 99.9 percent of its time waiting for the peripheral device to set up the next byte. The typical microcomputer uses interrupt-driven I/O because it has to process data while servicing a keyboard, a video display, and other peripheral devices.

Interrupt Priorities

Besides the hardware restarts, the 8085 also has an INTR interrupt, discussed later. Table 14-3 summarizes the 8085 interrupts. Notice that TRAP has the highest priority, RST 7.5 next highest, and so on. If two or more interrupts are active at the same time, the 8085 takes them in order of their priority level: TRAP is serviced first, then RST 7.5, and so forth.

TRAP

When an interrput pin goes high, the 8085 will complete the current instruction before looking at the pending interrupts. This means that a few microseconds may elapse from the time an interrupt pin goes high to the time the 8085 is aware of the interrupt. Because of this delay, you need to know how the different interrupts are activated.

As shown in Fig. 14-7, the 8085 is designed to respond to *edge triggering, level triggering,* or both. For instance, it takes a positive edge and a sustained high level to activate the TRAP interrupt. This means that *TRAP* must go high

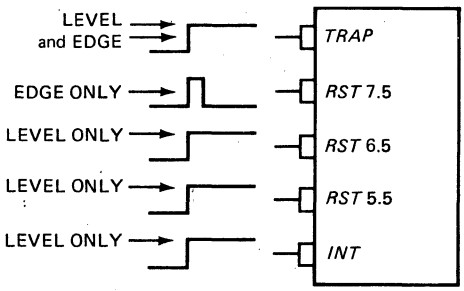


Fig. 14-7 Interrupt signals are edge- or level-sensitive.

and remain high until it is acknowledged. This avoids false triggering caused by noise and transients.

For example, suppose the 8085 is midway through an instruction cycle with another 2 μs to completion. If a 300-ns noise spike hits the TRAP input, it will edge-trigger but not level-trigger the TRAP interrupt because the 8085 is still working on the current instruction cycle. Because the TRAP input is both edge- and level-sensitive, the 8085 avoids responding to false TRAPS.

Since the TRAP input has the highest priority, it is used for catastrophic events such as power failures, parity errors, and other events that require immediate attention. In the case of brief power failures, it may be possible to save critical data. With parity errors, the data may be resampled or corrected before going on.

Other Interrupts

There are some applications where the peripheral device will send a pulse rather than a sustained high level. This is where the RST 7.5 interrupt is used. As shown in Fig. 14-7, it responds to edge triggering alone. The input pulse sets an internal flip-flop whose output is then sampled by the 8085 interrupt circuits. (False triggering is possible with the RST 7.5 input.)

The other interrupts are level-triggered; they must remain high until acknowledged.

14-4 INTERRUPT CIRCUITS

To get a better understanding of how interrupts work, look at Fig. 14-8. To begin with, notice the input waveforms that activate the interrupts: a positive edge and a sustained high level for the *TRAP*, the positive edge of a pulse for the RST 7.5, and sustained high levels for the RST 6.5 and RST 5.5.

TRAP

In Fig. 14-8 the positive edge of the *TRAP* signal (pin 6) will set the *D* flip-flop. Because of the AND gate, however, the final *TRAP* also depends on a sustained high-level TRAP input. This is why the TRAP is both edge- and level-sensitive.

The TRAP flip-flop can be cleared in either of two ways: a low *RESET IN* (system reset) or a high *TRAP ACKNOWLEDGE*. After the 8085 recognizes a TRAP interrupt, it will send a high *TRAP ACKNOWLEDGE* bit to the TRAP flip-flop; this clears it as a preparation for future TRAP interrupts.

RST 7.5

A pulse at pin 7 can activate the RST 7.5 interrupt because the positive edge will set the *D* flip-flop in Fig. 14-8. The

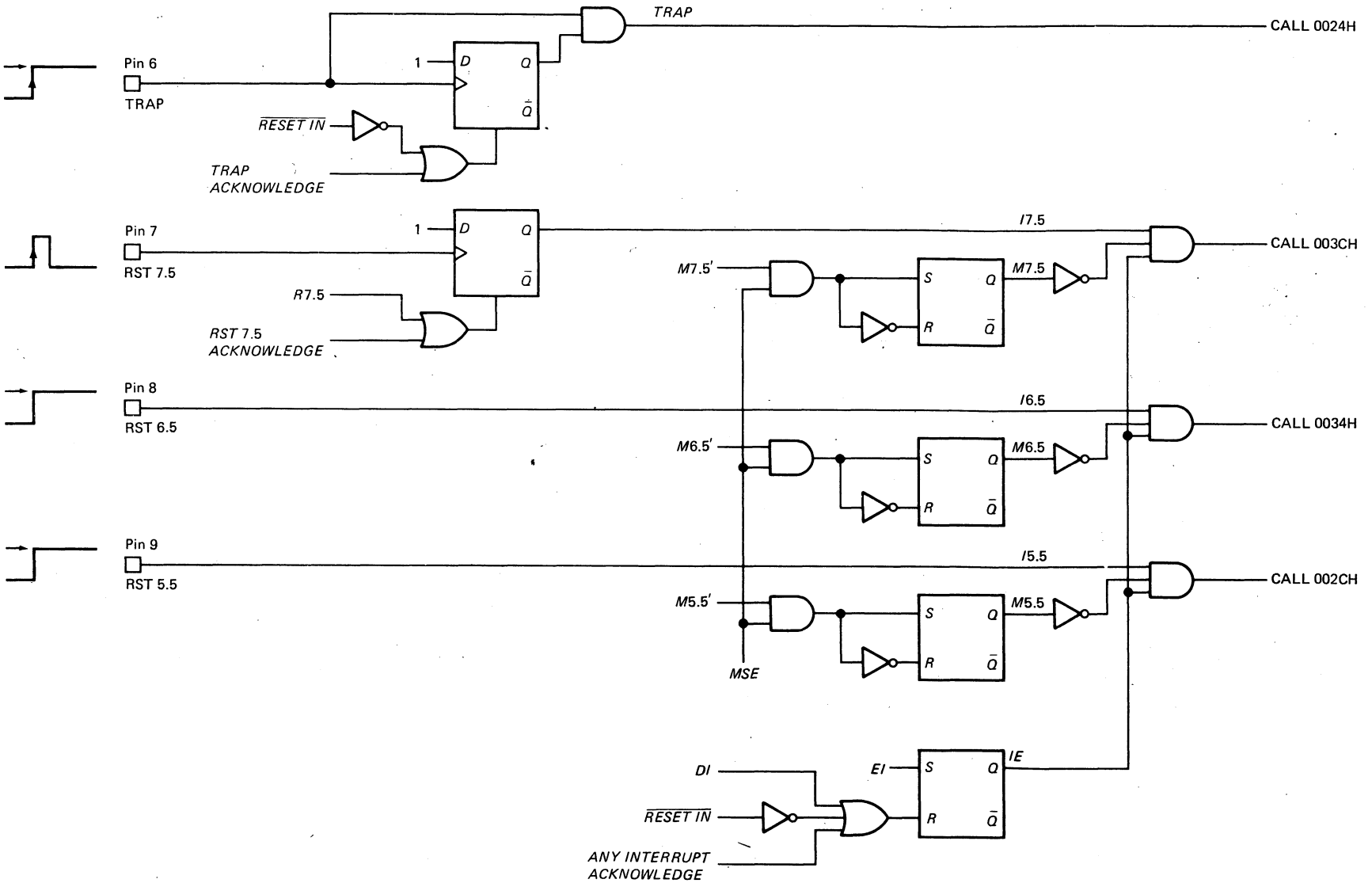


Fig. 14-8 Interrupt circuits.

output of this flip-flop is labeled *I7.5*. Notice that *I7.5* is one input to an AND gate.

One way to clear the RST 7.5 flip-flop is with a high *R7.5* bit, done with an instruction to be described later. Another way is with a high *RST 7.5 ACKNOWLEDGE*, internally produced as follows. After the 8085 recognizes a RST 7.5 interrupt, it sends a high *RST 7.5 ACKNOWLEDGE* bit to the second *D* flip-flop; this clears it for future RST 7.5 interrupts.

RST 6.5 and RST 5.5

The *RST 6.5* and *RST 5.5* inputs are connected directly to AND gates. Therefore, we need a sustained high level at pin 8 to enable the RST 6.5 AND gate and a sustained high level at pin 9 to enable the RST 5.5 AND gate.

Maskable Interrupts

The signals *I7.5, I6.5, I5.5,* and are called *pending interrupts*. The signal *IE* (bottom flip-flop) is called the *interrupt enable flag;* it must be high to activate the AND gates. Also notice the *M7.5, M6.5,* and *M5.5* signals; they must be low to enable the AND gates.

To activate the RST 7.5 interrupt, *I7.5* must be high, *M7.5* must be low, and *IE* must be high. Similarly, to get a high *RST 6.5, I6.5* must be high, *M6.5* low, and *IE* high. For the RST 5.5 interrupt to be active, *I5.5* must be high, *M5.5* low, and *IE* high.

The *M7.5, M6.5,* and *M5.5* signals are called *interrupt masks* because they can disable a pending interrupt. For example, if *M7.5* is high, it disables the AND gate it drives; this prevents a pending I7.5 interrupt from reaching the final output.

The RST 7.5, RST 6.5, and RST 5.5 interrupts are *maskable;* this means that they can be disabled by applying high *M7.5, M6.5,* and *M5.5* signals. The TRAP is *nonmaskable;* once it goes high and stays high, a TRAP interrupt appears at the final output.

14-5 INTERRUPT INSTRUCTIONS

Certain instructions are used with the interrupt circuits of Fig. 14-8. For instance, we might want to disable the interrupt system, or mask a particular interrupt, or examine pending interrupts, and so forth.

EI and DI

The 8085 has two instructions that can enable or disable all interrupts except the TRAP. The instruction

EI

stands for *enable interrupts*. When executed, this instruction will produce a high *EI* bit in Fig. 14-8 (bottom flip-flop). This sets the flip-flop and produces a high *IE* output.

The instruction

DI

stands for *disable interrupts*. When executed, it produces a high *DI* bit to the bottom flip-flop (Fig. 14-8). This clears the flip-flop and results in a low *IE*. The low *IE* then disables all interrupts except TRAP.

Besides the *DI* input, the OR gate has a $\overline{\text{RESET IN}}$ input and an ANY INTERRUPT ACKNOWLEDGE input. This means that the interrupts (except TRAP) are disabled by a system reset or by the acknowledgement of a previous interrupt. In other words, when the 8085 recognizes an interrupt, it produces a high *ANY INTERRUPT ACKNOWLEDGE* bit. This disables the interrupts and prevents a future interrupt (except TRAP) from interrupting a service subroutine.

Because the interrupts are automatically disabled by the *ANY INTERRUPT ACKNOWLEDGE* bit, the programmer usually includes an EI as the next to last instruction in the service subroutine. For instance, the last two instructions typically are

```
Subroutine:  ——
             ——
             ——
             EI
             RET
```

This subroutine cannot be interrupted (except by a TRAP). After the EI is executed, the processing returns to the main program with the interrupt system enabled.

The programmer who wants some critical part of the main program to run uninterrupted can use a DI at the beginning of the segment to be protected and an EI at the end:

```
Main program:  DI
               ——
               ——
               ——
               EI
```

This protects the program between the DI and EI because the interrupt system is disabled.

SIM

Here is another interrupt instruction:

SIM

SIM stands for *set interrupt mask*. To use this instruction, you first load the accumulator as shown in Fig. 14-9*a*.

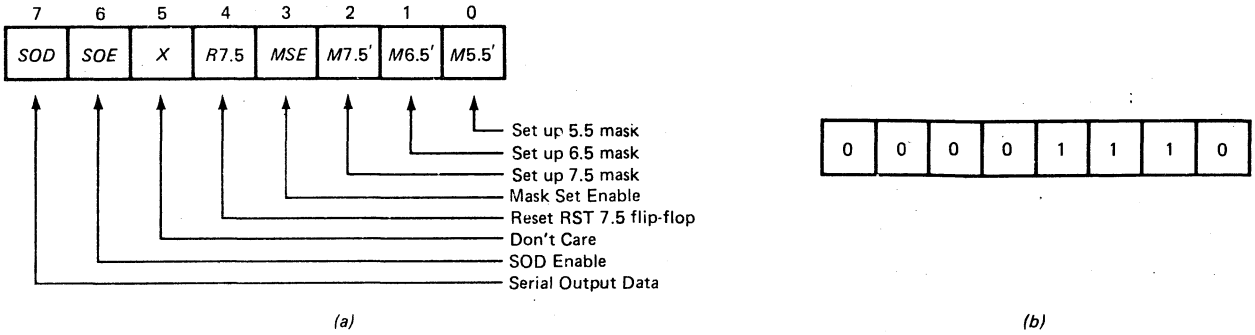


Fig. 14-9 Accumulator contents before executing SIM.

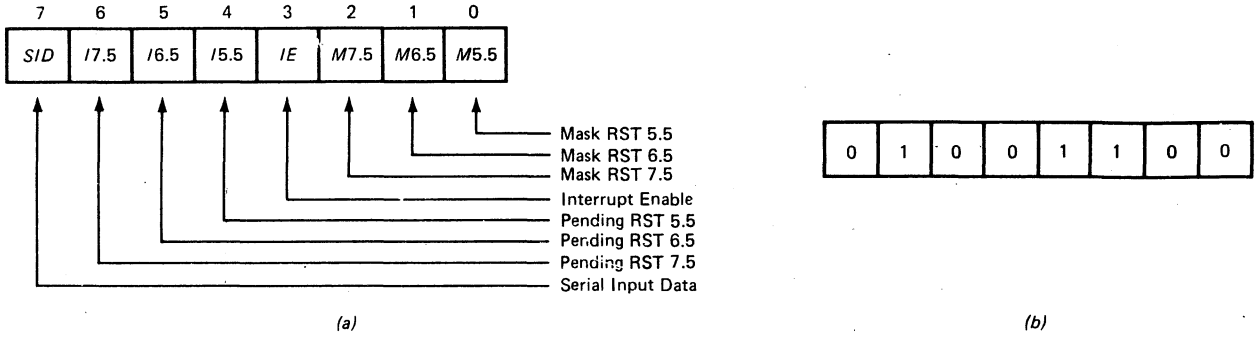


Fig. 14-10 Accumulator contents after executing RIM.

Then by executing a SIM the accumulator contents will be transferred to the appropriate locations.

The *R7.5* bit goes to the RST 7.5 flip-flop of Fig. 14-8; when high, this bit clears a pending I7.5 interrupt. The *MSE* (mask set enable) bit conditions the AND gates of Fig. 14-8; when high, it permits masking bits *M7.5'*, *M6.5'*, and *M5.5'* to set or reset the masking flip-flops.

Here is an example. Suppose we want to mask (disable) the RST 7.5 and RST 6.5 interrupts and unmask (enable) the RST 5.5 interrupt. Then, we can use

```
MVI A,0EH
SIM
```

After the MVI is executed, the accumulator contents appear as shown in Fig. 14-9*b*. Here you see a high *MSE*, high *M7.5'*, and high *M6.5'*; all other bits are low. The SIM then transfers these bits to the appropriate locations, shown in Fig. 14-8. (The SOD and SOE will be explained later.) The high *MSE* allows *M7.5'* and *M6.5'* to set their flip-flops; this produces high masking bits *M7.5* and *M6.5*, disabling the final AND gates and preventing interrupts I7.5 and I6.5 from arriving at the final outputs.

RIM

RIM stands for *read interrupt mask*. When executed, it loads the accumulator with the bits shown in Fig. 14-10*a*. Bit 7 is the serial input data (SID). Bits 6, 5, and 4 are the pending interrupts. Bit 3 is the interrupt-enable bit *IE*. Bits 2, 1, and 0 are the interrupt masks. Execution of a RIM allows the programmer to examine the status of the pending interrupts, masks, and the like. This may be necessary following an interrupt service subroutine.

For instance, suppose the accumulator contains 4CH after a RIM is executed. As shown in Fig. 14-10*b*, *I7.5*, *IE*, and *M7.5* are high. This means that a 7.5 interrupt is pending, the interrupt system is enabled, and the RST 7.5 interrupt is currently masked.

EXAMPLE 14-2

C3H is the op code for JMP. Figure 14-11 shows some starting addresses for interrupts. Identify the starting address of each service subroutine.

SOLUTION

The TRAP interrupt vectors to 0024H; therefore, the

```
C3H
00H
F0H
```

produces a JMP (C3H) to the service subroutine starting at F000H.

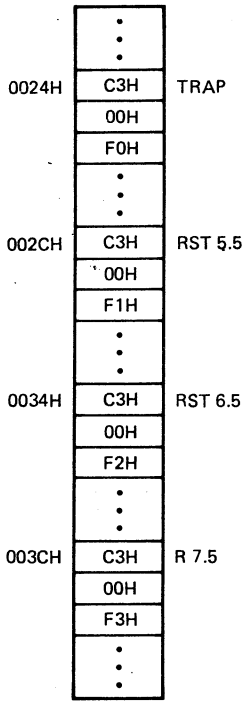


Fig. 14-11 Vector locations with subroutine starting addresses.

Likewise, the RST 5.5 service subroutine starts at F100H, the RST 6.5 at F200H, and the RST 7.5 at F300H.

EXAMPLE 14-3

Figure 14-12 shows a peripheral device connected to the RST 5.5 interrupt. After the CPU receives a data word in port 12H, it can send a high *ACKNOWLEDGE* bit (bit 7 of port 11H) back to the peripheral device.

The starting address in the RST 5.5 vector location is F100H. Show a service subroutine that inputs data from the peripheral device and stores the data at 3000H.

SOLUTION

Address	Mnemonic	Comment
F100H	PUSH PSW	;Save accumulator and flags
F101H	PUSH H	;Save HL contents
F102H	IN 12H	;Input data from device
F104H	LXI H,3000H	;Set pointer
F107H	MOV M,A	;Store data
F108H	MVI A,80H	;Set *ACKNOWLEDGE* bit
F10AH	OUT 11H	;Acknowledge data arrival
F10CH	POP H	;Restore HL contents
F10DH	POP PSW	;Restore accumulator and flags
F10EH	EI	;Enable interrupts
F10FH	RET	;Return

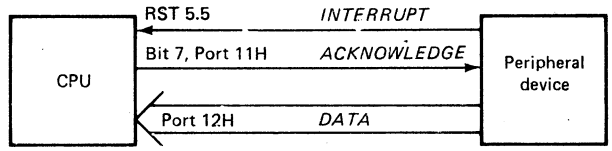


Fig. 14-12 Interrupt-driven I/O example.

When the peripheral device is ready for data transfer, it sends a high bit to the RST 5.5 input. After the 8085 recognizes this interrupt, it branches to vector location 002CH. Here it finds a JMP F100H. The jump takes the program to the starting address of the service subroutine.

The service subroutine usually destroys the contents of the accumulator and HL register. For this reason, the subroutine starts with a PUSH PSW and a PUSH H; this saves the accumulator contents, flags, and HL contents in the stack.

Next, the IN 12H inputs a data word from port 12H. After the HL pointer is set to 3000H, the data is stored at location 3000H. The next two instructions send a high *ACKNOWLEDGE* bit to the peripheral device.

The POP H and POP PSW restore the contents of the HL register, accumulator, and flag register. Because the stack operates as a first-in last-out memory, we pop in the reverse order that we pushed.

Finally comes the EI to enable the interrupts and the RFT to get us back to the main program.

By modifying this subroutine, we can store bytes in successive memory locations. For instance, using an INX H and some other instructions, we can update the HL pointer each time the subroutine is called. In this way, the incoming data words will be stored at 3000H, 3001H, 3002H, and so on.

14-6 SERIAL INPUT AND SERIAL OUTPUT

The 8085 has a SID (serial input data) pin. You can use this input to receive serial data from a peripheral device. The RIM instruction reads the interrupt mask into the accumulator (see Fig. 14-10). Bit 7 is the serial input data bit. This bit has nothing to do with the interrupt system; it is included in the RIM instruction to avoid having to include an extra instruction for SID operations.

Each time a new bit arrives at the SID input, we can execute a RIM instruction. By isolating and saving this bit, we can convert a serial data stream into a parallel 8-bit word (see Example 11-21 for programmed I/O). If interrupt-driven I/O is used, a service subroutine is called each time a new bit is at the SID input. This service subroutine would include a RIM plus rotate and store instructions for serial-to-parallel conversion.

The SOD output pin can deliver a serial data stream to a peripheral device. The SIM instruction sets the interrupt mask as shown earlier (Fig. 14-9). Bit 7, *SOD*, is latched

into the SOD output pin only if bit 6, *SOE* (SOD enable), is high. In other words, bit 6 acts like a switch for bit 7.

As an example, if want to send a high bit to the SOD output pin, we can use

```
MVI A,C0H
SIM
```

The MVI sets bits 7 and 6. The SIM then latches bit 7 into the SOD output pin. To send a low bit to the SOD output, we can use

```
MVI A,40H
SIM
```

By using rotate and other instructions we can write a program that converts an 8-bit parallel word into a serial data stream at the SOD output. (See Example 11-14 for the basic idea behind parallel-to-serial conversion.) With interrupt-driven I/O, the service subroutine would include a SIM, rotates, and other instructions for parallel-to-serial conversion.

14-7 EXTENDING THE INTERRUPT SYSTEM

The TRAP, RST 7.5, RST 6.5, and RST 5.5 give us four interrupt-request lines. Sometimes, we need more than this. One way to extend the interrupt system is with the INTR input.

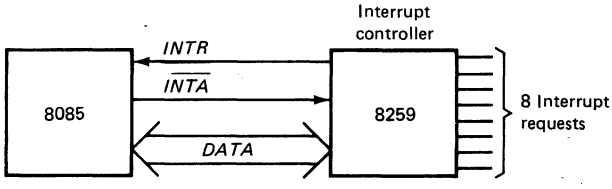


Fig. 14-13 The INTR interrupt extends interrupt capability.

Interrupt Controller

In Fig. 14-13 the 8259 receives interrupt requests from eight peripheral devices. Known as an *interrupt controller*, the 8259 stores the starting addresses of eight service subroutines, one for each of the peripheral devices. To service an interrupt, it sends a

CALL address

instruction to the 8085. This calls the service subroutine for the interrupting peripheral device.

Sending the CALL

Here are the details of how the CALL instruction is sent. When a peripheral device needs service, the 8259 sends a high bit to the INTR input. As soon as the 8085 recognizes a high *INTR*, it returns a low $\overline{INTA}$ (interrupt acknowledge) to the 8259. The 8259 responds by sending back the op code of a call.

Next, the 8085 sends another low $\overline{INTA}$. This $\overline{INTA}$ fetches the low address byte from the 8259. Finally, the 8085 sends another low $\overline{INTA}$. This third $\overline{INTA}$ fetches the high address byte.

An Example

Suppose the service subroutine for a peripheral device starts at address F400H. When this device needs service, the 8259 sends a high *INTR* to the 8085. After this interrupt is recognized, the 8085 sends back a low $\overline{INTA}$. This low $\overline{INTA}$ causes the 8259 to send

CDH

along the data bus. This is the op code for a CALL; it is stored in the 8085 instruction register.

Again, the 8085 sends a low $\overline{INTA}$. In response, the 8259 returns the low byte of the address:

00H

This low byte is stored in the Z register.

For a third time, the 8085 sends a low $\overline{INTA}$. The 8259 then returns the high byte of the address:

F4H

This high byte is stored in the W register. The instruction

CALL F400H

is now in the 8085. After the contents of the program counter are saved in the stack, the program branches to service subroutine.

Initializing

The eight starting addresses of the service subroutines are stored in the 8259 during initialization of the system. In other words, when you power up the system, the program counter is reset to 0000H. The early instructions in the program initialize the different chips like the 8156, 8355, 8259, etc. Initializing the 8259 means sending the starting addresses of the service subroutines to 8259; it has eight internal registers for storing these addresses. After the initialization is completed, the 8259 is ready to accept interrupt requests from the peripheral devices.

14-8 DIRECT-MEMORY ACCESS

A *floppy disk* is a thin plastic disk about 8 inches in diameter, coated with magnetic oxide. A *disk drive* is a peripheral device that can either read or write data on the disk, which can store a half million or more bytes. The only practical way to transfer data to and from the disk is with *direct memory access* (DMA). As described earlier, the 8085 can turn over control of its buses to a DMA controller for high-speed I/O transfers. In this way, large amounts of data can be transferred in a relatively short time.

Accumulator in the Middle

The details of DMA transfer are too complicated to go into here, but we can discuss the basic idea. The IN instruction is the usual way to input data from peripheral devices. The accumulator is involved because it receives the input data. Similarly, the OUT instruction transfers data from the accumulator to output devices. In either case, the accumulator serves as go-between.

One way to transfer data from the memory to peripheral devices is to use MOV and I/O instructions. For instance, to move 256 bytes from memory to an output device, we can use a loop that includes MOV A,M and OUT instructions. This approach will work, but it is too slow when large amounts of data are involved.

The Problem

The foregoing approach is slow for two reasons. First, the accumulator acts as a halfway station in each transfer of data from memory to I/O, or vice versa. Second, the 8085 is microprogrammed, which means that the microinstructions have to be read from a control ROM. The access time of this control ROM slows things down.

Basic Idea

DMA data transfers are faster because the accumulator is eliminated as a halfway station; the data goes directly from the memory to the peripheral device or vice versa. Also, the DMA controller has hardwired control instead of microprogramming. This eliminates the access time of the control ROM.

The *HOLD* and *HLDA* signals are used in DMA operations. In Fig. 14-14, when the DMA controller is ready to take over control, it sends a high *HOLD* signal to the 8085. The 8085 then three-states (floats) its address, data, and control buses. It also sends a high *HLDA* (hold acknowledge) to the DMA controller, indicating that it has turned over control. The DMA controller carries out the data transfers at a high speed and then returns control to the 8085 by sending back a low *HOLD* signal.

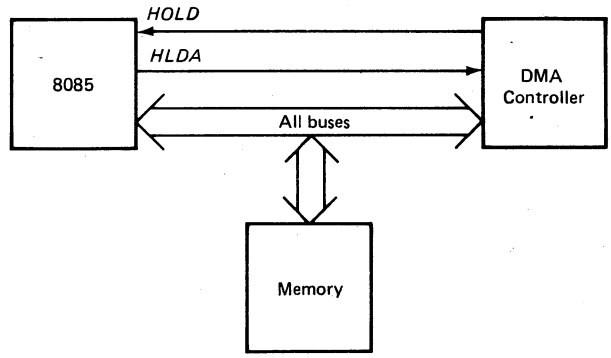


Fig. 14-14 Direct memory access.

GLOSSARY

DMA Direct memory access, the fastest type of I/O operation. The CPU passes control of its buses to a DMA controller, a chip optimized for high-speed data transfers from peripheral devices to memory and vice versa.

floppy disk A thin plastic disk about 8 inches in diameter and coated with magnetic oxide. It can store half a million or more bytes. A smaller version, called the mini floppy disk, is about 5 inches in diameter.

interrupt controller A peripheral chip used with the INTR interrupt to accommodate more interrupts. The 8259 handles interrupt requests from up to eight peripheral devices.

interrupt-driven I/O A type of I/O transfer that relies on both hardware and software. The CPU never waits on the peripheral device. Instead, the peripheral device sends an interrupt request when it is ready for servicing. After the CPU recognizes this interrupt, it calls a subroutine that services the peripheral device.

programmed I/O Input and output data transfers that rely exclusively on software. The CPU waits until a high status bit is received from the peripheral device. While waiting, the CPU wastes time, which is acceptable in simpler applications. Programmed I/O is also called polled I/O.

restart A special type of CALL in which the address is not programmed but built into the hardware. The 8085 has eight software restarts (RST 0 to RST 7) and four hardware restarts (TRAP, RST 7.5, RST 6.5, and RST 5.5).

SID Serial input data. The SID input of the 8085 can be loaded into bit 7 of the accumulator by executing a RIM instruction. With AND masks and rotates. a serial data stream into the SID input can be converted to a parallel 8-bit word.

SOD Serial output data. A SIM instruction latches bit 7 of the accumulator into the SOD output pin if the *SOE* (bit 6) is high.

start bit A bit used in programmed I/O. The CPU sends a high *START* bit to the peripheral device to begin an I/O transfer.

status bit One of the handshaking bits used in programmed I/O. It indicates that the peripheral device has data ready to send in an input operation or has received the data in an output operation.

TRAP The highest-priority interrupt, typically used for catastrophic events like temporary power failure, bus errors, and the like. Because of its critical nature TRAP is nonmaskable and can interrupt a service subroutine.

vector location The address to which a software or hardware restart branches. Usually the vector location and the next two memory locations contain a JMP address instruction that takes the processing to a long subroutine.

vectored interrupt An interrupt that takes the program to a vector location (a preassigned address). TRAP vectors to 0024H, RST 7.5 to 003CH, RST 6.5 to 0034H, and RST 5.5 to 002CH.

SELF-TESTING REVIEW

Read each of the following and provide the missing words. Answers appear at the beginning of the next question.

1. With _____ I/O the CPU has to wait for the peripheral device to send a high status bit. This waste of CPU time may be acceptable in some applications where the _____ has nothing better to do.
2. (*programmed, CPU*) The 8085 has eight _____ instructions: RST 0, RST 1, RST 2, RST 3, RST 4, RST 5, RST 6, and RST 7. Each is a special CALL to a preassigned address. This address is called a _____ location. Usually, the vector location and the next two memory locations contain a JMP instruction. This allows the program to branch to a longer _____.
3. (*restart, vector, subroutine*) The 8085 has four hardware restarts: TRAP, RST 7.5, RST 6.5, and RST 5.5. These are called _____ interrupts because they point to preassigned vector locations.
4. (*vectored*) TRAP has the _____ priority. It requires both edge and level triggering. This means that the TRAP input must go high and stay _____ until recognized. The RST 7.5 interrupt is edge-triggered only because it may be a _____. The RST 6.5 and RST 5.5 are level-triggered.
5. (*highest, high, pulse*) TRAP is nonmaskable. RST 7.5, RST 6.5, and RST 5.5 are _____; this means that we can disable these interrupts individually. To disable the whole interrupt system (except TRAP), the _____ instruction may be used.
6. (*maskable, DI*) When any interrupt is recognized, the 8085 automatically _____ the interrupt system (except for TRAP). This prevents an incoming interrupt from interrupting a service subroutine. The programmer must use an _____ instruction before the RET to enable the interrupt system when it returns to the main program.
7. (*disables, EI*) The SIM instruction allows us to _____ the interrupt mask. The RIM instruction lets us read the _____ mask. Besides setting and reading bits in the interrupt system, SIM and RIM are used for serial input data (SID) and serial output data (SOD).
8. (*set, interrupt*) An interrupt controller stores the starting addresses of service _____. When a peripheral device needs service, the interrupt controller sends a high *INTR* to the 8085, which returns three _____ $\overline{INTA}$s. In response to each of these low $\overline{INTA}$s, the interrupt controller sends back the op code of a _____, the low byte of the starting address, and the high byte of the starting address.
9. (*subroutines, low, CALL*) A _____ disk is a thin plastic disk coated with magnetic oxide. A disk driver is a peripheral device that can either read or write data on the disk. The only practical way to transfer data to and from a floppy disk is with _____ memory access (DMA).
10. (*floppy, direct*) DMA transfers are fast because the accumulator is not used during data transfers from memory to peripherals and vice versa. Also, the DMA controller has hardwired control instead of microprogramming. This eliminates the access time of the control ROM and allows the transfers to take place at much higher hardware speed.

PROBLEMS

14-1. In the input program of Sec. 14-1, what change is necessary if we want to store the incoming data at addresses 8000H to 80FFH?

14-2. How can we input and store 100 bytes instead of 256 bytes in the preceding problem?

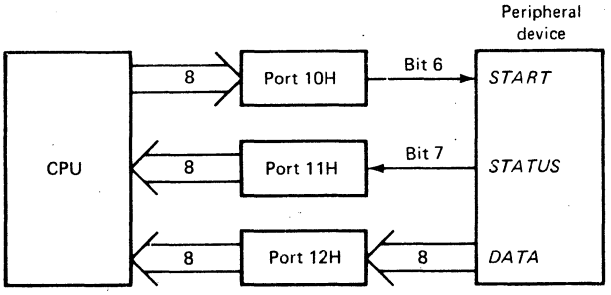


Fig. 14-15

14-3. Instead of the handshaking bits shown in Fig. 14-15, we prefer to use bit 0 of port 10H for the *START* bit, and bit 1 of port 11H for the *STATUS* bit. Write a program that inputs and stores 256 bytes of data at addresses 5000H to 50FFH.

14-4. Write a program that inputs and stores 512 bytes at addresses 5000H to 51FFH. Use the handshaking bits shown in Fig. 14-15.

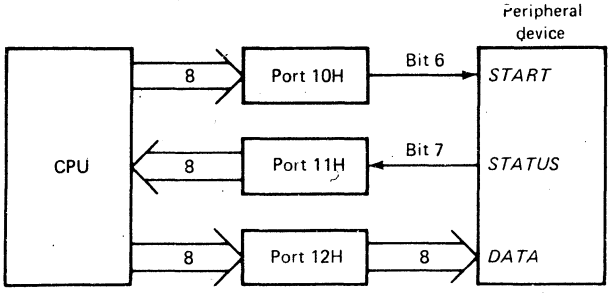


Fig. 14-16

14-5. Write a program that transfers 1,024 bytes from addresses 8000H to 83FFH to a peripheral device. Use the same handshaking bits as Fig. 14-16.

14-6. Here is a program segment:

```
LXI H,3000H
RST 5
MOV C,A
```

What is the vector location for the second instruction? What is the op code you almost always find in this location?

14-7. The CPU is processing the main program. At the end of an instruction cycle, it finds that pending interrupts *I7.5*, *I6.5*, and *I5.5* are all high in Fig. 14-17. Which interrupt does the CPU service first for each of the following:

a. *IE* is high, *M7.5* is high, *M6.5* is low, and *M5.5* is low.
b. *IE*, *M7.5*, *M6.5*, and *M5.5* are all high.
c. *IE* is high. *M7.5*, *M6.5* and *M5.5* are all low.
d. *IE* is low.

14-8. In Fig. 14-17, *M7.5*, *M6.5*, and *M5.5* are all low. What happens to interrupt masks for each of the following:

a. *M7.5'*, *M6.5'*, *M5.5'*, and *MSE* are low.
b. *M7.5'* and *M5.5'* are low. *M6.5'* and *MSE* are high.
c. *M7.5'*, *M6.5'*, *M5.5'*, and *MSE* are high.

14-9. Here are some initializing instructions:

```
MVI A,1DH
SIM
```

After the SIM is executed, which are the interrupts that are masked?

14-10. Write two initializing instructions like those of the preceding problem to mask RST 5.5 and RST 6.5. All unnecessary and don't care bits should be set to 0.

14-11. Here is how a service subroutine ends:

```
___
___
___
RIM
EI
RET
```

Suppose the accumulator contains CAH after the RIM is executed. Answer the following:

a. Is the serial input data low or high?
b. Which are the pending interrupts?
c. Is the interrupt-enable flag low or high?
d. Which interrupts are masked?

14-12. An RST 6.5 service subroutine looks like this:

```
PUSH PSW
PUSH B
PUSH D
PUSH H
MVI A,0AH
SIM
EI
___
___
___
RET
```

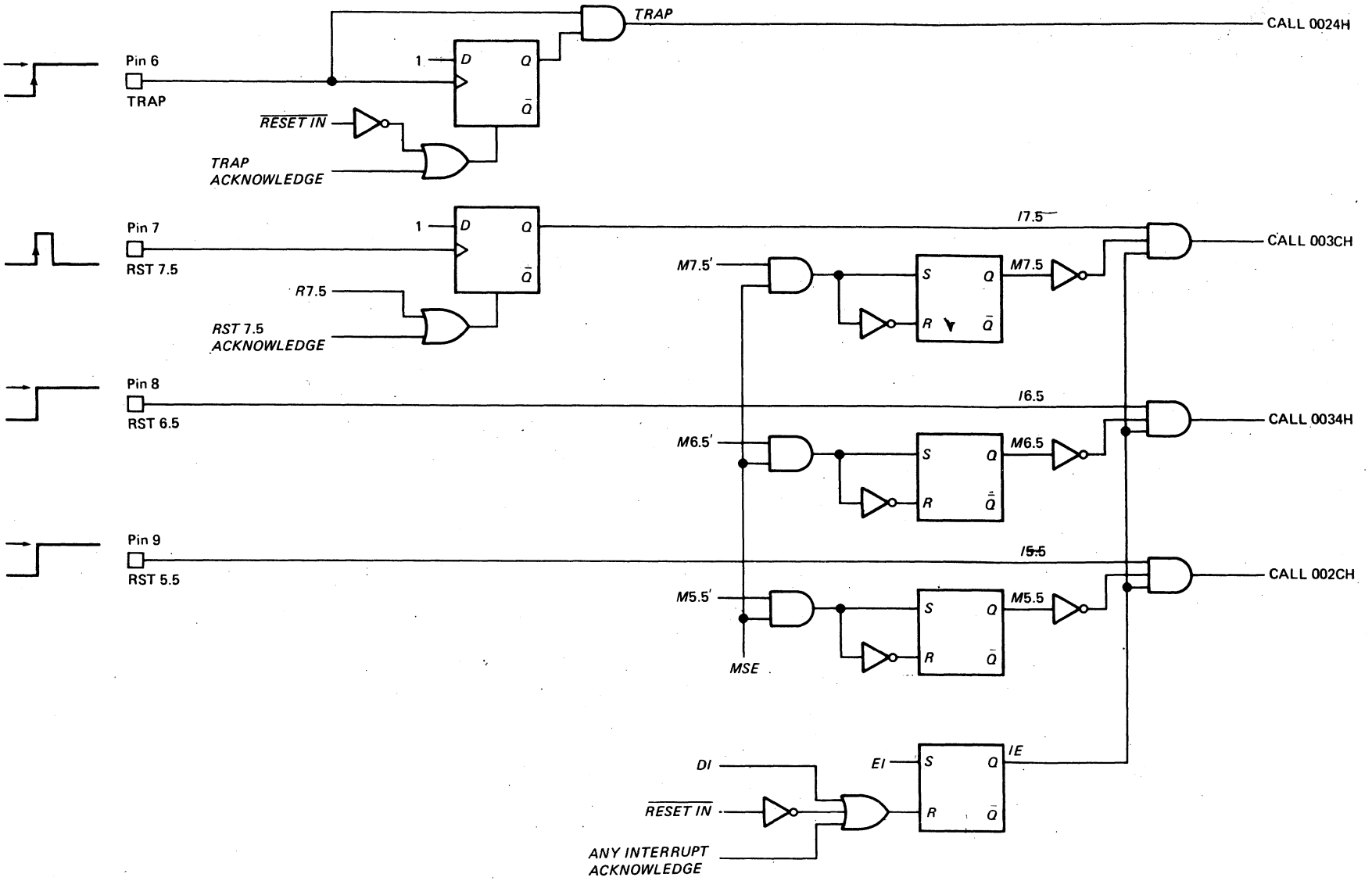

Pin 6
TRAP
1
D
Q
$\bar{Q}$
TRAP
RESET IN
TRAP
ACKNOWLEDGE
CALL 0024H
Pin 7
RST 7.5
1
D
Q
$\bar{Q}$
R7.5
RST 7.5
ACKNOWLEDGE
I7.5
M7.5'
S
Q
R
$\bar{Q}$
M7.5
CALL 003CH
Pin 8
RST 6.5
I6.5
M6.5'
S
Q
R
$\bar{Q}$
M6.5
CALL 0034H
Pin 9
RST 5.5
I5.5
M5.5'
S
Q
R
$\bar{Q}$
M5.5
MSE
CALL 002CH
DI
EI
S
Q
IE
RESET IN
R
$\bar{Q}$
ANY INTERRUPT
ACKNOWLEDGE

Fig. 14-17

a. Why are the PUSH instructions used?
b. After the SIM is executed, which are the interrupts that can interrupt the service subroutine?

14-13. Here is a program for serial input data:

Label	Mnemonic	Comment
	MVI B,00H	;Clear B register
	MVI C,08H	;Present counter to 8
LOOP:	RIM	;Get SID bit
	ANI 80H	;Isolate SID bit
	ORA B	;Update parallel word
	RRC	;Rotate right
	MOV B,A	;Save accumulator
	DCR C	;Count down
	JNZ LOOP	;Go back if not finished
	RLC	;Rotate left
	HLT	

The program loops 8 times. The successive SID bits after each RIM is executed are 1, 1, 0, 0, 0, 0, 1, and 0. Answer the following:

a. What does the B register contain after the first MOV B,A has been executed?
b. What does the B register contain after the second MOV B,A has been executed?
c. What are the contents of the accumulator after the RLC has been executed? What letter is this in ASCII code?
d. Was the SID data received LSB first or last?

14-14. Here is a program:

Label	Mnemonic	Comment
	MVI A,4DH	;Load ASCII M
	MVI C,08H	;Present counter to 8
LOOP:	RRC	;Rotate LSB into MSB
	MOV B,A	;Save accumulator
	ANI 80H	;Isolate SOD bit
	ORI 40H	;Enable SOE bit
	SIM	;Latch SOD bit
	MOV A,B	;Restore accumulator
	DCR C	;Count down
	JNZ LOOP	;Loop if not finished
	HLT	

Answer the following:

a. How many times does the processing pass through the loop?
b. Is the ASCII letter M being sent LSB first or last?
c. After the first ORI C0H is executed, what does the accumulator contain?
d. After the MOV A,B is executed for the last time, what does the accumulator contain?

Support Chips

15

This chapter continues our discussion of the 8156 and the 8355. Besides its 256-byte RAM, the 8156 contains three I/O ports. In addition to its 2K ROM, the 8355 has two I/O ports. We want to find out how to use these I/O ports.

Also included in this chapter are ways to create new ports, gate addressing, decoder addressing, and other I/O topics. Finally, we will look at how to expand the memory with 2114s, 1K static RAMs that have become industry standards.

15-1 THE 8156

As discussed in Chap. 13, the 8156 contains a 2,048-bit RAM organized as 256 words of 8 bits each. In a minimum system, A_{13} is connected to the CE input. This produces RAM locations of 2000H–20FFH.

The 8156 also has three I/O ports: port A, port B, and port C. To read from or write into these ports, *IO/$\overline{M}$* must be high. Each port has an internal register (PA, PB, or PC) that latches data during I/O operations. Ports A and B can each latch an 8-bit word. Port C, however, can only latch a 6-bit word. This is acceptable because port C is typically used for serial I/O or for handshaking where fewer than 8 bits are needed.

Figure 15-1 shows the pinout diagram of an 8156. Most of the pins are self-explanatory. For instance, port A has a data bus PA_7 through PA_0 (pins 28 to 21). Likewise, port B has a data bus PB_7 through PB_0 (pins 36 through 29). Port C, which has only a 6-bit data bus PC_5 through PC_0, uses pins 5, 2, 1, 39, 38, and 37.

The address-data bus, AD_7–AD_0, uses pins 19 to 12. Recall also the control signals *RESET* (pin 4), *IO/$\overline{M}$* (pin 7), *CE* (pin 8), *$\overline{RD}$* (pin 9), *$\overline{WR}$* (pin 10), and *ALE* (pin 11). These were discussed in Chap. 13 along with V_{SS} (pin 20, ground) and V_{CC} (pin 40, supply voltage). The only new signals are *TIMER IN* (pin 3) and *$\overline{TIMER\ OUT}$* (pin 6); they are discussed in Sec. 15-4.

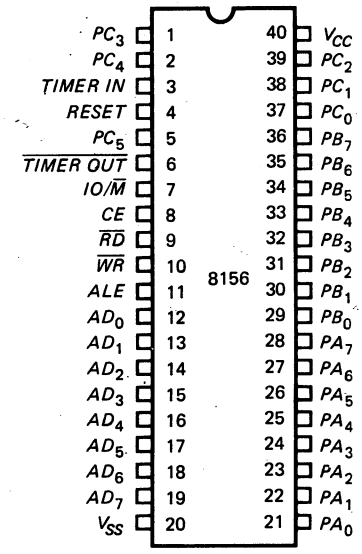


Fig. 15-1 8156 pinout.

15-2 PORT NUMBERS FOR THE 8156

When using the IN and OUT instructions, we have to include an immediate byte that specifies the port number. For instance, OUT 21H latches the accumulator contents into port 21H. Since the port numbers can vary from 00H to FFH, the 8085 can control up to 256 ports.

Duplication of Port Number

When an IN or OUT instruction is executed, the 8085 duplicates the port number on the address bus and on the address-data bus. For instance, during the execution of

IN 21H

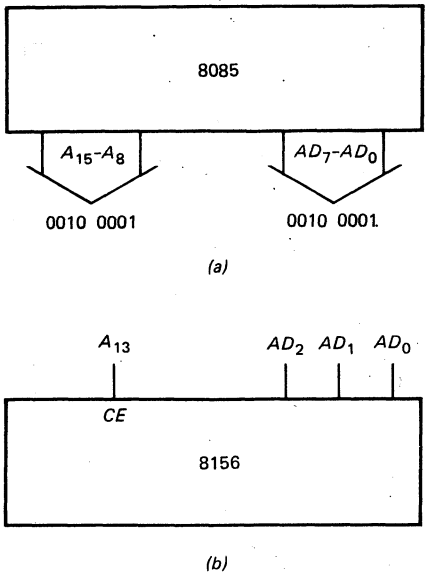


Fig. 15-2 (*a*) Port number is duplicated on buses; (*b*) active bits during I/O addressing.

the 8085 sends out

0010 0001

on its address bus and address-data bus, as shown in Fig. 15-2*a*.

As an equation,

$$A_{15}–A_8 = AD_7–AD_0 \qquad (15\text{-}1)$$

This says the upper address bits equal the lower address bits during I/O operations. Remember this duplication; it's important.

I/O Addresses

Besides ports A, B, and C, the 8156 contains a command register, a status register, and a timer. The command register determines whether the ports act like input ports or output ports. The status register contains information about the ports. The timer is a 14-bit down counter used for counting input pulses. Later sections discuss the command register, status register, and timer in detail.

During the execution of IN and OUT instructions, the 8156 uses only the three lower address bits (AD_2, AD_1, and AD_0). Table 15-1 shows the bit combinations for each internal register: 000 addresses the command-status registers, 001 addresses port A, 010 addresses port B, 011 addresses port C, 100 addresses the lower 8 bits of the timer count, and 101 addresses 2 bits of timer mode and 6 bits of timer count.

In other words, during I/O operations involving the 8156,

TABLE 15-1. I/O ADDRESSES

AD_2	AD_1	AD_0	Location
0	0	0	Command and status registers
0	0	1	Port A
0	1	0	Port B
0	1	1	Port C
1	0	0	Lower 8 bits of timer
1	0	1	2 bits of timer mode and upper 6 bits of timer

$AD_2AD_1AD_0$ is between 000 and 101. This means that the word on the address-data bus is from

$$AD_7–AD_0 = XXXX\ X000$$

to

$$AD_7–AD_0 = XXXX\ X101$$

Notice the don't cares. During I/O operations, the 8156 disregards bits AD_7, AD_6, AD_5, AD_4, and AD_3, which can therefore be 0s or 1s.

Duplication Equation

What is the port number of port A in a minimum system? To begin with, A_{13} must be high to enable the chip (see Fig. 15-2*b*). Furthermore, $AD_2AD_1AD_0$ must be 001 to address port A. Because both buses transmit the same port number during I/O operations, we can write

$$A_{15}–A_8 = AD_7–AD_0$$
$$XX1X\ XXXX = XXXX\ X001$$

On the left side of this equation, A_{13} is high and all other bits are don't cares. On the right side of the equation, $AD_2AD_1AD_0$ are 001; all other bits are don't cares.

To have equality, 0s and 1s on one side of the equation must appear on the other side. This means that $A_{10}A_9A_8$ must equal 001 and AD_5 must equal 1 to get

$$XX1X\ X001 = XX1X\ X001$$

Now both sides of the equation are equal; therefore, the port number is

$$\text{Port number} = XX1X\ X001 \qquad (15\text{-}2)$$

Since the remaining don't cares can be 0s or 1s, there are many solutions to Eq. 15-2. For instance, if all the remaining don't cares are set equal to zero,

$$\text{Port number} = 0010\ 0001$$

which is equivalent to 21H.

TABLE 15-2. 8156 PORT NUMBERS IN MINIMUM SYSTEM

Port Number	Selected Register
20H	Command-status register
21H	Port A
22H	Port B
23H	Port C
24H	Lower byte of timer count
25H	Timer mode and count (upper 6 bits)

Shadows

By checking other don't-care combinations, you will find these additional solutions to Eq. 15-2: 29H, 31H, 39H, . . ., F9H. In other words, the port addressing folds back and produces shadows at 29H, 31H, and so on.

Whenever a port number has shadows, the programmer almost always uses the lowest port number. This is why port A has a port number of 21H in our minimum system.

Other Port Numbers

By solving the duplication equation for the other registers in an 8156 we can arrive at the port numbers listed in Table 15-2. (Each of these port numbers has shadows which are not used.) As you see, port 20H is for the command-status registers, 21H for port A, 22H for port B, 23H for port C, and so on.

15-3 PROGRAMMING THE I/O PORTS

The 8156 is an interesting device because its I/O ports can be programmed as inputs or outputs. For instance, we can send the 8156 a command that makes port A an input port. Later, we can send another command that changes port A to an output port.

Command Word

The command register receives the command that controls the I/O ports and other functions. Figure 15-3*a* shows the contents of the command register. As indicated, bits 7 and 6 control the timer; bits 5 and 4, the interrupts; bits 3 and 2, port C; bit 1, port B; and bit 0, port A.

Table 15-3 shows how bit D_0 controls port A. When D_0 is a 0, port A is an input port; when D_0 is a 1, port A is an output port. Similarly, Table 15-4 illustrates the effect of bit D_1. When D_1 is a 0, port B acts like an input port. When D_1 is a 1, port B acts like an output port.

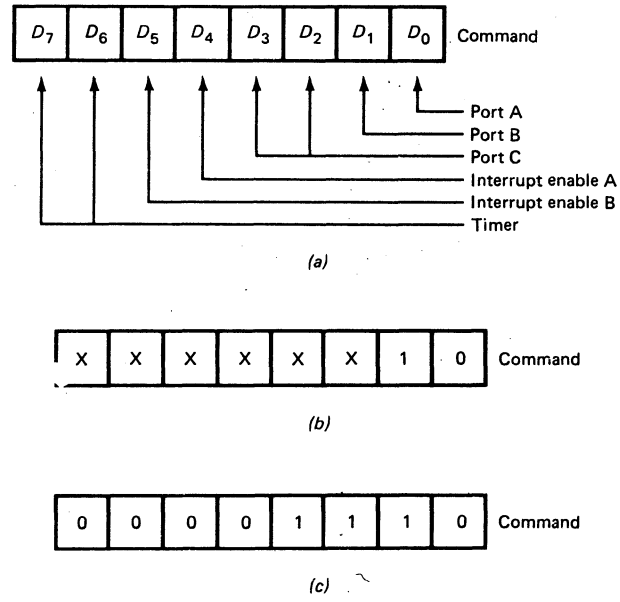


Fig. 15-3 (*a*) Command word; (*b*) B is output, A is input; (*c*) C and B are output, A is input.

To send a command to the command register, use an OUT 20H. As an example, suppose we want port A to be an input port and port B to be an output port. Then, D_0 must be a 0 and D_1 must be a 1, as shown in Fig. 15-3*b*. If we treat the don't cares as 0s, the command word becomes 02H. To load the command register with this word, we use

```
MVI A,02H
OUT 20H
```

This sets D_1 and resets D_0. After these two instructions have been executed, port A is an input and port B is an output.

If you prefer to make port A an output port and port B an input port, use this initialization:

```
MVI A,01H
OUT 20H
```

This sets D_0 in the command register and resets all other bits. The effect is to program port A as output and port B as input.

TABLE 15-3. PORT A

D_0	Effect
0	Input
1	Output

TABLE 15-4. PORT B

D_1	Effect
0	Input
1	Output

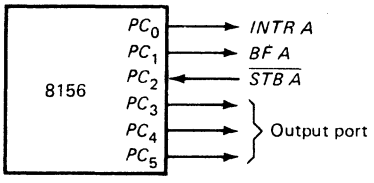
TABLE 15-5. PORT C

D_3	D_2	Mode
0	0	Input port (6 bits)
0	1	Port A handshaking and output (3 bits)
1	0	Ports A and B handshaking
1	1	Output port (6 bits)

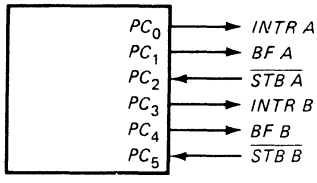
Port C

Port C is more complicated because it can act like an input port, an output port, or a handshaking port. Bits D_3 and D_2 determine how port C acts (see Table 15-5). When D_3D_2 is 00, port C acts like an input port. On the other hand, when D_3D_2 is 11, port C acts like an output port. As an example, if we want port A to be input, and ports B and C to be outputs, we can initialize the 8156 with these instructions:

```
MVI A,0EH
OUT 20H
```



(a)



(b)

Fig. 15-4 (*a*) Handshaking and output; (*b*) handshaking for A and B.

This will load the command register with the word shown in Fig. 15-3*c*.

Handshaking

When D_3D_2 is 01 in Table 15-5, half of port C is used as a handshaking port and the other half as an output port (see Fig. 15-4a). PC_0 sends out a bit called *INTR A*; this stands for interrupt at port A. PC_1 sends out *BF A*; this stands for buffer register full at port A. PC_2 receives the bit $\overline{STB\ A}$; this is a strobe signal into port A. (Examples 15-3 and 15-4 explain how these handshaking bits work.) The other bits (PC_3 to PC_5) act like a 3-bit output port.

If D_3D_2 is 10 in Table 15-5, port C provides handshaking for ports A and B (Fig. 15-4*b*). Three handshaking bits are used for each port: an interrupt, a buffer full, and a strobe. These allow us to use interrupt-driven I/O with handshaking.

Status Word

The status register can be read by using an IN 20H. This transfers the contents of the status register to the accumulator. Figure 15-5 shows the bits in the status register. When high, bit 0 (*INTR A*) indicates a pending interrupt at port A. When bit 1 (*BF A*) is high, it means that the buffer register is full at port A; that is, valid data is latched in port A and awaits an I/O transfer. Bit 2, *INTE A*, stands for interrupt enable at port A; when low, it prevents an interrupt from appearing at PC_0 (Fig. 15-4).

Bits 3 through 5 are the interrupt pending, buffer full, and interrupt enable for port B. Bit 6, *INTR TIMER*, is high when the terminal count has been reached. (Terminal count is explained in Sec. 15-4.) Bit 7 is an undefined (don't care).

As an example, suppose the execution of

```
IN 20H
```

produces accumulator contents of

```
A = X100 0111
```

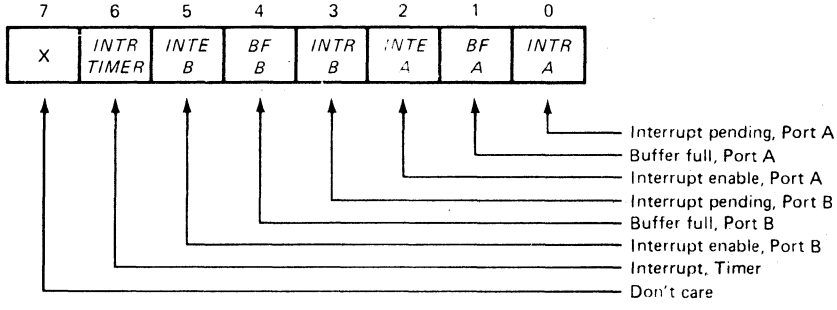


Fig. 15-5 Status register.

From left to right, we read a don't care, a high terminal count, a disabled B interrupt, an empty B buffer, no pending B interrupt, an enabled A interrupt, a full A buffer, and a pending A interrupt.

EXAMPLE 15-1

Show the instructions needed to make A an input port, B an output port, and C a full handshaking port. The A and B interrupts should be enabled and the timer bits reset to 00.

SOLUTION

Refer to Fig. 15-3*a* and Table 15-5. We need a command word of

$$0011\ 1010 = 3AH$$

This will reset the timer bits to 00 (bits 7 and 6), enable the A and B interrupts (bits 5 and 4), provide handshaking for A and B (bits 3 and 2), make B an output (bit 1), and A an input (bit 0). To send this command to the command register, we use

```
MVI A,3AH
OUT 20H
```

As soon as the OUT 20H has been executed, port A becomes an input, port B becomes an output, port C provides handshaking, both interrupts are enabled, and the timer bits are cleared.

EXAMPLE 15-2

What do the following initializing instructions do:

```
MVI A,2AH
OUT 20H
```

SOLUTION

The command word is

$$0010\ 1010 = 2AH$$

The only difference between this and the preceding example is that interrupt B is enabled but interrupt A is disabled.

EXAMPLE 15-3

Show an interrupt-driven input circuit that handshakes with port A.

SOLUTION

In Fig. 15-6*a* the peripheral device sends data (PA_7–PA_0) to port A of the 8156, which then sends the data (AD_7–AD_0) on to the 8085. Port C provides the handshaking bits *INTR A*, *BF A*, and $\overline{STB\ A}$.

Figure 15-6*b* is the timing diagram. First, the peripheral device sends a low $\overline{STB\ A}$ to the 8156. This loads the peripheral data into port A via the PA_7–PA_0 bus. The 8156 acknowledges receiving this data by sending a high *BF A* back to the peripheral device.

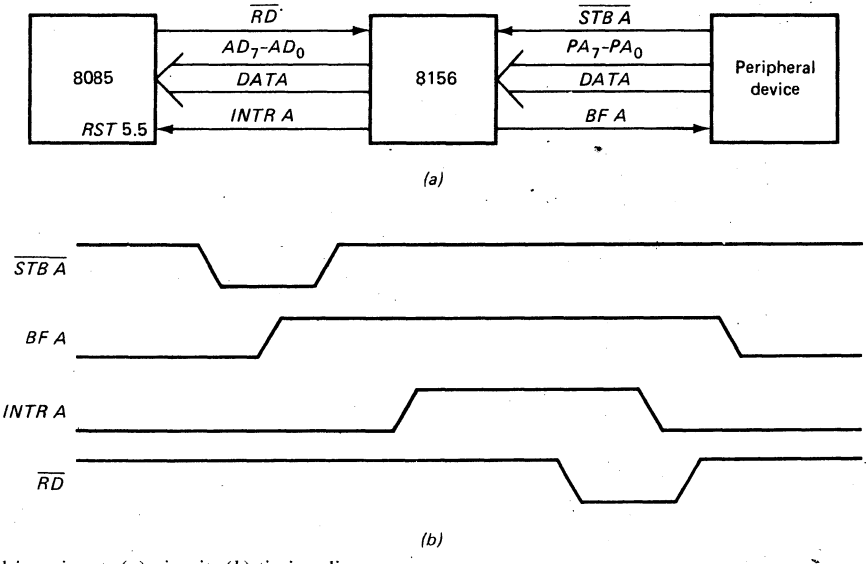


Fig. 15-6 Interrupt-driven input: (*a*) circuit; (*b*) timing diagram.

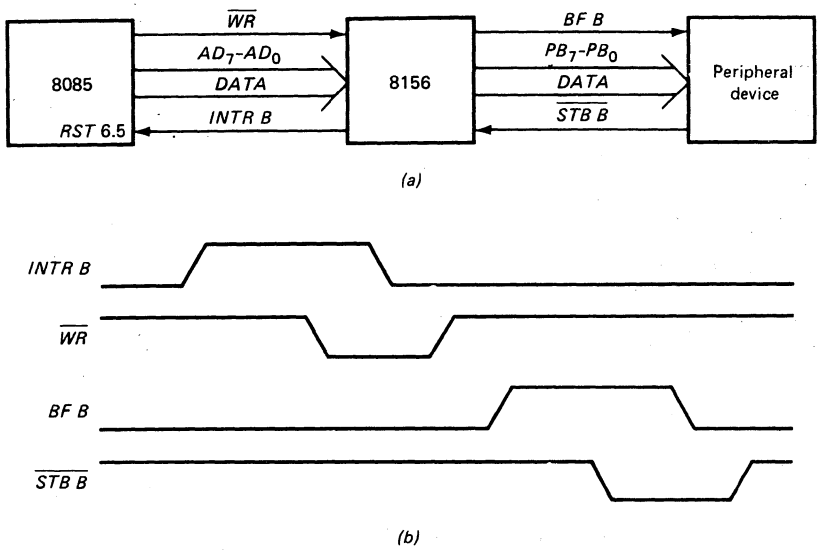


Fig. 15-7 Interrupt-driven output: (*a*) circuit; (*b*) timing diagram.

A little later, the 8156 is ready to transfer the data, so it sends a high *INTR A* to the RST 5.5 pin of the 8085. After the 8085 recognizes this interrupt, it branches to a service subroutine that includes an IN 21H. During the execution of this instruction, $\overline{RD}$ goes low; this transfers the data in port A to the accumulator via the address-data bus. Then the remainder of the service subroutine is completed.

EXAMPLE 15-4

Show an interrupt-driven output circuit that handshakes with port B.

SOLUTION

Figure 15-7*a* is the circuit, and Fig. 15-7*b* is the timing diagram. The action begins when the 8156 sends a high *INTR B* to the 8085. After this has been recognized, the 8085 branches to a service subroutine that includes an OUT 22H instruction. During the execution of this instruction, $\overline{WR}$ goes low; this transfers data from the accumulator to port B via the address-data bus. The *INTR B* then goes low, indicating that the data has been received.

A little later, the 8156 sends a high *BF B* to the peripheral device; this transfers the data from port B to the peripheral device by way of the PB_7–PB_0 bus. To indicate that it has received the data, the peripheral device returns a low $\overline{STB\ B}$.

15-4 PROGRAMMING THE TIMER

The timer is a 14-bit presettable down counter that counts the incoming *TIMER IN* pulses (pin 3). It can be preset with any number between 0002H and 3FFFH. This preset number is called the *terminal count*. For instance, if the timer is preset with 00FFH, it will reach the terminal count after 255 pulses have been received at the TIMER IN input.

Presetting the Terminal Count

As indicated in Table 15-2, port 24H addresses the lower 8 bits of the timer. Figure 15-8*a* shows how to visualize these bits. Similarly, port 25H addresses the upper 6 bits of the timer and the timer mode (see Fig. 15-8*b*). Let us ignore the timer mode for now by assuming that M_2M_1 is 00.

Here is an example. Suppose we want to preset the terminal count to 00FFH. Then these are the initializing instructions:

```
MVI A,FFH
OUT 24H
MVI A,00H
OUT 25H
```

The first two instructions load FFH into the lower 8 bits of the timer (Fig. 15-8*a*). The last two instructions load 00H into the timer mode and upper 6 bits (Fig. 15-8*b*).

Timer Mode

The output signal $\overline{TIMER\ OUT}$ (pin 6) depends on the timer mode M_2M_1 as indicated in Table 15-6. If M_2M_1 is 00, the timer produces a single square wave. If M_2M_1 is 01, you get a continuous square wave. When M_2M_1 is 10, $\overline{TIMER\ OUT}$ is a single pulse. And when M_2M_1 is 11, the timer generates a continuous pulse train.

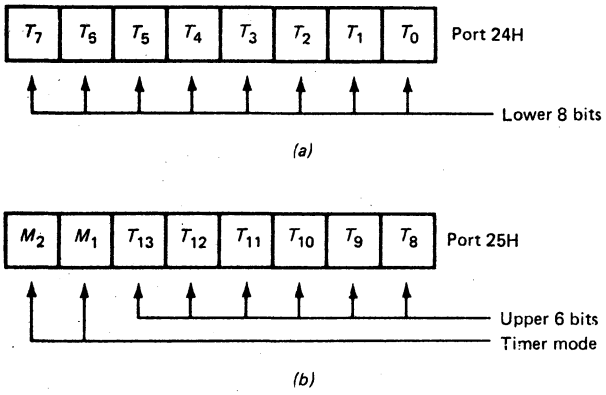


Fig. 15-8 (*a*) Lower timer byte; (*b*) upper timer byte.

Figure 15-9 shows how the *TIMER OUT* signals appear for each mode. When M_2M_1 is 00, the square wave begins when the timer starts counting. Halfway through the count, the square wave goes into the negative half cycle. The square wave ends when the terminal count is reached.

If we want continuous square waves, we have to program mode 01. In this case, the square waves begin when the timer starts counting and continue indefinitely.

When the timer mode is 10, we get a single pulse at the end of the count. In other words, when the terminal count is reached, the *TIMER OUT* goes negative for a brief interval, then returns positive. The width of this pulse equals the width of the *TIMER IN* pulse.

Finally, we can get a train of *TIMER OUT* pulses by programming mode 11. The pulses begin after the terminal count is reached the first time and continue to appear each time the terminal count is reached.

Timer Command

Bits D_7 and D_6 of the command register (Fig. 15-10) control the starting and stopping of the timer. As shown in Table 15-7, a D_7D_6 of 00 produces a nop. If D_7D_6 is 01, the timer stops immediately. If the bits are 10, the timer stops after reaching the terminal count. If D_7D_6 is 11, the timer starts counting.

For instance, suppose we want to start the timer, enable the port interrupts, make C and B output ports, and make A an input port. Then we need a command word of

$$D = 1111\ 1110$$

TABLE 15-6. TIMER MODE

M_2	M_1	Effect
0	0	Single square wave
0	1	Continuous square wave
1	0	Single pulse
1	1	Continuous pulse

TABLE 15-7. TIMER COMMAND

D_7	D_6	Effect
0	0	Nop
0	1	Stop immediately
1	0	Stop after TC is reached
1	1	Start

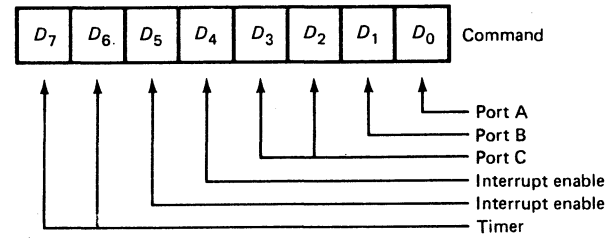


Fig. 15-10 Timer command bits.

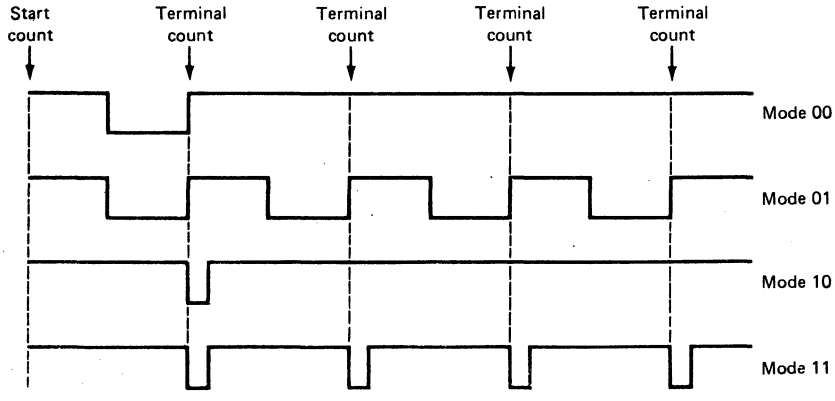


Fig. 15-9 Timer modes.

The initializing instructions are

```
MVI A,FEH
OUT 20H
```

EXAMPLE 15-5

In Fig. 15-11*a*, the system clock is connected to the TIMER IN input of the 8156. The clock has a frequency of 3 MHz. Show a program segment that produces a continuous square wave with a frequency of 1 kHz. Include a start timer command, disable the port interrupts, make C and B output ports, and make A an input port.

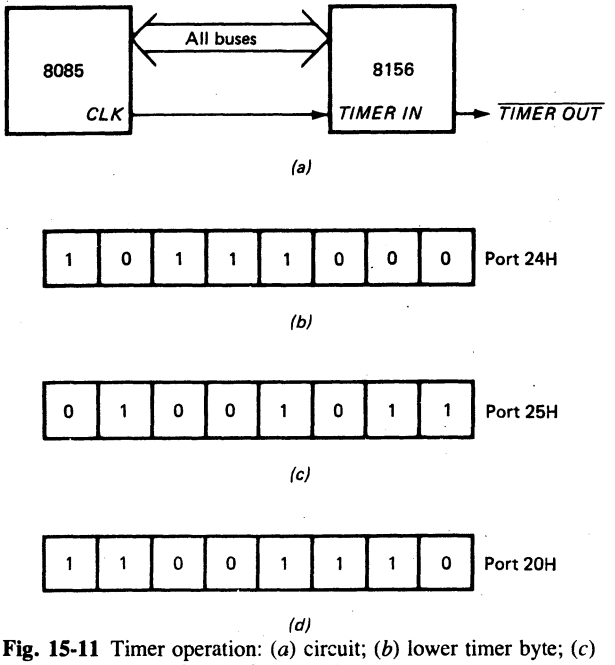


Fig. 15-11 Timer operation: (*a*) circuit; (*b*) lower timer byte; (*c*) upper timer byte; (*d*) command word.

SOLUTION

We have to divide the clock frequency by 3000, which is equivalent to 0BB8H. By setting the terminal count (Fig. 15-9) equal to 0BB8H, we will get one square-wave cycle out for each 3,000 clock pulses in. Therefore, the lower 8 bits of the terminal count are

1011 1000

This is the lower byte to be loaded into port 24H (see Fig. 15-11*b*).

The upper 6 bits of the terminal count are

00 1011

According to Table 15-6, the timer-mode bits for a continuous square wave are

$$M_2M_1 = 01$$

Therefore, port 25H must be loaded with

0100 1011

as shown in Fig. 15-11*c*.

The command word we need for port 20H is

1100 1110

as shown in Fig. 15-11*d*. Bits D_7D_6 come from Table 15-7; these bits start the timer. The programming of the other bits has already been explained; D_5D_4 disable the port interrupts (Fig. 15-10), D_3D_2 make C an output port (Table 15-5), D_1 makes B an output port, and D_0 makes A an input port.

Here is the program segment:

```
MVI A,B8H
OUT 24H
MVI A,4BH
OUT 25H
MVI A,CEH
OUT 20H
```

When these instructions are executed, ports 24H, 25H, and 20H are loaded as shown in Fig. 15-11*b* to *d*. The timer then starts counting clock pulses. For each 3,000 clock pulses received, one square wave comes out of the $\overline{\text{TIMER OUT}}$ pin. Therefore, we get a continuous square wave with a frequency of 1 kHz.

15-5 THE 8355

As discussed in Chap. 13, the 8355 contains a 16,384-bit ROM organized as 2,048 words of 8 bits each. In our minimum system, the ROM locations are from 0000H to 07FFH. The 8355 also has two 8-bit I/O ports, port A and port B.

Pinout

Figure 15-12 shows the pinout diagram of an 8355. Most of the pins are self-explanatory. Notice the two chip enables, $\overline{CE}_1$ and CE_2. Having one active low and the other active high allows design flexibility in addressing the memory. For the minimum system (Chap. 13), we connect $\overline{CE}_1$ to A_{13} and CE_2 to +5 V.

Port A has a data bus PA_7–PA_0 (pins 31 to 24). Similarly, port B has a data bus PB_7–PB_0 (pins 39 to 32). To address

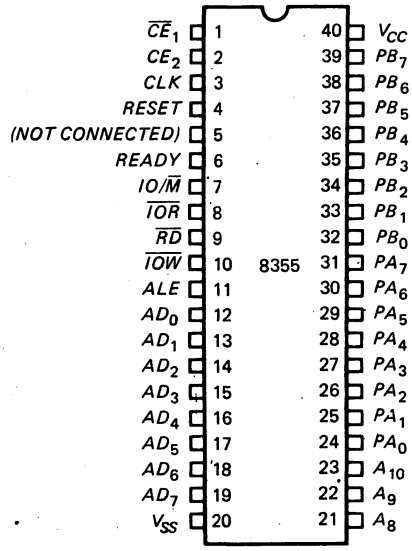


Fig. 15-12 8355 pinout.

the 2,048 stored bytes, we need 11 address lines. The upper address bits (A_{10}, A_9, A_8) connect to pins 23, 22, and 21. The lower address bits are multiplexed on the address-data bus (pins 19–12). Also discussed previously are the CLK (pin 3), RESET (pin 4), READY (pin 6), IO/$\overline{M}$ (pin 7), $\overline{RD}$ (pin 9), and ALE (pin 11).

Notice two new signals $\overline{IOR}$ (pin 8) and $\overline{IOW}$ (pin 10). A low $\overline{IOR}$ will read the selected port contents onto the address-data bus. $\overline{IOR}$ has the same effect as a high $IO/\overline{M}$ and a low $\overline{RD}$. Similarly, a low $\overline{IOW}$ will write the contents of the address-data bus into the selected port.

Port Numbers

The 8355 has two I/O ports and two internal registers called *data-direction registers* (DDR A and DDR B). These internal registers are like command registers because they determine whether the pins of ports A and B are inputs or outputs.

Table 15-8 shows the address bits for the 8355 registers: 00 for port A, 01 for port B, 10 for DDR A, and 11 for DDR B. By solving the duplication equations (similar to the derivation in Sec. 15-2) we can arrive at the port numbers listed in Table 15-9. As indicated, 00H addresses port A, 01H addresses port B, and so on. Because some address lines are not used in the minimum system, these port numbers have shadows.

TABLE 15-8. 8355 REGISTERS

AD_1	AD_0	Select
0	0	Port A
0	1	Port B
1	0	DDR A
1	1	DDR B

TABLE 15-9. 8355 PORT NUMBERS IN MINIMUM SYSTEM

Port Number	Selected Register
00H	Port A
01H	Port B
02H	DDR A
03H	DDR B

DDR Bits Control Port Pins

Ports A and B are pin-programmable. DDR A controls port A, and DDR B controls port B. For instance, we can send a command to the DDR A register to make some of the port A pins inputs and the others outputs. Later, we can send another command to change which pins are inputs and which outputs.

Tables 15-10 and 15-11 list the effect of each DDR bit on the corresponding bit in a port. As shown in Table 15-10, if a bit is 0 in DDR A, the corresponding pin of port A is an input pin; if the bit is a 1, the pin is an output pin.

Here is an example. Suppose DDR A has the contents shown in Fig. 15-13*a*. Then each 0 bit in DDR A produces an input pin in port A; each 1 bit in DDR A produces an output pin in port A.

As another example, look at the contents of DDR B in Fig. 15-13*b*. Each 0 bit in DDR B produces an input pin in port B, and each 1 bit produces an output pin.

Programming the Ports

To program the pins of a port as input or output, we have to load the appropriate bit pattern into the data direction

TABLE 15-10. PORT A

DDR A bit	Port A pin
0	Input
1	Output

TABLE 15-11. PORT B

DDR B bit	Port B pin
0	Input
1	Output

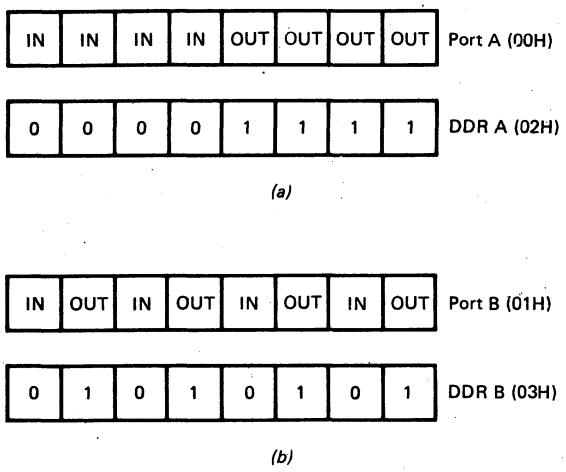


Fig. 15-13 Pin programming of ports A and B.

register for the port. For instance, to program port A as shown in Fig. 15-13*a*, we would use

```
MVI A,0FH
OUT 02H
```

After these two instructions are executed, PA_7 to PA_4 are input pins, and PA_3 to PA_0 are output pins.

As another example, we can program the port B pins as shown in Fig. 15-13*b* by using

```
MVI A,55H
OUT 03H
```

When these instructions are executed, PB_7 is an input pin, PB_6 is an output pin, PB_5 is an input pin, and so on.

Equivalent Circuit

For a better understanding of how the pins of port A are programmed, look at Fig. 15-14. This shows bit D_7; each of the remaining bits (D_6 to D_0) has a similar port latch and DDR latch. Initially, bit D_7 is loaded into DDR A during the execution of an OUT 02H; this sets or resets the DDR A latch. Since the output of the DDR A latch controls a three-state switch, the bit in DDR A determines input or output operation.

When DDR A contains a 1, it enables the three-state switch. During the execution of an OUT 00H, *WRITE* goes high and bit D_7 is latched into port A. With the three-state switch closed, bit D_7 is transmitted to pin PA_7. In this case, PA_7 acts like an output pin.

On the other hand, when DDR A contains a 0, it disables the three-state switch. During the execution of an IN 00H, *READ* goes high. Any peripheral data at pin PA_7 is then read onto the address-data bus. In other words, pin PA_7 now acts like an input pin.

EXAMPLE 15-6

What does the following program segment do?

```
MVI A,55H
OUT 03H
IN 01H
MVI A,FFH
OUT 01H
```

SOLUTION

The first two instructions program DDR B as shown earlier in Fig. 15-13*b*. This gives alternating input and output pins on port B. The IN 01H reads in PB_7, PB_5, PB_3, and PB_1 because these pins are programmed for input operation. The last two instructions send high bits to PB_6, PB_4, PB_2, and PB_0 because these pins are programmed for output operation.

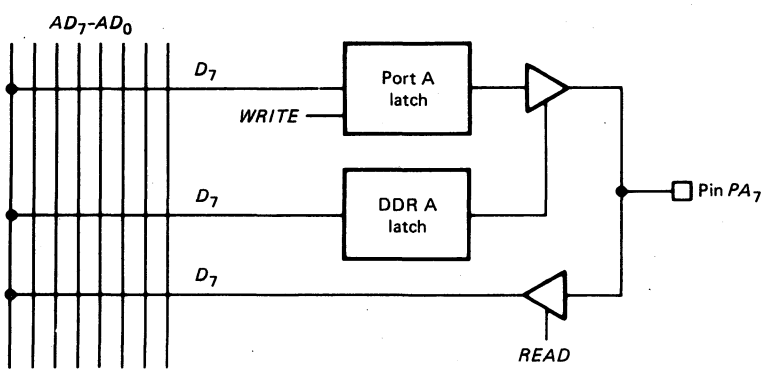


Fig. 15-14 Equivalent circuit for pin-programming bit 7.

15-6 FULLY DECODED MINIMUM SYSTEM

The minimum system has 2K of ROM, 256 bytes of RAM, and five I/O ports (two in the 8355 and three in the 8156). Some applications require more memory and I/O. The danger in adding more memory and I/O is one of inadvertently using memory and I/O shadows. The safest way to avoid this is to fully decode the address lines during memory and I/O operations.

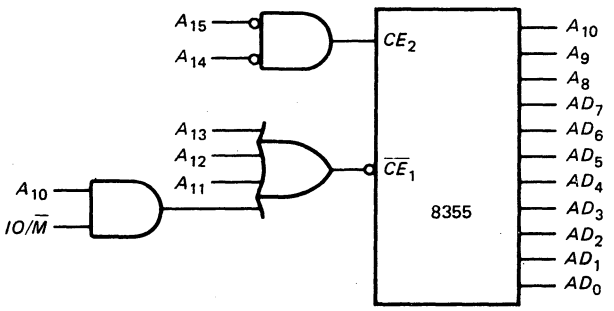


Fig. 15-15 Fully decoded 8355.

Fully Decoded 8355

The reason we get memory and I/O shadows in a minimum system is that some of the address lines are not used. If we used all the address lines, we would remove the don't cares that produce the shadows.

Figure 15-15 shows one way to eliminate all memory and I/O shadows in the 8355. For CE_2 to be active, A_{15} and A_{14} must be low. For $\overline{CE}_1$ to be active during memory operations, A_{13} through A_{11} must be low. (A_{10} has no effect because $IO/\overline{M}$ is low during memory operations.) This means that the ROM is enabled only when A_{15} through A_{11} are low. Symbolically,

$$A_{15}A_{14}A_{13}A_{12}A_{11} = 00000$$

As before, the lower 11 bits (A_{10}, A_9, A_8, and AD_7 to AD_0) select the memory location. In other words, the 8355 of Fig. 15-15 has a ROM range of

0000 0000 0000 0000

to

0000 0111 1111 1111

equivalent to 0000H to 07FFH. Since there are no don't cares, memory shadows no longer exist.

I/O shadows are also eliminated in Fig. 15-15. For CE_2 to be active, A_{15} and A_{14} must be low. For $\overline{CE}_1$ to be active, A_{13} to A_{10} must be low. (A_{10} must be low because $IO/\overline{M}$ is high during I/O operations.) Therefore, the duplication equation for port A is

0000 00XX = XXXX XX00

Equating 0s on both sides gives

0000 0000 = 0000 0000

This is the only solution. Therefore, port 00H has no I/O shadows. By a similar derivation, we can prove that the remaining ports have no shadows.

Fully Decoded 8156

Figure 15-16 shows one way to eliminate the memory and I/O shadows in the 8156. During a memory operation, CE is enabled only when

$A_{15}A_{14}A_{13}A_{12}A_{11} = 00100$ and $A_{10}A_9A_8 = 000$

As before, the lower 8 bits select the memory location. Therefore, the RAM range is from

0010 0000 0000 0000

to

0010 0000 1111 1111

This is equivalent to 2000H to 20FFH. Since there are no don't cares in the address lines, all RAM shadows have been eliminated.

During I/O operations, CE is enabled only when

$A_{15}A_{14}A_{13}A_{12}A_{11} = 00100$

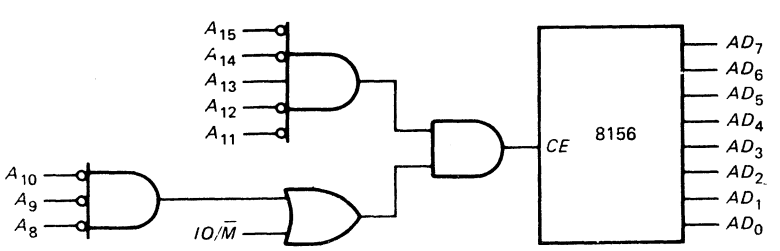


Fig. 15-16 Fully decoded 8156.

and $IO/\overline{M}$ is high. The duplication equation for port 20H is

$$0010\ 0XXX = XXXX\ X000$$

The only solution is 0010 0000, which means that port 20H has no shadows. Similarly, ports 21H to 25H have no shadows.

Conclusion

When all this discussion sinks in, here is what you have. The gates of Figs. 15-15 and 15-16 fully decode the address lines in a minimum system. Therefore, the only addressable ROM locations are 0000H–07FFH; the only addressable RAM locations are 2000H–20FFH; the only port numbers are 00H–03H and 20H–25H. The shadows are gone. This means that we are free to expand the memory and I/O of a minimum system without fear of new locations falling into the old shadows.

15-7 CREATING AND ADDRESSING NEW I/O PORTS

How can we add new I/O ports to a minimum system? By using three-state switches for the input ports and latches for the output ports.

Creating an Input Port

By connecting two 74LS126s and an inverter, as shown in Fig. 15-17, we can control 8 bits of data from a peripheral device. A high $\overline{ENABLE}$ floats the output lines; a low $\overline{ENABLE}$ connects the input data to the output lines. A circuit like this is called a *three-state driver;* a low $\overline{ENABLE}$ connects the peripheral data to the address-data bus; a high $\overline{ENABLE}$ disconnects the peripheral data from the address-data bus.

Gate Addressing an Input Port

The simplest way to address an input port is with *gate addressing*. This means using gates to produce the $\overline{ENABLE}$ signal for the three-state driver. As an example, Fig. 15-18*a* shows how to add port FFH to a fully decoded minimum system. To understand how the circuit works, look at the timing diagram for an IN instruction (Fig. 15-18*b*). During the M_3 cycle, the port address is placed on the upper address bus (A_{15} to A_8). Also notice how $IO/\overline{M}$ goes high and $\overline{RD}$ goes low during the M_3 cycle.

Here is what happens in Fig. 15-18*a*. When an IN FFH is executed, A_{15} to A_8 are high during the M_3 cycle. Since $IO/\overline{M}$ goes high and $\overline{RD}$ goes low during this cycle, the NAND gate momentarily produces a low $\overline{ENABLE}$. This

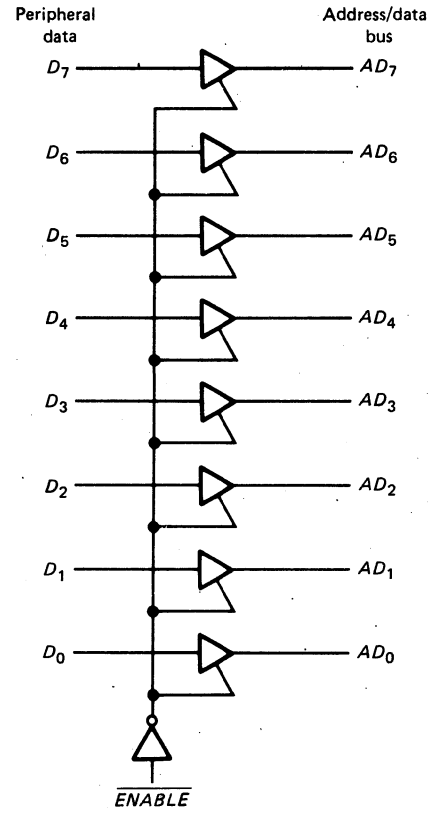


Fig. 15-17 Input port using three-state switches.

connects the peripheral data to the address-data bus; then the bus data is loaded into the accumulator.

For any other port address, the NAND gate produces a high $\overline{ENABLE}$ and the three-state driver is disabled. Likewise, during memory operations, $IO/\overline{M}$ is low and the three-state driver is disabled. As a result, the port of Fig. 15-18*a* is enabled only during the execution of an IN FFH instruction.

With different logic, we can create other ports. For instance, Fig. 15-19*a* shows the gate addressing for port FEH. Only during the execution of IN FEH is the three-state driver enabled. Figure 15-19*b* shows how port FDH is configured. This time it takes an IN FDH to enable the driver.

The 74LS138

Gate addressing is convenient when a few ports are being added. When many ports are involved, however, *decoder addressing* is better. Figure 15-20*a* shows the pinout for a 74LS138, a 1-of-8 decoder often used for decoder addressing. Pins 1, 2, and 3 are the inputs that select one of the output lines. Pins 4, 5, and 6 are gate enables that activate the decoder. Pin 8 is for ground and pin 16 for the supply voltage. The remaining pins are the output lines Y_0 to Y_7.

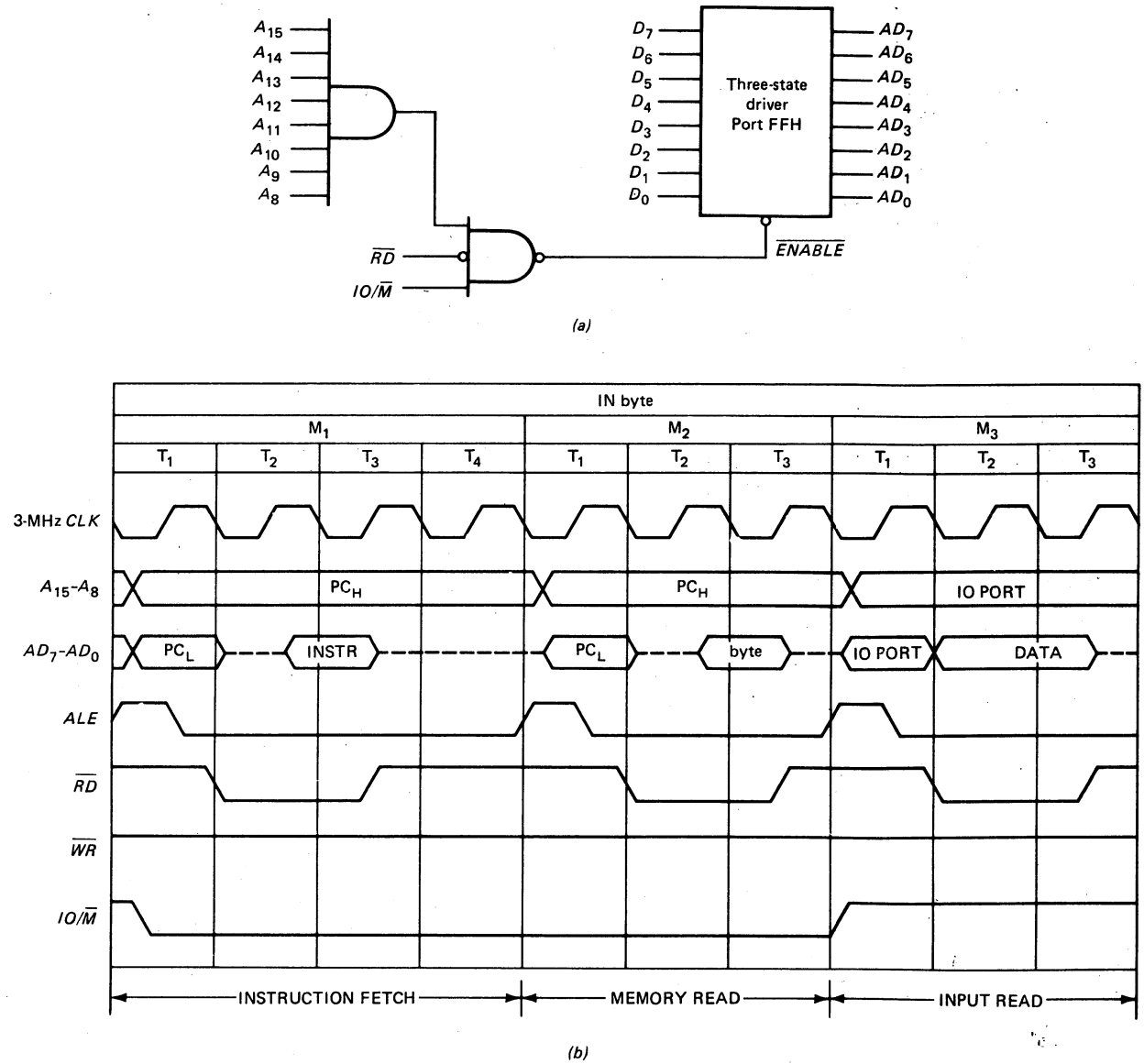


Fig. 15-18 (*a*) Gate addressing of port of port FFH; (*b*) timing diagrams.

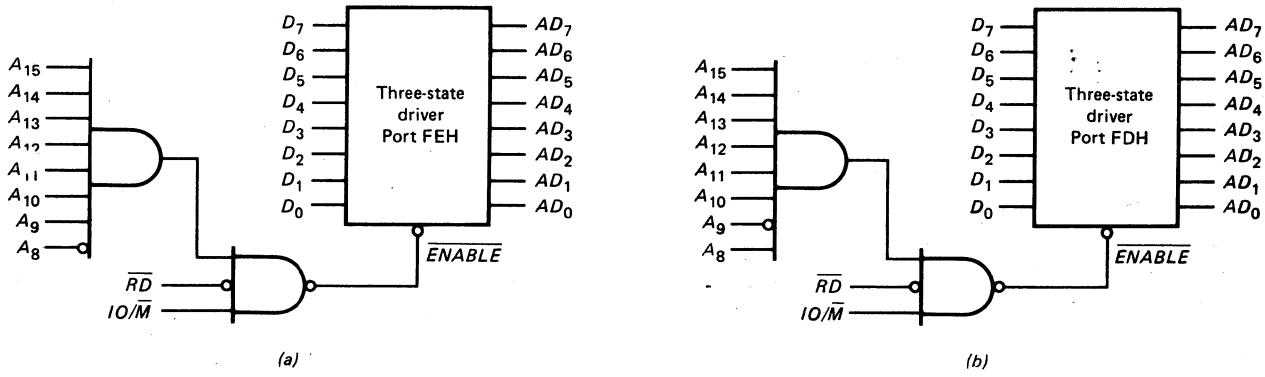


Fig. 15-19 Gate addressing: (*a*) port FEH; (*b*) port FDH.

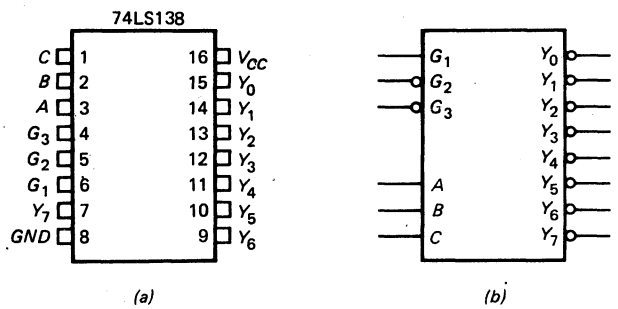


Fig. 15-20 74LS138: (*a*) pinout; (*b*) symbol.

This 1-of-8 decoder works as follows. When all gate enables are active, the *ABC* input makes only one of the output lines active. When $ABC = 000$, Y_0 is active; when $ABC = 001$, Y_1 is active; when $ABC = 010$, Y_2 is active; and so on. Table 15-12 summarizes the operation.

Notice three important features:

1. The decoded output line is low.
2. Decoding occurs only when all gate enables are active; this means G_1 high, G_2 low, and G_3 low.
3. If any gate enable is inactive, all output lines are high.

Figure 15-20*b* is the symbol we will use in our coming discussions. Bubbles on the input and output lines indicate active lows.

Decoder Addressing

With a 74LS138 and some gates, we can select up to 8 ports. As an example, Fig. 15-21 shows decoder addressing for ports F8H through FFH. When $IO/\overline{M}$ is low, the 74LS138 is disabled and all output lines are high. Therefore, none of the three-state drivers are enabled during memory operations.

During the M_3 cycle of the IN instruction, $IO/\overline{M}$ goes high and $\overline{RD}$ goes low; this enables G_3. For any port address from F8H to FFH, A_{15} to A_{11} are high. In symbols,

$$A_{15}A_{14}A_{13}A_{12}A_{11} = 11111$$

Therefore, G_1 is enabled. This leaves A_{10} to A_8 to select one of the eight output lines. When $A_{10}A_9A_8 = 000$, Y_0 is low; this sends a low $\overline{ENABLE}$ to port F8H. When $A_{10}A_9A_8 = 001$, port F9H is enabled. When $A_{10}A_9A_8 = 010$, port FAH is active, and so on for the remaining ports.

In other words, the execution of IN F8H enables port F8H; this connects the peripheral data at port F8H to the address-data bus; the data is then loaded into the accumulator. During the execution IN F9H, port F9H transfers peripheral data to the address-data bus, and so forth.

One 74LS138 controls eight ports. By using more 74LS138s and altering the addressing gates we can expand I/O to whatever level is necessary for the application.

Creating Output Ports

To create an output port, we can use latches like the 74LS75, 74LS173, 74LS175, etc. In our discussion, we emphasize 74LS173s (SAP-1 chips). Figure 15-22*a* shows how to connect two 74LS173s as output port FFH. Pins 14 through 11 of both chips are tied to the address-data bus. Pins 3 through 6 of both chips are connected to the peripheral device that receives the data.

During the M_3 cycle of an OUT instruction, the port address bits are placed on the upper address bus. When bits A_{15} through A_8 are all high (port FFH), the addressing gate delivers a low signal to pin 9. Since $IO/\overline{M}$ is also high during M_3 (see Fig. 15-22*b*), pin 10 goes low. Therefore, both gate enables (pin 9 and 10) are active. A bit later in the M_3 cycle, $\overline{WR}$ goes temporarily low, then high. The

TABLE 15-12. 74LS138 TRUTH TABLE

G_1	G_2	G_3	A	B	C	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
X	X	1	X	X	X	1	1	1	1	1	1	1	1
X	1	X	X	X	X	1	1	1	1	1	1	1	1
0	X	X	X	X	X	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	1	1	1	1	1	1	1
1	0	0	0	0	1	1	0	1	1	1	1	1	1
1	0	0	0	1	0	1	1	0	1	1	1	1	1
1	0	0	0	1	1	1	1	1	0	1	1	1	1
1	0	0	1	0	0	1	1	1	1	0	1	1	1
1	0	0	1	0	1	1	1	1	1	1	0	1	1
1	0	0	1	1	0	1	1	1	1	1	1	0	1
1	0	0	1	1	1	1	1	1	1	1	1	1	0

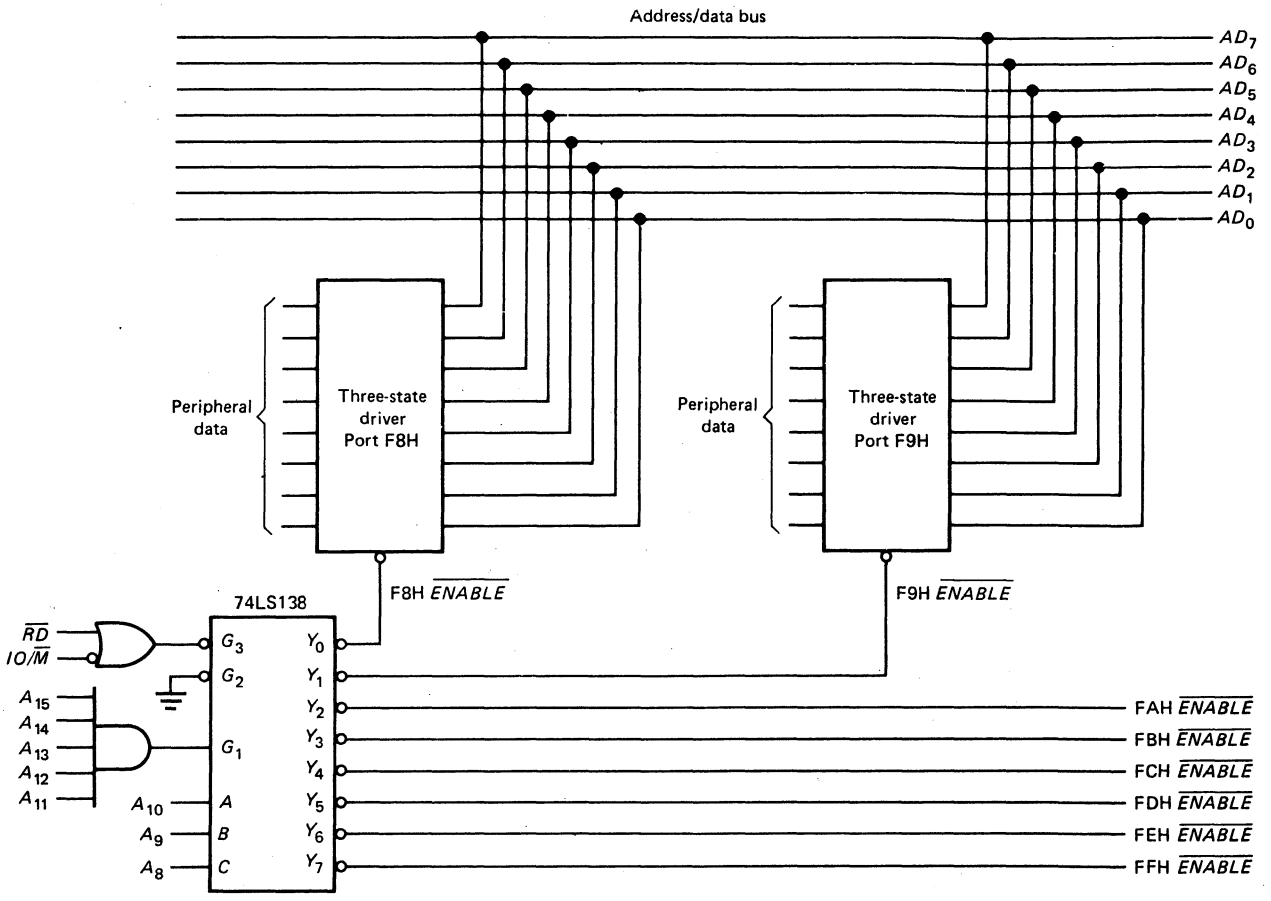


Fig. 15-21 Decoder addressing of ports F8H to FFH.

rising edge of the $\overline{WR}$ signal (Fig. 15-22*b*) latches the data into the 74LS173s.

If many output ports are to be added, we can use decoder addressing. This means that one 74LS138 can control eight output ports, as shown in Fig. 15-22*c*. Notice that G_2 and G_3 are grounded. Also, the upper address bits enable G_1 only when A_{15} to A_{11} are high. As before, A_{10} to A_8 activate one of the output lines. Each output line can be connected to pin 9 of an output port like Fig. 15-22*a*. When $IO/\overline{M}$ is high and $\overline{WR}$ returns high, the addressed port will latch bus data into the 74LS173s.

15-8 EXPANDING THE MEMORY WITH STATIC RAMS

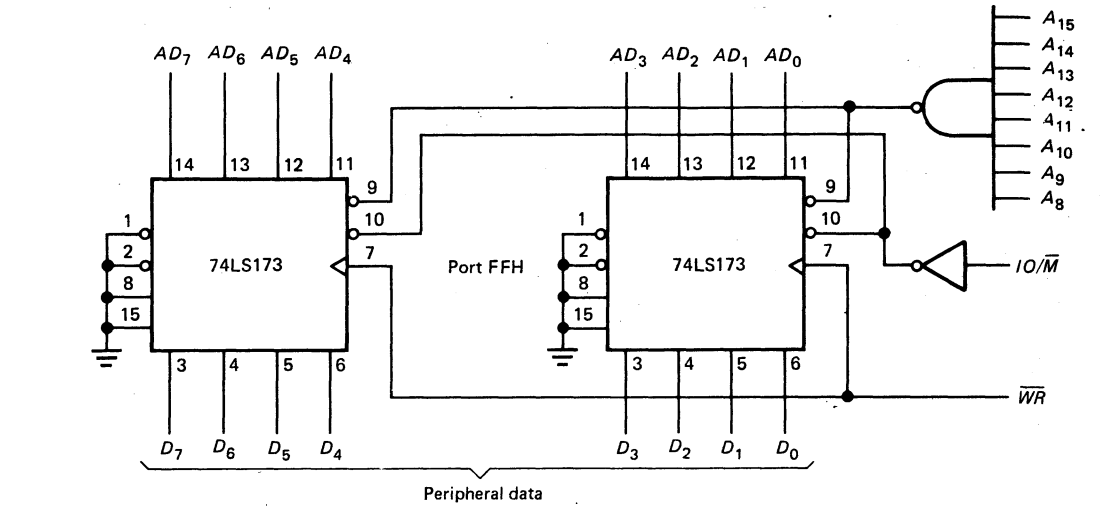
Our minimum system has 2K of ROM and 256 bytes of RAM. When the memory chips are fully decoded, this implies shadowless ROM and RAM locations from 0000H to 07FFH and from 2000H to 20FFH. In this section, we examine ways to add more memory.

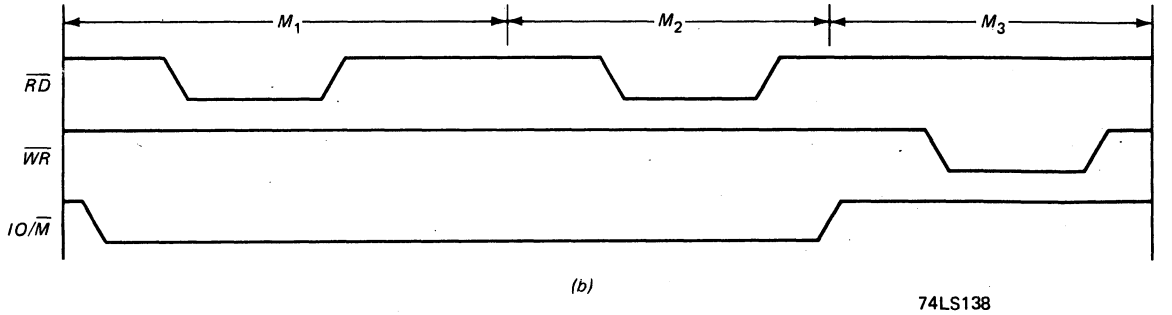
The 2114

This 18-pin chip is a 4,096-bit static RAM, which is organized as 1,024 words of 4 bits each. The 2114, now an industry standard, has a maximum access time of 450 ns. A newer version, the 2114A, can reduce the access time to 100 ns.

Figure 15-23*a* shows the pinout. The 10 address pins, A_0 to A_9, can access 1,024 memory locations. The four data pins are labeled D_0 to D_3. When the chip select $\overline{CS}$ is high, the data lines float. To enable the memory, $\overline{CS}$ must go low; then a high $\overline{WE}$ produces a read operation, and a low $\overline{WE}$ a write.

To get byte length, we need two 2114s in parallel, as shown in Fig. 15-23*b*. This produces 1,024 words of 8 bits each. When used in a microprocessor-based system, the data pins are connected to the data bus; the address pins (not shown) are connected to a memory-address register (MAR). To simplify later figures, we will draw Fig. 15-23*b* as shown in Fig. 15-23*c*.





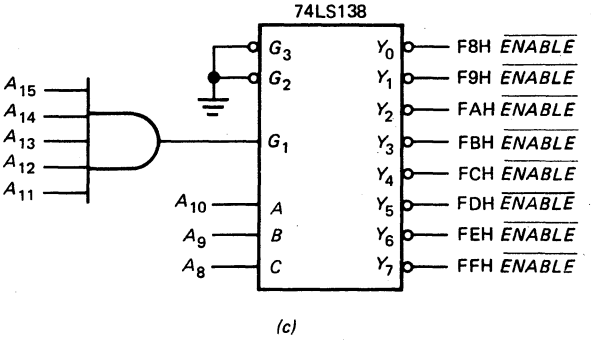
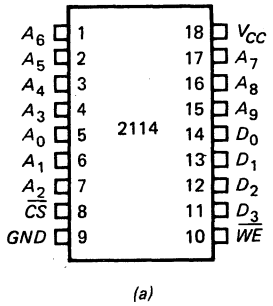
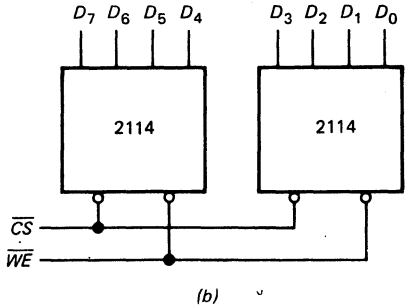


Fig. 15-22 Output port: (*a*) gate addressing port FFH; (*b*) timing diagrams; (*c*) decoder addressing ports F8H to FFH.





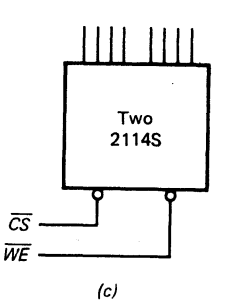


Fig. 15-23 2114 Static RAM: (*a*) pinout; (*b*) two in parallel produce byte; (*c*) symbol for two 2114s.

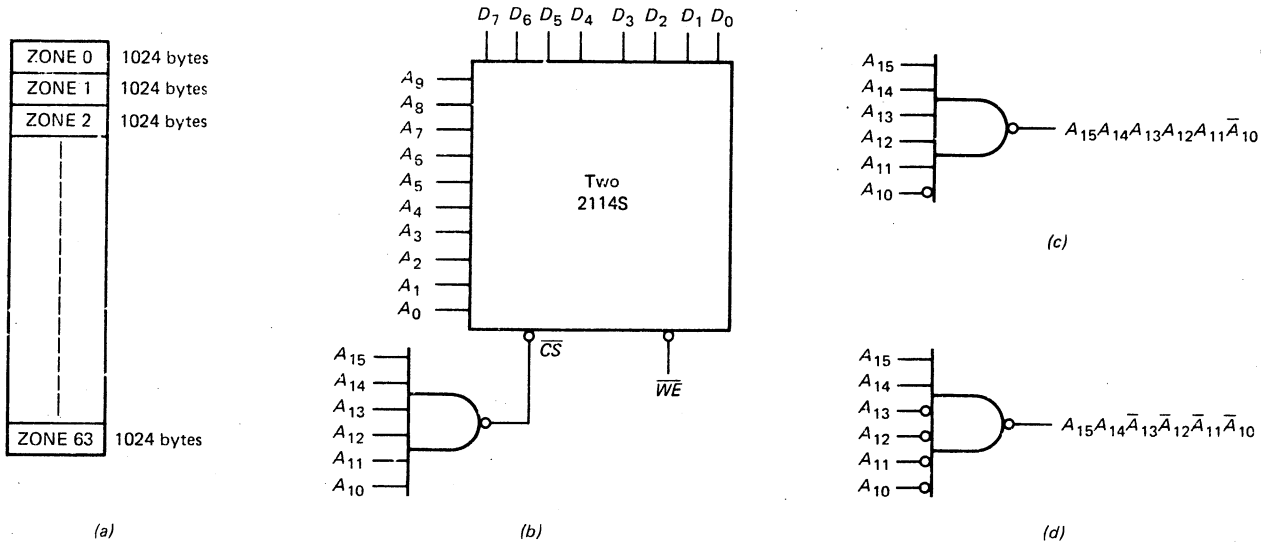


Fig. 15-24 (*a*) Memory zones for intervals of 1K; (*b*) gate addressing zone 63; (*c*) zone 62; (*d*) zone 48.

Memory Zones

A pair of 2114s has 10 address bits A_9 to A_0, which point to 1,024 bytes in memory. By decoding bits A_{15} to A_{10} these 1,024 bytes can be assigned to any of the 64 zones shown in Fig. 15-24*a*. Bits A_{15} to A_{10} are called *zone bits* because they determine which zone is used. For instance, with the gate addressing of Fig. 15-24*b*, zone bits A_{15} to A_{10} produce a low $\overline{CS}$ only when

$$A_{15}A_{14}A_{13}A_{12}A_{11}A_{10} = 111111$$

Therefore, the 1,024 bytes are located in zone 63.

Change the gate addressing and you get a different zone. Figure 15-24*c* shows the gate addressing for zone 62. The output of the NAND gate is $A_{15}A_{14}A_{13}A_{12}A_{11}\overline{A}_{10}$. This output is low only for zone bits of 111110, equivalent to decimal 62.

As another example, the NAND gate of Fig. 15-24*d* has an output of $A_{15}A_{14}\overline{A}_{13}\overline{A}_{12}\overline{A}_{11}\overline{A}_{10}$. This is low only for zone bits of 110000, whose decimal equivalent is 48. Therefore, this gate addressing locates the 1,024 bytes in zone 48.

As an aid, Appendix 10 shows how a 64K memory is divided into 64 zones. You will find this useful in analysis and design because it lists the address bits along with the hexadecimal and decimal equivalents for each zone. For Fig. 15-24*b* the 1,024 bytes are in zone 63, which means hexadecimal locations FC00H to FFFFH. For Fig. 15-24*c* the 1,024 bytes are in zone 62, equivalent to hexadecimal F800H to FBFFH. For Fig. 15-24*d* zone 48 is used; this means hexadecimal locations C000H to C3FFH.

Example of Gate-Addressed 2114s

Figure 15-25*a* shows how to add zones 48, 62, and 63 to a minimum system. As before, A_{15} to A_{10} select the zone. We have added $IO/\overline{M}$, $\overline{RD}$, and $\overline{WR}$ to produce read and write memory operations. When $IO/\overline{M}$ is high (during I/O operations), a high output appears at each OR gate and none of the 2114s is chip-selected; therefore, the memory is inoperative.

On the other hand, with $IO/\overline{M}$ low, we can get a memory read when $\overline{RD}$ goes low or a memory write when $\overline{WR}$ goes low. For instance, if

$$A_{15}A_{14}A_{13}A_{12}A_{11}A_{10} = 111111$$

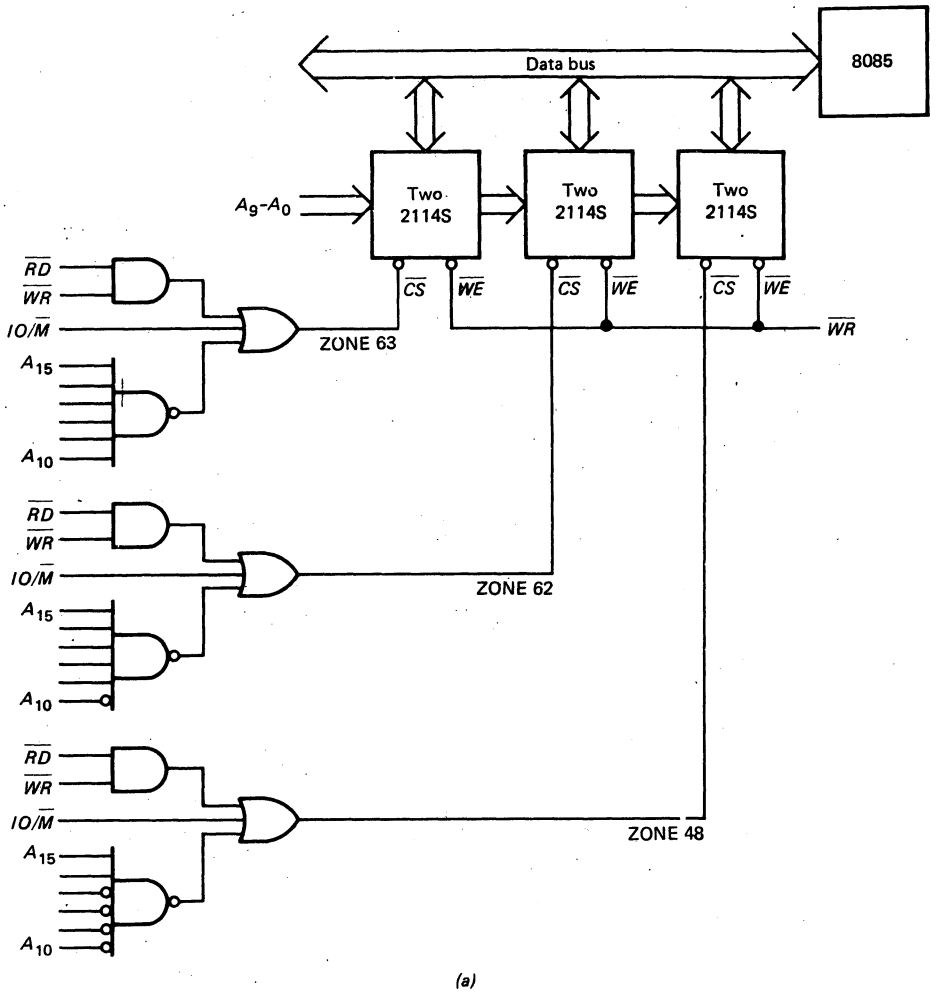
a memory read occurs in zone 63 when $IO/\overline{M}$ and $\overline{RD}$ are low. If

$$A_{15}A_{14}A_{13}A_{12}A_{11}A_{10} = 110000$$

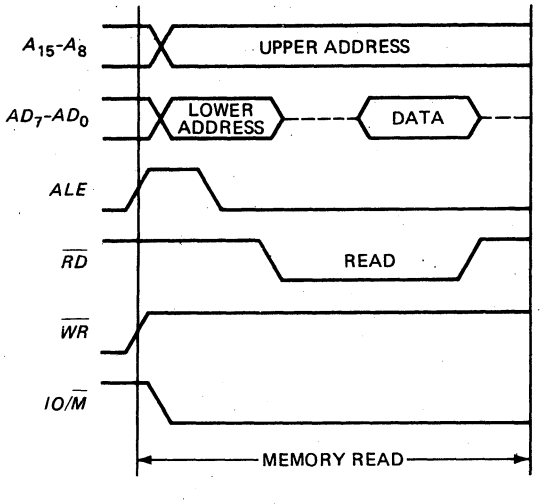
a memory write takes place in zone 48 when $IO/\overline{M}$ and $\overline{WR}$ are low.

Read Timing

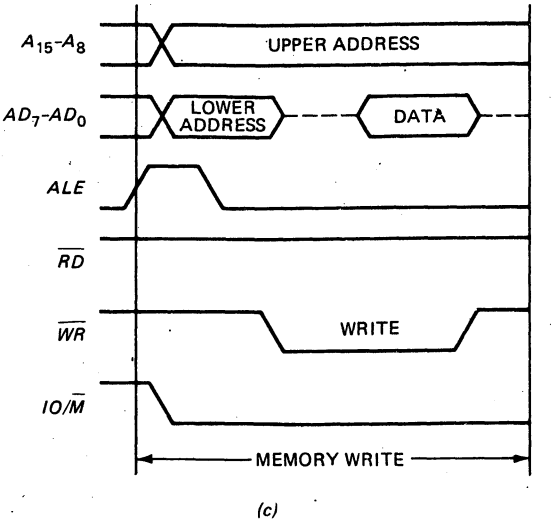
Notice the timing diagram for a memory read (Fig. 15-25*b*). The upper address bits are on the address bus throughout the read cycle; this means that we can connect bits A_{15} to A_{10} of the address bus directly to the zoning gates. Also, inputs A_9 and A_8 to the 2114s can be connected directly to lines A_9 and A_8 of the address bus.



(a)



(b)



(c)

Fig. 15-25 (*a*) Gate addressing zones 48, 62, and 63; (*b*) memory read; (*c*) memory write.

As shown in Fig. 15-25*b*, the lower address bits AD_7 to AD_0 are on the address-data bus only during the early part of a read cycle; therefore, these must be stored in a MAR (not shown) using the falling edge of the ALE to latch the bits. This MAR then addresses the 2114s. (The 8156 and 8355 have on-chip MARs that do this automatically.)

After the lower address bits have been latched in the MAR, the lower address bits disappear from the address-data bus; freeing it for the upcoming memory operation. As shown in Fig. 15-25*b*, $\overline{RD}$ goes low while bracketed by a low $IO/\overline{M}$. This reads the contents of the addressed memory location onto the address-data bus, allowing the 8085 to transfer the data to the target register (instruction register, A, B, C, etc.).

Write Timing

Figure 15-24*c* shows the timing diagram for a memory-write operation. As before, the lower address bits are present on the address-data bus only during the early part of the cycle. The trailing edge of the *ALE* signal is used to latch these bits into a MAR. Later in the cycle, the low $\overline{WR}$ will write the data on the address-data bus into the selected memory location.

Example of Decoder-Addressed 2114s

Gate addressing the memory zones is all right if only a few zones are involved. When many zones are being added, decoder addressing is more convenient. Figure 15-26 shows one way to do it. A_{15} to A_{10} are the zone bits. To enable G_1, A_{15} to A_{13} must be high. As before, A_{12} through A_{10} decode the output lines of the 74LS138. This means that we can zone from

$$A_{15}A_{14}A_{13}A_{12}A_{11}A_{10} = 111000$$

to

$$A_{15}A_{14}A_{13}A_{12}A_{11}A_{10} = 111111$$

equivalent to zones 56 to 63. With Appendix 10, we can list the RAM locations being covered:

Zone 56	E000H–E3FFH
Zone 57	E400H–E7FFH
Zone 58	E800H–EBFFH
Zone 59	EC00H–EFFFH
Zone 60	F000H–F3FFH
Zone 61	F400H–F7FFH
Zone 62	E800H–FBFFH
Zone 63	FC00H–FFFFH

In effect, we have added 8K of RAM to the upper memory.

By adding one or more inverters to the A_{15}, A_{14}, and A_{13} lines we can relocate this 8K of RAM. For example, put an inverter on the A_{13} line and you zone into

$$A_{15}A_{14}A_{13}A_{12}A_{11}A_{10} = 110000$$

to

$$A_{15}A_{14}A_{13}A_{12}A_{11}A_{10} = 110111$$

equivalent to zones 48 to 55.

15-9 DYNAMIC RAMS

The details of addressing and refreshing dynamic RAMs are too complicated to go into here, but a few comments are appropriate. As an approximation, smaller applications (appliances, control systems, instruments, toys, and the like) need from 1K to 8K of ROM and from 64 bytes to 1K of RAM, the mixture of ROM and RAM depending on what is being done. Larger applications (microcomputers, programmable instruments, word processors, etc.) require 1K to 48K of ROM and 4K to 63K of RAM.

ROM expansion is straightforward. We can add more 8355s (or other available ROMs) using either gate addressing or decoder addressing to locate the new ROM zones. RAM expansion, however, raises the question of whether to use static or dynamic RAMs.

As mentioned in Chap. 9, a dynamic RAM has the advantage of containing more memory cells than a static RAM of the same physical size. But the disadvantage is the need to refresh the stored data every few milliseconds. The complicated refresh circuitry is usually not justified unless you need at least 16K of RAM. In other words, static RAMs are adequate for smaller applications and for some larger ones too.

The 2117 typifies the dynamic RAMs that are commercially available. This 16-pin chip is a 16,384 × 1 dynamic RAM, organized as 16,384 words of 1 bit each. This means that you have to use eight 2117s in parallel to get 16K of RAM memory (1 byte in each location). Because 14 bits are needed to address 16,384 locations, the address bits are multiplexed by loading in 7 bits followed by another 7 bits.

Appendix 7 shows how a 64K memory is zoned when using 16K chips. A_{15} and A_{14} are the zone bits. As shown, zone 0 is hexadecimal locations 0000H–3FFFH, zone 1 is 4000H–7FFFH, zone 2 is 8000H–BFFFH, and zone 3 is C000H–FFFFH.

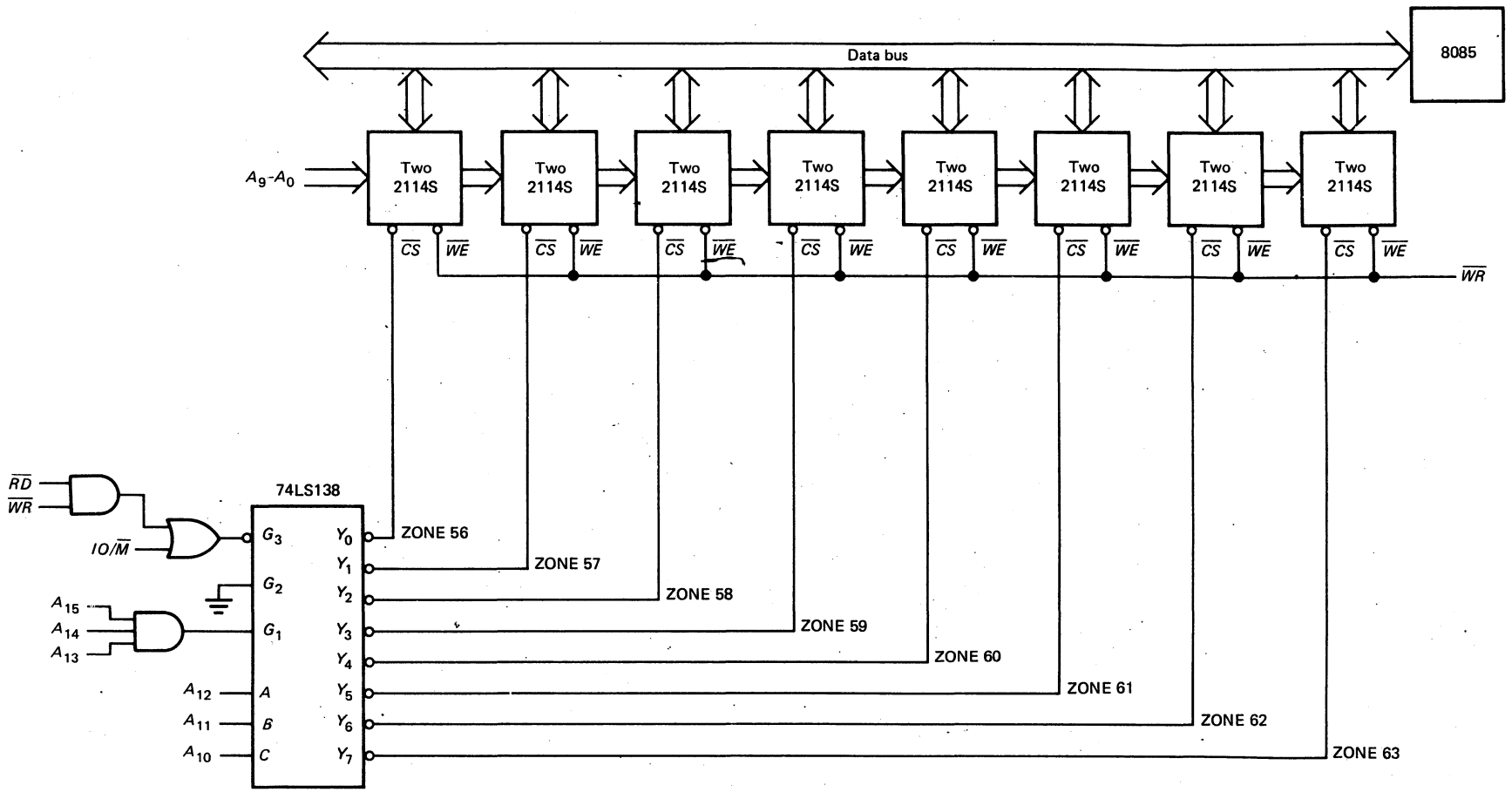


Fig. 15-26 Decoder addressing of zones 56 to 63.

GLOSSARY

command register A register in the 8156 that controls the timer, interrupts, and I/O ports. In a minimum system this register has an address of 20H. This register is loaded with an OUT 20H.

data direction register A register in which each bit determines whether the corresponding pin in a port is an input or output. Abbreviated DDR.

decoder addressing Using the output lines of a decoder to enable ports or memory chips.

full decoding Decoding all the address bits so that each port or memory location has no shadows.

gate addressing Using gates to enable a port or memory chip.

input port A three-state driver that connects input data to the address-data bus when addressed by its port number during the execution of an IN instruction.

memory zone The locations that a memory chip can read from or write into.

output port A latch or register that receives output data from the address-data bus when addressed by its port number during the execution of an OUT instruction.

status register A register in the 8156 that contains information about the timer and the ports. This register is read with an IN 20H.

terminal count The number that is preset in the timer; it determines how many input pulses produce one output square wave or pulse.

strobe $\overline{STB\ A}$ and $\overline{STB\ B}$ are strobe signals when port C is used for handshaking. For input operations these signals indicate that the peripheral device is ready to send data; for output operations the signals indicate that the device has received the data.

timer A presettable 14-bit down counter in the 8156 that produces square waves or pulses.

SELF-TESTING SUMMARY

Read each of the following and provide the missing words. Answers appear at the beginning of the next question.

1. For the 8156 to be active, the _____ input must be high. Ignoring shadows, the RAM locations in the minimum system are _____ to _____.
2. (*CE, 2000H, 20FFH*) In a minimum system, port A has an address of 21H, port B of _____, and port C of _____. The command and status registers both have an address of 20H.
3. (*22H, 23H*) The lower timer byte is addressed with 24H, and the upper timer byte with _____.
4. (*25H*) When port C is used for handshaking, it provides three signals: a strobe, a _____ full, and an _____. Port C can provide handshaking for port A plus 3 bits of output, or it can provide handshaking for ports A and _____.
5. (*buffer, interrupt, B*) The _____ is a presettable 14-bit counter that counts *TIMER IN* pulses. The number that is preset in the timer is called the _____ count.
6. (*timer, terminal*) The timer can produce a single or _____ square wave; or it can produce a single or continuous _____.
7. (*continuous, pulse*) In the 8355 the ROM is organized as _____ words of 8 bits each. For this chip to be active the $\overline{CE}_1$ input must be low and the CE_2 input high. Ignoring foldback, the ROM locations in the minimum system are _____ to _____. To read these memory locations, the $IO/\overline{M}$ signal must be low.
8. (*2,048, 0000H, 07FFH*) When a bit is 0 in a DDR, it makes the corresponding port pin an _____. On the other hand, a 1 bit programs an _____ pin.
9. (*input, output*) To eliminate I/O and memory _____ in a minimum system we can fully decode the address lines to the 8355 and 8156. With full decoding, the only addressable _____ locations in a minimum system are 0000H–07FFH; the only addressable RAM locations are _____.
10. (*shadows, ROM, 2000H–20FFH*) The simplest way to address a port is with _____ addressing. This means using gates to produce the enable signal. When many ports are involved, _____ addressing is more convenient. A 74LS138 can select up to _____ ports.
11. (*gate, decoder, eight*) A pair of 2114s can store _____ words of _____ bits each. Bits A_{15} to A_{10} are called _____ bits because they determine which zone is used. With 2114s, the zones are from 0 to 63. The zone bits can either gate-address or _____ address the desired zone.
12. (*1,024, 8, zone, decoder-*) Static RAMs are adequate for smaller applications and some larger ones. Dynamic RAMs are practical when the RAM needs at least 16K.

PROBLEMS

15-1. Refer to Fig. 15-27.

a. To transfer data during IN operations, is $IO/\overline{M}$ low or high?

b. While data is transferred during an IN operation, is $\overline{WR}$ low or high?

c. Does $\overline{WR}$ go low during OUT operations?

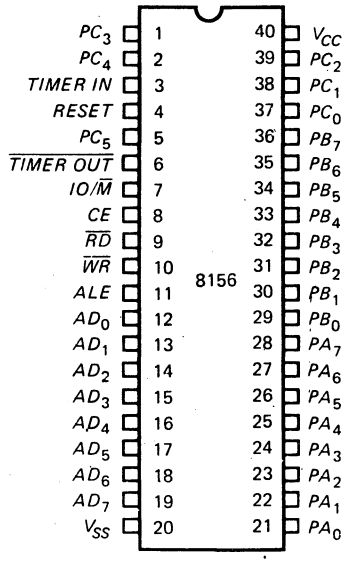


Fig. 15-27

15-2. During the execution of OUT 25H, what are the binary numbers appearing on the buses of Fig. 15-28*a*?

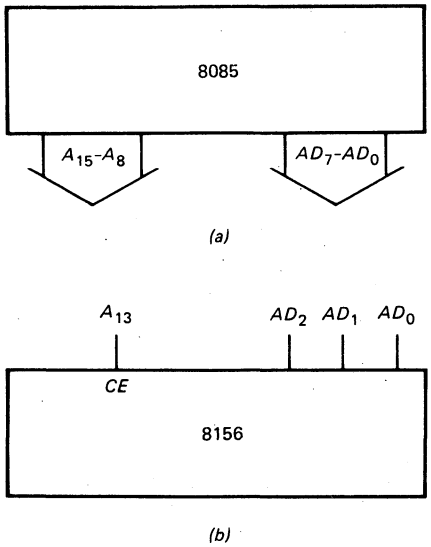


Fig. 15-28

15-3. What do these instructions do?

```
MVI 3EH
OUT 20H
```

15-4. What are the two instructions that do the following: reset timer bits, disable A and B interrupts, provide handshaking for port A only, make port B an input, make port A an output?

15-5. The contents of the command register are 23H. How does port C act?

15-6. After executing an IN 20H, the accumulator contains

$$\mathbf{A} = \text{X011 1000}$$

What do these contents mean?

15-7. Here is a program segment:

```
MVI A, 2BH
OUT 24H
MVI A,86H
OUT 25H
```

What is the terminal count in decimal form? What kind of signal does the timer produce?

15-8. A 3-MHz system clock is connected to the TIMER IN input of an 8156. Write a program that divides this clock by a factor of 2,500 and produces a train of pulses from the *TIMER OUT* pin.

15-9. An 8156 has A_{15} connected to its CE input. A_{14} to A_8 are unconnected, and AD_7 to AD_0 are connected. Ignoring shadows, what are the RAM locations? The ports addresses?

15-10. In an 8355, DDR A contains 8CH. How does each pin in port A act?

15-11. What do these instructions do:

```
MVI 3DH
OUT 03H
```

15-12. The $\overline{CE}_1$ input of an 8355 is connected to A_{13} and the CE_2 input to A_{12}. Ignoring foldover, what are the ROM addresses? The port addresses?

15-13. The 8156 of Fig. 15-29 has RAM locations from 2000H to 20FFH. How many zone bits are there? If all bubbles are removed, what are the new RAM locations?

15-14. Inverters are placed on the A_9 and A_8 address lines of Fig. 15-30*a*. What is the new port number?

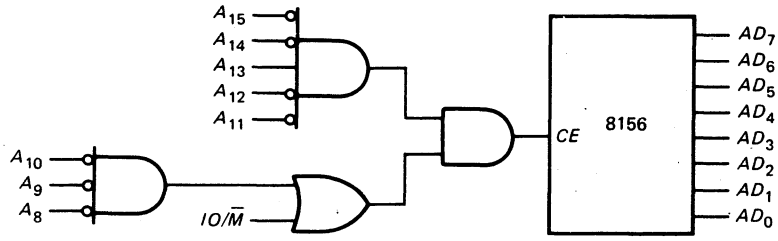
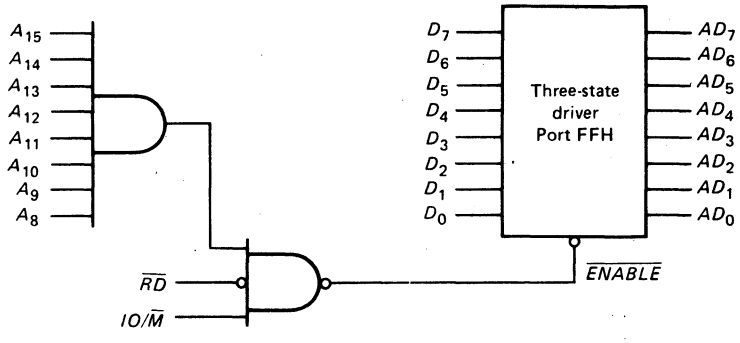
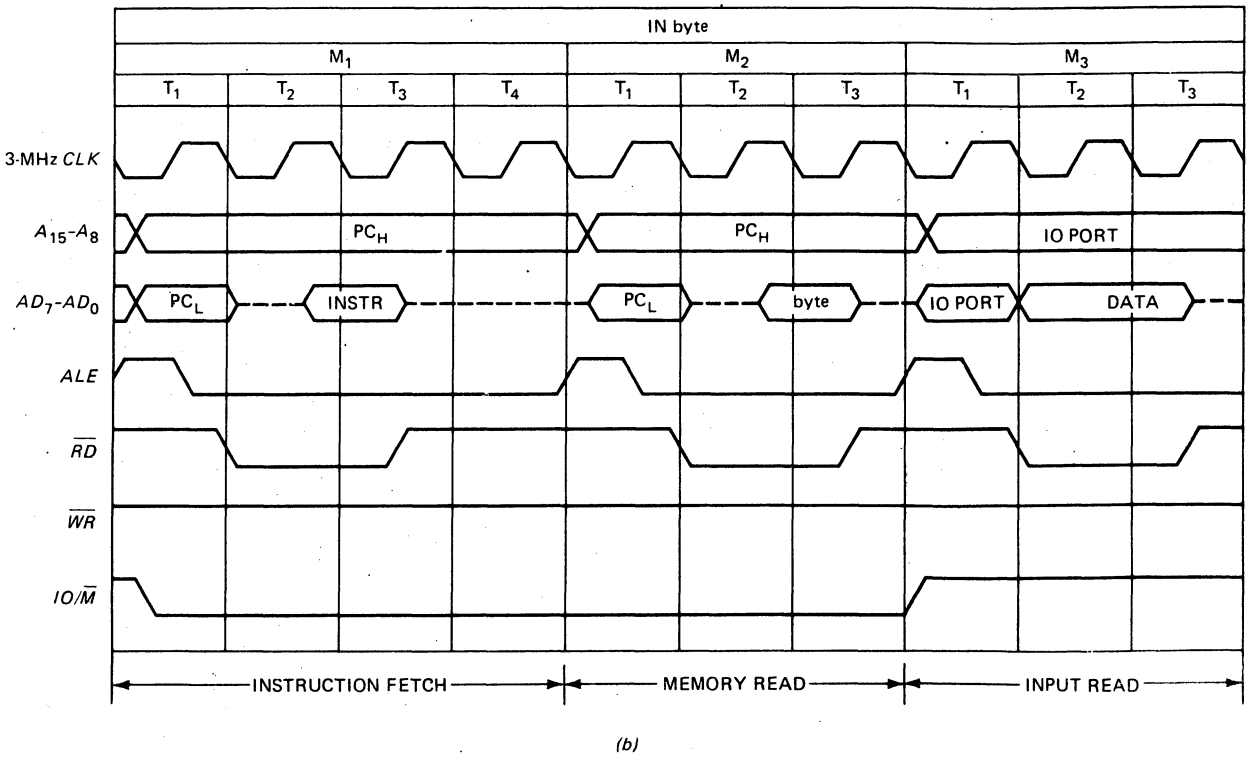


Fig. 15-29



(a)



(b)

Fig. 15-30

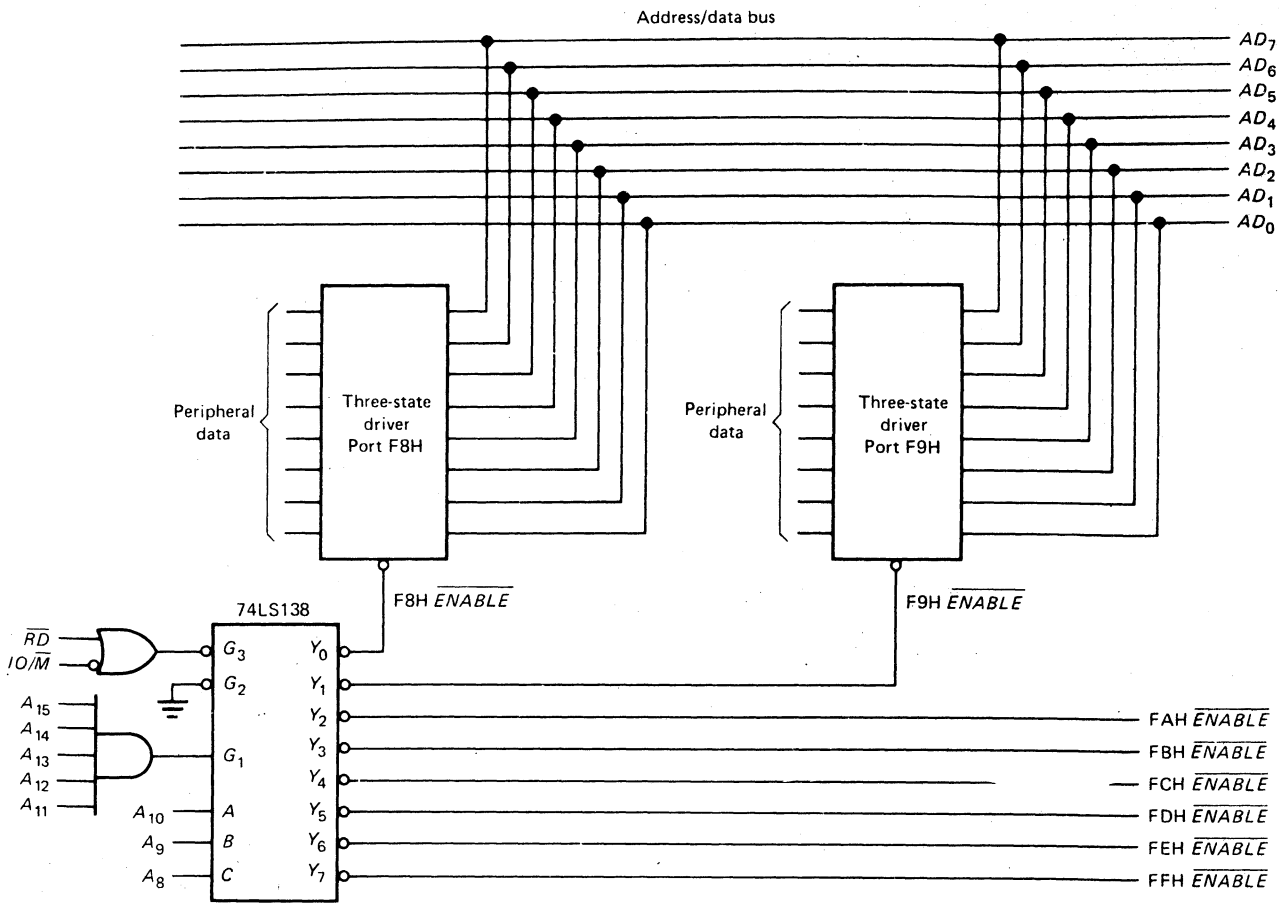
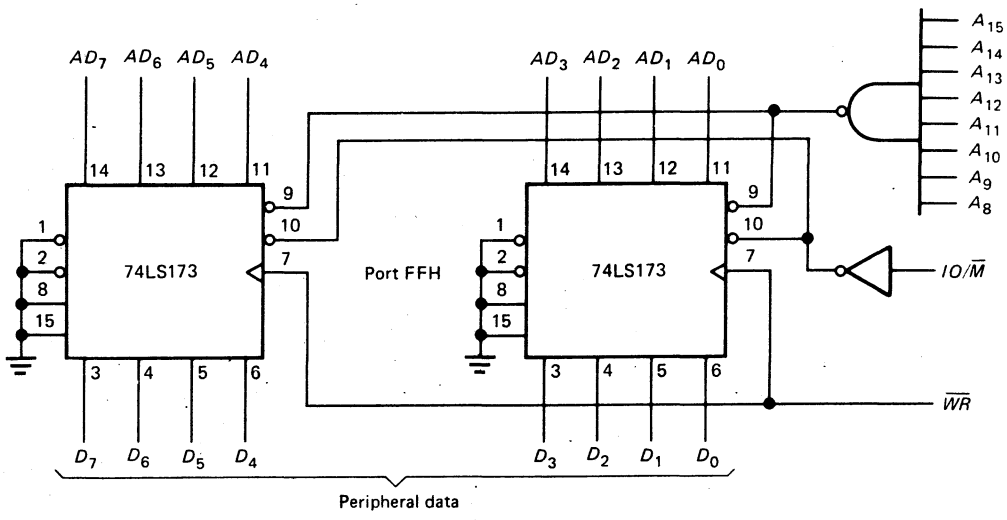
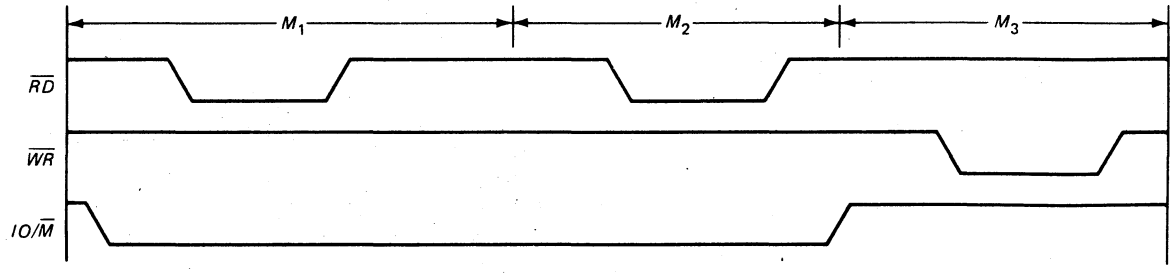

Address/data bus
AD_7
AD_6
AD_5
AD_4
AD_3
AD_2
AD_1
AD_0
Peripheral data
Three-state driver Port F8H
Peripheral data
Three-state driver Port F9H
F8H $\overline{ENABLE}$
F9H $\overline{ENABLE}$
74LS138
$\overline{RD}$
$IO/\overline{M}$
A_{15}
A_{14}
A_{13}
A_{12}
A_{11}
A_{10}
A_9
A_8
G_3
G_2
G_1
A
B
C
Y_0
Y_1
Y_2
Y_3
Y_4
Y_5
Y_6
Y_7
FAH $\overline{ENABLE}$
FBH $\overline{ENABLE}$
FCH $\overline{ENABLE}$
FDH $\overline{ENABLE}$
FEH $\overline{ENABLE}$
FFH $\overline{ENABLE}$

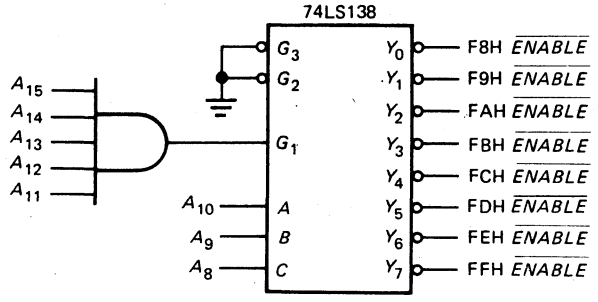
Fig. 15-31


AD_7 AD_6 AD_5 AD_4
AD_3 AD_2 AD_1 AD_0
A_{15} A_{14} A_{13} A_{12} A_{11} A_{10} A_9 A_8
74LS173
Port FFH
74LS173
$IO/\overline{M}$
$\overline{WR}$
D_7 D_6 D_5 D_4
D_3 D_2 D_1 D_0
Peripheral data

(a)

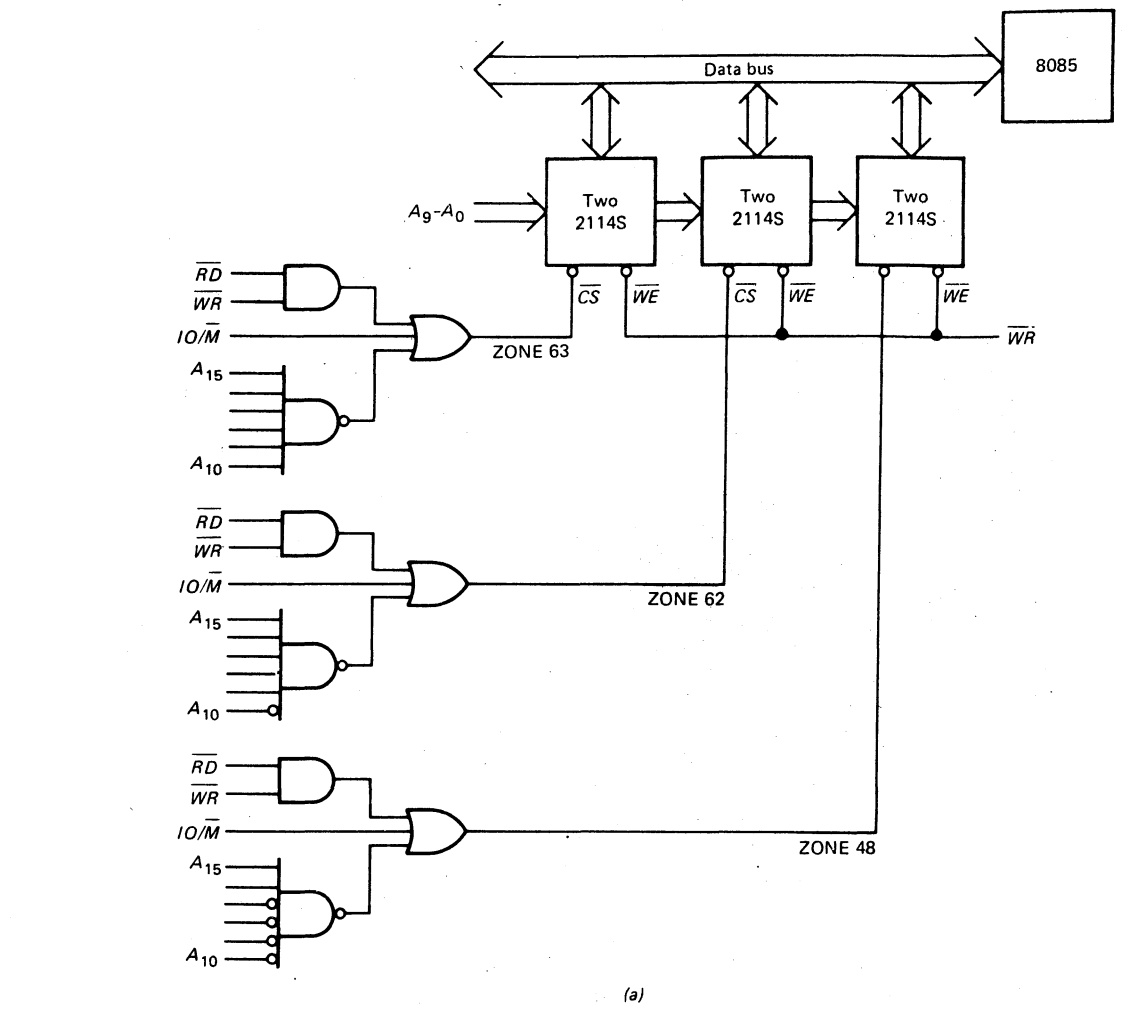

M_1
M_2
M_3
$\overline{RD}$
$\overline{WR}$
$IO/\overline{M}$

(b)

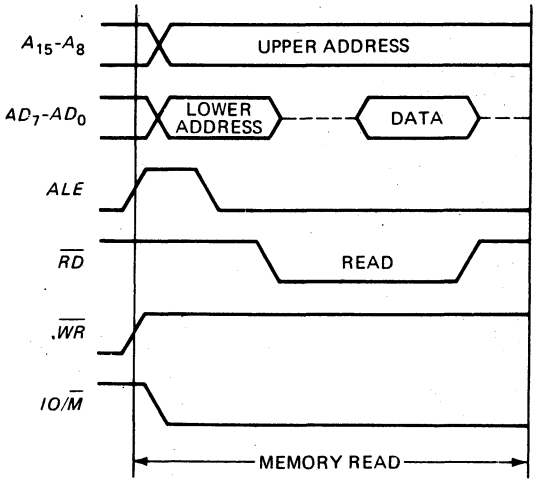

74LS138
A_{15} A_{14} A_{13} A_{12} A_{11}
A_{10} A_9 A_8
G_3 G_2 G_1 A B C
Y_0 F8H $\overline{ENABLE}$
Y_1 F9H $\overline{ENABLE}$
Y_2 FAH $\overline{ENABLE}$
Y_3 FBH $\overline{ENABLE}$
Y_4 FCH $\overline{ENABLE}$
Y_5 FDH $\overline{ENABLE}$
Y_6 FEH $\overline{ENABLE}$
Y_7 FFH $\overline{ENABLE}$

(c)

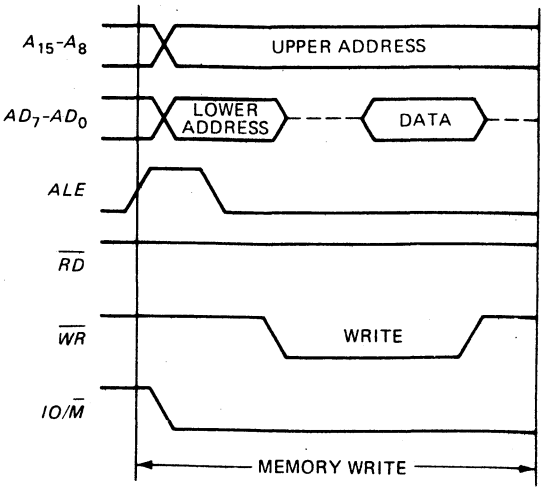
Fig. 15-32


Data bus
8085
Two 2114S
Two 2114S
Two 2114S
A_9-A_0
$\overline{RD}$
$\overline{WR}$
IO/$\overline{M}$
A_{15}
A_{10}
$\overline{CS}$
$\overline{WE}$
ZONE 63
ZONE 62
ZONE 48
$\overline{WR}$

(a)


A_{15}-A_8
UPPER ADDRESS
AD_7-AD_0
LOWER ADDRESS
DATA
ALE
$\overline{RD}$
READ
$\overline{WR}$
IO/$\overline{M}$
MEMORY READ

(b)


A_{15}-A_8
UPPER ADDRESS
AD_7-AD_0
LOWER ADDRESS
DATA
ALE
$\overline{RD}$
$\overline{WR}$
WRITE
IO/$\overline{M}$
MEMORY WRITE

(c)

Fig. 15-33

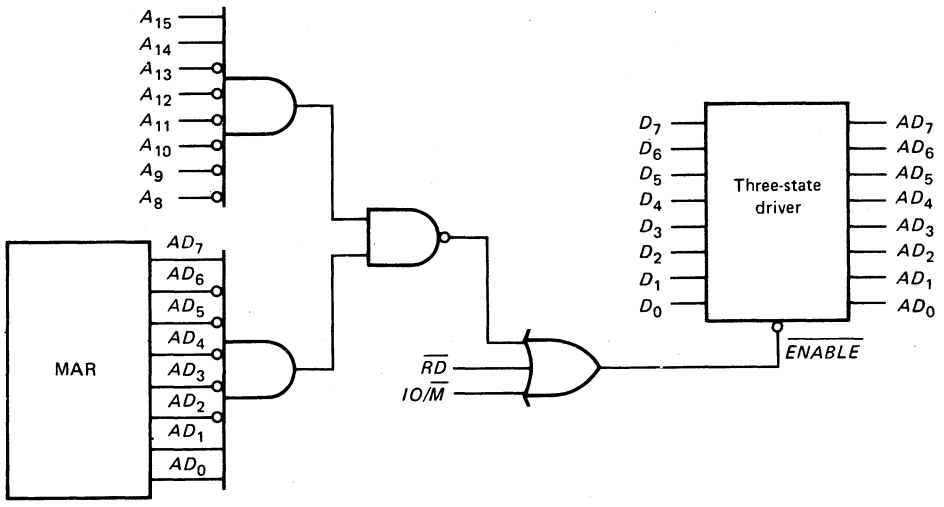


Fig. 15-34 Memory-mapped I/O.

15-15. How would you change Fig. 15-30*a* to address port 40H?

15-16. An inverter is used on the A_{15} input of Fig. 15-31. What are the new port numbers for the circuit?

15-17. What change can you make in Fig. 15-32*a* to create output port 32H?

15-18. An inverter is added to each of the A_{15} inputs in Fig. 15-33*a*. What are the new memory zones? Express the answer in zone numbers and hexadecimal equivalents.

15-19. The 2141 is a 4,096 × 1 static RAM. Eight of these chips in parallel produce 4,096 bytes. How many zone bits does this imply? How many zones are there? If these chips are gate addressed into zone 12, what are the hexadecimal equivalents of the memory locations? (Use Appendix 8.)

15-20. Using IN and OUT instructions is only one method of I/O operation. Another approach relies on *memory-mapped I/O*. The idea is to treat a port as a memory location and to access it with memory-reference instructions like

```
MOV reg,M
LDA address
ADD M
XRA M
```

and others. The advantage of memory-mapped I/O is the added control these new instructions bring. The disadvantage is more decoding of the address bits. Figure 15-34 is an example of a memory-mapped input port. Answer the following:

a. Can the three-state driver be enabled when $IO/\overline{M}$ is high?

b. If $\overline{RD}$ and $IO/\overline{M}$ are low, what are the 16 address bits that enable the three-state driver?

c. If $D_7D_6D_5D_4D_3D_2D_1D_0$ = 1001 1110, what are the contents of the accumulator after an LDA C083H is executed?

d. Suppose the accumulator contents are

$$A = 1011\ 0011$$

If the $D_7D_6D_5D_4D_3D_2D_1D_0$ = 1110 0111, what does the accumulator contain after LXI H,C083H and an ANA M are executed?

15-21. How can you change Fig. 15-34 to get an input port memory-mapped into location FFFFH?

The Analog Interface

16

The data in a microprocessor is in digital form. This differs from the outside world where data is in analog (continuous) form. To get digital data, we need to use an *analog-to-digital* (A/D) *converter;* it will convert analog voltage or current into an equivalent digital word.

Conversely, after a CPU has processed data, it is often necessary to convert the digital answer into an analog voltage or current. This conversion requires a *digital-to-analog* (D/A) converter.

The *analog interface* is the boundary where digital and analog meet, where the microcomputer connects to the outside world. At this interface, we find either an A/D converter (input side) or a D/A converter (output side). This chapter discusses some of the hardware and software found at the analog interface.

16-1 OP-AMP BASICS

Let us briefly review the *operational amplifier* (op amp) because this device is used with D/A and A/D converters. We will zero in on the key features that make the op amp useful at the analog interface.

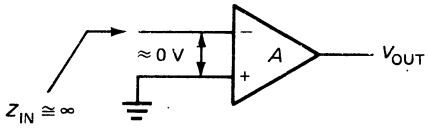


Fig. 16-1 Operational amplifier.

Virtual Ground

Figure 16-1 shows the symbol for an op amp. V_{OUT} is the output voltage with respect to ground. A is the open-loop voltage gain of the op amp, often more than 100,000. When connected as an inverter, the noninverting input (+ input) is grounded. The inverting input (− input) receives the signal voltage.

Because the voltage gain of an op amp is so large, the input voltage is in microvolts. To a first approximation, the input voltage may be treated as 0 V. Furthermore, the input impedance of the inverting input approaches infinity (sometimes FETs are used for the input stage, as in BIFET op amps). These key features, zero input voltage and infinite input impedance, make the inverting input a *virtual ground point.*

How is a virtual ground different from an ordinary ground? An ordinary ground has zero voltage while sinking any amount of current. A virtual ground, however, is a ground for voltage but not for current; it has zero voltage but can sink no current. In the discussion that follows, we will approximate the inverting input of an op amp as a virtual ground point: this means zero voltage and zero current.

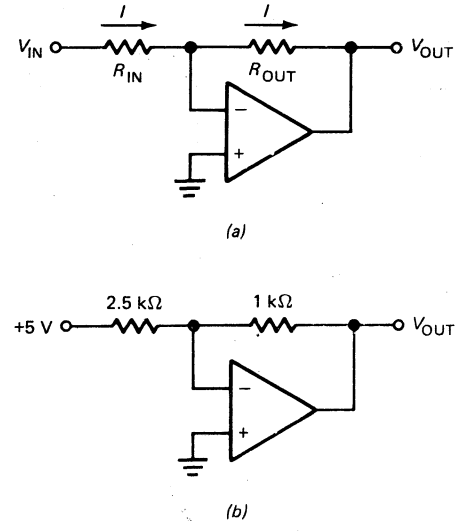


Fig. 16-2 Output current equals input current.

Output Voltage and Current

Figure 16-2*a* shows an inverting op amp with input and output resistors. V_{IN} is the input voltage with respect to ground, and V_{OUT} is the output voltage with respect to ground. Because of the high gain and input impedance, we

can approximate the inverting input as a virtual ground point. Therefore, all the input voltage appears across the input resistor, which means that the input current is

$$I = \frac{V_{IN}}{R_{IN}} \qquad (16\text{-}1)$$

Since none of the input current can enter the virtual ground point, it must pass through the output resistor. In other words, the output current equals the input current. And the output voltage is

$$V_{OUT} = -IR_{OUT} \qquad (16\text{-}2)$$

The minus sign indicates phase inversion. If the input voltage is positive, the output voltage is negative.

As an example of calculating input current and output voltage, look at Fig. 16-2*b*. The input current is

$$I = \frac{5\text{ V}}{2.5\text{ k}\Omega} = 2\text{ mA}$$

The output voltage is

$$V_{OUT} = -2\text{ mA} \times 1\text{ k}\Omega = -2\text{ V}$$

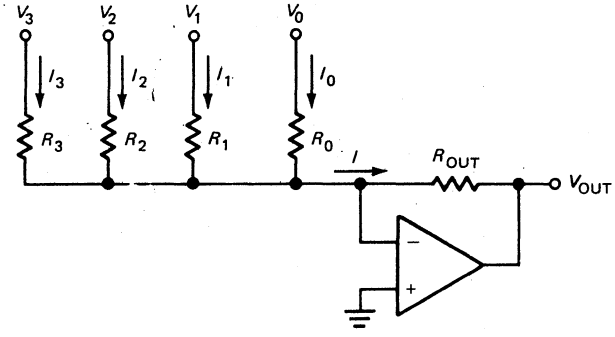


Fig. 16-3 Output current equals sum of input currents.

Summing Circuit

Figure 16-3 is an op-amp circuit whose output current is the *sum* of the input currents. Here is the proof. Because of the virtual ground point, each input voltage appears across its resistor. This means that the input currents are

$$I_3 = \frac{V_3}{R_3} \quad I_2 = \frac{V_2}{R_2} \quad I_1 = \frac{V_1}{R_1} \quad I_0 = \frac{V_0}{R_0}$$

Kirchhoff's current law gives a total input current of

$$I = I_3 + I_2 + I_1 + I_0$$

Again, the virtual ground guarantees that all this input current goes through the output resistor. As before,

$$V_{OUT} = -IR_{OUT}$$

16-2 A BASIC D/A CONVERTER

The op-amp summing circuit can be used to build a D/A converter by selecting input resistors that are weighted in binary progression. Figure 16-4 gives you the idea. V_{REF} is an accurate reference voltage, and the resistors are precision resistors to get accurate input currents. The switches can be open or closed. When all switches are open, all input currents are zero and the output current is zero.

All Bits High

When all switches are closed, the input currents are

$$I_3 = \frac{V_{REF}}{R} \quad I_2 = \frac{V_{REF}}{2R} \quad I_1 = \frac{V_{REF}}{4R} \quad I_0 = \frac{V_{REF}}{8R}$$

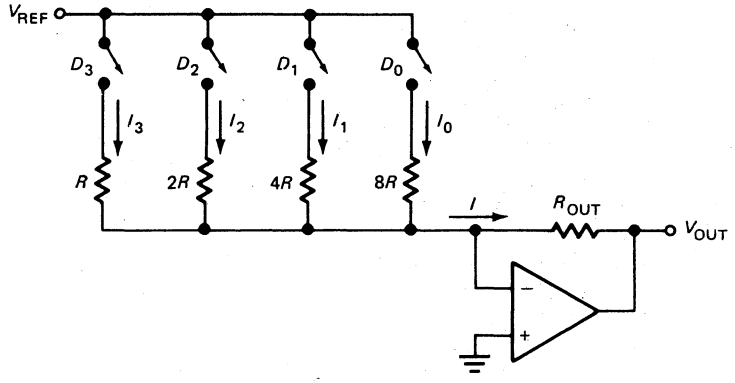


Fig. 16-4 D/A conversion with binary-weighted resistors.

The output current with all switches closed is the sum of all input currents and equals

$$I = \frac{V_{REF}}{R}(1 + 0.5 + 0.25 + 0.125) \quad (16\text{-}3)$$

$$I = 1.875\frac{V_{REF}}{R}$$

By opening and closing switches we can produce 16 different output currents from 0 to $1.875V_{REF}/R$.

Any Digital Input

If 0 stands for an open switch and 1 for a closed switch, we can rewrite Eq. 16-3 as

$$I = \frac{V_{REF}}{R}(D_3 + 0.5D_2 + 0.25D_1 + 0.125D_0) \quad (16\text{-}4)$$

In powers of 2,

$$I = \frac{V_{REF}}{R}(D_3 + 2^{-1}D_2 + 2^{-2}D_1 + 2^{-3}D_0) \quad (16\text{-}5)$$

This says that the output current is the sum of binary-weighted input currents. In other words, we have a D/A converter. For instance, suppose V_{REF} = 5 V and R = 5 kΩ. Then the total output current varies from 0 to 1.875 mA, as shown in Table 16-1.

TABLE 16-1. WEIGHTED D/A CONVERTER

D_3	D_2	D_1	D_0	Output current, mA	Fraction of maximum
0	0	0	0	0	0
0	0	0	1	0.125	$\frac{1}{15}$
0	0	1	0	0.25	$\frac{2}{15}$
0	0	1	1	0.375	$\frac{3}{15}$
0	1	0	0	0.5	$\frac{4}{15}$
0	1	0	1	0.625	$\frac{5}{15}$
0	1	1	0	0.75	$\frac{6}{15}$
0	1	1	1	0.875	$\frac{7}{15}$
1	0	0	0	1	$\frac{8}{15}$
1	0	0	1	1.125	$\frac{9}{15}$
1	0	1	0	1.25	$\frac{10}{15}$
1	0	1	1	1.375	$\frac{11}{15}$
1	1	0	0	1.5	$\frac{12}{15}$
1	1	0	1	1.625	$\frac{13}{15}$
1	1	1	0	1.75	$\frac{14}{15}$
1	1	1	1	1.875	$\frac{15}{15}$

Current Switches

Figure 16-5 shows how we can transistorize the switching. Data bits D_3 through D_0 drive the bases of the transistors through the current-limiting resistors. When a bit is high, it produces enough base current to saturate its transistor. When a bit is low, the transistor is cut off. Since each transistor is saturated or cut off, it acts like a closed or open switch. (Base resistance is not critical; it need only be less than collector resistance multiplied by β_{dc}.)

If the lower 4 bits of an output port are connected to D_3 to D_0, the circuit of Fig. 16-5 will convert digital data to analog current. For instance, assume port 22H has been programmed as an output port in a minimum system. If the lower 4 bits of port 22H are connected to D_3 to D_0, this program segment will operate the D/A converter:

Label	Mnemonic	Comment
	MVI A,FFH	;Initialize accumulator
LOOP:	INR A	;Count up
	OUT 22H	;Output nibble
	JMP LOOP	;Get next nibble

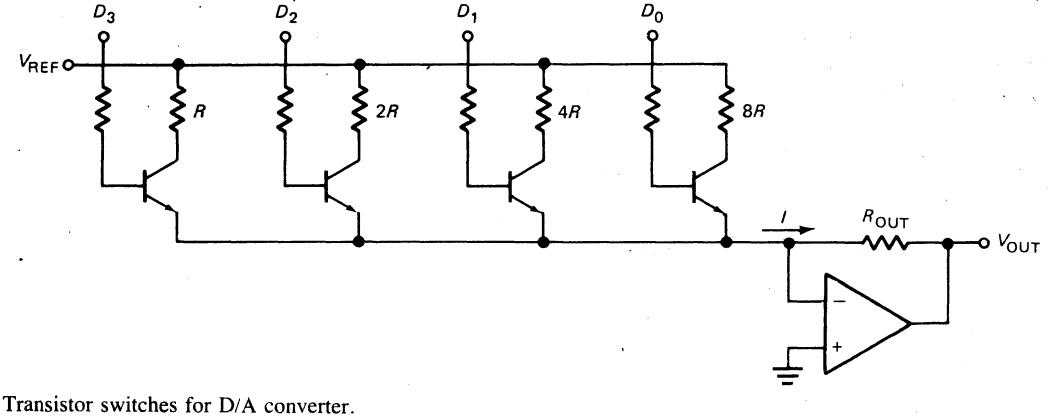


Fig. 16-5 Transistor switches for D/A converter.

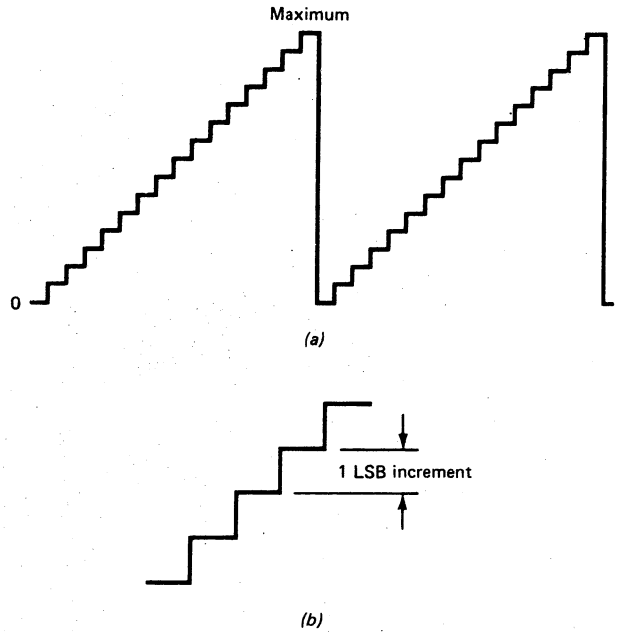


Fig. 16-6 (*a*) Staircase output current; (*b*) each step equals an LSB increment.

The first INR A produces accumulator contents of 00H. Subsequent INR executions produce 01H, 02H, . . . , 0FH, 10H, 11H, . . . , 1FH, 20H, 21H, . . . , FFH. As far as D_3 to D_0 are concerned, they see a nibble stream of 0000, 0001, 0010, 0011, . . . , 1111, 0000, 0001, and so on.

Figure 16-6*a* illustrates how the output current of the D/A converter appears. As each input nibble is latched into port 22H, the output current moves one step higher until reaching the maximum current. Then the cycle repeats. If all resistors are exact and all transistors matched, all steps are identical in size.

Resolution

In the perfect staircase of Fig. 16-6*b* a step is called an *LSB increment* because it is produced by a change in the LSB. One way to measure the quality of a D/A converter is its *resolution*, the ratio of the LSB increment to the maximum output. As a formula,

$$\text{Resolution} = \frac{1}{2^n - 1} \qquad (16\text{-}6)$$

For instance, a 4-bit D/A converter has a resolution of

$$\text{Resolution} = \frac{1}{2^4 - 1} = \frac{1}{15}$$

This is sometimes read as 1 part in 15.

The number of different steps an n-bit converter produces is

$$\text{Steps} = 2^n - 1 \qquad (16\text{-}6a)$$

Therefore, an alternative way to think of resolution is

$$\text{Resolution} = \frac{1}{\text{steps}} \qquad (16\text{-}6b)$$

Percent resolution is given by

$$\text{Percent resolution} = \text{resolution} \times 100\% \qquad (16\text{-}7)$$

If the resolution is 1 part in 15, then

$$\text{Percent resolution} = \tfrac{1}{15} \times 100\% = 6.67\%$$

The greater the number of bits, the better the resolution. With Eqs. 16-6 and 16-7 we can calculate the resolution and percent resolution for more bits. Table 16-2 is a summary of the resolution for converters with 4 to 18 bits.

Because the number of bits determines the resolution in Eq. 16-6, an indirect way to specify resolution is by stating the number of bits. For instance, an 8-bit converter has 8-bit resolution, a 10-bit converter has 10-bit resolution, and so on. This is a quick and easy way to pin down the resolution. When necessary, Eqs. 16-6, 16-6*a*, and 16-7 can give additional information.

Accuracy

In a D/A converter, *absolute accuracy* refers to how close each output current is to its ideal value. In Fig. 16-5 absolute accuracy depends on the reference voltage, resistor tolerance, transistor mismatch, and so forth. In a typical application, a trimmer adjustment is included to set the full-scale output at a preassigned value.

Relative accuracy refers to how close each output level is to its ideal fraction of full-scale output. With a 4-bit

TABLE 16-2. RESOLUTION

Bits	Resolution	Percent
4	1 part in 15	6.67
6	1 part in 63	1.59
8	1 part in 255	0.392
10	1 part in 1,023	0.0978
12	1 part in 4,095	0.0244
14	1 part in 16,383	0.0061
16	1 part in 65,535	0.00153
18	1 part in 262,143	0.000381

converter, the ideal output levels as a fraction of full-scale should be 0, $\frac{1}{15}$, $\frac{2}{15}$, $\frac{3}{15}$, and so on. Because data sheets specify relative accuracy rather than absolute accuracy, our subsequent discussions will emphasize relative accuracy.

Relative accuracy depends mainly on the tolerance of the weighted resistors in Fig. 16-5. If they are exactly R, $2R$, $4R$, and $8R$, all steps equal 1 LSB increment in Fig. 16-6*a*. When the resistors depart from ideal values, the steps may be larger or smaller than 1 LSB increment.

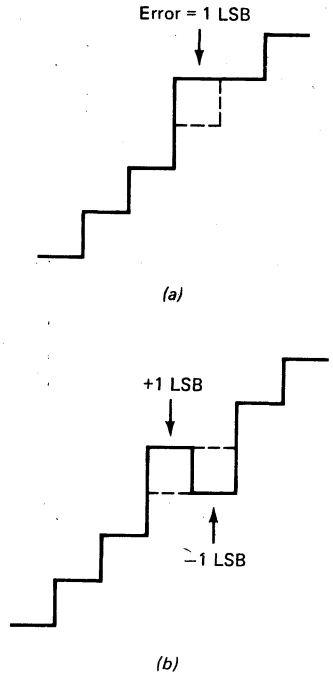


Fig. 16-7 Error specified in LSB increments.

Errors are specified in terms of LSB increments. For instance, Fig. 16-7*a* shows an error of 1 LSB; the actual output (solid line) differs from the ideal output (dashed line) by 1 LSB increment. If a negative error follows a positive error, the staircase can fall as shown in Fig. 16-7*b*. Here you see an error of +1 LSB followed by an error of −1 LSB.

Monotonicity

A *monotonic D/A converter* is one that produces an increase in output current for each successive digital input. The staircases of Fig. 16-7*a* and *b* are not monotonic because they do not produce an increase for each digital input. Figure 16-7*a* is almost monotonic, but Fig. 16-7*b* is far from monotonic. Monotonicity is the least we can expect from a D/A converter because it only makes sense; the output should increase when the input does.

For a D/A converter to be monotonic the error must be less than $\pm\frac{1}{2}$ LSB at each output level. Why? Because in the worst case, a $+\frac{1}{2}$-LSB error followed by a $-\frac{1}{2}$-LSB error produces the critical level where monotonicity is about to be lost. Figure 16-8 illustrates this critical case, an error of $+\frac{1}{2}$ LSB followed by an error of $-\frac{1}{2}$ LSB. If the error of a converter is less than $\pm\frac{1}{2}$ LSB for each output level, we are guaranteed a rising current for each successive digital input. Almost all commercially available D/A converters are monotonic because they have an accuracy of better than $\pm\frac{1}{2}$ LSB at each output level.

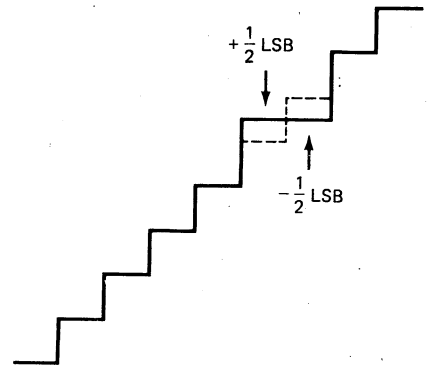


Fig. 16-8 Critical level for monotonicity.

Settling Time

After you apply a digital input, it takes a D/A converter anywhere from nanoseconds to microseconds to produce the correct output. *Settling time* is defined as the time it takes for the converter output to stabilize to within $\frac{1}{2}$ LSB of its final value. This time depends on the stray capacitance, saturation delay time, and other factors. Settling time is important because it places a limit on how fast you can change the digital inputs.

Disadvantages of Weighted Resistors

For a weighted-resistor circuit to be monotonic the tolerance of the resistors must be less than the percent resolution. For instance, if the resolution is $\frac{1}{15}$ (6.67 percent), resistors with a tolerance of less than ±6.67 percent will produce a monotonic staircase. If the resolution is $\frac{1}{255}$ (about 0.4 percent), the resistors need a tolerance of better than ±0.4 percent for a monotonic output. As you see, 4 bits are no problem, but 8 bits are.

Another difficulty arises with weighted resistors. As the number of bits increases, the range of resistance values gets awkward. For 8 bits, we need resistances of R, $2R$, $4R$, . . . , $128R$. The largest resistance is 128 times the smallest. For a 12-bit converter, the largest resistance needs to be 2,048 times the smallest. Because of the tolerance and range problems, mass production of weighted-resistor D/A converters is impractical.

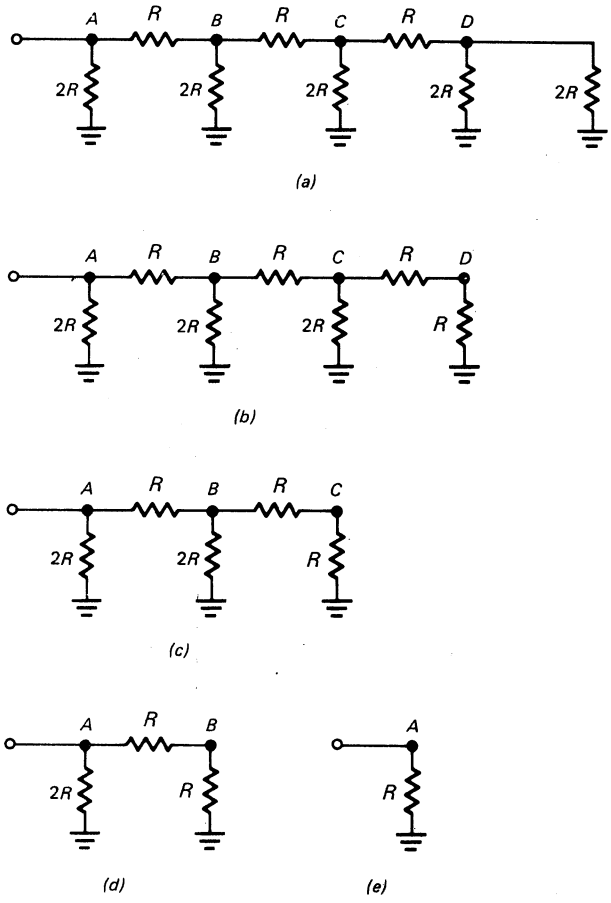


Fig. 16-9 *R*-2*R* ladder.

16-3 THE LADDER METHOD

One way to get around the problems of a binary-weighted resistors is to use a *ladder* circuit. Figure 16-9*a* is an example of the *R*-2*R* ladder commonly used in integrated D/A converters. Only two resistance values are needed; this eliminates the range problem. Furthermore, since the resistors are on the same chip, they have almost identical characteristics; this minimizes the tolerance problem. In other words, as the number of bits increases, an integrated ladder can divide the current much more accurately than a binary-weighted circuit.

Ladder Properties

An *R*-2*R* ladder does something interesting to the impedance at different points in the circuit. To begin with, the two resistors at node D in Fig. 16-9*a* are in parallel and may be reduced to an equivalent resistance *R*, shown in Fig. 16-9*b*. Now, to the right of node C we have *R* in series with *R*, a total of 2*R*. Since node C has 2*R* is in parallel with 2*R*, the circuit reduces to Fig. 16-9*c*.

Looking into the left side of node B (Fig. 16-9*c*), we see 2*R* in parallel with 2*R*. Therefore, the circuit reduces to Fig. 16-9*d*. Again, 2*R* is in parallel with 2*R*, so the circuit reduces to the single *R* shown in Fig. 16-9*e*.

Figure 16-10 summarizes ladder impedances. Do you see the point? Looking into the left side of a node, we always see an equivalent resistance of *R*. Just to the right of each node, we always see a resistance of 2*R*. This impedance phenomenon is the key to analyzing modern D/A converters because they use the ladders instead of weighted resistors.

Binary Division of Current

Figure 16-11 shows how a ladder can divide the current into binary levels. The typical D/A converter has a reference current set by the user. In this example, the reference current is 2 mA. The bottom of each 2*R* resistor is grounded in either switch position. When a switch is to the right, the current through a 2*R* resistor flows to the upper ground. When a switch is to the left, the lower ground sinks the current. With all the switches to the right, as shown in Fig. 16-11, I_{OUT} is zero.

Here is how the ladder divides the 2 mA of reference current. Just to the right of node *A* we see an equivalent resistance of 2*R*. Therefore, the 2 mA of input current divides equally at node *A*. Similarly, at node *B* we see 2*R* in parallel with 2*R*; again, the current divides equally into 0.5-mA branch currents. This process continues through the ladder, so that we wind up with the upper grounds sinking 1, 0.5, 0.25, and 0.125 mA.

Other Switch Positions

When we move the switches, we do not change the way the current divides at the nodes. It still divides equally at each node. But when a switch is to the left, it steers the

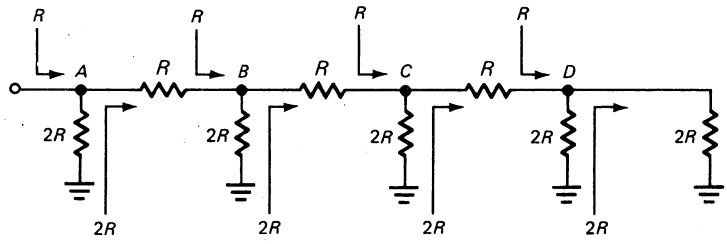


Fig. 16-10 Ladder impedances.

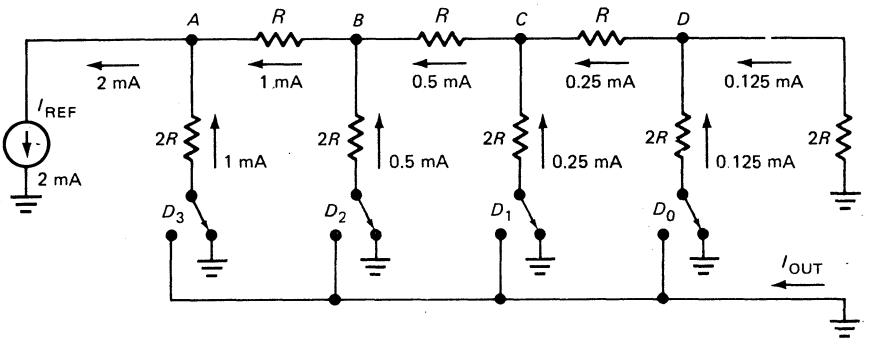


Fig. 16-11 D/A conversion with *R-2R* ladder.

current into the lower ground. Bits D_3 to D_0 control the transistorized switches. From previous discussions, we can see that

$$I_{OUT} = (D_3 + 2^{-1}D_2 + 2^{-2}D_1 + 2^{-3}D_0)\frac{I_{REF}}{2} \quad (16\text{-}8)$$

Therefore, the output current of a 4-bit ladder is from 0 to $\frac{15}{16}I_{REF}$.

More Bits

A similar analysis applies to longer ladders. The output current is

$$I_{OUT} = (D_{n-1} + 2^{-1}D_{n-2} + \cdots + 2^{1-n}D_0)\frac{I_{REF}}{2} \quad (16\text{-}9)$$

For instance, an 8-bit ladder produces a maximum output current of $\frac{255}{256}I_{REF}$. The LSB increment is $\frac{1}{255}I_{REF}$.

Why Steer Current

Current steering may seem more complicated than necessary, but there is good reason for it. The currents throughout the ladder remain constant; all that changes are the ground points. Constant current implies constant voltage, which means that stray capacitance in the ladder has little effect. In other words, we do not get the usual exponential charge and discharge associated with a change in voltage. This reduces the settling time. For this reason, IC converters often use the current-steering approach shown in Fig. 16-11.

16-4 THE DAC0808

There are many commercially available D/A converters. The least expensive have resolutions of 8 to 12 bits. The most expensive have resolutions of 16 to 18 bits. Almost all are monotonic with less than $\pm\frac{1}{2}$-LSB error at each output level.

The DAC0808 is an example. This inexpensive and widely used 8-bit D/A converter contains a reference current source, an *R-2R* ladder, and eight transistor switches to steer the binary currents as previously discussed. An external voltage and resistor are used to set the reference current to a typical value of 2 mA. The DAC0808 has a settling time of 150 ns and a relative accuracy of $\pm\frac{1}{2}$LSB.

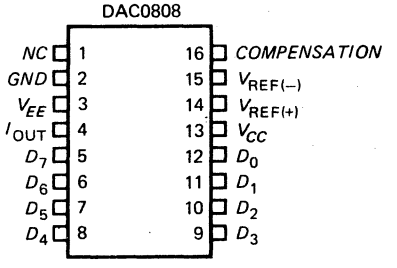


Fig. 16-12 Pinout for DAC0808.

Pinout

Figure 16-12 shows the pinout. Typically, the pins are used as follows. Pin 1 is unused (NC stands for no connection). Pin 2 is chip ground. Pin 3 (V_{EE}) is -15 V. Pin 4 is the ground return for the current out of the ladder; this pin usually connects to an op amp. Pins 5 to 12 are for the 8 bits of input data. Pin 13 (V_{CC}) is $+5$ V. Pin 14 is connected to a positive supply through a resistance R_{14}, and pin 15 is grounded through a resistance R_{15}. Finally, a capacitor between pin 16 and pin 3 frequency-compensates the device.

A Circuit

Figure 16-13 shows the data bits of a DAC0808 connected to port 22H of a minimum system. Pin 2 of the DAC0808 is grounded, and a 15-pF compensating capacitor is between pins 16 and 3. A $+5$-V supply sets up a reference current for the ladder. Trimmer R_{14} allows you to adjust this to 2

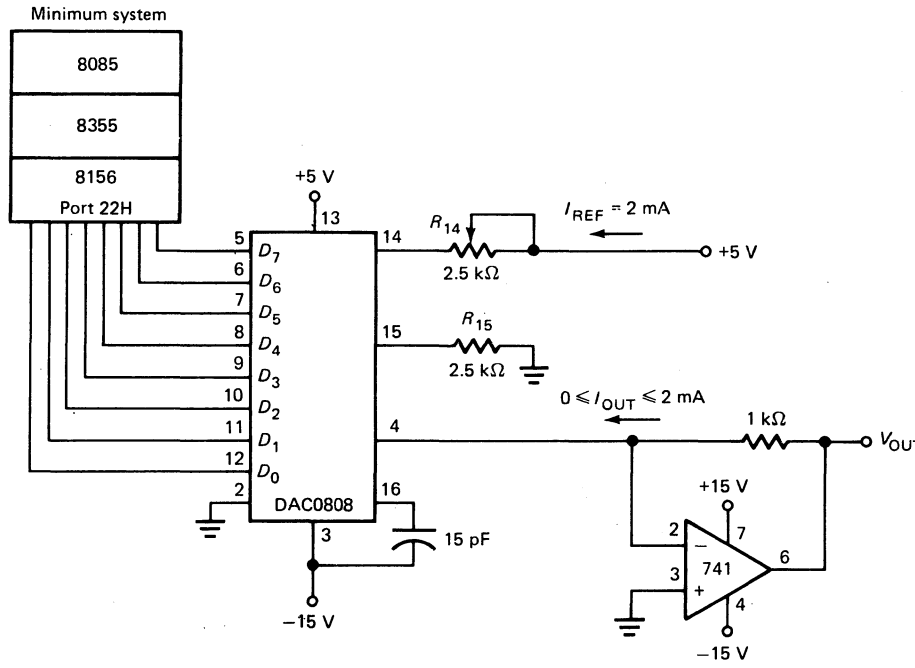


Fig. 16-13 Interfacing the DAC0808.

mA. R_{15} is the same size as R_{14}; this compensates for drift in the input stage of the converter. Notice that I_{OUT} drives the inverting input of an op amp; therefore, the final output voltage ranges from 0 to +2 V (actually 1.992 V).

EXAMPLE 16-1

What does the following program do in Fig. 16-13?

Label	Mnemonic	Comment
	MVI A,02H	;Load command word for 8156
	OUT 20H	;Make port 22H an output
	MVI A,FFH	;Initialize counter
LOOP:	INR A	;Increment counter
	OUT 22H	;Output data
	JMP LOOP	;Get next byte

SOLUTION

The first two instructions initialize the 8156 to make port 22H an output port. After the accumulator is set to FFH, the program enters a loop. In this loop the accumulator acts like a counter that counts from 00H to FFH; then the cycle repeats. After each accumulator byte is latched in port 22H, the D/A converter produces an equivalent analog current. Because of the direction of the current, the output voltage is a positive staircase with 255 steps from 0 to 1.992 V. The staircase is monotonic because the DAC0808 has a relative accuracy of better than $\pm\frac{1}{2}$ LSB.

EXAMPLE 16-2

If a clock frequency of 3 MHz drives the 8085, what is the approximate frequency of the staircase in the preceding example?

SOLUTION

We have to work out the period of the staircase, which is 256 times the duration of each output level. The duration of each output level depends on how long it takes to pass through the loop. Each T state has a duration of

$$T = \frac{1}{f} = \frac{1}{3\text{ MHz}} = 330\text{ ns}$$

Here are the calculations for loop time:

INR	$4 \times 330\text{ ns} = 1.32\ \mu\text{s}$
OUT	$10 \times 330\text{ ns} = 3.3$
JMP	$10 \times 330\text{ ns} = \underline{3.3}$
	$7.92\ \mu\text{s}$

This is how long each output level lasts. Since it takes only 150 ns to settle to within $\frac{1}{2}$ LSB of the final level, each step is well defined.

Because there are 256 levels, the period of a staircase is

$$T = 256 \times 7.92\ \mu\text{s} = 2.03\text{ ms}$$

which gives a frequency of

$$f = \frac{1}{T} = \frac{1}{2.03 \times 10^{-3}} = 493 \text{ Hz}$$

EXAMPLE 16-3

The SDK-85 is a microprocessor trainer containing an 8085 minimum system with a hexadecimal keyboard and display. The 8355 ROM contains a monitor program that allows you to enter and run user programs. Hand-assemble the program of Example 16-1 for entry into an SDK-85.

SOLUTION

Address	Hex code
2000	3E
2001	02
2002	D3
2003	20
2004	3E
2005	FF
2006	3C
2007	D3
2008	22
2009	C3
200A	06
200B	20

Notice that the user program starts at memory location 2000H. This is because user programs and data are stored in RAM locations 2000H to 20FFH.

16-5 THE COUNTER METHOD OF A/D CONVERSION

Figure 16-14 shows the simplest but least used method of A/D conversion. V_{IN} is the analog input voltage. D_7 to D_0 are the digital output. The digital output drives a D/A converter, which produces an analog output V_{OUT}. When *COUNT* is high, the counter counts upward. When *COUNT* is low, the counter stops. For convenience, an 8-bit D/A converter and 8-bit counter are used, but the idea applies to any number of bits.

Operation

The A/D conversion takes place as follows. First, the *START* pulse goes low, clearing the counter. When the *START* pulse returns high, the counter is ready to go. Initially, V_{OUT} is zero; therefore, the op amp has a high output and *COUNT* is high. The counter starts counting upward from zero. Since the output of the counter drives a D/A converter, the converter output is a positive voltage staircase. As long as V_{IN} is greater than V_{OUT}, the op amp has a positive output, *COUNT* remains high, and the staircase voltage keeps rising.

At some point along the staircase, the next step makes V_{OUT} greater than V_{IN}. This forces *COUNT* to go low, and the counter stops. Now, the digital output D_7 to D_0 is the digital equivalent of the analog input. The negative-going edge of the *COUNT* signal is used as an *end-of-conversion* signal; this tells other circuits that the A/D conversion is finished.

If the analog input V_{IN} is changed, external circuits must send another *START* pulse to *start the conversion*. This clears the count and a new cycle begins. When the digital data is ready, the end-of-conversion signal has a falling edge.

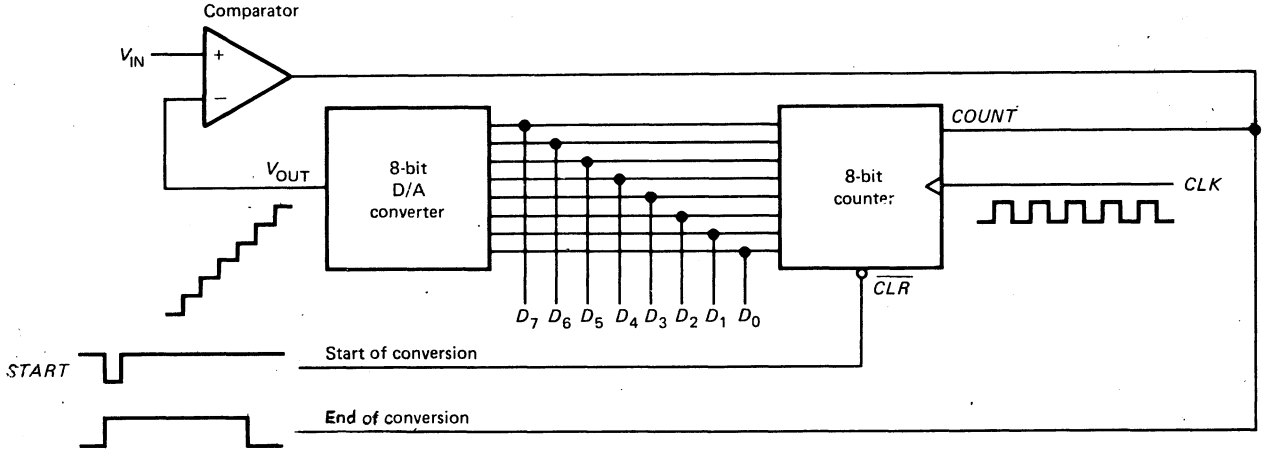


Fig. 16-14 A/D conversion with counter.

Disadvantage

The main disadvantage of the *counter method* is its slow speed. In the worst case (maximum analog input) the counter has to reach the maximum count before the staircase voltage is greater than the analog input. For an 8-bit converter, this means a conversion time of 255 clock periods. For a 12-bit converter, the conversion time is 4,095 clock periods.

16-6 SUCCESSIVE APPROXIMATION

The most widely used approach in A/D conversion is the *successive-approximation method* (see Fig. 16-15). As before, the output of a D/A converter drives the inverting input of an op-amp comparator. The difference, however, is in how the SAR register converges on the digital equivalent. (SAR stands for *successive-approximation register*.) When the conversion is finished, the digital equivalent is transferred to the output buffer register.

MSB First

When the start-of-conversion signal goes low, the SAR register is cleared and V_{OUT} drops to zero. When the start-of-conversion signal goes high, the conversion begins. Instead of counting up 1 bit at a time, the successive-approximation method starts by setting the MSB. In other words, during the first clock pulse the control circuit loads a high MSB into the SAR register, whose output then equals

1000 0000

As soon as this digital output appears, V_{OUT} jumps to $\frac{128}{255}$ times full-scale. If this is more than V_{IN}, the negative output of the comparator signals the control circuit to reset the MSB. On the other hand, if V_{OUT} is less than V_{IN}, the positive output of the comparator indicates that the MSB is to remain set. In some designs, setting and testing the MSB take place during the first clock pulse following the start of conversion. In other designs, several clock pulses may be needed to set the MSB, test it, and reset it if necessary.

Remaining Bits

Let us assume that the MSB was not reset. The SAR register contents are now 1000 0000. The next clock pulse will set D_6, giving a digital output of

1100 0000

V_{OUT} now steps to $\frac{192}{255}$ times full-scale. If V_{OUT} is greater than V_{IN}, the negative op-amp output causes D_6 to reset. If V_{OUT} is less than V_{IN}, D_6 remains set.

During the remaining clock pulses, successive bits are set and tested. Whenever a bit causes V_{OUT} to exceed V_{IN}, the bit is reset. In this way, all bits are set, tested, and reset if necessary. With the fastest circuits, the conversion is finished after eight clock pulses, and the D/A output is the analog equivalent of the register contents. Slower designs take longer because more clock pulses are needed to set, test, and possibly reset each bit.

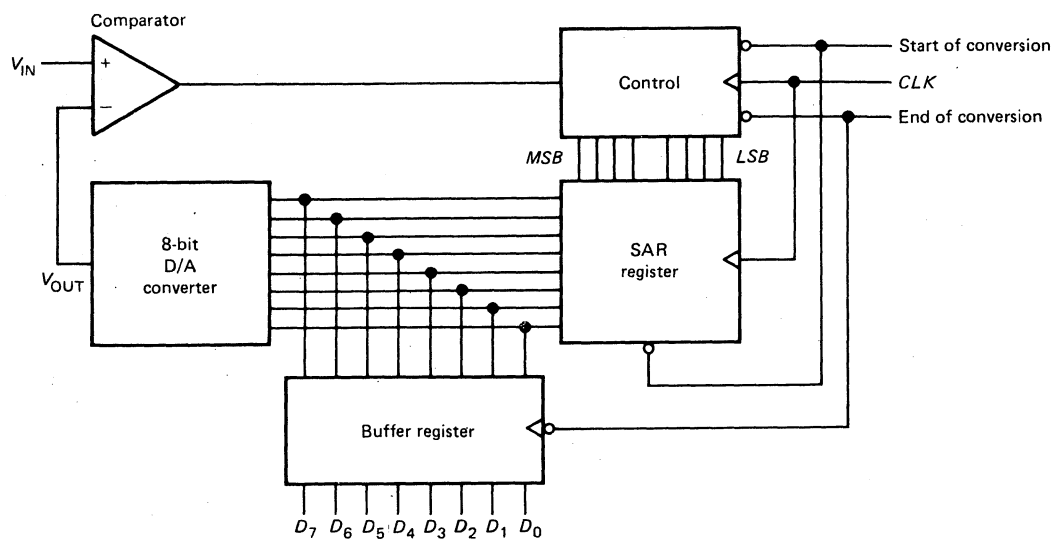


Fig. 16-15 A/D conversion by successive approximation.

Output Buffer

When the conversion is finished, the control circuit sends out a low end-of-conversion signal. The falling edge of this signal loads the digital equivalent into the buffer register. In this way, the digital output will remain even though we start a new conversion cycle.

Advantage

The main advantage of the successive-approximation method is speed. At best, it takes only n clock pulses to produce n-bit resolution of the analog signal. This is a big improvement over the counter method. Even with slower designs, the successive-approximation method is still considerably better than the counter method.

16-7 THE ADC0801

A/D converters are commercially available as integrated circuits with 8- to 16-bit resolution. This section introduces the ADC0801, an 8-bit A/D converter that is easily interfaced to an 8080 or 8085. The device uses successive approximation, converting an analog input (0 to 5 V) to an 8-bit digital equivalent. The ADC0801 has an on-chip clock generator, needs a supply of only +5 V, and has an optimum conversion time of approximately 100 μs.

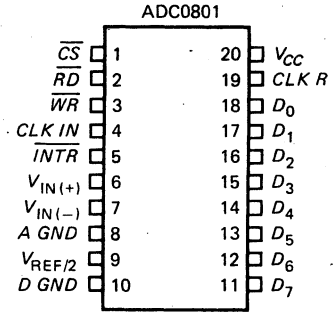


Fig. 16-16 Pinout of ADC0801.

Pinout Diagram

Figure 16-16 shows the pinout. Pins 11 to 18 (digital output) are a three-state output; this allows a direct connection to the address-data bus if desired. If $\overline{CS}$ (pin 1) or $\overline{RD}$ (pin 2) is high, pins 11 to 18 float; when $\overline{CS}$ and $\overline{RD}$ are both low, the digital output appears on the output lines.

The start-of-conversion signal has been labeled $\overline{WR}$ (pin 3). To start a conversion, $\overline{CS}$ must be low. When $\overline{WR}$ goes low, the converter is reset; when $\overline{WR}$ returns to the high state, the conversion begins.

The converter clock frequency must be in the range of 100 to 800 kHz. The CLK IN (pin 4) may be derived from the CPU clock. If the system clock is running at a frequency greater than 800 kHz, we can divide it down with one or more flip-flops. Alternatively, we can use an on-chip clock generator by connecting an external RC circuit between CLK IN (pin 4) and CLK R (pin 19).

Pin 5 is $\overline{INTR}$, the end-of-conversion signal. $\overline{INTR}$ goes high at the start of a conversion; it is *asserted* (made active low) when the conversion is finished. The falling edge of $\overline{INTR}$ can be used to interrupt a microprocessor, which then branches to a service subroutine processing the converter output.

Pins 6 and 7 are a differential input for the analog signal; this type of input allows you to ground pin 7 for single-ended positive input, to ground pin 6 for single-ended negative input, or to drive both pins for differential input.

The device has two grounds, A GND (pin 8) and D GND (pin 10). Both must be grounded. Pin 20 is the supply voltage, +5 V in a microprocessor-based system.

Pin 9

In the ADC0801, V_{REF} is the maximum analog input voltage; this is the voltage that produces a maximum digital output of FFH. If pin 9 is unconnected, V_{REF} equals the supply voltage V_{CC}. This means that a supply of +5 V allows an analog input range of 0 to +5 V for single-ended positive input (input on pin 6 with pin 7 grounded).

In some applications we may prefer a different analog range. This is where pin 9 comes in. The voltage you connect to pin 9 overrides the supply voltage and controls the maximum analog input voltage. If you want a maximum analog input of +4 V, for instance, you must apply +2 V to pin 9. If you want a maximum analog input of +3 V, then apply +1.5 V to pin 9.

In our discussions, we leave pin 9 open and let V_{CC} set the maximum analog input. In this case, the analog input range is 0 to +5 V because a supply voltage of +5 V is used in the remainder of this chapter.

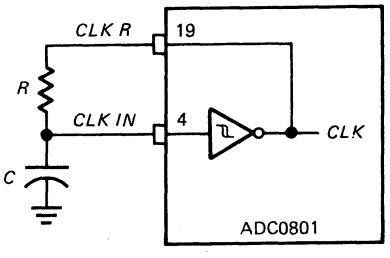


Fig. 16-17 RC circuit for on-chip clock.

Using the On-Chip Clock

Self-clocking the ADC0801 requires an external RC circuit connected to the CLK IN and CLK R pins, as shown in Fig. 16-17. The Schmitt-trigger inverter produces an internal clock frequency of

$$f = \frac{1}{1.1RC} \qquad (16\text{-}10)$$

A typical resistance range is from 10 to 50 kΩ. As an example, 10kΩ and 120 pF produce a converter clock frequency of

$$f = \frac{1}{(1.1 \times 10^{4})(120 \times 10^{-12})} = 758 \text{ kHz}$$

Continuous Operation

The ADC0801 can be connected to produce continuous A/D conversion. To do this, we have to ground $\overline{CS}$ and connect $\overline{WR}$ to $\overline{INTR}$, as shown in Fig. 16-18. Also, to enable the output register, $\overline{RD}$ is grounded; then a continuous digital output appears. With pin 7 grounded, the circuit is set up for single-ended positive input to pin 6.

The action is continuous because the $\overline{INTR}$ signal (equivalent to end of conversion) drives the $\overline{WR}$ input (equivalent to start of conversion). At the end of each conversion, $\overline{INTR}$ goes low. This resets the converter and drives $\overline{INTR}$ high. As soon as $\overline{INTR}$ goes high, a new conversion begins. As each conversion is finished, the digital equivalent is loaded into the output buffer register. In this way, the digital output is being continuously updated to reflect changes in the analog input.

To get the data into the microprocessor, an IN 21H is executed. This transfers the data into the accumulator. Then additional instructions can process this data as needed in the application.

A final point. When the power is first turned on, the circuit of Fig. 16-18 may not start under certain conditions. To ensure proper starting, it is necessary to have additional circuitry that applies a negative-going pulse to the $\overline{WR}$ input during power-up. Once started, the conversion is continuous.

Software Handshaking

Programmed handshaking adds an extra degree of software control over the operation of an A/D converter. As an example, we can connect $\overline{INTR}$ to bit 7 of port 00H and $\overline{WR}$ to bit 0, as shown in Fig. 16-19. In this way, the software has to start a conversion by sending a start-of-conversion pulse to the $\overline{WR}$ input. When the conversion is finished, the low end-of-conversion signal ($\overline{INTR}$) tells the CPU that the data is ready.

Here is a program segment illustrating the handshaking:

Label	Mnemonic	Comment
		;Group 1
	MVI A,01H	;Reset bit 7, set bit 0
	OUT 02H	;Initialize port 00H
	MVI A,00H	;Reset bit 0
	OUT 20H	;Initialize port 21H
		;Group 2
NEXT:	MVI A,00H	;Reset bit 0
	OUT 00H	;Send low *SOC* signal
	MVI A,01H	;Set bit 0
	OUT 00H	;Send high *SOC* signal
		;Group 3
WAIT:	IN 00H	;Begin wait loop
	ANI 80H	;Mask all except bit 7
	JNZ WAIT	;Loop if *EOC* signal high

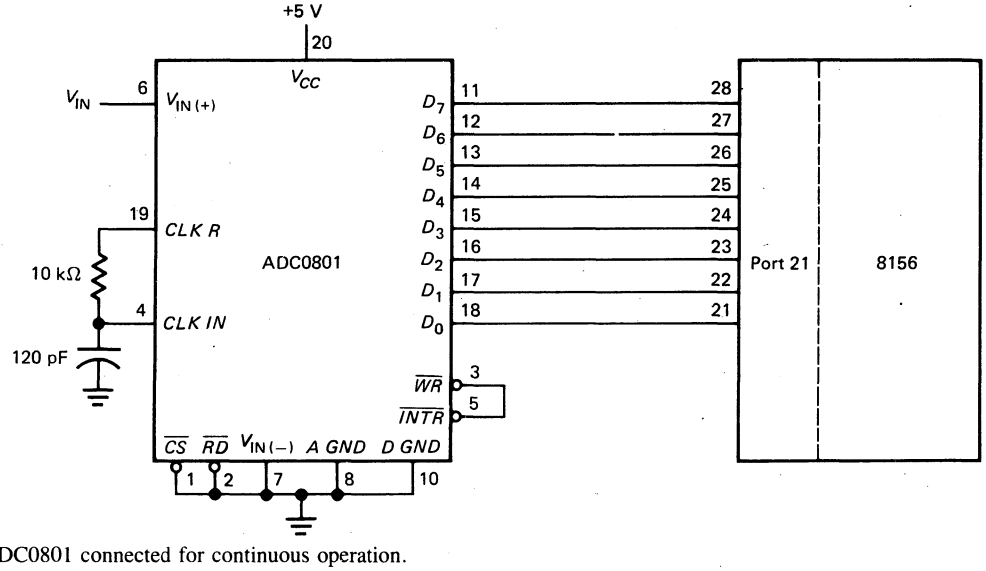


Fig. 16-18 ADC0801 connected for continuous operation.

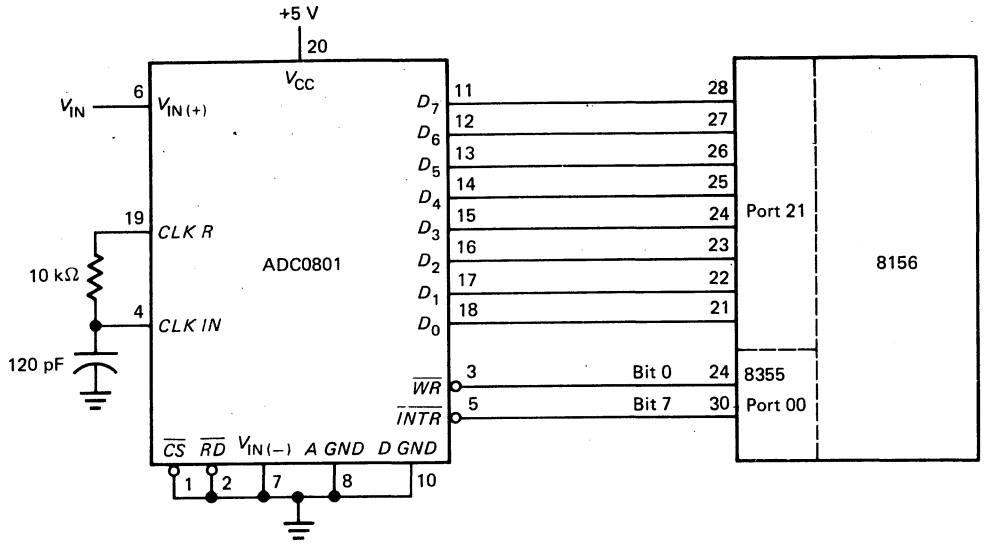


Fig. 16-19 Software handshaking with the ADC0801.

```
                ;Group 4
IN 21H          ;Input converter data
      .         ;Additional instructions
      .         ;to process data as
      .         ;required
JMP NEXT        ;Return for next sample
```

The instructions have been grouped into their functions. For instance, the first group of instructions

```
MVI A,01H
OUT 02H
MVI A,00H
OUT 20H
```

program ports 00H and 21H. The first two instructions make bit 0 an output to the start-of-conversion line and bit 7 an input for the end-of-conversion signal (Fig. 16-19). The next two instructions make port 21H an input port.

The second group of instructions sends a negative-going start pulse to the A/D converter. The first two instructions produce a low $\overline{WR}$, and the next two produce a high $\overline{WR}$. At this point, the A/D conversion begins.

The program now enters a WAIT loop. As long as the end-of-conversion signal is high, bit 7 is high. In this case, the program will loop back to the WAIT entry point. After the A/D converter has converged on the digital equivalent, the end-of-conversion $\overline{INTR}$ signal goes low, forcing bit 7 low. This time, the IN 00H and ANI 80H set the zero flag, and the program falls through the JNZ instruction.

The final group of instructions begins by loading the converter data into the accumulator. Then additional instructions can process the data as needed. Finally, the JMP NEXT takes the program back to the NEXT entry point where a new conversion cycle begins.

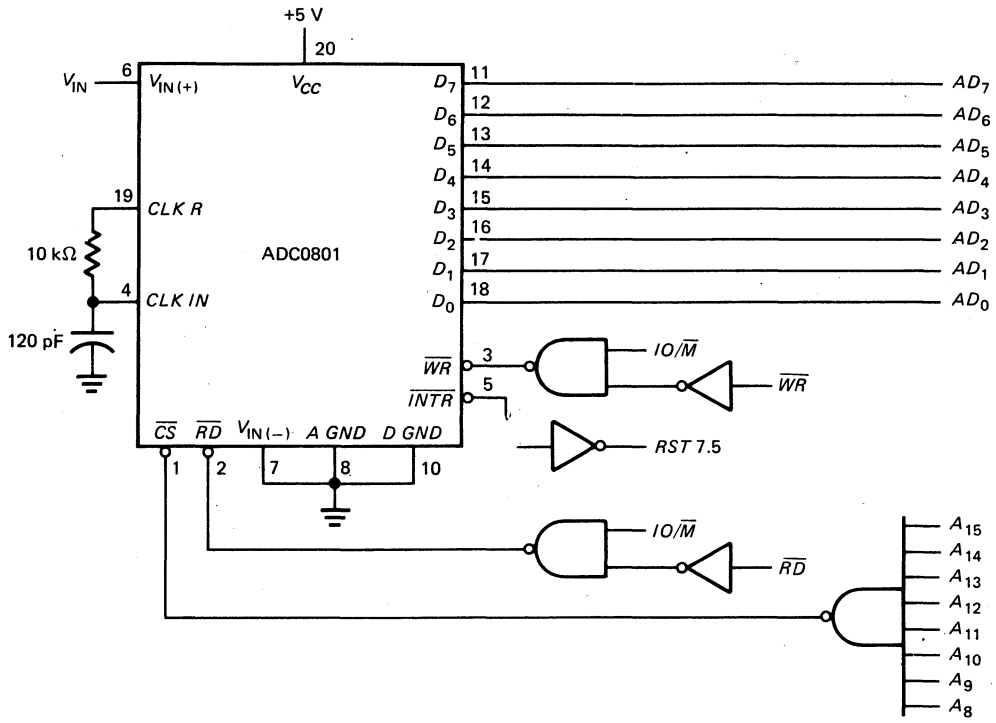
Hardware Handshaking

In the preceding program, we used software to send and receive the handshaking bits. The ADC0801 can be interfaced with an 8080/8085 for hardware handshaking. In other words, instead of software control of the $\overline{WR}$ and $\overline{INTR}$ signals, we can let the 8080/8085 control bus send and receive the handshaking bits.

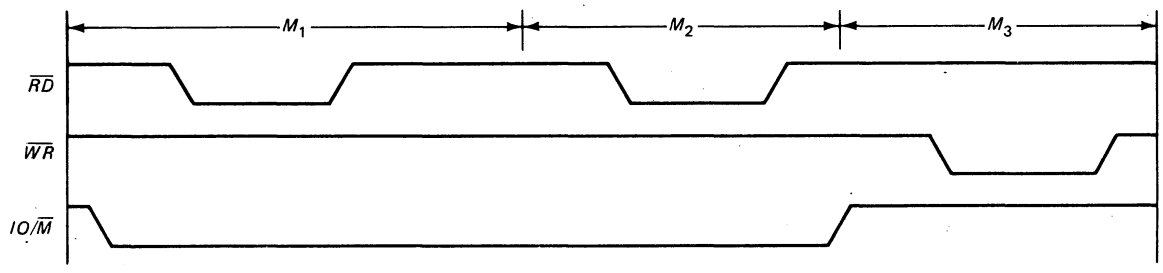
Figure 16-20*a* illustrates the idea. In this circuit, the ADC0801 is being gate-addressed as an I/O device with a port address of FFH. Why? Because the 8-input NAND gate produces a low $\overline{CS}$ output only when FFH is on the upper half of the address bus. Furthermore, notice how the $IO/\overline{M}$ signal is being gated with $\overline{WR}$ and $\overline{RD}$. Since $IO/\overline{M}$ is high only when IN and OUT instructions are executed, the A/D converter is being controlled as an I/O device rather than a memory location.

The end-of-conversion signal $\overline{INTR}$ goes low when the data is ready. Because of the inverter, the RST 7.5 interrupt receives a rising edge. This interrupt allows the 8085 to input and process the converter data.

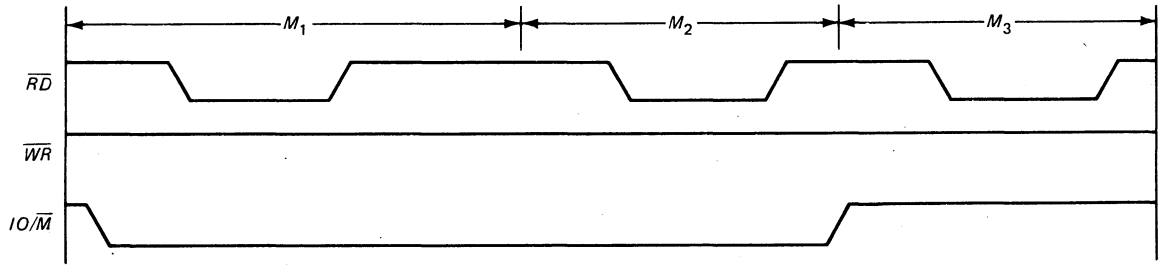
As previously discussed, the vector location for an RST 7.5 interrupt is 003CH. Suppose the 3 bytes at 003CH–003EH contain machine code for JMP DATA. Then an RST 7.5 interrupt sends the program to the address labeled DATA. With this in mind we give a program to illustrate how the microprocessor controls the converter:



(a)



(b)



(c)

Fig. 16-20 Hardware handshaking with the ADC0801.

```
Label     Mnemonic      Comment
          MVI A,1BH     ;Mask 6.5 and 5.5
          SIM           ;Set interrupt mask
NEXT:     OUT FFH       ;Start conversion
          EI            ;Enable the interrupts
LOOP:     NOP           ;Waste time
          JMP LOOP      ;Loop
            .
            .
            .
DATA:     IN FFH        ;Input converter data
            .           ;Additional instructions
            .           ;to process converter data
            .           ;for your application
          JMP NEXT      ;Go back for next sample
```

The first two instructions set the interrupt mask. The RST 7.5 interrupt is unmasked; the RST 6.5 and 5.5 interrupts are masked.

Figure 16-20*b* shows the 8085 timing for the OUT instruction. The first two machine cycles (M_1 and M_2) are memory-read operations (op code followed by port number). During the third machine cycle, $IO/\overline{M}$ goes high. A little later, $\overline{WR}$ goes low and returns high. This means that pin 3 of the ADC0808 (Fig. 16-20*a*) pulses temporarily low. In other words, the execution of OUT FFH starts the conversion process.

Returning to the program, notice the EI instruction; this enables the interrupt system. Next, the program enters what appears to be an endless loop. As long as the conversion process is going on, the $\overline{INTR}$ output is high and the program loops, doing nothing but waiting. When the conversion is finished, however, $\overline{INTR}$ goes low and RST 7.5 goes high.

After the RST 7.5 interrupt is acknowledged, the 8085 branches to vector location 003CH. Here it finds a JMP DATA which sends the 8085 to the DATA label of the program. When the 8085 executes the IN FFH, it will load the converter data into the accumulator. Why? Because the IN instruction has the control timing shown in Fig. 16-20*c*. Notice how $IO/\overline{M}$ is high and $\overline{RD}$ is low during the third machine cycle. At this time, the converter data is read onto the address-data bus and loaded into the accumulator.

Additional instructions (not shown) can process the data as needed in the particular application. The final instruction JMP NEXT then returns the program to the NEXT entry point.

Here is a fine point. Notice that we do not use a RET to get back to the NEXT instruction. Since the RST 7.5 interrupted an endless loop, a RET instruction would bring the program back to the loop with no way of escaping. Only by using a JMP NEXT can we then start a new conversion.

16-8 SUCCESSIVE APPROXIMATION WITH SOFTWARE

If speed is not important, you can use software to implement the successive-approximation method of A/D conversion. Figure 16-21 illustrates the idea. The output of a comparator is connected to bit 0 of port 21H, programmed as an input. Port 22H, programmed as an output, drives a D/A converter. The following program segment will produce an output at port 22H that converges on the digital equivalent of the analog input.

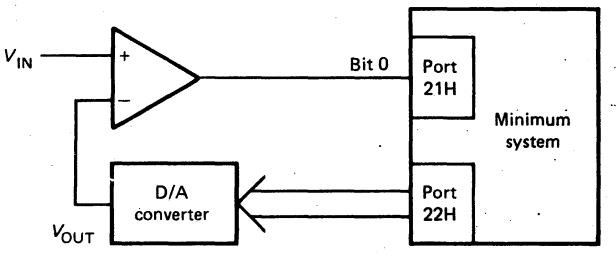


Fig. 16-21 A/D conversion with software.

```
Label     Mnemonic      Comment
                        ;Group 1
          STC           ;Set carry
          MVI B,09H     ;Preset B counter
          MVI C,00H     ;Clear ring counter
          MVI D,00H     ;Clear SAR

                        ;Group 2
LOOP:     DCR B         ;Decrement B counter
          JZ END        ;Jump to end after 8 bits

                        ;Group 3
          MOV A,C       ;Load next bit
          RAR           ;Rotate right
          MOV C,A       ;Save shifted bit

                        ;Group 4
          MOV A,D       ;Load SAR
          ADD C         ;Add next bit
          MOV D,A       ;Set latest bit in SAR
          OUT 22H       ;Send to output

                        ;Group 5
          IN 21H        ;Input op amp
          ANI 01H       ;Mask all but bit 0
          JNZ LOOP      ;Jump if op amp positive
          MOV A,D       ;Load SAR
          SUB C         ;Reset bit
          MOV D,A       ;Store SAR
          JMP LOOP      ;Go back for next bit

END:      MOV A,D       ;Load digital output into A
```

First Group

The instructions have been grouped by their function. The first group (first four instructions) initializes the necessary registers. The carry is set, and the B counter is preset with decimal 9. Also, the C and D registers are cleared.

Second Group

The next group of instructions (DCR B and JZ END) act like a counter to check for 8-bit resolution. The first 8 times through the loop, the program falls through the JZ END. On the ninth time the B counter drops to zero and the program jumps to END because the conversion is finished.

Third Group

The third group of instructions (MOV A,C through MOV C,A) rotate a single bit to the right. For instance, the first time through this group, the carry bit is rotated into the MSB, which is saved in the C register:

$$\mathbf{C} = 1000\ 0000$$

The second time through the loop, the bit is rotated to the right to get

$$\mathbf{C} = 0100\ 0000$$

Successive loops produce

$$\mathbf{C} = 0010\ 0000$$
$$\mathbf{C} = 0001\ 0000$$

and so on. Notice that the C register acts like a ring counter. In other words, register C contains the latest bit being set in the successive-approximation method.

Fourth Group

The fourth group of instructions (MOV A,D through OUT 22H) sends the latest approximation to the output port. The first time through the loop the ADD C produces

$$\mathbf{A} = 1000\ 0000$$

The MOV D,A saves this result. The OUT 22H sends this to port 22H. Then the D/A converter produces an analog output of $\frac{128}{255}$ times full-scale.

Fifth Group

The fifth group of instructions (IN 21H through the JMP LOOP) tests the effect of each bit and resets it when necessary. For instance, the first time through the loop the D/A converter produces a V_{OUT} equal to $\frac{128}{255}$ times full-scale. If this is less than V_{IN}, bit 0 of port 21H is high. The IN 21H and ANI 01H test bit 0. Since this bit is high, the zero flag is reset. The JNZ LOOP returns the program to the LOOP point.

In the foregoing discussion, if V_{OUT} is greater than V_{IN}, bit 0 is low. The program then falls through the JNZ LOOP to the MOV A,D, followed by the SUB C and MOV D,A. These instructions have the effect of resetting the MSB.

Conclusion

The program will pass through the loop 8 times because of the second group of instructions. The next bit is added to the successive approximation during the fourth group of instructions. The fifth group of instructions then tests the effect of the latest bit. If V_{OUT} becomes greater than V_{IN}, the latest bit is reset. After 8 passes through the loop, the software conversion is finished and the program falls through to the final instruction.

16-9 VOLTAGE-CONTROLLED OSCILLATOR

Figure 16-22 illustrates another method of A/D conversion used in low-cost applications. The analog input voltage drives a *voltage-controlled oscillator* (VCO). As an example, the VCO could be a 555 timer with the analog voltage driving the control input. The output of the VCO is a rectangular wave whose period is directly proportional to V_{IN}, the analog voltage. The rectangular wave is connected to bit 7 of port 21H. Here is a program that performs A/D conversion.

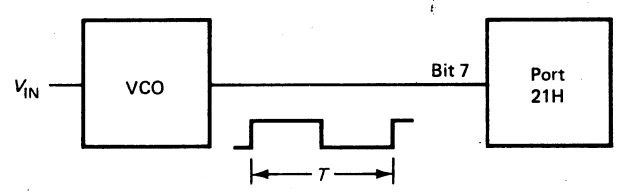


Fig. 16-22 A/D conversion using software to measure period.

Label	Mnemonic	Comment
		;Group 1
	MVI A,00H	;Reset bit 0
	OUT 20H	;Make port 21H an input
	MVI C,00H	;Clear the counter
		;Group 2
LOOP1:	IN 21H	;Input signal
	ANI 80H	;Mask all but bit 7
	JZ LOOP1	;Loop if bit 7 low

```
                        ;Group 3
LOOP2:    INR C         ;Count up
          IN 21H        ;Input signal
          ANI 80H       ;Mask all but bit 7
          JNZ LOOP2     ;Loop if bit 7 high

                        ;Group 4
LOOP3:    INR C         ;Count up
          IN 21H        ;Input signal
          ANI 80H       ;Mask all but bit 7
          JZ LOOP3      ;Loop if bit 7 low

          MOV A,C       ;Load data intc A
```

First Group

The first group of instructions programs port 21H as an input port. Also, register C is cleared. This register acts like a counter during the program.

Second Group

The next group of instructions is LOOP1. The program stays in this loop as long as the signal is low. After the signal goes high, the program falls through the JZ. The purpose of LOOP1 is to wait until a positive edge arrives. Then the counting can begin.

Third Group

The third group of instructions, LOOP2, starts by incrementing the counter. Then bit 7 is tested to determine if the signal is high or low. As long as the signal is high, the program loops and the counter keeps counting. When the signal goes low, the program falls through the JNZ. The purpose of LOOP2 is to count the length of the positive half cycle.

Fourth Group

The fourth group of instructions is LOOP3. The purpose of this loop is to count the length of the negative half cycle of the input signal. As long as the signal is low, the looping continues. When the signal goes high again (end of cycle), the program falls through LOOP3 to the final instruction. The MOV A,C then loads the digital equivalent of analog voltage into the accumulator. The larger the analog input voltage, the longer the period of the input signal and the greater the final value in the accumulator.

16-10 SAMPLE-AND-HOLD CIRCUITS

In our discussion of A/D converters, we treated the analog input voltage as a constant. What happens if it changes while a conversion is taking place? The answer depends on the A/D conversion method, the conversion time, and other factors. In any case, if the input voltage changes by a significant amount while the A/D conversion is going on, the digitized output is ambiguous.

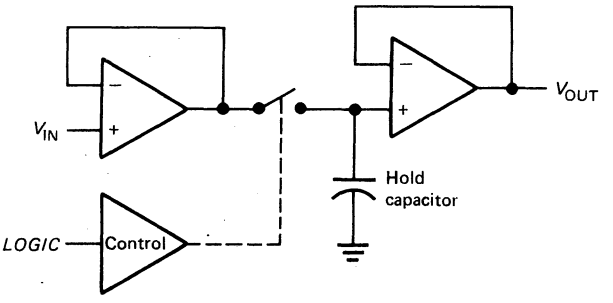


Fig. 16-23 Sample-and-hold amplifier.

The Basic Idea

The way to get around a changing input voltage is to use a *sample-and-hold amplifier* (Fig. 16-23). As discussed earlier, the voltage between the inverting and noninverting inputs of an op amp is in microvolts, so we can approximate this voltage as zero. This implies the voltage from the inverting input (− input) to ground is approximately V_{IN}. Because of the direct connection, the output of the first op amp is approximately V_{IN}. In other words, the first op amp acts like a unity-gain amplifier.

The switch is *logic-controlled,* meaning that a high input closes the switch and a low input opens it. When the switch is closed, the capacitor rapidly charges to V_{IN}. Since the second op amp is also a unity-gain amplifier, V_{OUT} equals V_{IN} to a close approximation. When the switch opens, the capacitor retains its charge. Ideally, the output holds at a value of V_{IN}.

If the input voltage changes rapidly while the switch is closed, the capacitor can follow this voltage because the charging time constant is very short. If the switch is suddenly opened, the capacitor voltage represents a sample of the input voltage at the instant the switch was opened. The capacitor then holds this sample until the switch is again closed and a new sample taken.

Acquisition Time

One way to specify the quality of a sample-and-hold amplifier is its *acquisition time*. This is the time needed to get an accurate sample (typically to within 0.1 percent) after the switch is closed. Ideally, acquisition time is zero, but in a real sample-and-hold amplifier the charging time constant of the hold capacitor plus other factors produce a nonzero acquisition time.

Aperture Time

Another measure of a sample-and-hold amplifier is its *aperture time*. This is defined as the time required for the switch to open. Since the switch is a transistor switch, there is a short time before it appears open and no longer affects the hold capacitor.

Droop Rate

The *droop rate* is the rate at which the output voltage decreases in the hold condition. There are leakage paths for the capacitor charge, and this is why the output voltage will droop slowly when the switch is open. The larger the capacitor, the smaller the droop in a given time.

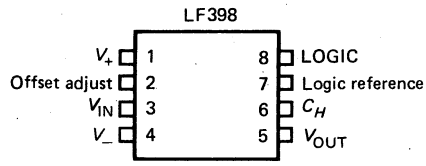


Fig. 16-24 Pinout for LF398.

The LF398

The LF398 is a commercially available sample-and-hold amplifier. For a hold capacitor of 0.001 μF, it has an acquisition time of 4 μs, an aperture time of 150 ns, and a droop rate of 30 mV/s.

Figure 16-24 shows the pinout. The device uses a *split supply* (equal positive and negative voltages). The positive supply goes to pin 1, and the negative to pin 4. These supply voltages may be from +5 to +18 V.

The *offset adjust* (pin 2) is used to zero the output for a zero input condition. Since the error is quite small, we can leave pin 2 unconnected in typical applications. Pin 3 is for the input voltage to be sampled. Pin 5 is the output voltage. Pin 6 is for the *hold capacitor*. Pin 7 is usually grounded. Pin 8 is the logic signal; when this signal is high, the device is sampling; when the signal is low, the device is holding.

The hold capacitor can be in the range of 100 pF to 1 μF, the exact size being determined by a tradeoff between acquisition time and droop rate. The smaller the hold capacitor, the shorter the acquisition time but the larger the droop rate. For our later discussions, we will use a hold capacitance of 0.001 μF. According to the data sheets for the LF398, this results in an acquisition time of 4 μs and a droop rate of 30 mV/s.

Software-Controlled Sample-and-Hold

Figure 16-25 shows how bit 0 of port 23H can control a sample-and-hold amplifier. The idea is to send a high bit to sample and a low bit to hold. Here is the program:

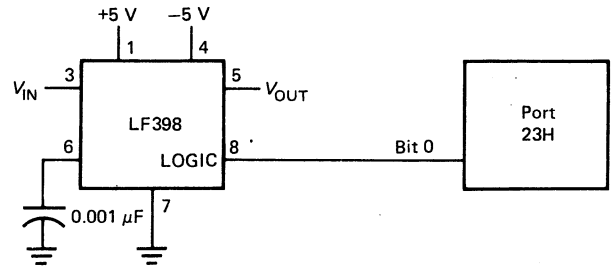


Fig. 16-25 Software control of sample-and-hold-amplifier.

Label	Mnemonic	Comment
	MVI A,0CH	;Set bits 3 and 2
	OUT 20H	;Make port 23H an output
	MVI A,01H	;Set bit 0
	OUT 23H	;Sample
	CALL ACQTIM	;Wait for 4 μs
	MVI A,00H	;Clear bit 0
	OUT 23H	;Hold

The first two instructions make port 23H an output. The next two send a high bit 0 to port 23H, which puts the sample-and-hold amplifier into the sample state. Next comes CALL ACQTIM. This subroutine (not shown) produces a delay of 4 μs (acquisition time) to ensure enough time for an accurate sample. The last two instructions clear bit 0 and the sample-and-hold amplifier goes into the hold state.

EXAMPLE 16-4

Figure 16-26 shows how we can combine many ideas discussed so far. The analog input voltage V_{IN} drives the LF398, which is controlled by bit 0 of port 23H. The output of the sample-and-hold amplifier goes to the ADC0801. The A/D converter is controlled as an I/O device with a port address of FFH. The converter output connects to the address-data bus of a fully decoded minimum system. After the 8085 has processed the data, it can send the results to port 22H. The DAC0808 then converts this processed data to an analog output V_{OUT}.

Here is the program that runs the system:

Label	Mnemonic	Comment
		;Group 1
	MVI A,0EH	;Set bits 3, 2, 1
	OUT 20H	;Port 22H and 23H are outputs
	MVI A,1BH	;Mask 6.5 and 5.5
	SIM	;Set interrupt mask

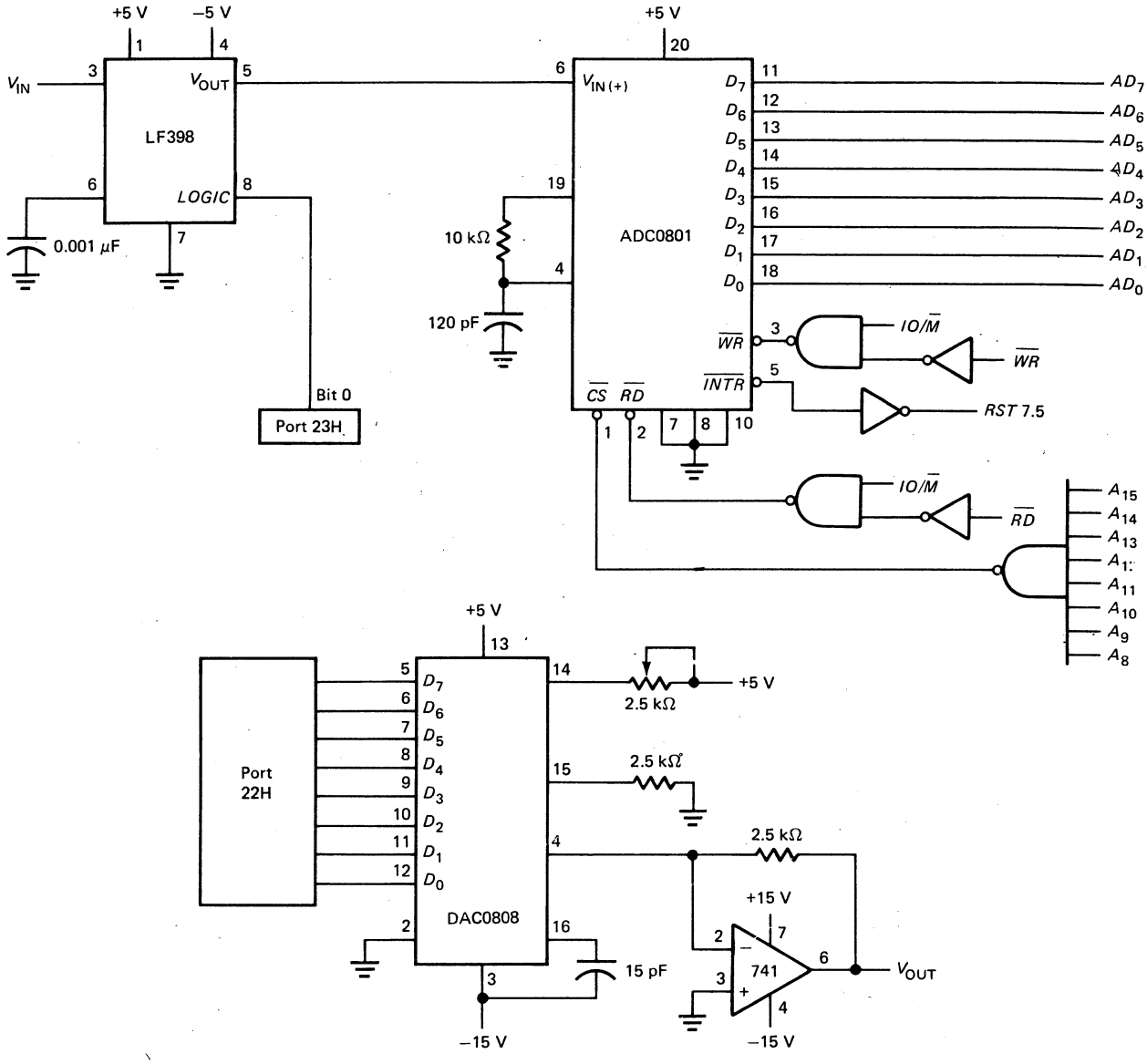


Fig. 16-26 Data acquisition and processing.

```
                    ;Group 2
NEXT:   MVI A,01H   ;Set bit 0
        OUT 23H     ;Sample analog input;
        CALL ACQTIM Wait for 4 μs
        MVI A,00H   ;Clear bit 0
        OUT 23H     ;Hold

                    ;Group 3
        OUT FFH     ;Start conversion
        EI          ;Enable the interrupts
LOOP:   NOP         ;Waste time
        JMP LOOP    ;Loop
```

```
                    ;Group 4
DATA:   IN FFH      ;Input converter data
          .         ;Additional instructions
          .         ;to process converter data
          .         ;for your application
        OUT 22H     ;Output processed data
        JMP NEXT    ;Get next sample
```

Group 1 initializes the ports and interrupt masks. Group 2 controls the sample-and-hold amplifier. After the OUT 23H places the sample-and-hold amplifier on hold, group 3 starts the A/D conversion and loops until the data is ready. As

discussed in Sec. 16-7, the RST 7.5 vectors to a JMP DATA instruction, which takes the program to the fourth group of instructions. Group 4 inputs the converter data, processes it as needed in the application, and sends the result to port 22H. After 150 ns, the DAC0808 has converted the digital data into its analog equivalent.

GLOSSARY

acquisition time The time after the switch is closed needed to get an accurate sample (typically to within 0.1 percent) in a sample-and-hold amplifier.

aperture time The time required to open the switch in a sample-and-hold amplifier.

droop rate The rate at which the output of a sample-and-hold amplifier decreases when the switch is open.

monotonicity The ability of a D/A converter to produce a rising output for each successive digital input. A converter is monotonic if its relative error is less than $\pm\frac{1}{2}$ LSB at each output level.

operational amplifier A direct-coupled amplifier with high input impedance, low output impedance, and high voltage gain. Commonly known as an op amp.

relative accuracy An indication, specified in terms of LSB increments, of how close each output is to its ideal value as a fraction of full scale.

resolution The number of bits in a D/A or A/D converter.

sample-and-hold amplifier An amplifier with a logic-controlled switch and a capacitor. When the switch is closed (sampling), the capacitor charges to the input voltage. When the switch is open (holding), the capacitor maintains the output at approximately the sample voltage.

settling time The time it takes a D/A converter to stabilize to within $\frac{1}{2}$ LSB of its final value.

successive approximation One of the methods used in A/D converters. It involves testing the effect of each bit starting with the MSB. If a bit produces an output less than the input, it is used; if not, the bit is reset.

virtual ground A phenomenon in an op amp when its noninverting input is grounded. Because the input impedance and voltage gain approach infinity, the inverting input appears grounded to voltage and open to current.

SELF-TESTING REVIEW

Read each of the following and provide the missing words. Answers appear at the beginning of the next question.

1. Two key features of an op amp are its _____ voltage gain and its _____ input impedance. When the noninverting input is grounded, the inverting input looks like a _____ ground.
2. (*high, high, virtual*) A virtual ground is a ground for _____ but not for _____. With respect to current, a virtual ground appears _____.
3. (*voltage, current, open*) In an op-amp summing circuit, the _____ current is the sum of all the input currents. By using binary-weighted _____ in an op-amp summing circuit we can build a _____ converter.
4. (*output, resistors, D/A*) One way to measure the quality of a D/A converter is its _____, the ratio of the LSB increment to the maximum output. The number of output steps in a D/A converter is 2^{n-1}, where n is the number of _____. Another way to specify resolution is to state the number of _____.
5. (*resolution, bits, bits*) Relative accuracy refers to how close each output level of a D/A converter is to its ideal fraction of _____ output. If the relative accuracy of each output level is within $\pm\frac{1}{2}$ LSB, the D/A converter is _____. Monotonicity ensures that the output will _____ for successive digital inputs.
6. (*full-scale, monotonic, increase*) The limit on how fast you can change the digital inputs to a D/A converter is the _____ time, defined as the time it takes for the converter output to stabilize within _____ LSB of its final value.
7. (*settling, $\frac{1}{2}$*) One way to get around the problems of binary-weighted resistors is to use a _____ circuit. With this circuit, only two resistor values are needed. An R-$2R$ ladder divides the input _____ into binary components.
8. (*ladder, current*) One approach to A/D conversion is the counter method, where a _____ drives a D/A converter whose output goes to an op amp. The other input to the op amp is the _____ voltage. The output of the op amp stops the counter.
9. (*counter, analog*) The most widely used method for A/D conversion relies on _____ approximation. Each bit, starting with the _____, is tested for its effect on the op-amp output; if the bit contributes too much, it is _____. An end-of-conversion signal indicates when the conversion is finished.

10. (*successive, MSB, reset*) Software handshaking with an A/D converter means that an output port sends a start-of-conversion bit to begin the conversion; when the CPU detects an _____ bit at an input port, it can process the digital equivalent of the _____ input.

11. (*end-of-conversion, analog*) Hardware handshaking with an A/D converter means addressing the converter like a port, using a gated $\overline{WR}$ signal to start the conversion. The _____ bit goes to one of the interrupt inputs of the microprocessor.

12. (*end-of-conversion*) If the analog input to an A/D converter is changing faster than the conversion time, a sample-and-hold amplifier can be used to sample the analog input. When the switch is opened, the output of the sample-and-hold amplifier delivers an almost constant input volage to the A/D converter.

PROBLEMS

16-1. In Fig. 16-27*a*, V_{IN} is 15 V, R_{IN} is 10 kΩ, and R_{OUT} is 2 kΩ. What does the input current equal? The output current? The output voltage?

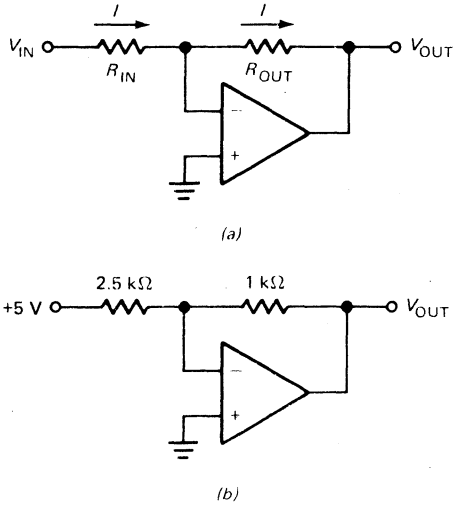


Fig. 16-27

16-2. In Fig. 16-28, V_{REF} = 1 V, R = 2 kΩ, and R_{OUT} = 1 kΩ. What does the output current equal when all the switches are closed? The output voltage?

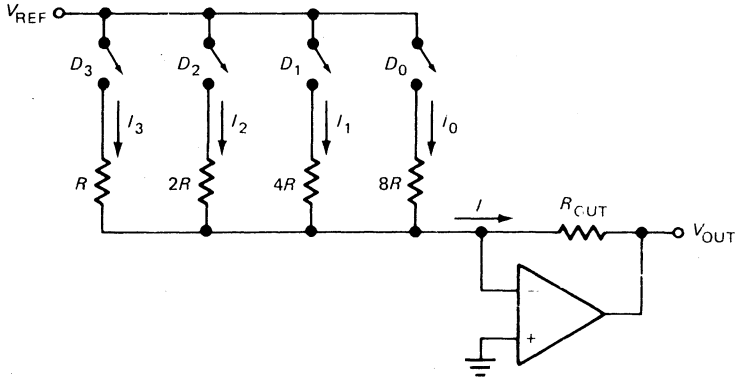


Fig. 16-28

16-3. How many steps are there in the output of a 10-bit D/A converter? What is the percent resolution?

16-4. All switches are to the left in Fig. 16-29. What does I_{OUT} equal? The D_2 and D_0 switches are moved to the right. What is the new value of I_{OUT}?

16-5. If I_{REF} is changed to 5 mA in Fig. 16-29, what is the maximum value of I_{OUT}?

16-6. In the program of Example 16-1, if

```
MVI A,FFH
```

is changed to

```
MVI A,00H
```

and

```
INR A
```

is changed to

```
DCR A
```

Describe the output waveform in Fig. 16-30.

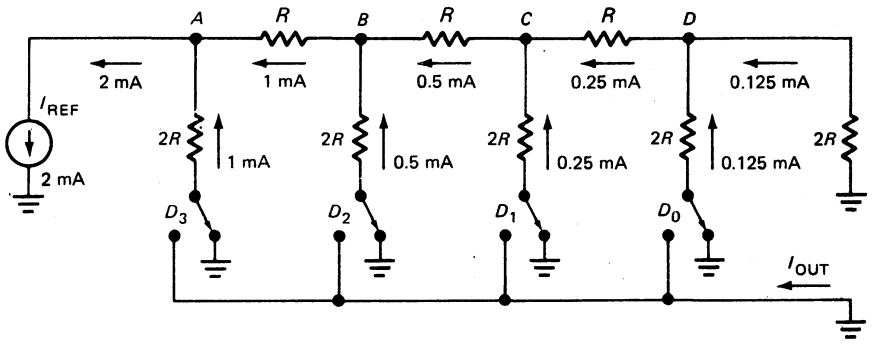


Fig. 16-29

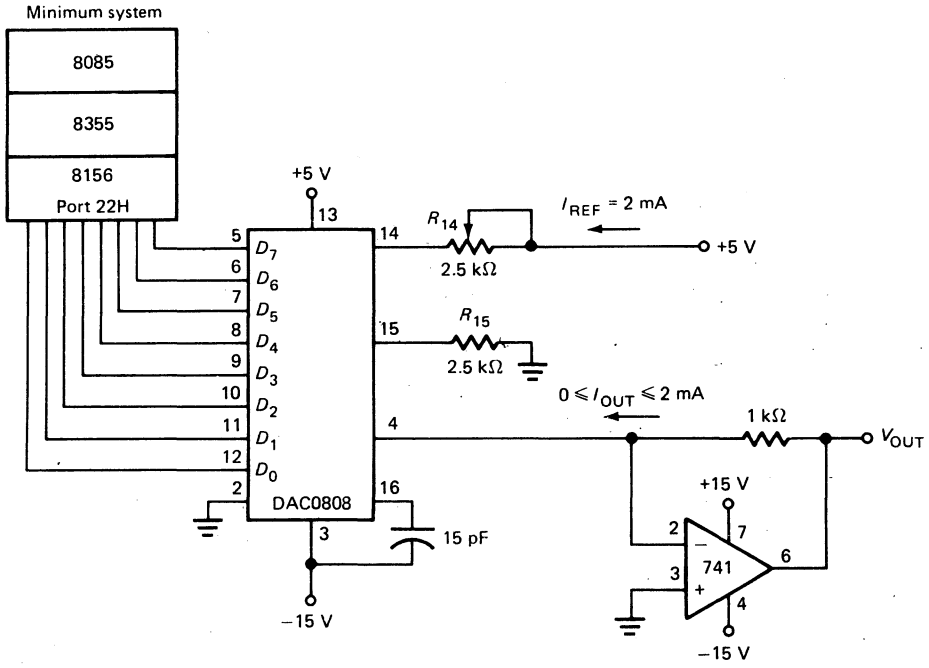


Fig. 16-30

16-7. Here is a program used with Fig. 16-30:

Label	Mnemonic
	MVI A,02H
	OUT 20H
	MVI A,FFH
LOOP1:	INR A
	OUT 22H
	CPI FFH
	JZ LOOP2
	JMP LOOP1
LOOP2:	DCR A
	OUT 22H
	JZ LOOP1
	JMP LOOP2

What does the output voltage look like when the program is run?

16-8. Hand-assemble the program of Prob. 16-7 starting at address 2000H.

16-9. The 8-bit D/A converter of Fig. 16-31 has a maximum output voltage of 2 V. If the V_{IN} = 1.5 V, what is the digital output D_7 to D_0 at the end of the conversion?

16-10. In Fig. 16-32, the digital output D_7 to D_0 has a hexadecimal equivalent of 7C at the end of the conversion. If the maximum output voltage of the D/A converter is 2 V, what is the analog input voltage?

16-11. In Fig. 16-33 R = 20 kΩ and C = 75 pF. What is the converter clock frequency?

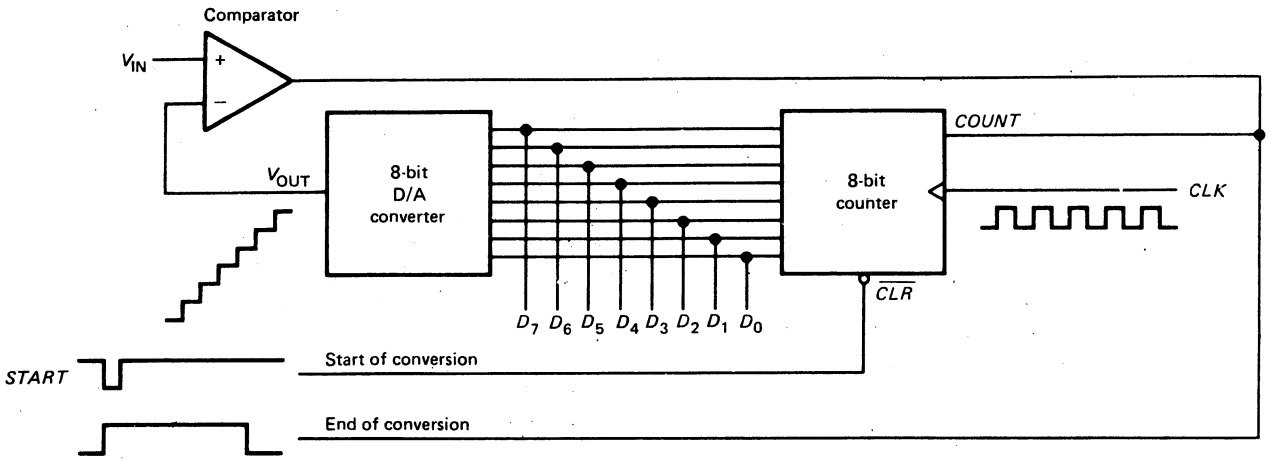


Fig. 16-31

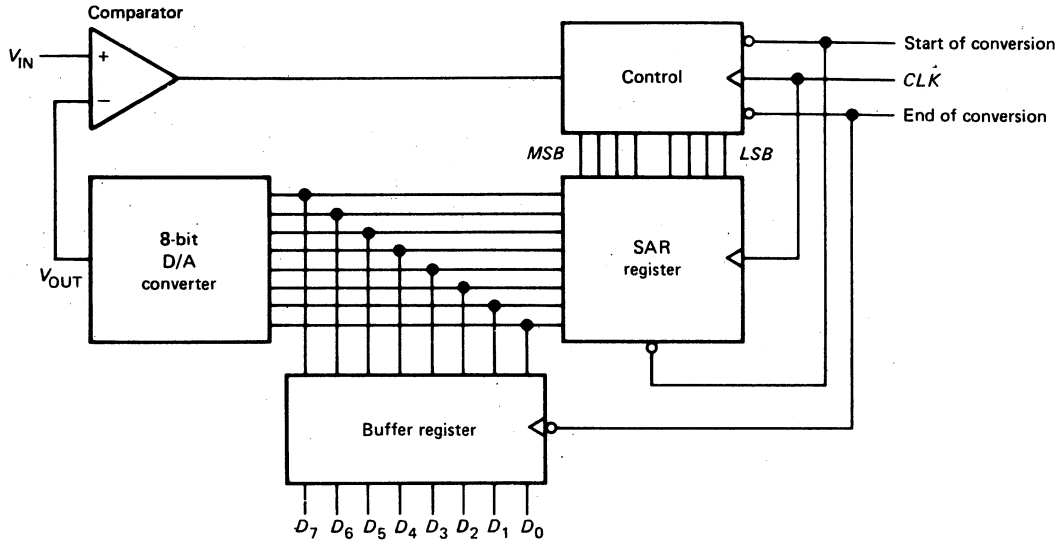


Fig. 16-32

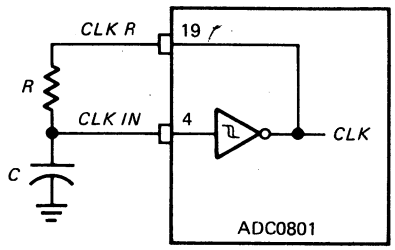


Fig. 16-33

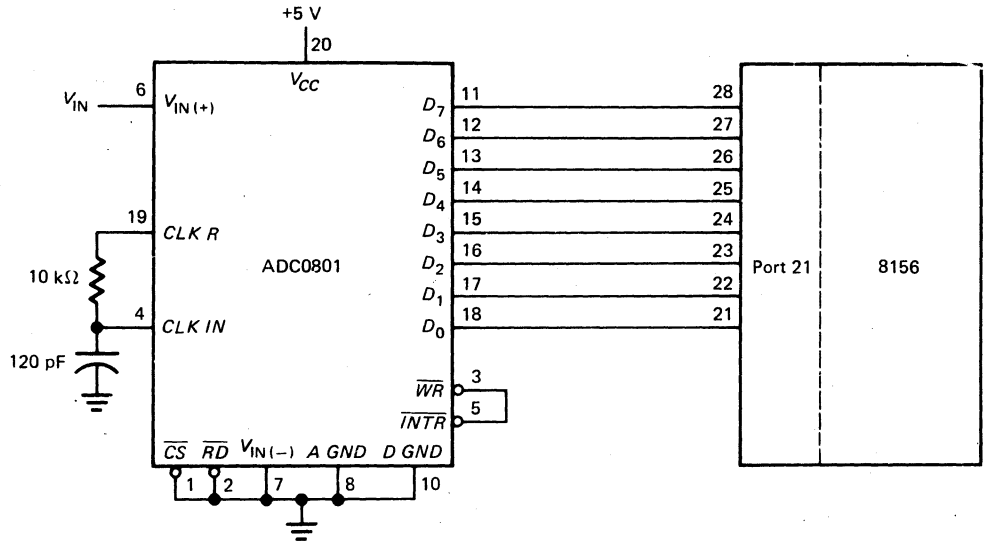


Fig. 16-34

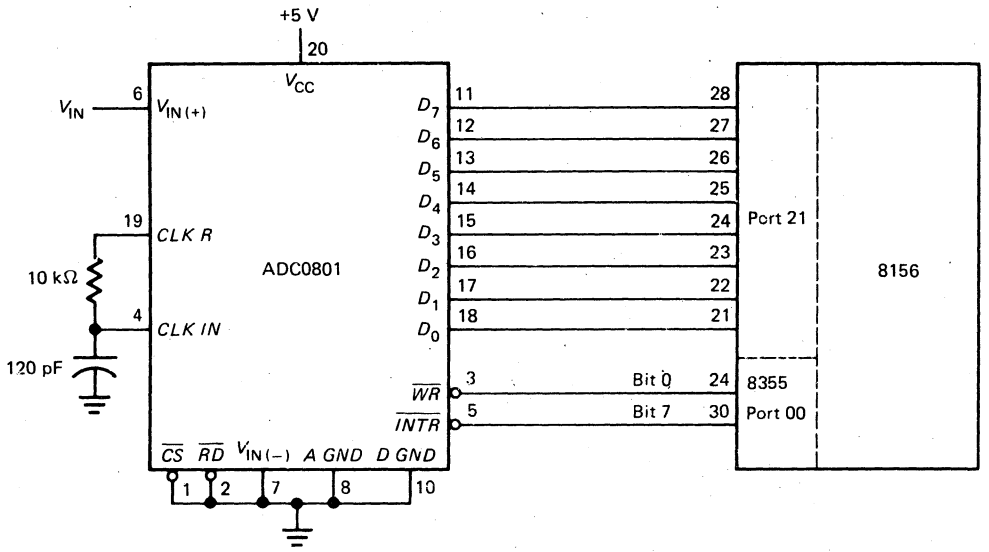


Fig. 16-35

16-12. The digital output in Fig. 16-34 is updated to BEH at the end of a conversion. What is the analog input?

16-13. The software-handshaking program of Sec. 16-7 is run with the A/D converter of Fig. 16-35. If V_{IN} = 1.23 V, what does the accumulator contain after the execution of the IN 21H?

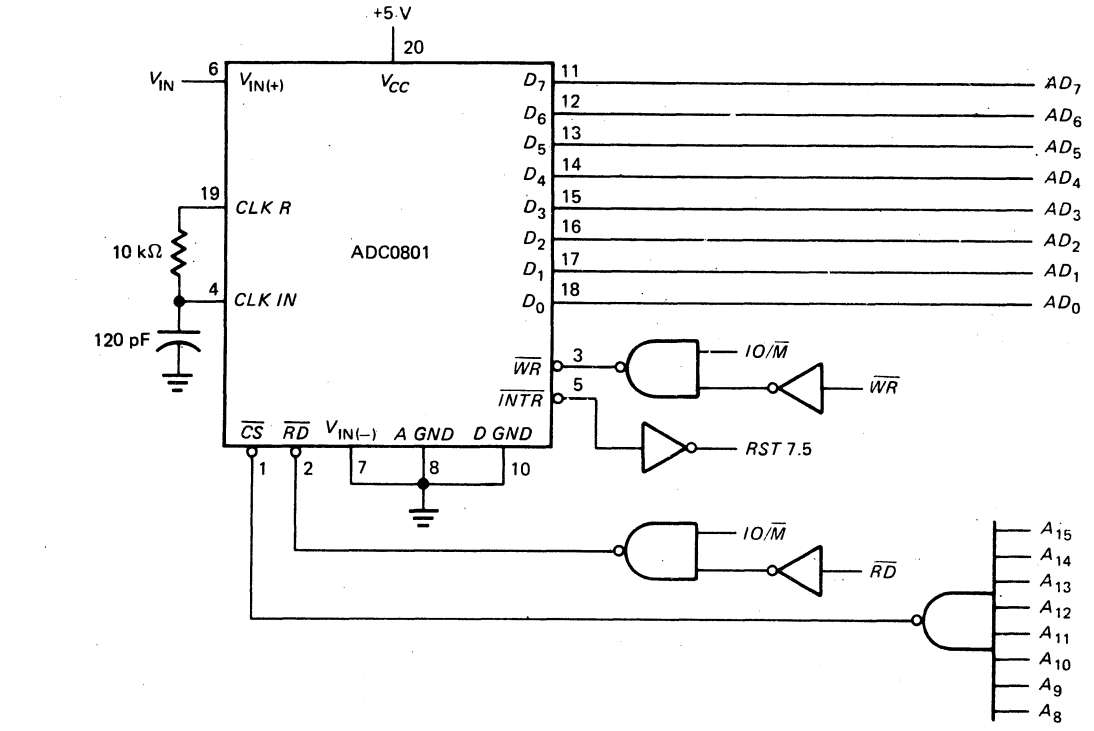
16-14. The hardware-handshaking program of Sec. 16-7 is controlling the converter of Fig. 16-36.

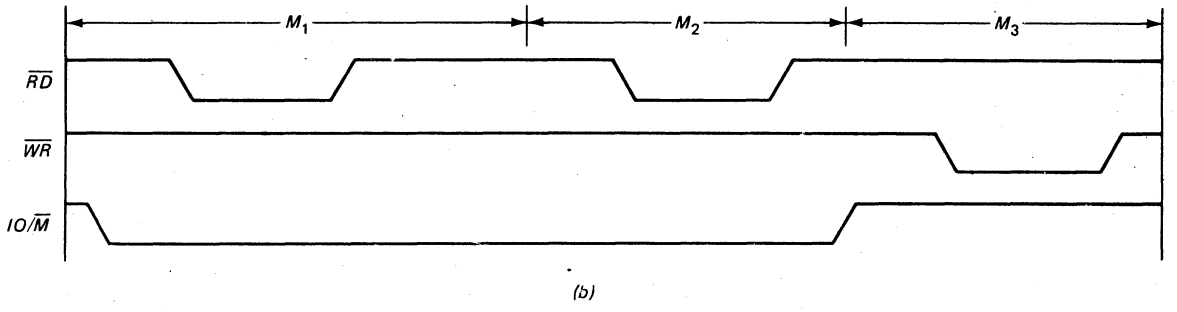
a. If the digital output is 5AH at the end of a conversion, what is the analog input?

b. The program is looping. What does the accumulator contain?

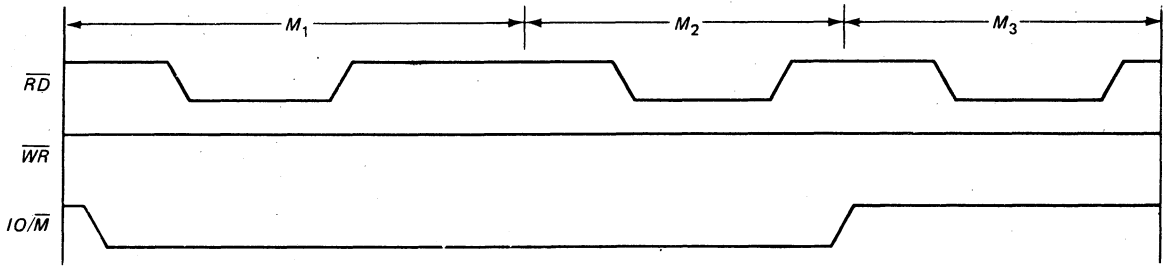
c. If the analog input voltage is 3.99 V, what does the accumulator contain after the IN FFH is executed?

16-15. In the VCO program of Sec. 16-9, how many T states does it take to pass through LOOP2 once (assume JNZ jumps back)? If the VCO of Fig. 16-37 has a frequency of 1 kHz and the system clock a frequency of 3 MHz, what is the approximate count in the accumulator after the MOV A,C has been executed?


+5 V
20
V_{IN}
6
$V_{IN(+)}$
V_{CC}
D_7 11
D_6 12
D_5 13
D_4 14
D_3 15
D_2 16
D_1 17
D_0 18
AD_7
AD_6
AD_5
AD_4
AD_3
AD_2
AD_1
AD_0
19
CLK R
10 kΩ
ADC0801
4
CLK IN
120 pF
$\overline{WR}$ 3
$\overline{INTR}$ 5
$IO/\overline{M}$
$\overline{WR}$
RST 7.5
$\overline{CS}$ 1
$\overline{RD}$ 2
$V_{IN(-)}$ 7
A GND 8
D GND 10
$IO/\overline{M}$
$\overline{RD}$
A_{15}
A_{14}
A_{13}
A_{12}
A_{11}
A_{10}
A_9
A_8


M_1
M_2
M_3
$\overline{RD}$
$\overline{WR}$
$IO/\overline{M}$

(b)


M_1
M_2
M_3
$\overline{RD}$
$\overline{WR}$
$IO/\overline{M}$

(c)

Fig. 16-36

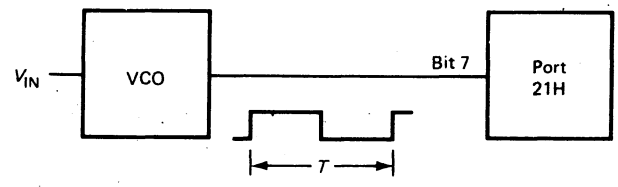


Fig. 16-37

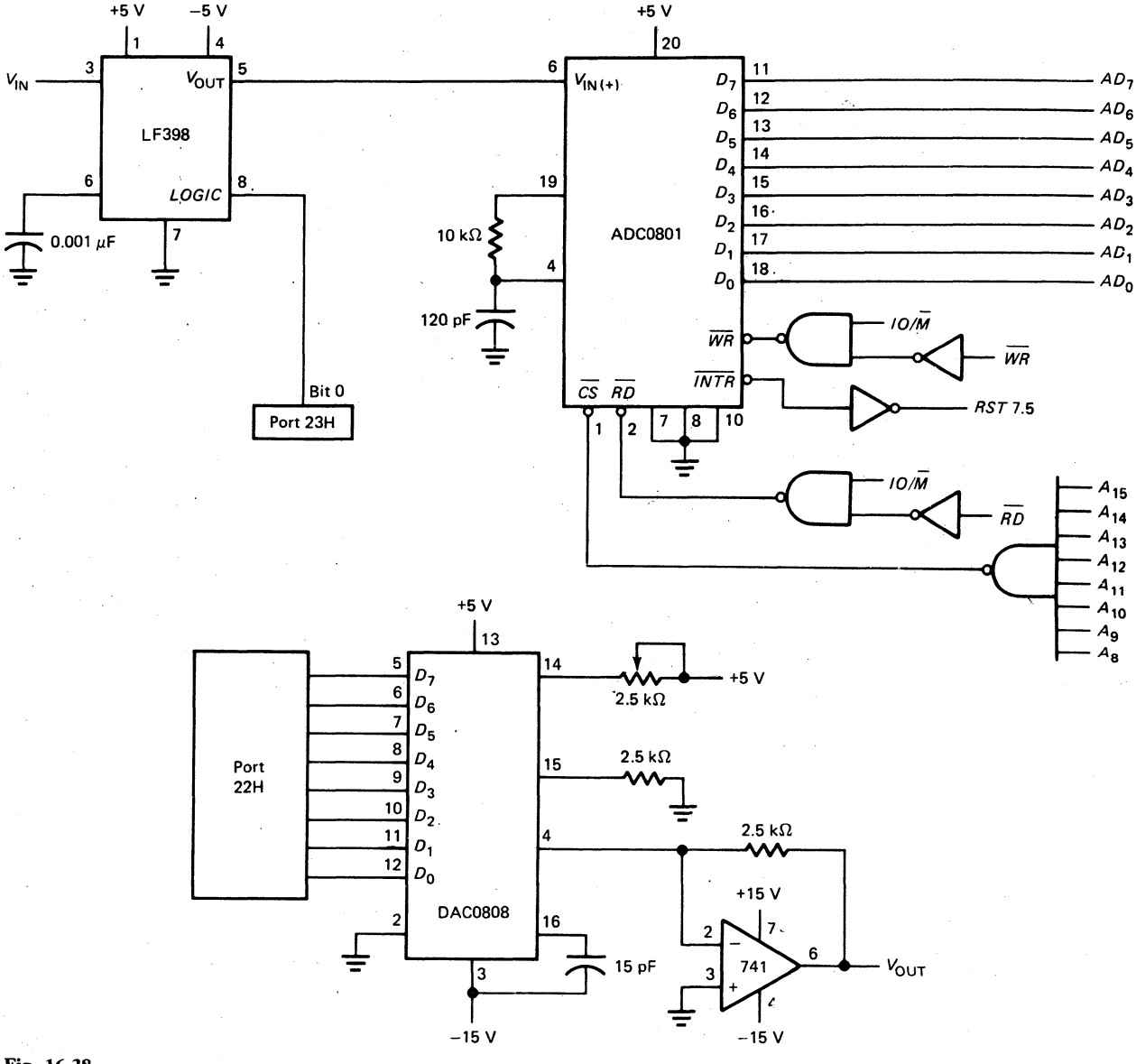


Fig. 16-38

16-16. The software-controlled sample-and-hold program of Sec. 16-10 calls a subroutine labeled ACQTIM, which produces a delay of 4 μs. This ensures adequate time for the sample to be taken.

Write a subroutine ACQTIM that produces a time delay of at least 4 μs for a system clock of 3 MHz.

16-17. Answer the following for Fig. 16-38:

a. What is the normal range of analog input voltage?

b. If V_{IN} is 0.99 V, what is the digital output of the ADC0801 after $\overline{INTR}$ goes low?

c. What is the maximum output voltage from the 741 op amp?

16-18. In the program of Example 16-4, additional instructions process the data after the IN FFH is executed. Suppose the additional instructions are as follows:

Label	Mnemonic	Comment
	STC	;Set carry flag
	CMC	;Clear carry flag
	ADI 80H	;Add 128
	JC OFF	;If carry set, go to OFF
ON:	MVI A,FFH	;Prepare to turn on heat
	JMP OUT	;Escape
OFF:	MVI A,00H	;Prepare to turn off heat
OUT:	OUT 22H	;Tell heater what to do

Answer the following:

a. For what range of analog input voltage does port 22H get loaded with FFH?

b. If the ADI 80H is changed to ADI 70H, what is the range of analog input voltage that sets the carry flag when the ADI is executed?

16-19. Suppose a transducer-controlled circuit linearly converts temperatures from 50 to 100°F into analog voltages from 0 to 5 V. This means that 50°F produces 0 V, 60°F produces 1 V, 70°F produces 2 V, etc. The analog voltage out of the transducer circuit is connected to the pin 6 of the ADC0801 in Fig. 16-38. (We don't need the sample-and-hold amplifier in this problem.)

Using all the program given in the Prob. 16-18, answer the following:

a. If the temperature is 72°F, what does port 22H contain after the OUT 22H has been executed?

b. If the temperature is 78°F, what does port 22H contain after the OUT 22H has been executed?

c. If the ADI 80H is changed to ADI 60H, what is the highest temperature that produces FFH in port 22H?

Appendixes

APPENDIX 1. BINARY-HEXADECIMAL-DECIMAL EQUIVALENTS

Binary	Hexadecimal	UB Decimal	LB Decimal
0000 0000	00	0	0
0000 0001	01	256	1
0000 0010	02	512	2
0000 0011	03	768	3
0000 0100	04	1,024	4
0000 0101	05	1,280	5
0000 0110	06	1,536	6
0000 0111	07	1,792	7
0000 1000	08	2,048	8
0000 1001	09	2,304	9
0000 1010	0A	2,560	10
0000 1011	0B	2,816	11
0000 1100	0C	3,072	12
0000 1101	0D	3,328	13
0000 1110	0E	3,584	14
0000 1111	0F	3,840	15
0001 0000	10	4,096	16
0001 0001	11	4,352	17
0001 0010	12	4,608	18
0001 0011	13	4,864	19
0001 0100	14	5,120	20
0001 0101	15	5,376	21
0001 0110	16	5,632	22
0001 0111	17	5,888	23
0001 1000	18	6,144	24
0001 1001	19	6,400	25
0001 1010	1A	6,656	26
0001 1011	1B	6,912	27
0001 1100	1C	7,168	28
0001 1101	1D	7,424	29
0001 1110	1E	7,680	30
0001 1111	1F	7,936	31
0010 0000	20	8,192	32
0010 0001	21	8,448	33
0010 0010	22	8,704	34
0010 0011	23	8,960	35
0010 0100	24	9,216	36
0010 0101	25	9,472	37
0010 0110	26	9,728	38
0010 0111	27	9,984	39
0010 1000	28	10,240	40
0010 1001	29	10,496	41
0010 1010	2A	10,752	42
0010 1011	2B	11,008	43
0010 1100	2C	11,264	44
0010 1101	2D	11,520	45
0010 1110	2E	11,776	46
0010 1111	2F	12,032	47
0011 0000	30	12,288	48
0011 0001	31	12,544	49
0011 0010	32	12,800	50
0011 0011	33	13,056	51
0011 0100	34	13,312	52
0011 0101	35	13,568	53
0011 0110	36	13,824	54
0011 0111	37	14,080	55
0011 1000	38	14,336	56
0011 1001	39	14,592	57
0011 1010	3A	14,848	58
0011 1011	3B	15,104	59
0011 1100	3C	15,360	60
0011 1101	3D	15,616	61
0011 1110	3E	15,872	62
0011 1111	3F	16,128	63
0100 0000	40	16,384	64
0100 0001	41	16,640	65
0100 0010	42	16,896	66
0100 0011	43	17,152	67
0100 0100	44	17,408	68
0100 0101	45	17,664	69
0100 0110	46	17,920	70
0100 0111	47	18,176	71
0100 1000	48	18,432	72
0100 1001	49	18,688	73
0100 1010	4A	18,944	74
0100 1011	4B	19,200	75
0100 1100	4C	19,456	76
0100 1101	4D	19,712	77
0100 1110	4E	19,968	78
0100 1111	4F	20,224	79
0101 0000	50	20,480	80

Binary	Hexadecimal	UB Decimal	LB Decimal	Binary	Hexadecimal	UB Decimal	LB Decimal
0101 0001	51	20,736	81	1000 0011	83	33,536	131
0101 0010	52	20,992	82	1000 0100	84	33,792	132
0101 0011	53	21,248	83	1000 0101	85	34,048	133
0101 0100	54	21,504	84	1000 0110	86	34,304	134
0101 0101	55	21,760	85	1000 0111	87	34,560	135
0101 0110	56	22,016	86	1000 1000	88	34,816	136
0101 0111	57	22,272	87	1000 1001	89	35,072	137
0101 1000	58	22,528	88	1000 1010	8A	35,328	138
0101 1001	59	22,784	89	1000 1011	8B	35,584	139
0101 1010	5A	23,040	90	1000 1100	8C	35,840	140
0101 1011	5B	23,296	91	1000 1101	8D	36,096	141
0101 1100	5C	23,552	92	1000 1110	8E	36,352	142
0101 1101	5D	23,808	93	1000 1111	8F	36,608	143
0101 1110	5E	24,064	94	1001 0000	90	36,864	144
0101 1111	5F	24,320	95	1001 0001	91	37,120	145
0110 0000	60	24,576	96	1001 0010	92	37,376	146
0110 0001	61	24,832	97	1001 0011	93	37,632	147
0110 0010	62	25,088	98	1001 0100	94	37,888	148
0110 0011	63	25,344	99	1001 0101	95	38,144	149
0110 0100	64	25,600	100	1001 0110	96	38,400	150
0110 0101	65	25,856	101	1001 0111	97	38,656	151
0110 0110	66	26,112	102	1001 1000	98	38,912	152
0110 0111	67	26,368	103	1001 1001	99	39,168	153
0110 1000	68	26,624	104	1001 1010	9A	39,424	154
0110 1001	69	26,880	105	1001 1011	9B	39,680	155
0110 1010	6A	27,136	106	1001 1100	9C	39,936	156
0110 1011	6B	27,392	107	1001 1101	9D	40,192	157
0110 1100	6C	27,648	108	1001 1110	9E	40,448	158
0110 1101	6D	27,904	109	1001 1111	9F	40,704	159
0110 1110	6E	28,160	110	1010 0000	A0	40,960	160
0110 1111	6F	28,416	111	1010 0001	A1	41,216	161
0111 0000	70	28,672	112	1010 0010	A2	41,472	162
0111 0001	71	28,928	113	1010 0011	A3	41,728	163
0111 0010	72	29,184	114	1010 0100	A4	41,984	164
0111 0011	73	29,440	115	1010 0101	A5	42,240	165
0111 0100	74	29,696	116	1010 0110	A6	42,496	166
0111 0101	75	29,952	117	1010 0111	A7	42,752	167
0111 0110	76	30,208	118	1010 1000	A8	43,008	168
0111 0111	77	30,464	119	1010 1001	A9	43,264	169
0111 1000	78	30,720	120	1010 1010	AA	43,520	170
0111 1001	79	30,976	121	1010 1011	AB	43,776	171
0111 1010	7A	31,232	122	1010 1100	AC	44,032	172
0111 1011	7B	31,488	123	1010 1101	AD	44,288	173
0111 1100	7C	31,744	124	1010 1110	AE	44,544	174
0111 1101	7D	32,000	125	1010 1111	AF	44,800	175
0111 1110	7E	32,256	126	1011 0000	B0	45,056	176
0111 1111	7F	32,512	127	1011 0001	B1	45,312	177
1000 0000	80	32,768	128	1011 0010	B2	45,568	178
1000 0001	81	33,024	129	1011 0011	B3	45,824	179
1000 0010	82	33,280	130	1011 0100	B4	46,080	180

APPENDIX 1. BINARY-HEXADECIMAL-DECIMAL EQUIVALENTS (*Continued*)

Binary	Hexadecimal	UB Decimal	LB Decimal	Binary	Hexadecimal	UB Decimal	LB Decimal
1011 0101	B5	46,336	181	1101 1101	DD	56,576	221
1011 0110	B6	46,592	182	1101 1110	DE	56,832	222
1011 0111	B7	46,848	183	1101 1111	DF	57,088	223
1011 1000	B8	47,104	184	1110 0000	E0	57,344	224
1011 1001	B9	47,360	185	1110 0001	E1	57,600	225
1011 1010	BA	47,616	186	1110 0010	E2	57,856	226
1011 1011	BB	47,872	187	1110 0011	E3	58,112	227
1011 1100	BC	48,128	188	1110 0100	E4	58,368	228
1011 1101	BD	48,384	189	1110 0101	E5	58,624	229
1011 1110	BE	48,640	190	1110 0110	E6	58,880	230
1011 1111	BF	48,896	191	1110 0111	E7	59,136	231
1100 0000	C0	49,152	192	1110 1000	E8	59,392	232
1100 0001	C1	49,408	193	1110 1001	E9	59,648	233
1100 0010	C2	49,664	194	1110 1010	EA	59,904	234
1100 0011	C3	49,920	195	1110 1011	EB	60,160	235
1100 0100	C4	50,176	196	1110 1100	EC	60,416	236
1100 0101	C5	50,432	197	1110 1101	ED	60,672	237
1100 0110	C6	50,688	198	1110 1110	EE	60,928	238
1100 0111	C7	50,944	199	1110 1111	EF	61,184	239
1100 1000	C8	51,200	200	1111 0000	F0	61,440	240
1100 1001	C9	51,456	201	1111 0001	F1	61,696	241
1100 1010	CA	51,712	202	1111 0010	F2	61,952	242
1100 1011	CB	51,968	203	1111 0011	F3	62,208	243
1100 1100	CC	52,224	204	1111 0100	F4	62,464	244
1100 1101	CD	52,480	205	1111 0101	F5	62,720	245
1100 1110	CE	52,736	206	1111 0110	F6	62,976	246
1100 1111	CF	52,992	207	1111 0111	F7	63,232	247
1101 0000	D0	53,248	208	1111 1000	F8	63,488	248
1101 0001	D1	53,504	209	1111 1001	F9	63,744	249
1101 0010	D2	53,760	210	1111 1010	FA	64,000	250
1101 0011	D3	54,016	211	1111 1011	FB	64,256	251
1101 0100	D4	54,272	212	1111 1100	FC	64,512	252
1101 0101	D5	54,528	213	1111 1101	FD	64,768	253
1101 0110	D6	54,784	214	1111 1110	FE	65,024	254
1101 0111	D7	55,040	215	1111 1111	FF	65,280	255
1101 1000	D8	55,296	216				
1101 1001	D9	55,552	217				
1101 1010	DA	55,808	218				
1101 1011	DB	56,064	219				
1101 1100	DC	56,320	220				

APPENDIX 2. 7400 SERIES TTL

Number	Function
7400	Quad 2-input NAND gates
7401	Quad 2-input NAND gates (open collector)
7402	Quad 2-input NOR gates
7403	Quad 2-input NOR gates (open collector)
7404	Hex inverters
7405	Hex inverters (open collector)
7406	Hex inverter buffer-driver
7407	Hex buffer-drivers
7408	Quad 2-input AND gates
7409	Quad 2-input AND gates (open collector)
7410	Triple 3-input NAND gates
7411	Triple 3-input AND gates
7412	Triple 3-input NAND gates (open collector)
7413	Dual Schmitt triggers
7414	Hex Schmitt triggers
7416	Hex inverter buffer-drivers
7417	Hex buffer-drivers
7420	Dual 4-input NAND gates
7421	Dual 4-input AND gates
7422	Dual 4-input NAND gates (open collector)
7423	Expandable dual 4-input NOR gates
7425	Dual 4-input NOR gates
7226	Quad 2-input TTL-MOS interface NAND gates
7427	Triple 3-input NOR gates
7428	Quad 2-input NOR buffer
7430	8-input NAND gate
7432	Quad 2-input OR gates
7437	Quad 2-input NAND buffers
7438	Quad 2-input NAND buffers (open collector)
7439	Quad 2-input NAND buffers (open collector)
7440	Dual 4-input NAND buffers
7441	BCD-to-decimal decoder–Nixie driver
7442	BCD-to-decimal decoder
7443	Excess 3-to-decimal decoder
7444	Excess Gray-to-decimal
7445	BCD-to-decimal decoder-driver
7446	BCD-to-seven segment decoder-drivers (30-V output)
7447	BCD-to-seven segment decoder-drivers (15-V output)
7448	BCD-to-seven segment decoder-drivers
7450	Expandable dual 2-input 2-wide AND-OR-INVERT gates
7451	Dual 2-input 2-wide AND-OR-INVERT gates
7452	Expandable 2-input 4-wide AND-OR gates
7453	Expandable 2-input 4-wide AND-OR-INVERT gates
7454	2-input 4-wide AND-OR-INVERT gates
7455	Expandable 4-input 2-wide AND-OR-INVERT gates
7459	Dual 2-3 input 2-wide AND-OR-INVERT gates
7460	Dual 4-input expanders
7461	Triple 3-input expanders
7462	2-2-3-3 input 4-wide expanders
7464	2-2-3-4 input 4-wide AND-OR-INVERT gates
7465	4-wide AND-OR-INVERT gates (open collector)
7470	Edge-triggered *JK* flip-flop
7472	*JK* master-slave flip-flop
7473	Dual *JK* master-slave flip-flop
7474	Dual *D* flip-flop
7475	Quad latch
7476	Dual *JK* master-slave flip-flop
7480	Gates full adder
7482	2-bit binary full adder
7483	4-bit binary full adder
7485	4-bit magnitude comparator
7486	Quad EXCLUSIVE-OR gate
7489	64-bit random-access read-write memory
7490	Decade counter
7491	8-bit shift register
7492	Divide-by-12 counter
7493	4-bit binary counter
7494	4-bit shift register
7495	4-bit right-shift–left-shift register
7496	5-bit parallel-in–parallel-out shift register
74100	4-bit bistable latch
74104	*JK* master-slave flip-flop
74105	*JK* master-slave flip-flop
74107	Dual *JK* master-slave flip-flop
74109	Dual *JK* positive-edge-triggered flip-flop
74116	Dual 4-bit latches with clear
74121	Monostable multivibrator
74122	Monostable multivibrator with clear
74123	Monostable multivibrator
74125	Three-state quad bus buffer
74126	Three-state quad bus buffer
74132	Quad Schmitt trigger
74136	Quad 2-input EXCLUSIVE-OR gate
74141	BCD-to-decimal decoder-driver
74142	BCD counter-latch-driver
74145	BCD-to-decimal decoder-driver
74147	10/4 priority encoder
74148	Priority encoder
74150	16-line-to-1-line multiplexer
74151	8-channel digital multiplexer
74152	8-channel data selector-multiplexer

APPENDIX 2. 7400 SERIES TTL (*Continued*)

Number	Function	Number	Function
74153	Dual 4/1 multiplexer	74190	Up-down decade counter
74154	4-line–to–16-line decoder-demultiplexer	74191	Synchronous binary up-down counter
74155	Dual 2/4 demultiplexer	74192	Binary up-down counter
74156	Dual 2/4 demultiplexer	74193	Binary up-down counter
74157	Quad 2/1 data selector	74194	4-bit directional shift register
74160	Decade counter with asynchronous clear	74195	4-bit parallel-access shift register
74161	Synchronous 4-bit counter	74196	Presettable decade counter
74162	Synchronous 4-bit counter	74197	Presettable binary counter
74163	Synchronous 4-bit counter	74198	8-bit shift register
74164	8-bit serial shift register	74199	8-bit shift register
74165	Parallel-load 8-bit serial shift register	74221	Dual one-shot Schmitt trigger
74166	8-bit shift register	74251	Three-state 8-channel multiplexer
74173	4-bit three-state register	74259	8-bit addressable latch
74174	Hex *F* flip-flop with clear	74276	Quad *JK* flip-flop
74175	Quad *D* flip-flop with clear	74279	Quad debouncer
74176	35-MHz presettable decade counter	74283	4-bit binary full adder with fast carry
74177	35-MHz presettable binary counter	74284	Three-state 4-bit multiplexer
74179	4-bit parallel-access shift register	74285	Three-state 4-bit multiplexer
74180	8-bit odd-even parity generator-checker	74365	Three-state hex buffers
74181	Arithmetic-logic unit	74366	Three-state hex buffers
74182	Look-ahead carry generator	74367	Three-state hex buffers
74184	BCD-to-binary converter	74368	Three-state hex buffers
74185	Binary-to-BCD converter	74390	Individual clocks with flip-flops
74189	Three-state 64-bit random-access memory	74393	Dual 4-bit binary counter

APPENDIX 3. PINOUTS AND FUNCTION TABLES

74LS83

The 74LS83 is a 4-bit full adder; the binary output is

$$\mathbf{S = A + B}$$

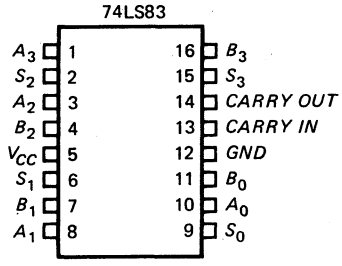


Fig. A-1

In Fig. A-1, pins 1, 3, 8, and 10 are the **A** input (A_3, A_2, A_1, A_0); pins 16, 4, 7, and 11 are the **B** input (B_3, B_2, B_1, B_0); and pins 15, 2, 6, and 9 are the **S** output (S_3, S_2, S_1, S_0). Pin 13 is the CARRY IN, and pin 14 is the CARRY OUT.

74LS157

This chip is a word multiplexer. Two words of 4 bits each are the inputs; one word of 4 bits is the output. The two input words are designated **L** (left) and **R** (right); the output word is **Y.** In Fig A-2, pin 1 (SELECT) and pin 15 (STROBE) are control inputs. The **L** word goes to pins 14, 11, 5, 2 (L_3, L_2, L_1, L_0), and the **R** word goes to pins 13, 10, 6, and 3 (R_3, R_2, R_1, R_0).

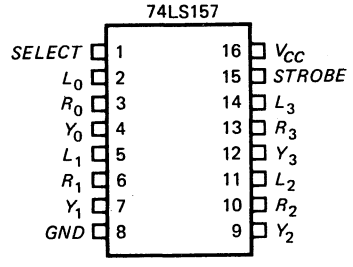


Fig. A-2

TABLE A-1. FUNCTION TABLE

STROBE	*SELECT*	Y	Comment
1	X	0	Output goes low
0	0	**L**	Output equals left word
0	1	**R**	Output equals right word

As indicated in Table A-1, a high *STROBE* input produces a low output, no matter what the input words. When *STROBE* is low, the *SELECT* input controls the operation. A low *SELECT* will send the **L** word to the output; a high *SELECT* sends the **R** word to the output.

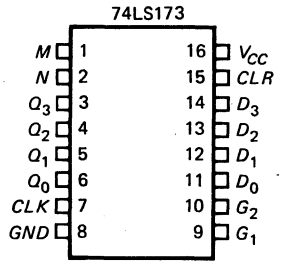


Fig. A-3

74LS173

The 74LS173 is a 4-bit buffer register with three-state outputs. In Fig. A-3, pins 14, 13, 12, and 11 are the data inputs (D_3, D_2, D_1, D_0). Pins 3, 4, 5, and 6 are the data outputs (Q_3, Q_2, Q_1, Q_0). Pins 9 and 10 (G_1 and G_2) are the input control. Pins 1 and 2 (M and N) are the output control.

As shown in Table A-2, both *M* and *N* must be low to get a *Q* output. If either *M* or *N* (or both) is high, the output is three-stated (floating or high impedance).

When *M* and *N* are both low, Table A-3 applies. As indicated, a high *CLEAR* will clear all *Q* bits to 0. When *CLEAR* is low, G_1 and G_2 control input loading. If either G_1 or G_2 (or both) are high, no change takes place in the *Q* bits. When both G_1 and G_2 are low, the next positive clock edge loads the input data.

TABLE A-2. OUTPUT CONTROL

M	*N*	Output
0	0	Connected
0	1	Hi-Z
1	0	Hi-Z
1	1	Hi-Z

TABLE A-3. FUNCTION TABLE FOR *M* = 0 AND *N* = 0

CLEAR	*CLOCK*	G_1	G_2	D_n	Q_n	Comment
1	X	X	X	X	0	Clear output
0	0	X	X	X	NC	No change
0	↑	1	X	X	NC	No change
0	↑	X	1	X	NC	No change
0	↑	0	0	0	0	Reset bit *n*
0	↑	0	0	1	1	Set bit *n*

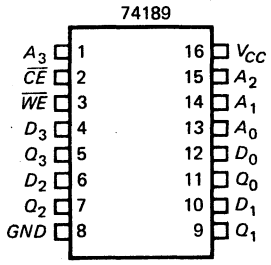


Fig. A-4

74189

The 74189 is a 64-bit RAM organized as 16 words of 4 bits each. In Fig. A-4 pins 1, 15, 14, and 13 are the address inputs (A_3, A_2, A_1, A_0). Pins 4, 6, 10, and 12 are the data inputs (D_3, D_2, D_1, D_0). Pins 5, 7, 9, and 11 are the data outputs (Q_3, Q_2, Q_1, Q_0).

TABLE A-4. FUNCTION TABLE

$\overline{CE}$	$\overline{WE}$	Output	Comment
1	X	Hi-Z	Do nothing
0	0	Hi-Z	Write complement
0	1	Stored word	Read

Table A-4 summarizes the operation of this read-write memory. When $\overline{CE}$ is high, the output is three-stated (high impedance). When $\overline{CE}$ is low and $\overline{WE}$ is low, the complement of the input data word is stored at the addressed memory location; during this write operation, the output is three-stated. When $\overline{CE}$ is low and $\overline{WE}$ is high, the stored word appears at the output.

APPENDIX 4. SAP-1 PARTS LIST

Chips

C1: 74LS107, dual *JK* master-slave flip-flop
C2: 74LS107
C3: 74LS126, quad three-state normally open switches
C4: 74LS173, buffer register, three-state outputs, 4 bits
C5: 74LS157, 2-to-1 nibble multiplexer
C6: 74189, 64-bit (16 × 4) static RAM, three-state outputs
C7: 74189
C8: 74LS173
C9: 74LS173
C10: 74LS173
C11: 74LS173
C12: 74LS126
C13: 74LS126
C14: 74LS86, quad 2-input EXCLUSIVE-OR gates
C15: 74LS86
C16: 74LS83, quad full adders
C17: 74LS83
C18: 74LS126
C19: 74LS126
C20: 74LS173
C21: 74LS173
C22: 74LS173
C23: 74LS173
C24: 7400, quad 2-input NAND gates
C25: 74LS10, triple 3-input NAND gates
C26: 74LS00
C27: 7404, hex inverter
C28: NE555, timer
C29: 74LS107
C30: LM340T-5, voltage regulator, 5 V
C31: 74LS04, hex inverter
C32: 74LS20, dual 4-input NAND gates
C33: 74LS20
C34: 74LS20
C35: 74LS04
C36: 74LS107
C37: 74LS107
C38: 74LS107
C39: 74LS00
C40: 74LS00
C41: 74LS00
C42: 74LS00
C43: 74LS00
C44: 74LS20
C45: 74LS10
C46: 74LS00
C47: 74LS04
C48: 74LS04

Diodes

D1: 1N4001, rectifier diode, 50 PIV, 1 A
D2: 1N4001
D3: 1N4001
D4: 1N4001

Switches

S1: SPST DIP switch, 4 bits
S2: DPST on-off
S3: SPST DIP, 8 bits
S4: SPST push button, momentary, normally open
S5: SPDT push button, momentary
S6: SPDT push button, momentary
S7: SPDT on-on switch

Miscellaneous

Resistors: eight 1-kΩ, fourteen 10-kΩ, one 18-kΩ, one 36-kΩ
Capacitors: 0.01-μF, 0.1-μF, 1000-μF (50 V)
Transformer: F-25X = 115 V primary, 12.6 V secondary CT, 1.5 A
Fuse: $\frac{3}{8}$-A slow blow

Totals

1N4001–4	74LS20–4
LM340T-5–1	74LS83–2
NE555–1	74LS86–2
7400–1	74LS107–6
74LS00–7	74LS126–5
7404–1	74LS157–1
74LS04–4	74LS173–9
74LS10–2	74189–2

Instruction	Op Code	*T* states	Flags	Main Effect
ACI byte	CE	7	All	A ← A + CY + byte
ADC A	8F	4	All	A ← A + A + CY
ADC B	88	4	All	A ← A + B + CY
ADC C	89	4	All	A ← A + C + CY
ADC D	8A	4	All	A ← A + D + CY
ADC E	8B	4	All	A ← A + E + CY
ADC H	8C	4	All	A ← A + H + CY
ADC L	8D	4	All	A ← A + L + CY
ADC M	8E	7	All	A ← A + M_{HL} + CY
ADD A	87	4	All	A ← A + A
ADD B	80	4	All	A ← A + B
ADD C	81	4	All	A ← A + C
ADD D	82	4	All	A ← A + D
ADD E	83	4	All	A ← A + E
ADD H	84	4	All	A ← A + H
ADD L	85	4	All	A ← A + L
ADD M	86	7	All	A ← A + M_{HL}
ADI byte	C6	7	All	A ← A + byte
ANA A	A7	4	All	A ← A AND A
ANA B	A0	4	All	A ← A AND B
ANA C	A1	4	All	A ← A AND C
ANA D	A2	4	All	A ← A AND D
ANA E	A3	4	All	A ← A AND E
ANA H	A4	4	All	A ← A AND H
ANA L	A5	4	All	A ← A AND L
ANA M	A6	7	All	A ← A AND M_{HL}
ANI byte	E6	7	All	A ← A AND byte
CALL address	CD	18	None	PC ← address
CC address	DC	18/9	None	PC ← address if *CY* = 1
CM address	FC	18/9	None	PC ← address if *S* = 1
CMA	2F	4	None	A ← $\overline{A}$
CMC	3F	4	CY	CY ← $\overline{CY}$
CMP A	BF	4	All	Z ← 1 if A = A
CMP B	B8	4	All	Z ← 1 if A = B
CMP C	B9	4	All	Z ← 1 if A = C
CMP D	BA	4	All	Z ← 1 if A = D
CMP E	BB	4	All	Z ← 1 if A = E
CMP H	BC	4	All	Z ← 1 if A = H
CMP L	BD	4	All	Z ← 1 if A = L
CMP M	BE	7	All	Z ← 1 if A = M_{HL}
CNC address	D4	18/9	None	PC ← address if *CY* = 0
CNZ address	C4	18/9	None	PC ← address if *Z* = 0
CP address	F4	18/9	None	PC ← address if *S* = 0
CPE address	EC	18/9	None	PC ← address if *P* = 1
CPI byte	FE	7	All	Z ← 1 if *A* = byte
CPO address	E4	18/9	None	PC ← address if *P* = 0
CZ address	CC	18/9	None	PC ← address if *Z* = 1
DAA	27	4	All	A ← BCD number
DAD B	09	10	CY	HL ← HL + BC
DAD D	19	10	CY	HL ← HL + DE
DAD H	29	10	CY	HL ← HL + HL

Instruction	Op Code	*T* states	Flags	Main Effect
DAD SP	39	10	CY	HL ← HL + SP
DCR A	3D	4	All but CY	A ← A − 1
DCR B	05	4	All but CY	B ← B − 1
DCR C	0D	4	All but CY	C ← C − 1
DCR D	15	4	All but CY	D ← D − 1
DCR E	1D	4	All but CY	E ← E − 1
DCR H	25	4	All but CY	H ← H − 1
DCR L	2D	4	All but CY	L ← L − 1
DCR M	35	10	All but CY	$M_{HL} \leftarrow M_{HL} - 1$
DCX B	0B	6	None	BC ← BC − 1
DCX D	1B	6	None	DE ← DE − 1
DCX H	2B	6	None	HL ← HL − 1
DCX SP	3B	6	None	SP ← SP − 1
DI	F3	4	None	Disable interrupts
EI	FB	4	None	Enable interrupts
HLT	76	5	None	Stop processing
IN byte	DB	10	None	A ← byte
INR A	3C	4	All but CY	A ← A + 1
INR B	04	4	All but CY	B ← B + 1
INR C	0C	4	All but CY	C ← C + 1
INR D	14	4	All but CY	D ← D + 1
INR E	1C	4	All but CY	E ← E + 1
INR H	24	4	All but CY	H ← H + 1
INR L	2C	4	All but CY	L ← L + 1
INR M	34	10	All but CY	$M_{HL} \leftarrow M_{HL} + 1$
INX B	03	6	None	BC ← BC + 1
INX D	13	6	None	DE ← DE + 1
INX H	23	6	None	HL ← HL + 1
INX SP	33	6	None	SP ← SP + 1
JC address	DA	10/7	None	PC ← address if $CY = 1$
JM address	FA	10/7	None	PC ← address if $S = 1$
JMP address	C3	10	None	PC ← address
JNC address	D2	10/7	None	PC ← address if $CY = 0$
JNZ address	C2	10/7	None	PC ← address if $Z = 0$
JP address	F2	10/7	None	PC ← address if $S = 0$
JPE address	EA	10/7	None	PC ← address if $P = 1$
JPO address	E2	10/7	None	PC ← address if $P = 0$
JZ address	CA	10/7	None	PC ← address if $Z = 1$
LDA address	3A	13	None	$A \leftarrow M_{adr}$
LDAX B	0A	7	None	$A \leftarrow M_{BC}$
LDAX D	1A	7	None	$A \leftarrow M_{DE}$
LHLD address	2A	16	None	$H \leftarrow M_{adr}$
LXI B, dble	01	10	None	BC ← dble
LXI D, dble	11	10	None	DE ← dble
LXI H, dble	21	10	None	HL ← dble
LXI SP, dble	31	10	None	SP ← dble
MOV A,A	7F	4	None	A ← A
MOV A,B	78	4	None	A ← B
MOV A,C	79	4	None	A ← C
MOV A,D	7A	4	None	A ← D
MOV A,E	7B	4	None	A ← E
MOV A,H	7C	4	None	A ← H
MOV A,L	7D	4	None	A ← L

Instruction	Op Code	*T* states	Flags	Main Effect
MOV A,M	7E	7	None	A ← M_{HL}
MOV B,A	47	4	None	B ← A
MOV B,B	40	4	None	B ← B
MOV B,C	41	4	None	B ← C
MOV B,D	42	4	None	B ← D
MOV B,E	43	4	None	B ← E
MOV B,H	44	4	None	B ← H
MOV B,L	45	4	None	B ← L
MOV B,M	46	7	None	B ← M_{HL}
MOV C,A	4F	4	None	C ← A
MOV C,B	48	4	None	C ← B
MOV C,C	49	4	None	C ← C
MOV C,D	4A	4	None	C ← D
MOV C,E	4B	4	None	C ← E
MOV C,H	4C	4	None	C ← H
MOV C,L	4D	4	None	C ← L
MOV C,M	4E	7	None	C ← M_{HL}
MOV D,A	57	4	None	D ← A
MOV D,B	50	4	None	D ← B
MOV D,C	51	4	None	D ← C
MOV D,D	52	4	None	D ← D
MOV D,E	53	4	None	D ← E
MOV D,H	54	4	None	D ← H
MOV D,L	55	4	None	D ← L
MOV D,M	56	7	None	D ← M_{HL}
MOV E,A	5F	4	None	E ← A
MOV E,B	58	4	None	E ← B
MOV E,C	59	4	None	E ← C
MOV E,D	5A	4	None	E ← D
MOV E,E	5B	4	None	E ← E
MOV E,H	5C	4	None	E ← H
MOV E,L	5D	4	None	E ← L
MOV E,M	5E	7	None	E ← M_{HL}
MOV H,A	67	4	None	H ← A
MOV H,B	60	4	None	H ← B
MOV H,C	61	4	None	H ← C
MOV H,D	62	4	None	H ← D
MOV H,E	63	4	None	H ← E
MOV H,H	64	4	None	H ← H
MOV H,L	65	4	None	H ← L
MOV H,M	66	7	None	H ← M_{HL}
MOV L,A	6F	4	None	L ← A
MOV L,B	68	4	None	L ← B
MOV L,C	69	4	None	L ← C
MOV L,D	6A	4	None	L ← D
MOV L,E	6B	4	None	L ← E
MOV L,H	6C	4	None	L ← H
MOV L,L	6D	4	None	L ← L
MOV L,M	6E	7	None	L ← M_{HL}
MOV M,A	77	7	None	M_{HL} ← A
MOV M,B	70	7	None	M_{HL} ← B

Instruction	Op Code	*T* states	Flags	Main Effect
MOV M,C	71	7	None	$M_{HL} \leftarrow C$
MOV M,D	72	7	None	$M_{HL} \leftarrow D$
MOV M,E	73	7	None	$M_{HL} \leftarrow E$
MOV M,H	74	7	None	$M_{HL} \leftarrow H$
MOV M,L	75	7	None	$M_{HL} \leftarrow L$
MVI A,byte	3E	7	None	A ← byte
MVI B,byte	06	7	None	B ← byte
MVI C,byte	0E	7	None	C ← byte
MVI D,byte	16	7	None	D ← byte
MVI E,byte	1E	7	None	E ← byte
MVI H,byte	26	7	None	H ← byte
MVI L,byte	2E	7	None	L ← byte
MVI M,byte	36	10	None	$M_{HL} \leftarrow$ byte
NOP	00	4	None	Delay
ORA A	B7	4	All	A ← A OR A
ORA B	B0	4	All	A ← A OR B
ORA C	B1	4	All	A ← A OR C
ORA D	B2	4	All	A ← A OR D
ORA E	B3	4	All	A ← A OR E
ORA H	B4	4	All	A ← A OR H
ORA L	B5	4	All	A ← A OR L
ORA M	B6	7	All	A ← A OR M_{HL}
ORI byte	F6	7	All	A ← A OR byte
OUT byte	D3	10	None	Port byte ← A
PCHL	E9	6	None	PC ← HL
POP B	C1	10	None	B ← M_{stk}
POP D	D1	10	None	D ← M_{stk}
POP H	E1	10	None	H ← M_{stk}
POP PSW	F1	10	None	F ← M_{stk}, A ← $M_{stk} - 1$
PUSH B	C5	12	None	$M_{stk} - 1 \leftarrow B$, $M_{stk} - 2 \leftarrow C$
PUSH D	D5	12	None	$M_{stk} - 1 \leftarrow D$, $M_{stk} - 2 \leftarrow E$
PUSH H	E5	12	None	$M_{stk} - 1 \leftarrow H$, $M_{stk} - 2 \leftarrow L$
PUSH PSW	F5	12	None	$M_{stk} - 1 \leftarrow A$, $M_{stk} - 2 \leftarrow F$
RAL	17	4	CY	Rotate all left
RAR	1F	4	CY	Rotate all right
RC	D8	12/6	None	PC ← return address if $CY = 1$
RET	C9	10	None	PC ← return address
RIM	20	4	None	A ← I
RLC	07	4	CY	Rotate left with carry
RM	F8	12/6	None	PC ← return address if $S = 1$
RNC	D0	12/6	None	PC ← return address if $CY = 0$
RNZ	C0	12/6	None	PC ← return address if $Z = 0$
RP	F0	12/6	None	PC ← return address if $S = 0$
RPE	E8	12/6	None	PC ← return address if $P = 1$
RPO	E0	12/6	None	PC ← return address if $P = 0$
RRC	0F	4	CY	Rotate right with carry
RST 0	C7	12	None	PC ← 0000H
RST 1	CF	12	None	PC ← 0008H
RST 2	D7	12	None	PC ← 0010H
RST 3	DF	12	None	PC ← 0018H
RST 4	E7	12	None	PC ← 0020H
RST 5	EF	12	None	PC ← 0028H
RST 6	F7	12	None	PC ← 0030H

Instruction	Op Code	*T* states	Flags	Main Effect
RST 7	FF	12	None	PC ← 0038H
RZ	C8	12/6	None	PC ← return address if $Z = 1$
SBB A	9F	4	All	A ← A − A − CY
SBB B	98	4	All	A ← A − B − CY
SBB C	99	4	All	A ← A − C − CY
SBB D	9A	4	All	A ← A − D − CY
SBB E	9B	4	All	A ← A − E − CY
SBB H	9C	4	All	A ← A − H − CY
SBB L	9D	4	All	A ← A − L − CY
SBB M	9E	7	All	A ← A − M − CY
SBI byte	DE	7	All	A ← A − byte − CY
SHLD address	22	16	None	M_{adr+1} ← H, M_{adr} ← L
SIM	30	4	None	I ← A
SPHL	F9	6	None	SP ← HL
STA address	32	13	None	M_{adr} ← A
STAX B	02	7	None	M_{BC} ← A
STAX D	12	7	None	M_{DE} ← A
STC	37	4	CY	CY ← 1
SUB A	97	4	All	A ← A − A
SUB B	90	4	All	A ← A − B
SUB C	91	4	All	A ← A − C
SUB D	92	4	All	A ← A − D
SUB E	93	4	All	A ← A − E
SUB H	94	4	All	A ← A − H
SUB L	95	4	All	A ← A − L
SUB M	96	7	All	A ← A − M
SUI byte	D6	7	All	A ← A − byte
XCHG	EB	4	None	HL ↔ DE
XRA A	AF	4	All	A ← A XOR A
XRA B	A8	4	All	A ← A XOR B
XRA C	A9	4	All	A ← A XOR C
XRA D	AA	4	All	A ← A XOR D
XRA E	AB	4	All	A ← A XOR E
XRA H	AC	4	All	A ← A XOR H
XRA L	AD	4	All	A ← A XOR L
XRA M	AE	7	All	A ← A XOR M
XRI byte	EE	7	All	A ← A XOR byte
XTHL	E3	16	None	HL ↔ stack

APPENDIX 6. MEMORY LOCATIONS: POWERS OF 2

Address Bits	Hexadecimal	Decimal	Power of 2
0000 0000 0000 0001	0001H	1	0
0000 0000 0000 0010	0002H	2	1
0000 0000 0000 0100	0004H	4	2
0000 0000 0000 1000	0008H	8	3
0000 0000 0001 0000	0010H	16	4
0000 0000 0010 0000	0020H	32	5
0000 0000 0100 0000	0040H	64	6
0000 0000 1000 0000	0080H	128	7
0000 0001 0000 0000	0100H	256	8
0000 0010 0000 0000	0200H	512	9
0000 0100 0000 0000	0400H	1,024	10
0000 1000 0000 0000	0800H	2,048	11
0001 0000 0000 0000	1000H	4,096	12
0010 0000 0000 0000	2000H	8,192	13
0100 0000 0000 0000	4000H	16,384	14
1000 0000 0000 0000	8000H	32,768	15

APPENDIX 7. MEMORY LOCATIONS: 16K AND 8K INTERVALS

Address Bits	Hexadecimal	Decimal	Zone
Zone bits = $A_{15}A_{14}$			
0000 0000 0000 0000	0000H	0	0
0011 1111 1111 1111	3FFFH	16,383	
0100 0000 0000 0000	4000H	16,384	1
0111 1111 1111 1111	7FFFH	32,767	
1000 0000 0000 0000	8000H	32,768	2
1011 1111 1111 1111	BFFFH	49,151	
1100 0000 0000 0000	C000H	49,152	3
1111 1111 1111 1111	FFFFH	65,535	
Zone bits = $A_{15}A_{14}A_{13}$			
0000 0000 0000 0000	0000H	0	0
0001 1111 1111 1111	1FFFH	8,191	
0010 0000 0000 0000	2000H	8,192	1
0011 1111 1111 1111	3FFFH	16,383	
0100 0000 0000 0000	4000H	16,384	2
0101 1111 1111 1111	5FFFH	24,575	
0110 0000 0000 0000	6000H	24,576	3
0111 1111 1111 1111	7FFFH	32,767	
1000 0000 0000 0000	8000H	32,768	4
1001 1111 1111 1111	9FFFH	40,959	
1010 0000 0000 0000	A000H	40,960	5
1011 1111 1111 1111	BFFFH	49,151	
1100 0000 0000 0000	C000H	49,152	6
1101 1111 1111 1111	DFFFH	57,343	
1110 0000 0000 0000	E000H	57,344	7
1111 1111 1111 1111	FFFFH	65,535	

APPENDIX 8. MEMORY LOCATIONS: 4K INTERVALS

Address Bits	Hexadecimal	Decimal	Zone
Zone bits = $A_{15}A_{14}A_{13}A_{12}$			
0000 0000 0000 0000	0000H	0	0
0000 1111 1111 1111	0FFFH	4,095	
0001 0000 0000 0000	1000H	4,096	1
0001 1111 1111 1111	1FFFH	8,191	
0010 0000 0000 0000	2000H	8,192	2
0010 1111 1111 1111	2FFFH	12,287	
0011 0000 0000 0000	3000H	12,288	3
0011 1111 1111 1111	3FFFH	16,383	
0100 0000 0000 0000	4000H	16,384	4
0100 1111 1111 1111	4FFFH	20,479	
0101 0000 0000 0000	5000H	20,480	5
0101 1111 1111 1111	5FFFH	24,575	
0110 0000 0000 0000	6000H	24,576	6
0110 1111 1111 1111	6FFFH	28,671	
0111 0000 0000 0000	7000H	28,672	7
0111 1111 1111 1111	7FFFH	32,767	
1000 0000 0000 0000	8000H	32,768	8
1000 1111 1111 1111	8FFFH	36,863	
1001 0000 0000 0000	9000H	36,864	9
1001 1111 1111 1111	9FFFH	40,959	
1010 0000 0000 0000	A000H	40,960	10
1010 1111 1111 1111	AFFFH	45,055	
1011 0000 0000 0000	B000H	45,056	11
1011 1111 1111 1111	BFFFH	49,151	
1100 0000 0000 0000	C000H	49,152	12
1100 1111 1111 1111	CFFFH	53,247	
1101 0000 0000 0000	D000H	53,248	13
1101 1111 1111 1111	DFFFH	57,343	
1110 0000 0000 0000	E000H	57,344	14
1110 1111 1111 1111	EFFFH	61,439	
1111 0000 0000 0000	F000H	61,440	15
1111 1111 1111 1111	FFFFH	65,535	

APPENDIX 9. MEMORY LOCATIONS: 2K INTERVALS

Address Bits	Hexadecimal	Decimal	Zone	Address Bits	Hexadecimal	Decimal	Zone
Zone bits = $A_{15}A_{14}A_{13}A_{12}A_{11}$							
0000 0000 0000 0000	0000H	0	0	1000 0000 0000 0000	8000H	32,768	16
0000 0111 1111 1111	07FFH	2,047		1000 0111 1111 1111	87FFH	34,815	
0000 1000 0000 0000	0800H	2,048	1	1000 1000 0000 0000	8800H	34,816	17
0000 1111 1111 1111	0FFFH	4,095		1000 1111 1111 1111	8FFFH	36,863	
0001 0000 0000 0000	1000H	4,096	2	1001 0000 0000 0000	9000H	36,864	18
0001 0111 1111 1111	17FFH	6,143		1001 0111 1111 1111	97FFH	38,911	
0001 1000 0000 0000	1800H	6,144	3	1001 1000 0000 0000	9800H	38,912	19
0001 1111 1111 1111	1FFFH	8,191		1001 1111 1111 1111	9FFFH	40,959	
0010 0000 0000 0000	2000H	8,192	4	1010 0000 0000 0000	A000H	40,960	20
0010 0111 1111 1111	27FFH	10,239		1010 0111 1111 1111	A7FFH	43,007	
0010 1000 0000 0000	2800H	10,240	5	1010 1000 0000 0000	A800H	43,008	21
0010 1111 1111 1111	2FFFH	12,287		1010 1111 1111 1111	AFFFH	45,055	
0011 0000 0000 0000	3000H	12,288	6	1011 0000 0000 0000	B000H	45,056	22
0011 0111 1111 1111	37FFH	14,335		1011 0111 1111 1111	B7FFH	47,103	
0011 1000 0000 0000	3800H	14,336	7	1011 1000 0000 0000	B800H	47,104	23
0011 1111 1111 1111	3FFFH	16,383		1011 1111 1111 1111	BFFFH	49,151	
0100 0000 0000 0000	4000H	16,384	8	1100 0000 0000 0000	C000H	49,152	24
0100 0111 1111 1111	47FFH	18,431		1100 0111 1111 1111	C7FFH	51,199	
0100 1000 0000 0000	4800H	18,432	9	1100 1000 0000 0000	C800H	51,200	25
0100 1111 1111 1111	4FFFH	20,479		1100 1111 1111 1111	CFFFH	53,247	
0101 0000 0000 0000	5000H	20,480	10	1101 0000 0000 0000	D000H	53,248	26
0101 0111 1111 1111	57FFH	22,527		1101 0111 1111 1111	D7FFH	55,295	
0101 1000 0000 0000	5800H	22,538	11	1101 1000 0000 0000	D800H	55,296	27
0101 1111 1111 1111	5FFFH	24,575		1101 1111 1111 1111	DFFFH	57,343	
0110 0000 0000 0000	6000H	24,576	12	1110 0000 0000 0000	E000H	57,344	28
0110 0111 1111 1111	67FFH	26,623		1110 0111 1111 1111	E7FFH	59,391	
0110 1000 0000 0000	6800H	26,624	13	1110 1000 0000 0000	E800H	59,392	29
0110 1111 1111 1111	6FFFH	28,671		1110 1111 1111 1111	EFFFH	61,439	
0111 0000 0000 0000	7000H	28,672	14	1111 0000 0000 0000	F000H	61,440	30
0111 0111 1111 1111	77FFH	30,719		1111 0111 1111 1111	F7FFH	63,487	
0111 1000 0000 0000	7800H	30,720	15	1111 1000 0000 0000	F800H	63,488	31
0111 1111 1111 1111	7FFFH	32,767		1111 1111 1111 1111	FFFFH	65,535	

APPENDIX 10. MEMORY LOCATIONS: 1K INTERVALS

Address Bits	Hexadecimal	Decimal	Zone	Address Bits	Hexadecimal	Decimal	Zone
Zone bits = $A_{15}A_{14}A_{13}A_{12}A_{11}A_{10}$							
0000 0000 0000 0000	0000H	0	0	0101 0000 0000 0000	5000H	20,480	20
0000 0011 1111 1111	03FFH	1,023		0101 0011 1111 1111	53FFH	21,503	
0000 0100 0000 0000	0400H	1,024	1	0101 0100 0000 0000	5400H	21,504	21
0000 0111 1111 1111	07FFH	2,047		0101 0111 1111 1111	57FFH	22,527	
0000 1000 0000 0000	0800H	2,048	2	0101 1000 0000 0000	5800H	22,528	22
0000 1011 1111 1111	0BFFH	3,071		0101 1011 1111 1111	5BFFH	23,551	
0000 1100 0000 0000	0C00H	3,072	3	0101 1100 0000 0000	5C00H	23,552	23
0000 1111 1111 1111	0FFFH	4,095		0101 1111 1111 1111	5FFFH	24,575	
0001 0000 0000 0000	1000H	4,096	4	0110 0000 0000 0000	6000H	24,576	24
0001 0011 1111 1111	13FFH	5,119		0110 0011 1111 1111	63FFH	25,599	
0001 0100 0000 0000	1400H	5,120	5	0110 0100 0000 0000	6400H	25,600	25
0001 0111 1111 1111	17FFH	6,143		0110 0111 1111 1111	67FFH	26,623	
0001 1000 0000 0000	1800H	6,144	6	0110 1000 0000 0000	6800H	26,624	26
0001 1011 1111 1111	1BFFH	7,167		0110 1011 1111 1111	6BFFH	27,647	
0001 1100 0000 0000	1C00H	7,168	7	0110 1100 0000 0000	6C00H	27,648	27
0001 1111 1111 1111	1FFFH	8,191		0110 1111 1111 1111	6FFFH	28,671	
0010 0000 0000 0000	2000H	8,192	8	0111 0000 0000 0000	7000H	28,672	28
0010 0011 1111 1111	23FFH	9,215		0111 0011 1111 1111	73FFH	29,695	
0010 0100 0000 0000	2400H	9,216	9	0111 0100 0000 0000	7400H	29,696	29
0010 0111 1111 1111	27FFH	10,239		0111 0111 1111 1111	77FFH	30,719	
0010 1000 0000 0000	2800H	10,240	10	0111 1000 0000 0000	7800H	30,720	30
0010 1011 1111 1111	2BFFH	11,263		0111 1011 1111 1111	7BFFH	31,743	
0010 1100 0000 0000	2C00H	11,264	11	0111 1100 0000 0000	7C00H	31,744	31
0010 1111 1111 1111	2FFFH	12,287		0111 1111 1111 1111	7FFFH	32,767	
0011 0000 0000 0000	3000H	12,288	12	1000 0000 0000 0000	8000H	32,768	32
0011 0011 1111 1111	33FFH	13,311		1000 0011 1111 1111	83FFH	33,791	
0011 0100 0000 0000	3400H	13,312	13	1000 0100 0000 0000	8400H	33,792	33
0011 0111 1111 1111	37FFH	14,335		1000 0111 1111 1111	87FFH	34,815	
0011 1000 0000 0000	3800H	14,336	14	1000 1000 0000 0000	8800H	34,816	34
0011 1011 1111 1111	3BFFH	15,359		1000 1011 1111 1111	8BFFH	35,839	
0011 1100 0000 0000	3C00H	15,360	15	1000 1100 0000 0000	8C00H	35,840	35
0011 1111 1111 1111	3FFFH	16,383		1000 1111 1111 1111	8FFFH	36,863	
0100 0000 0000 0000	4000H	16,384	16	1001 0000 0000 0000	9000H	36,864	36
0100 0011 1111 1111	43FFH	17,407		1001 0011 1111 1111	93FFH	37,887	
0100 0100 0000 0000	4400H	17,408	17	1001 0100 0000 0000	9400H	37,888	37
0100 0111 1111 1111	47FFH	18,431		1001 0111 1111 1111	97FFH	38,911	
0100 1000 0000 0000	4800H	18,432	18	1001 1000 0000 0000	9800H	38,912	38
0100 1011 1111 1111	4BFFH	19,455		1001 1011 1111 1111	9BFFH	39,935	
0100 1100 0000 0000	4C00H	19,456	19	1001 1100 0000 0000	9C00H	39,936	39
0100 1111 1111 1111	4FFFH	20,479		1001 1111 1111 1111	9FFFH	40,959	

APPENDIX 10. MEMORY LOCATIONS: 1K INTERVALS (*Continued*)

Address Bits	Hexadecimal	Decimal	Zone	Address Bits	Hexadecimal	Decimal	Zone
Zone bits = $A_{15}A_{14}A_{13}A_{12}A_{11}A_{10}$							
1010 0000 0000 0000	A000H	40,960	40	1101 0000 0000 0000	D000H	53,248	52
1010 0011 1111 1111	A3FFH	41,983		1101 0011 1111 1111	D3FFH	54,271	
1010 0100 0000 0000	A400H	41,984	41	1101 0100 0000 0000	D400H	54,272	53
1010 0111 1111 1111	A7FFH	43,007		1101 0111 1111 1111	D7FFH	55,295	
1010 1000 0000 0000	A800H	43,008	42	1101 1000 0000 0000	D800H	55,296	54
1010 1011 1111 1111	ABFFH	44,031		1101 1011 1111 1111	DBFFH	56,319	
1010 1100 0000 0000	AC00H	44,032	43	1101 1100 0000 0000	DC00H	56,320	55
1010 1111 1111 1111	AFFFH	45,055		1101 1111 1111 1111	DFFFH	57,343	
1011 0000 0000 0000	B000H	45,056	44	1110 0000 0000 0000	E000H	57,344	56
1011 0011 1111 1111	B3FFH	46,079		1110 0011 1111 1111	E3FFH	58,367	
1011 0100 0000 0000	B400H	46,080	45	1110 0100 0000 0000	E400H	58,368	57
1011 0111 1111 1111	B7FFH	47,103		1110 0111 1111 1111	E7FFH	59,391	
1011 1000 0000 0000	B800H	47,104	46	1110 1000 0000 0000	E800H	59,392	58
1011 1011 1111 1111	BBFFH	48,127		1110 1011 1111 1111	EBFFH	60,415	
1011 1100 0000 0000	BC00H	48,128	47	1110 1100 0000 0000	EC00H	60,416	59
1011 1111 1111 1111	BFFFH	49,151		1110 1111 1111 1111	EFFFH	61,439	
1100 0000 0000 0000	C000H	49,152	48	1111 0000 0000 0000	F000H	61,440	60
1100 0011 1111 1111	C3FFH	50,175		1111 0011 1111 1111	F3FFH	62,463	
1100 0100 0000 0000	C400H	50,176	49	1111 0100 0000 0000	F400H	62,464	61
1100 0111 1111 1111	C7FFH	51,199		1111 0111 1111 1111	F7FFH	63,487	
1100 1000 0000 0000	C800H	51,200	50	1111 1000 0000 0000	F800H	63,488	62
1100 1011 1111 1111	CBFFH	52,223		1111 1011 1111 1111	FBFFH	64,511	
1100 1100 0000 0000	CC00H	52,224	51	1111 1100 0000 0000	FC00H	64,512	63
1100 1111 1111 1111	CFFFH	53,247		1111 1111 1111 1111	FFFFH	65,535	

Answers to Odd-Numbered Problems

CHAP. 1. 1-1. a. 1 b. 2 c. 2½ 1-3. a. 10 b. 2 c. 5 d. 16 1-5. 1,024, 4,096, 8K 1-7. 1010 1100, 172 1-9. 201 1-11. 1100 0111, 199 1-13. 111000 1-15. 1001 0110 1-17. F52B, F52C, F52D, F52E, F52F, F530 1-19. a. 1111 1111 b. 1010 1011 1100 c. 1100 1101 0100 0010 d. 1111 0011 0010 1001 1-21. 0011 1110, 0000 1110, 1101 0011, 0010 0000, 0111 0110 1-23. a. 4,095 b. 16,383 c. 32,740 d. 46,040 1-25. 16,384, 16K 1-27. 0000, FFFF 1-29. a. EE b. 1D7B c. 3BFF d. B8B5 1-31. a. 87 b. 9,043 c. 597,266 1-33. 100 1100, 100 1001, 101 0011, 101 0100

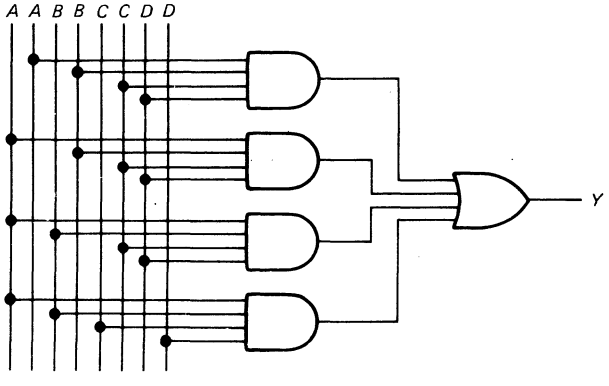
CHAP. 2. 2-1. One or more, one 2-3. Noninverter 2-5. 64, 000000 2-7. 3, 9, C, F 2-9. 128, 1111111 2-11. 0, 59 2-13. $Y = \overline{A + B}$, low 2-15. 8 2-17. 0, $Y = \overline{A} + \overline{B} + \overline{C}$, 000 to 110, 111 2-19. $Y = \overline{ABC}$, 0 2-21. $Y = \overline{AB + CD}$, 16, 0000, 0001, 0010, 0100, 0101, 0110, 1000, 1001, 1010 2-23. a. 0000 b. 0001 c. *JIM* d. *OPR* 2-25. a. Positive b. Negative c. Positive d. Negative

CHAP. 3. 3-1. High; low; inverter 3-3. None, Z_5, Z_6 3-5. Q is 1, $\overline{Q}$ is 0 3-7. Change the output NOR gate of Fig. 3-28*a* to a bubbled AND gate; all bubbles cancel leaving the simplified circuit of Fig. 3-28*b*. 3-9. 0, 1 3-11. 512 3-13. 16; 0, 1, 1, 0 3-15. 1, 0, inverter 3-17. a. None b. Z_7 c. Z_2 d. X_2 and Y_2 3-19. 0, 1 3-21. 512 3-23. Low, high 3-25. a. 0 b. 1 c. 1 d. 1 3-27. a. 11010 b. 01001 c. 11111 d. 10010 3-29. Remove the inverter 3-31. a. *CARRY* = 0, *SUM* = 0 b. 0, 1 c. 0, 1 d. 1, 0 3-33. a. 0011 1100 b. 0101 0000 1100 c. 0001 1110 0101 1100 d. 1111 0000 1101 0010

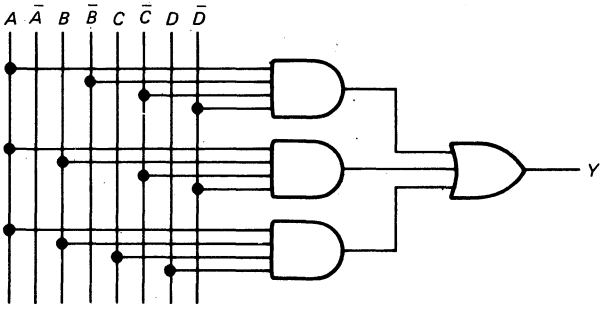
CHAP. 4. 4-1. 1.075 mA, 1.387 mA 4-3. 5 4-5. All; b, c, f, g

CHAP. 5. 5-1. $\overline{A}\overline{B}C\overline{D}$, $AB\overline{C}D$, $ABC\overline{D}$

5-3.



5-5.

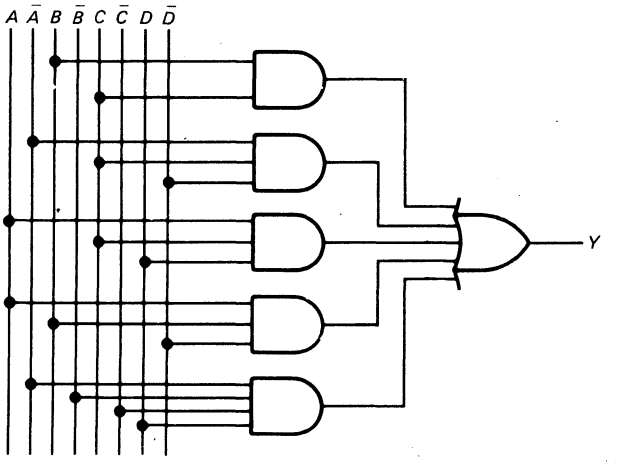


5-7

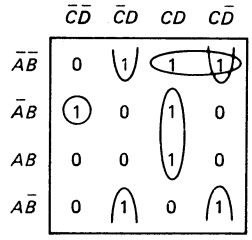
	$\overline{C}\overline{D}$	$\overline{C}D$	CD	$C\overline{D}$
$\overline{A}\overline{B}$	0	0	0	0
$\overline{A}B$	0	0	0	0
AB	1	1	1	1
$A\overline{B}$	1	1	1	1

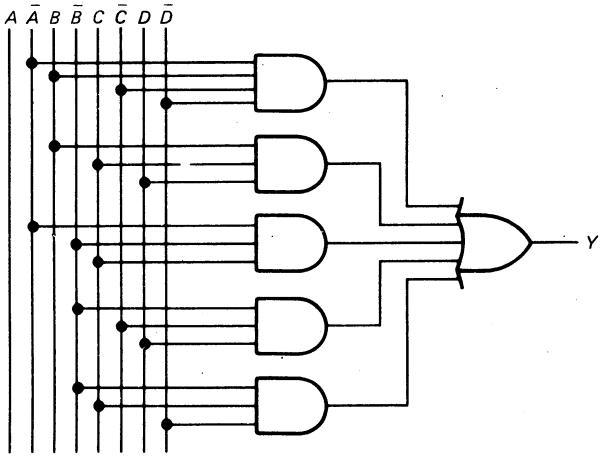
5-9.

	$\overline{C}\overline{D}$	$\overline{C}D$	CD	$C\overline{D}$
$\overline{A}\overline{B}$	0	1	0	1
$\overline{A}B$	0	0	1	1
AB	1	0	1	1
$A\overline{B}$	0	0	1	0

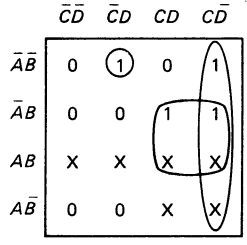


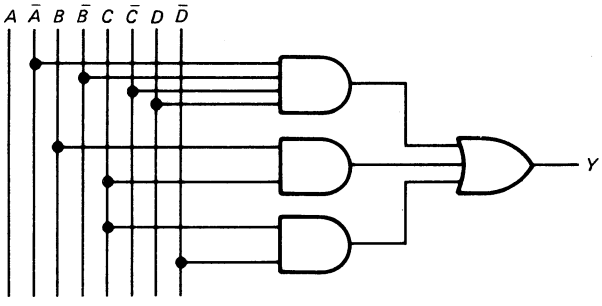
5-11.



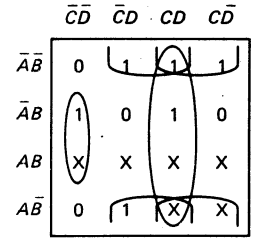


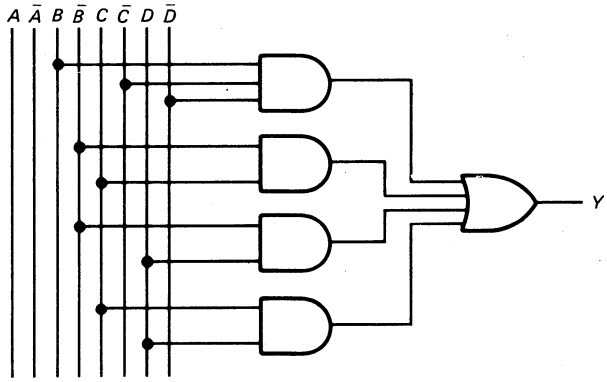
5-13.





5-15.





CHAP. 6. 6-1. a. 0001 1000, 18H b. 0010 0100, 24H c. 0010 1010, 2AH d. 0110 0011, 63H 6-3. a. 7BH b. 78H c. A8H d. D1H 6-5. a. +30 b. −7 c. −28 d. +49 6-7. a. F9H b. 01H c. 03H d. 1FH 6-9. a. 1110 1101, EDH b. 1101 0000, D0H c. 0010 0101, 25H d. 1101 1111, DFH 6-11. 9BH, DDH

CHAP. 7. 7-1. a. C b. G 7-3. a. 0000 b. 1001 7-5. 3 MHz; the output frequency is half the input frequency 7-7. $Q = 0, Y = 1; Q = 1, Y = CLK$

CHAP. 8. 8-1. a. 0001 0111 b. 1000 1101 8-3. 385 Ω 8-5. 4 μs 8-7. 6.4 μs 8-9. 65,535 8-11. 1 μs, 6 μs 8-13. 1.6 μs, 0.2 μs 8-15. Two answers: 7490 (divide by 10) and 7492 (divide by 6), or 7490 (divide by 5) and 7492 (divide by 12) 8-17. 136 8-19. a. 0, 1 b. 1, 1 c. 0

CHAP. 9. 9-1. 16,384 9-3. 12

9-5.

Address	Data
DDDD	UDDD UDDU
DDDU	DUUU UUDD
DDUD	DDUU DUUD
DDUU	DDUD DDUU
DUDD	DDDU DUUU
DUDU	DUDU UUUU
DUUD	UUUD UUDU
DUUU	UUUU UDDD

9-7. 63 9-9. BFFFH; 49,151 9-11. a. 47, 212, 207, 110, 83, 122 b. 36,357

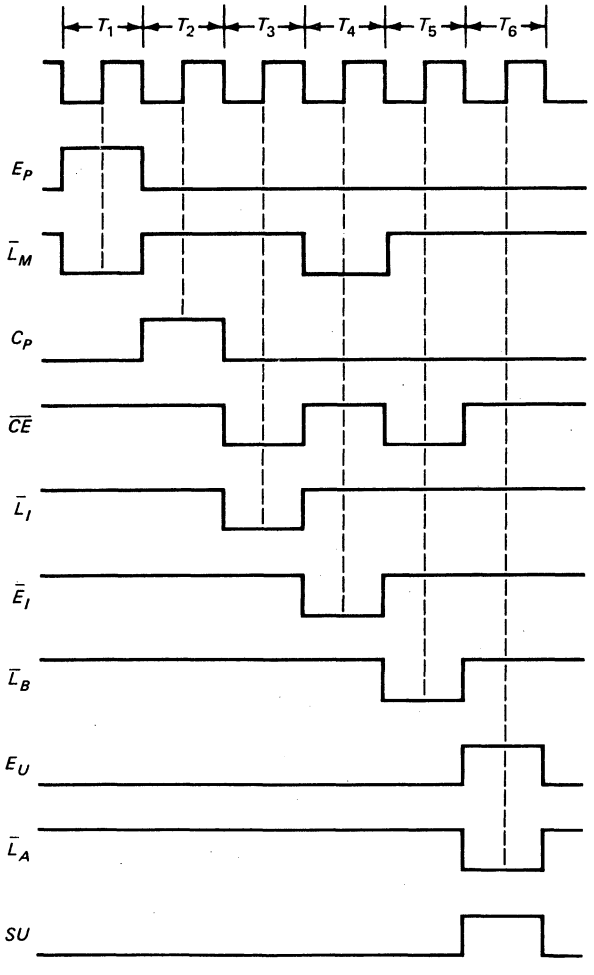
CHAP. 10. 10-1.

Address	Mnemonic
0H	LDA DH
1H	ADD EH
2H	SUB FH
3H	OUT
4H	HLT
DH	05H
EH	04H
FH	06H

10-3.

Address	Mnemonic
0H	LDA BH
1H	ADD CH
2H	SUB DH
3H	ADD EH
4H	SUB FH
5H	HLT
BH	08H
CH	04H
DH	03H
EH	05H
FH	02H

10-5.



10-7. LDA: 1A3H or 0001 1010 0011, 2C3H or 0010 1100 0011, 3E3H or 0011 1110 0011; SUB: 1A3H or 0001 1010 0011, 2E1H or 0010 1110 0001, 3CFH or 0011 1100 1111 10-9. a. Negative edge; *CLK* is on its rising edge b. High c. Low d. High 10-11. a. Low b. Low c. High

CHAP. 11. 11-1.

```
Mnemonic
MVI A,64H
MVI B,96H
MVI C,C8H
HLT
```

11-3.

```
Mnemonic
MVI A,32H
STA 4000H
MVI A,33H
STA 4001H
MVI A,34H
STA 4002H
HLT
```

11-5.

```
Mnemonic
MVI A,44H
MVI B,22H
ADD B
STA 5000H
HLT
```

11-7. a. 120 b. 119 c. Change the first instruction to MVI C,D2H

11-9.

```
Mnemonic
MVI A,00H
MVI B,19H
MVI C,07H
CALL F006H
STA 20FFH
HLT
```

11-11.

```
Label    Mnemonic
         IN 01H
         ANI 01H
         JNZ ODD
         MVI A,45H
         JMP DONE
ODD:     MVI A,4FH
DONE:    MVI C,08H
AGAIN:   OUT 04H
         RAR
         DCR C
         JNZ AGAIN
         HLT
```

11-13.

Address	Contents
2000H	DBH
2001H	02H
2002H	E6H
2003H	01H
2004H	CAH
2005H	00H
2006H	20H
2007H	DBH
2008H	01H
2009H	32H
200AH	00H
200BH	40H
200CH	76H

11-15.

Address	Contents
2000H	0EH
2001H	23H
2002H	0DH
2003H	C2H
2004H	02H
2005H	20H
2006H	C9H

11-17.

Label	Mnemonic
	MVI A,05H
LOOP:	CALL F020H
	DCR A
	JNZ LOOP
	RET

Address	Contents
E100H	3EH
E101H	05H
E102H	CDH
E103H	20H
E104H	F0H
E105H	3DH
E106H	C2H
E107H	02H
E108H	E1H
E109H	C9H

11-19.

Address	Contents
F080H	3EH
F081H	06H
F082H	32H
F083H	93H
F084H	F0H
F085H	CDH
F086H	60H
F087H	F0H
F088H	3AH
F089H	93H
F08AH	F0H
F08BH	3DH
F08CH	32H
F08DH	93H
F08EH	F0H
F08FH	C2H
F090H	85H
F091H	F0H
F092H	C9H

11-21.

Address	Contents
2000H	D3H
2001H	04H
2002H	0EH
2003H	42H
2004H	0DH
2005H	C2H
2006H	04H
2007H	20H
2008H	2FH
2009H	00H
200AH	C3H
200BH	00H
200CH	20H

CHAP. 12. 12-1.

Mnemonic
MVI A,00H
MVI B,01H
MVI C,59H
MVI D,02H
MVI E,F1H
ADD C
ADD E
MOV L,A
MVI A,00H
ADC B
ADD D
MOV H,A
HLT

An alternative solution is

Mnemonic
MVI A,F1H
ADI 59H
MOV L,A
MVI A,02H
ACI 01H
MOV H,A
HLT

12-3.

Label	Mnemonic
	LXI H,4FFFH
LOOP:	INX H
	MOV B,M
	MOV A,H

```
        ADI 40H
        MOV H,A
        MOV M,B
        SUI 40H
        MOV H,A
        CPI 53H
        JNZ LOOP
        MOV A,L
        CPI FFH
        JNZ LOOP
        HLT
```

12-5.

Label	Mnemonic
	LXI SP,E000H
	MVI A,00H
	MVI B,FFH
LOOP:	INR A
	OUT 22H
	CALL F010H
	DCR B
	JNZ LOOP
	HLT

12-7.

Label	Mnemonic
	LXI SP,E000H
	LXI H,5FFFH
LOOP:	INX H
	MOV A,M
	OUT 22H
	CALL F020H
	MOV A,H
	CPI 61H
	JNZ LOOP
	MOV A,L
	CPI FFH
	JNZ LOOP
	HLT

12-9.

Label	Mnemonic
	LXI SP,E000H
	LXI H,4FFFH
LOOP:	INX H
	MOV A,M
	MOV B,08H
AGAIN:	OUT 22H
	CALL F010H
	RAR
	DCR B
	JNZ AGAIN
	MOV A,L
	CPI FFH
	JNZ LOOP
	HLT

CHAP. 13. 13-1. 6.25 MHz 13-3. a. LHLD 2000H b. PCHL c. XTHL 13-5. a. High b. Low c. High d. Low 13-7. 8000H–80FFH 13-9. 2002H 13-11. a. JMP 3090H b. 4 c. 30H d. 90H

CHAP. 14. 14-1. LXI H,8000H

14-3.

Label	Mnemonic
	LXI H,5000H
	MVI C,00H
LOOP:	MVI A,01H
	OUT 10H
WAIT:	IN 11H
	ANI 02H
	JZ WAIT
	IN 12H
	MOV M,A
	INX H
	MVI A,00H
	OUT 10H
	DCR C
	JNZ LOOP
	HLT

14-5.

Label	Mnemonic
	LXI H,8000H
LOOP:	MOV A,M
	OUT 12H
	MVI A,40H
	OUT 10H
WAIT:	IN 11H
	ANI 80H
	JZ WAIT
	INX H
	MVI A,00H
	OUT 10H
	MOV A,H
	CPI 83H
	JNZ LOOP
	MOV A,L
	CPI FFH
	JNZ LOOP
	HLT

14-7. a. 6.5 b. None c. 7.5 d. None 14-9. 7.5 and 5.5 14-11. a. High b. 7.5 c. High d. 6.5 14-13. a. 40H b. 60H c. 43H, letter C d. First

CHAP. 15. 15-1. a. High b. High c. Yes 15-3. Enable A and B interrupts, port C output, port B output, and port A input. 15-5. Port C is an input port 15-7. 1579, single pulse 15-9. 8000H–80FFH, 80H to 85H 15-11. The instructions program port 01H as follows: bits 7, 6, and 1 become inputs; bits 5, 4, 3, 2, and 0 become outputs.

15-13. 8, FF00H–FFFFH 15-15. Add inverter to each zone bit except A_{14}, so that the gate output is $\overline{A}_{15}A_{14}\overline{A}_{13}\overline{A}_{12}\overline{A}_{11}\overline{A}_{10}\overline{A}_9\overline{A}_8$. 15-17. Add inverters to zone bits A_{15}, A_{14}, A_{13}, A_{10}, and A_8 to produce gate output of $\overline{A}_{15}\overline{A}_{14}A_{13}A_{12}\overline{A}_{11}\overline{A}_{10}A_9\overline{A}_8$. 15-19. 4, 15, C000H–CFFFH 15-21. Remove inverters on all address bits

CHAP. 16. 16-1. 1.5 mA, 1.5 mA, 3 V 16-3. 1,023, 0.0978% 16-5. 4.688 mA 16-7. Triangular wave from level 0 to 1.992 V. 16-9. 1100 0000 16-11. 606 kHz 16-13. 0011 1111 16-15. 31 *T* states, approximately 96 (decimal) 16-17. a. 0 to 5 V b. 0011 0011 c. Approximately 5 V 16-19. a. FFH b. 00H c. Approximately 81°C

Index