

STRUCTURED COMPUTER ORGANIZATION

SECOND EDITION

ANDREW S. TANENBAUM

*Vrije Universiteit
Amsterdam, The Netherlands*

PRENTICE-HALL, INC.

ENGLEWOOD CLIFFS, NEW JERSEY 07632

Library of Congress Cataloging in Publication Data

Tanenbaum, Andrew, S. (date)
Structured Computer Organization

Includes index.

1. Electronic digital computers—Programming.

I. Title

QA76.6.T38 1984 001.64'2 83-2916
ISBN 0-13-854489-1

To Suzanne, Barbara, Marvin and the memory of Sweetie π

Production/editorial supervision: Nancy Milnamow

Manufacturing buyer: Gordon Osbourne

© 1984 by Prentice-Hall, Inc., Englewood Cliffs, N. J. 07632

All rights reserved. No part of this book
may be reproduced in any form or by any means
without permission in writing from the publisher.

10 9 8 7 6 5 4 3

Printed in the United States of America

ISBN 0-13-854489-1

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA PTY. LTD., *Sydney*
EDITORIA PRENTICE-HALL DO BRASIL, LTDA., *Rio de Janeiro*
PRENTICE-HALL OF CANADA, LTD., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LTD., *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
PRENTICE-HALL OF SOUTHEAST ASIA PTE. LTD., *Singapore*
WHITEHALL BOOKS LTD., *Wellington, New Zealand*

CONTENTS

PREFACE	x
----------------	----------

1 INTRODUCTION	1
-----------------------	----------

- 1.1. LANGUAGES, LEVELS, AND VIRTUAL MACHINES 3
- 1.2. CONTEMPORARY MULTILEVEL MACHINES 4
- 1.3. HISTORICAL EVOLUTION OF MULTILEVEL MACHINES 8
- 1.4. HARDWARE, SOFTWARE, AND MULTILEVEL MACHINES 10
- 1.5. PROCESSES 13
- 1.6. OUTLINE OF THIS BOOK 15

2 COMPUTER SYSTEMS ORGANIZATION	19
--	-----------

- 2.1. PROCESSORS 19
 - 2.1.1 Instruction Execution 20
 - 2.1.2 Parallel Instruction Execution 23
 - 2.1.3 Processor Classification 25
- 2.2. MEMORY 26
 - 2.2.1 Bits 26
 - 2.2.2 Memory Addresses 27
 - 2.2.3 Metabits 29
 - 2.2.4 Secondary Memory 30

- 2.3. INPUT/OUTPUT 34
 - 2.3.1 I/O Devices 35
 - 2.3.2 I/O Processors 35
 - 2.3.3 Character Codes 36
 - 2.3.4 Error-Correcting Codes 36
 - 2.3.5 Frequency-Dependent Codes 40
- 2.4. COMPUTER NETWORKS AND DISTRIBUTED SYSTEMS 43
 - 2.4.1 Long-Haul Networks 43
 - 2.4.2 Telecommunication 46
 - 2.4.3 Local Networks 51
 - 2.4.4 Distributed Systems 52
- 2.5. SUMMARY 53

3 THE DIGITAL LOGIC LEVEL

58

- 3.1. GATES AND BOOLEAN ALGEBRA 58
 - 3.1.1 Gates 59
 - 3.1.2 Boolean Algebra 61
 - 3.1.3 Implementation of Boolean Functions 62
 - 3.1.4 Circuit Equivalence 64
- 3.2. BASIC DIGITAL LOGIC CIRCUITS 68
 - 3.2.1 Integrated Circuits 69
 - 3.2.2 Combinational Circuits 70
 - 3.2.3 Arithmetic Circuits 76
 - 3.2.4 Clocks 80
- 3.3. MEMORY 83
 - 3.3.1 Latches 83
 - 3.3.2 Flip-Flops and Registers 86
 - 3.3.3 Memory Organization 87
 - 3.3.4 Memory Properties 91
- 3.4. MICROPROCESSORS AND MICROCOMPUTERS 94
 - 3.4.1 Microprocessor Chips 94
 - 3.4.2 Microcomputer Buses 95
 - 3.4.3 The Z80 Microprocessor 97
 - 3.4.4 The 68000 Microprocessor
- 3.5. INTERFACING 105
 - 3.5.1 I/O Chips 105
 - 3.5.2 Address Decoding 107
 - 3.5.3 An Example Microcomputer 110
- 3.6. SUMMARY 112

4 THE MICROPROGRAMMING LEVEL

117

- 4.1. REVIEW OF THE DIGITAL LOGIC LEVEL 118
 - 4.1.1 Registers 118
 - 4.1.2 Buses 119
 - 4.1.3 Multiplexers and Decoders 121
 - 4.1.4 ALUs and Shifters 122
 - 4.1.5 Clocks 123
 - 4.1.6 Main Memory 124
 - 4.1.7 Component Packaging 124
- 4.2. AN EXAMPLE MICROARCHITECTURE 126
 - 4.2.1 The Data Path 126
 - 4.2.2 Microinstructions 128
 - 4.2.3 Microinstruction Timing 129
 - 4.2.4 Microinstruction Sequencing 133
- 4.3. AN EXAMPLE MACROARCHITECTURE 134
 - 4.3.1 Stacks 135
 - 4.3.2 The Macroinstruction Set 140
- 4.4. AN EXAMPLE MICROPROGRAM 142
 - 4.4.1 The Micro Assembly Language 142
 - 4.4.2 The Example Microprogram 143
 - 4.4.3 Remarks about the Microprogram 148
 - 4.4.4 Perspective 148
- 4.5. DESIGN OF THE MICROPROGRAMMING LEVEL 149
 - 4.5.1 Horizontal versus Vertical Microprogramming 149
 - 4.5.2 Nanoprogramming 156
 - 4.5.3 Improving Performance 158
- 4.6. THE IBM 370/125 MICROPROGRAMMING LEVEL 162
 - 4.6.1 The IBM 370/125 Microarchitecture 163
 - 4.6.2 IBM 3125 Microinstructions 166
- 4.7. THE PDP-11/60 MICROPROGRAMMING LEVEL 169
 - 4.7.1 The PDP-11/60 Microarchitecture 169
 - 4.7.2 The PDP-11/60 Microinstructions 172
- 4.8. SUMMARY 177

5 THE CONVENTIONAL MACHINE LEVEL

181

- 5.1. EXAMPLES OF THE CONVENTIONAL MACHINE LEVEL 181
 - 5.1.1 IBM System/370 182
 - 5.1.2 DEC PDP-11 185
 - 5.1.3 Motorola MC68000 190
 - 5.1.4 Zilog Z80 196

- 5.2. INSTRUCTION FORMATS 200
 - 5.2.1 Design Criteria for Instruction Formats 202
 - 5.2.2 Expanding Opcodes 203
 - 5.2.3 Examples of Instruction Formats 204
- 5.3. ADDRESSING 213
 - 5.3.1 Immediate Addressing 213
 - 5.3.2 Direct Addressing 214
 - 5.3.3 Register Addressing 214
 - 5.3.4 Indirect Addressing 215
 - 5.3.5 Indexing 217
 - 5.3.6 Base-Register Addressing 218
 - 5.3.7 Stack Addressing 220
 - 5.3.8 Addressing on the PDP-11 and the 68000 227
 - 5.3.9 Discussion of Addressing Modes 231
- 5.4. INSTRUCTION TYPES 232
 - 5.4.1 Data Movement Instructions 233
 - 5.4.2 Dyadic Operations 235
 - 5.4.3 Monadic Operations 236
 - 5.4.4 Comparisons and Conditional Jumps 238
 - 5.4.5 Procedure Call Instructions 240
 - 5.4.6 Loop Control 241
 - 5.4.7 Input/Output 243
- 5.5. FLOW OF CONTROL 246
 - 5.5.1 Sequential Flow of Control and Jumps 247
 - 5.5.2 Procedures 248
 - 5.5.3 Coroutines 255
 - 5.5.4 Traps 263
 - 5.5.5 Interrupts 263
- 5.7. SUMMARY 269

6 THE OPERATING SYSTEM MACHINE LEVEL

274

- 6.1. IMPLEMENTATION OF THE OPERATING SYSTEM MACHINE LEVEL 274
- 6.2. VIRTUAL I/O INSTRUCTIONS 277
 - 6.2.1 Sequential Files 277
 - 6.2.2 Random Access Files 279
 - 6.2.3 Implementation of Virtual I/O Instructions 280
 - 6.2.4 Directory Management Instructions 285
 - 6.2.5 IBM 370 Virtual I/O 286
 - 6.2.6 UNIX Virtual I/O 290
 - 6.2.7 CP/M Virtual I/O 295
- 6.3. VIRTUAL INSTRUCTIONS USED IN PARALLEL PROCESSING 298
 - 6.3.1 Process Creation 299
 - 6.3.2 Race Conditions 301
 - 6.3.3 Process Synchronization Using Semaphores 304

- 6.4. VIRTUAL MEMORY 308
 - 6.4.1 Paging 308
 - 6.4.2 Implementation of Paging 311
 - 6.4.3 Demand Paging and the Working Set Model 315
 - 6.4.4 Page Replacement Policy 318
 - 6.4.5 Page Size and Fragmentation 320
 - 6.4.6 Cache Memory 321
 - 6.4.7 Segmentation 322
 - 6.4.8 The MULTICS Virtual Memory 327
 - 6.4.9 Virtual Memory on the IBM 370 331
 - 6.4.10 Virtual Memory on the PDP-11 333
 - 6.4.11 Virtual Memory on the 68000 338
- 6.5. JOB CONTROL LANGUAGES 341
- 6.6. SUMMARY 344

7 THE ASSEMBLY LANGUAGE LEVEL

350

- 7.1. INTRODUCTION TO ASSEMBLY LANGUAGE 351
 - 7.1.1 What Is an Assembly Language? 351
 - 7.1.2 Format of an Assembly Language Statement 352
 - 7.1.3 Comparison of Assembly Language and Problem-Oriented Languages 354
 - 7.1.4 Program Tuning 355
- 7.2. THE ASSEMBLY PROCESS 357
 - 7.2.1 Two-Pass Assemblers 358
 - 7.2.2 Pass One 359
 - 7.2.3 Pass Two 363
 - 7.2.4 Symbol Table 365
- 7.3. MACROS 367
 - 7.3.1 Macro Definition, Call, and Expansion 367
 - 7.3.2 Macros with Parameters 369
 - 7.3.3 Implementation of a Macro Facility in an Assembler 370
- 7.4. LINKING AND LOADING 371
 - 7.4.1 Tasks Performed by the Linker 372
 - 7.4.2 Structure of an Object Module 375
 - 7.4.3 Binding Time and Dynamic Relocation 377
 - 7.4.4 Dynamic Linking 380
- 7.5. SUMMARY 382

8 MULTILEVEL MACHINES 385

- 8.1. METHODS OF IMPLEMENTING NEW LEVELS 385
 - 8.1.1 Interpretation 385
 - 8.1.2 Translation 386
 - 8.1.3 Procedural Extension 390
- 8.2. DESIGN STRATEGIES FOR MULTILEVEL MACHINES 391
 - 8.2.1 Top-Down Design 391
 - 8.2.2 Bottom-Up Design 393
 - 8.2.3 Middle-Out Design 395
- 8.3. PROGRAM PORTABILITY 396
 - 8.3.1 A Universal Programming Language 397
 - 8.3.2 The Brute Force Approach 398
 - 8.3.3 UNCOL 399
 - 8.3.4 Abstract Machine Language 402
 - 8.3.5 Portable Compilers 404
 - 8.3.6 Emulation 405
- 8.4. SELF-VIRTUALIZING MACHINES 406
 - 8.4.1 IBM VM/370 System 408
 - 8.4.2 Goals of Self-Virtualizing Machines 408
 - 8.4.3 Implementation of a Self-Virtualizing Machine 411
- 8.5. THE COMPILER-INTERPRETER INTERFACE 418
 - 8.5.1 High-Level Interfaces 419
 - 8.5.2 Discussion of High-Level Interfaces 420
- 8.6. SUMMARY 423

9 READING LIST AND BIBLIOGRAPHY 426

- 9.1. SUGGESTIONS FOR FURTHER READING 426
 - 9.1.1 Introduction and General Works 426
 - 9.1.2 Computer Systems Organization 427
 - 9.1.3 The Digital Logic Level 428
 - 9.1.4 The Microprogramming Level 429
 - 9.1.5 The Conventional Machine Level 429
 - 9.1.6 The Operating System Machine Level 430
 - 9.1.7 The Assembly Language Level 431
 - 9.1.8 Multilevel Machines 432
 - 9.1.9 Binary Numbers and Arithmetic 433
- 9.2. ALPHABETICAL BIBLIOGRAPHY 433

APPENDIXES

A	BINARY NUMBERS	439
A.1.	FINITE PRECISION NUMBERS	439
A.2.	RADIX NUMBER SYSTEMS	441
A.3.	CONVERSION FROM ONE RADIX TO ANOTHER	442
A.4.	NEGATIVE BINARY NUMBERS	445
A.5.	BINARY ARITHMETIC	448
B	FLOATING-POINT NUMBERS	451
	INDEX	457

3

THE DIGITAL LOGIC LEVEL

At the bottom of the hierarchy of Fig. 1-2 we find the digital logic level, the computer's real hardware. In this chapter, we will examine many aspects of digital logic, as a building block for the study of higher levels in subsequent chapters. Our study will emphasize microcomputers over larger machines not only because they are simpler (and thus easier to study) but also because they are becoming increasingly important compared to larger machines.

The basic elements from which all digital computers are constructed are amazingly simple. We will begin our study by looking at these basic elements and also at the special two-valued algebra (Boolean algebra) used to analyze them. Next we will examine some fundamental circuits that can be built using gates in simple combinations, including circuits for doing arithmetic. The following topic is how gates can be combined to store information, that is, how memories are organized. After that, we come to the subject of CPUs and especially how single chip CPUs interface with memory and peripheral devices.

3.1. GATES AND BOOLEAN ALGEBRA

Digital circuits can be constructed from a small number of primitive elements by combining them in innumerable ways. In the following sections we will describe these primitive elements, show how they can be combined, and introduce a powerful tool for analyzing their behavior.

3.1.1. Gates

A digital circuit is one in which only two logical values are present. Typically, a signal between 0 and 1 volt represents one value (e.g., binary 0) and a signal between 2 and 5 volts represents the other value (e.g., binary 1). Voltages outside these two ranges are not permitted. Tiny electronic devices, called **gates**, can compute various functions of these two-valued signals. These gates form the hardware basis on which all digital computers are built.

Although the details of how gates work inside is really beyond the scope of this book, belonging to a level below our level 0, the **device level**, we will now digress ever so briefly to take a quick look at the basic idea, which is really quite simple. All modern digital logic ultimately rests on the fact that a transistor can be made to operate as a very fast binary switch. In Fig. 3-1(a) we have shown a single bipolar transistor (the circle) embedded in a simple circuit. This transistor has three connections to the outside world: the **collector**, the **base**, and the **emitter**. When the input voltage, V_{in} is below a certain critical value, the transistor turns off and acts like an infinite resistance, causing the output of the circuit, V_{out} , to take on a value close to V_{cc} . When V_{in} exceeds the critical value, the transistor switches on and acts like a perfect conductor, causing V_{out} to be pulled down to ground (by convention, 0 volts).

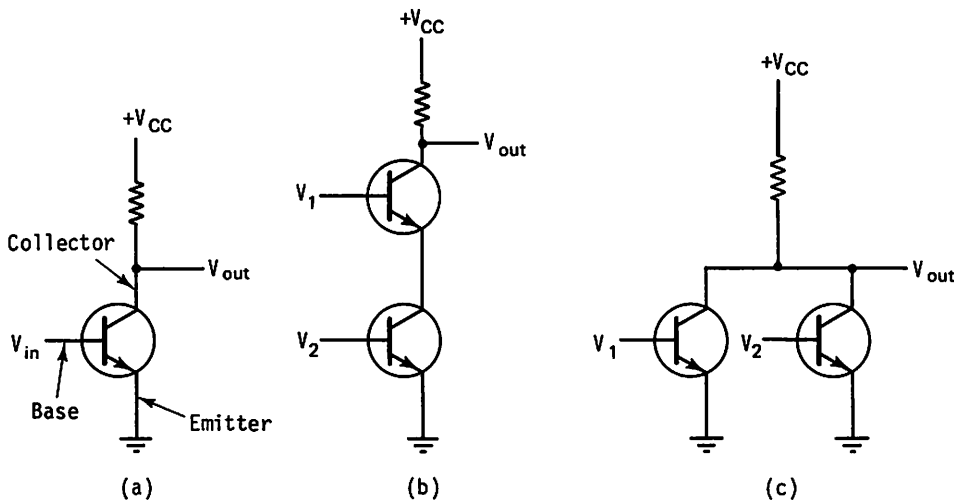


Fig. 3-1. (a) A transistor inverter. (b) A NAND gate. (c) A NOR gate.

The important thing to notice is that when V_{in} is low, V_{out} is high, and vice versa. This circuit is thus an inverter, converting a logical 0 to a logical 1, and a logical 1 to a logical 0. The resistor is needed to limit the amount of current drawn by the transistor. The time required to switch from one state to the other is typically a few nanoseconds.

In Fig. 3-1(b) two transistors are cascaded in series. If both V_1 and V_2 are high, both transistors will conduct and V_{out} will be pulled low. If either input is low, the corresponding transistor will turn off, and the output will be high. In other words, V_{out} will be low if and only if both V_1 and V_2 are high.

In Fig. 3-1(c) the two transistors are wired in parallel instead of in series. In this configuration, if either input is high, the corresponding transistor will turn on and pull the output down to ground. If both inputs are low, the output will remain high.

These three circuits, or their equivalents, form the three simplest gates. They are called NOT, NAND, and NOR gates, respectively. NOT gates are often called **inverters**; we will use the two terms interchangeably. If we now adopt the convention that "high" (V_{cc} volts) is a logical 1, and that "low" (ground) is a logical 0, we can express the output value as a function of the input values. The conventional symbols used to depict these three gates are shown in Fig. 3-2(a)-(c), along with the functional behavior for each circuit.

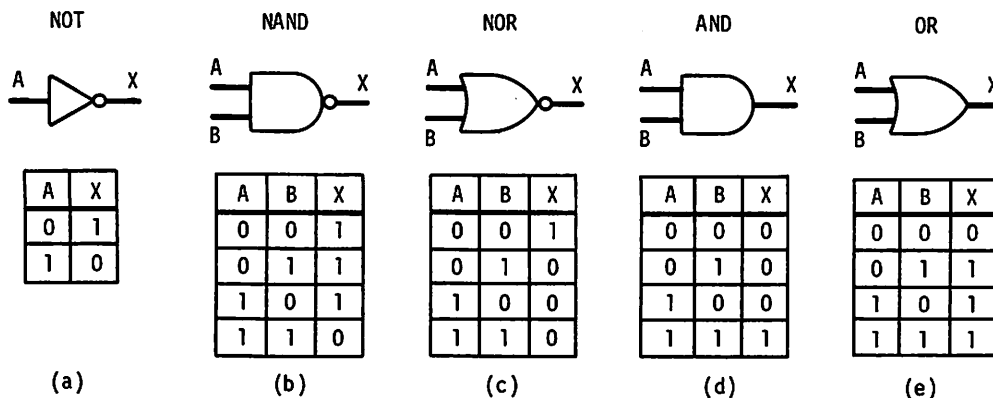


Fig. 3-2. The symbols and functional behavior for the five basic gates.

If the output signal of Fig. 3-1(b) is fed into an inverter circuit, we get another circuit with precisely the inverse of the NAND gate—namely, a circuit whose output is 1 if and only if both inputs are 1. Such a circuit is called an AND gate; its symbol and functional description are given in Fig. 3-2(d). Similarly, the NOR gate can be connected to an inverter to yield a circuit whose output is 1 if either or both inputs is a 1 but 0 if both inputs are 0. The symbol and functional description of this circuit, called an OR gate, are given in Fig. 3-2(e). The small circles used as part of the symbols for the inverter, NAND gate, and NOR gate are called **inversion bubbles**. They are often used in other contexts as well to indicate an inverted signal.

The five gates of Fig. 3-2 are the principal building blocks of the digital logic level. From the foregoing discussion, it should be clear that the NAND and NOR gates require two transistors each, whereas the AND and OR gates require three each. For this reason, many computers are based on NAND and NOR gates rather than the more familiar AND and OR gates. (In practice, all the gates are implemented somewhat

differently, but NAND and NOR are still simpler than AND and OR.) In passing it is worth noting that gates may have more than two inputs. In principle, a NAND gate, for example, may have arbitrarily many inputs, but in practice more than eight inputs is unusual.

Although the subject of how gates are constructed belongs to the device level, we would like to mention the major families of manufacturing technology because they are referred to frequently. The two major technologies are **bipolar** and **MOS** (Metal Oxide Semiconductor). The major bipolar types are **TTL** (Transistor-Transistor Logic), which has been the workhorse of digital electronics for years, and **ECL** (Emitter-Coupled Logic), which is used when very high-speed operation is required.

MOS gates are 10 times slower than TTL and 100 times slower than ECL but require almost no power and almost no space, so large numbers of them can be packed together tightly. MOS comes in many varieties, including PMOS, NMOS, and CMOS, with new varieties continually appearing.

3.1.2. Boolean Algebra

To describe the circuits that can be built by combining gates, a new type of algebra is needed, one in which variables and functions can take on only the values 0 and 1. Such an algebra is called a **Boolean algebra**, after its discoverer, the English mathematician George Boole (1815-1864). Strictly speaking, we are really referring to a specific type of Boolean algebra, a **switching algebra**, but the term “Boolean algebra” is so widely used to mean “switching algebra” that we will not make the distinction.

Just as there are functions in “ordinary” (i.e., secondary-school) algebra, so are there functions in Boolean algebra. A Boolean function has one or more input variables and yields a result that depends only on the values of these variables. A simple function, f , can be defined by saying that $f(A)$ is 1 if A is 0 and $f(A)$ is 0 if A is 1. This function is the NOT function of Fig. 3-2(a).

Because a Boolean function of n variables has only 2^n possible sets of input values, the function can be completely described by giving a table with 2^n rows, each row telling the value of the function for a different combination of input values. Such a table is called a **truth table**. The tables of Fig. 3-2 are examples of truth tables. If we agree to always list the rows of a truth table in numerical order (base 2), that is, for two variables in the order 00, 01, 10, and 11, the function can be completely described by the 2^n -bit binary number obtained by reading the result column of the truth table vertically. Thus NAND is 1110, NOR is 1000, AND is 0001, and OR is 0111. Obviously, only 16 Boolean functions of two variables exist, corresponding to the 16 possible 4-bit result strings. In contrast, ordinary (i.e., high school) algebra has an infinite number of functions of two variables, none of which can be described by giving a table of outputs for all possible inputs because each variable can take on any one of an infinite number of possible values. Clearly, finiteness has its virtues.

Figure 3-3(a) shows the truth table for a Boolean function of three variables: $M = f(A, B, C)$. This function is the majority logic function, that is, it is 0 if a

majority of its inputs are 0, and 1 if a majority of its inputs are 1. Although any Boolean function can be fully specified by giving its truth table, as the number of variables increases, this notation becomes increasingly cumbersome. Instead, another notation is frequently used.

To see how this other notation comes about, note that any Boolean function can be specified by telling which combinations of input variables give an output value of 1. For the function of Fig. 3-3(a) there are four combinations of input variables that make M 1. By convention, we will place a bar over an input variable to indicate that its value is 0. The absence of a bar means that it is 1. Furthermore, we will use implied multiplication or a dot to mean AND and $+$ to mean OR. Thus, for example, $\bar{A}BC$ means $A = 1$ and $B = 0$ and $C = 1$. Also, $\bar{A}\bar{B} + B\bar{C}$ means $(A = 1 \text{ and } B = 0) \text{ or } (B = 1 \text{ and } C = 0)$. The four rows of Fig. 3-3(a) producing 1 bits in the output are: $\bar{A}BC$, $\bar{A}\bar{B}C$, $A\bar{B}\bar{C}$, and ABC . The function, M , is true if any one of these four conditions is true; hence we can write

$$M = \bar{A}BC + \bar{A}\bar{B}C + A\bar{B}\bar{C} + ABC$$

as a compact way of giving the truth table. A function of n variables can thus be described by giving a "sum" of at most 2^n n -variable "product" terms. This formulation is especially important, as we will see shortly, because it leads directly to an implementation of the function using standard gates.

3.1.3. Implementation of Boolean Functions

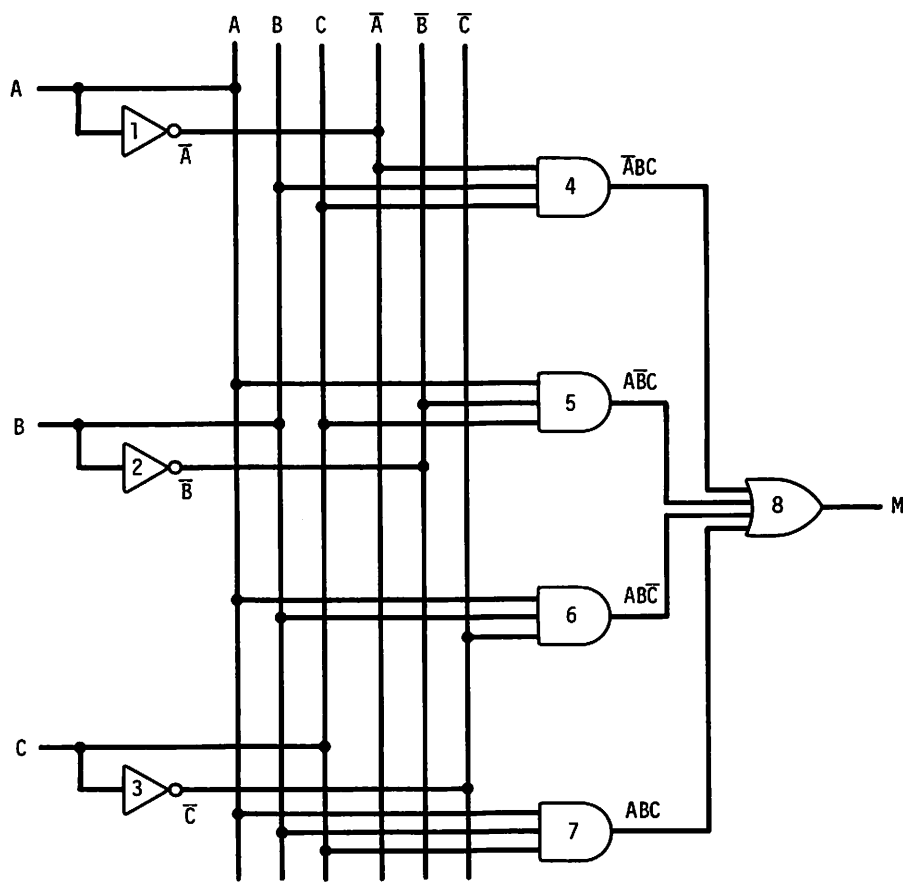
As mentioned above, the formulation of a Boolean function as a sum of up to 2^n product terms leads directly to a possible implementation. Using Fig. 3-3 as an example, we can see how this implementation is accomplished. In Fig. 3-3(b), the inputs, A , B , and C , are shown at the left edge and the output function, M , is shown at the right edge. Because complements of the input variables are needed, they are generated by tapping the inputs and passing them through the inverters labeled 1, 2, and 3. To keep the figure from becoming cluttered, we have drawn in six vertical lines, three of which are connected to the input variables, and three of which are connected to their complements. These lines provide a convenient source for the inputs to subsequent gates. For example, gates 5, 6, and 7 all use A as an input. In an actual circuit these gates would probably be wired directly to A without using any intermediate "vertical" wires.

The circuit contains four AND gates, one for each term in the equation for M (i.e., one for each row in the truth table having a 1 bit in the result column). Each AND gate computes one row of the truth table, as indicated. Finally, all the product terms are ORed together to get the final result.

The circuit of Fig. 3-3(b) uses a convention that we will need repeatedly throughout this book: when two lines cross, no connection is implied unless a heavy dot is present at the intersection. For example, the output of gate 3 crosses all six vertical lines but it is connected only to $\bar{C}$. Be warned that some authors use other conventions.

A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(a)



(b)

Fig. 3-3. (a) The truth table for the majority function of three variables.
 (b) A circuit for (a).

From the example of Fig. 3-3 it should be clear how to implement a circuit for any Boolean function:

1. Write down the truth table for the function.
2. Provide inverters to generate the complement of each input.
3. Draw an AND gate for each term with a 1 in the result column.
4. Wire the AND gates to the appropriate inputs.
5. Feed the output of all the AND gates into an OR gate.

Although we have shown how any Boolean function can be implemented using NOT, AND, and OR gates, it is often convenient to implement circuits using only a single type of gate. Fortunately, it is straightforward to convert circuits generated by the preceding algorithm to pure NAND or pure NOR form. To make such a conversion, all we need is a way to implement NOT, AND, and OR using a single gate type. The top row of Fig. 3-4 shows how all three of these can be implemented using only NAND gates; the bottom row shows how it can be done using only NOR gates. In both cases a value can be inverted using a single gate, whereas both AND and OR require three gates each. Thus to implement a Boolean function using only NAND or only NOR gates, first follow the procedure given above for constructing it with NOT, AND, and OR. Then replace the multi-input gates with equivalent circuits using two-input gates. For example, $A + B + C + D$ can be computed as $(A + B) + (C + D)$, using three two-input OR gates. Finally, the NOT, AND, and OR gates are replaced by the circuits of Fig. 3-4.

Although this procedure does not lead to the optimal circuits, in the sense of the minimum number of gates, it does show that a solution is always feasible. Both NAND and NOR gates are said to be **complete**, because any Boolean function can be computed using either of them. No other gate has this property, which is another reason they are often preferred for the building blocks of circuits.

3.1.4. Circuit Equivalence

Circuit designers naturally try to reduce the number of gates in their products to reduce component cost, printed circuit board space, power consumption, and so on. To reduce the complexity of a circuit, the designer must find another circuit that computes the same function as the original but does so with fewer gates (or perhaps with simpler gates, for example, two-input gates instead of four-input gates). In the search for equivalent circuits, Boolean algebra can be a valuable tool.

As an example of how Boolean algebra can be used, consider the circuit and truth table for $AB + AC$ shown in Fig. 3-5(a). Although we have not discussed them yet, many of the rules of ordinary algebra also hold for Boolean algebra. In particular, $AB + AC$ can be factored into $A(B + C)$ using the distributive law. Figure 3-5(b) shows the circuit and truth table for $A(B + C)$. Because two functions are

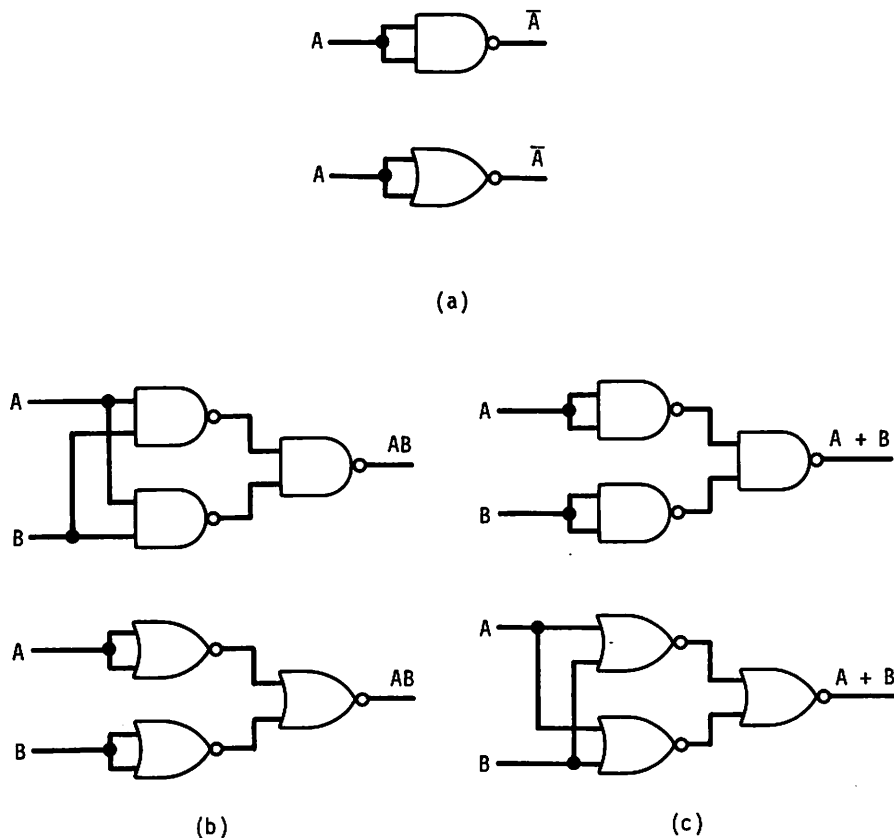


Fig. 3-4. Construction of (a) NOT, (b) AND, and (c) OR gates using only NAND gates or only NOR gates.

equivalent if and only if they have the same output for all possible inputs, it is easy to see from the truth tables of Fig. 3-5 that $A(B + C)$ is equivalent to $AB + AC$. Despite this equivalence, the circuit of Fig. 3-5(b) is clearly better than that of Fig. 3-5(a) because it contains fewer gates.

In general, a circuit designer can represent a circuit as a Boolean function and then apply the laws of Boolean algebra to this representation in an attempt to find a simpler but equivalent one. From the final representation, a new circuit can be constructed.

To use this approach, we need some identities from Boolean algebra. Figure 3-6 shows some of the major ones. It is interesting to note that each law has two forms that are **duals** of each other. By interchanging AND and OR and also 0 and 1, either form can be produced from the other one. All the laws can be easily proven by

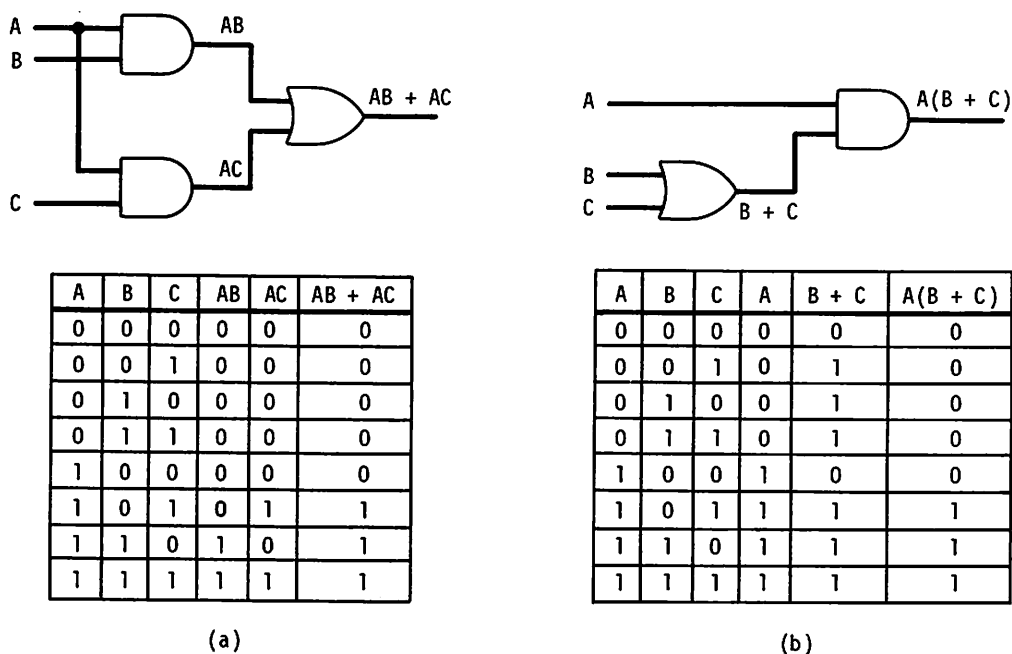


Fig. 3-5. Two equivalent functions. (a) $AB + AC$. (b) $A(B + C)$.

constructing their truth tables. Except for de Morgan's law, the absorption law, and the AND form of the distributive law, the results are reasonably intuitive. De Morgan's law can be extended to more variables, for example, $\overline{ABC} = \overline{A} + \overline{B} + \overline{C}$.

Name	AND form	OR form
Identity law	$1A = A$	$0 + A = A$
Null law	$0A = 0$	$1 + A = 1$
Idempotent law	$AA = A$	$A + A = A$
Inverse law	$A\overline{A} = 0$	$A + \overline{A} = 1$
Commutative law	$AB = BA$	$A + B = B + A$
Associative law	$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
Distributive law	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$
Absorption law	$A(A + B) = A$	$A + AB = A$
De Morgan's law	$\overline{AB} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A}\overline{B}$

Fig. 3-6. Some identities of Boolean algebra.

De Morgan's law suggests an alternative notation. In Fig. 3-7(a) the AND form is shown with negation indicated by inversion bubbles, both for input and output. Thus

an OR gate with inverted inputs is equivalent to a NAND gate. From Fig. 3-7(b), the dual form of de Morgan's law, it is clear that a NOR gate can be drawn as an AND gate with negated inputs. By negating both forms of de Morgan's law, we arrive at Fig. 3-7(c) and (d), which shown equivalent representations of the AND and OR gates. Analogous symbols exist for the multiple variable forms of de Morgan's law (e.g., an n input NAND gate becomes an OR gate with n inverted inputs).

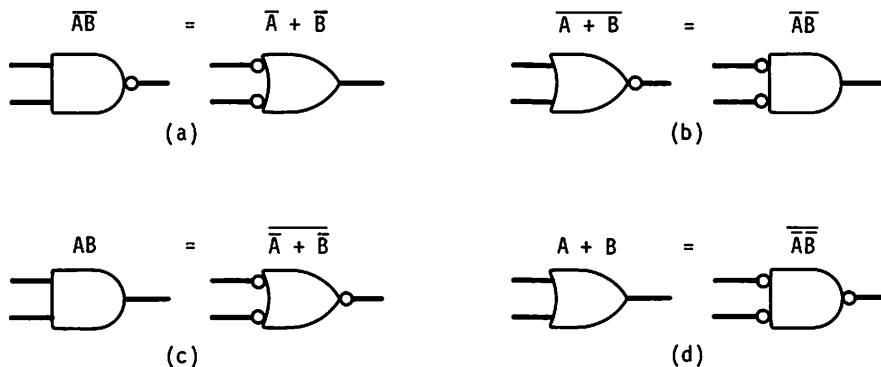


Fig. 3-7. Alternative symbols for some gates: (a) NAND. (b) NOR. (c) AND. (d) OR.

Using the identities of Fig. 3-7 and the analogous ones for multi-input gates, it becomes much easier to convert the sum-of-products representation of a truth table to pure NAND or pure NOR form. As an example, consider the XOR (exclusive or) function of Fig. 3-8(a). The standard sum-of-products circuit is shown in Fig. 3-8(b). To convert to NAND form, the lines connecting the output of the AND gates to the input of the OR gate should be redrawn with two inversion bubbles, as shown in Fig. 3-8(c). Finally, using Fig. 3-7(a), we arrive at Fig. 3-8(d). The variables $\bar{A}$ and $\bar{B}$ can be generated from A and B using NAND or NOR gates with their inputs tied together. Note that inversion bubbles can be moved along a line at will, for example, from the outputs of the input gates in Fig. 3-8(d) to the inputs of the output gate.

As a final note on circuit equivalence, we will now demonstrate the surprising result that the same physical gate can compute different functions, depending on the conventions used. In Fig. 3-9(a) we show the output of a certain gate, F , for different input combinations. Both inputs and outputs are shown in volts. If we adopt the convention that 0 volts is logical 0 and 5 volts is logical 1, called **positive logic**, we get the truth table of Fig. 3-9(b), the AND function. If, however, we adopt **negative logic**, which has 0 volts as logical 1 and 5 volts as logical 0, we get the truth table of Fig. 3-9(c), the OR function.

Thus the convention chosen to map voltages onto logical values is of great importance. Except where otherwise specified, we will henceforth use positive logic, so the terms logical 1, true, and high are synonyms, as are logical 0, false, and low.

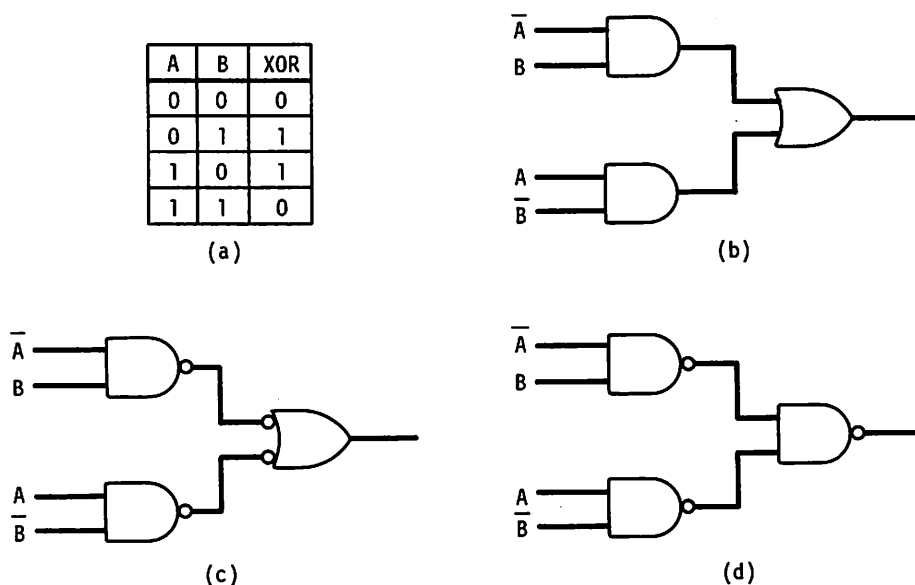


Fig. 3-8. (a) Truth table for EXCLUSIVE OR. (b)-(d) Three circuits for computing it.

A	B	F
0 ^v	0 ^v	0 ^v
0 ^v	5 ^v	0 ^v
5 ^v	0 ^v	0 ^v
5 ^v	5 ^v	5 ^v

(a)

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

(b)

A	B	F
1	1	1
1	0	1
0	1	1
0	0	0

(c)

Fig. 3-9. (a) Electrical characteristics of a device. (b) Positive logic. (c) Negative logic.

3.2. BASIC DIGITAL LOGIC CIRCUITS

In the previous sections we saw how to implement truth tables and other simple circuits using individual gates. In practice, few circuits are actually constructed gate-by-gate anymore, although this once was common. Nowadays, the usual building blocks are modules containing a number of gates. In the following sections we will examine these building blocks more closely and see how they are used and how they can be constructed from individual gates.

3.2.1. Integrated Circuits

Gates are not manufactured or sold individually but rather in units called **integrated circuits**, often called **ICs** or **chips**. An IC is a square piece of silicon about 5×5 mm on which some gates have been deposited. ICs are usually mounted in rectangular plastic or ceramic packages measuring 5 to 15 mm wide and 20 to 50 mm long. Along the long edges are two parallel rows of pins about 5 mm long that can be inserted in sockets or soldered to printed circuit boards. Each pin connects to the input or output of some gate on the chip or to power or to ground. The packages with two rows of pins outside and ICs inside are technically known as **dual inline packages** or **DIPs**, but everyone calls them chips, thus blurring the distinction between the piece of silicon and its package. The most common packages have 14, 16, 18, 20, 22, 24, 28, 40, 64, or 68 pins. For some special applications square packages with pins on all four sides are sometimes used.

Chips can be divided into rough classes based on the number of gates they contain:

SSI (Small Scale Integrated) circuit: 1 to 10 gates

MSI (Medium Scale Integrated) circuit: 10 to 100 gates

LSI (Large Scale Integrated) circuit: 100 to 100,000 gates

VLSI (Very Large Scale Integrated) circuit: >100,000 gates

These classes have different properties and are used in different ways.

An SSI chip typically contains two to six independent gates, each of which can be used individually, in the style of the previous sections. Figure 3-10 illustrates a schematic drawing of a common SSI chip containing four NAND gates. Each of these gates has two inputs and one output, requiring a total of 12 pins for the four gates. In addition, the chip needs power (typically about 5 volts, denoted by V_{cc}), and ground (GND), which are shared by all gates. The package generally has a notch near pin 1 to identify the orientation. To avoid clutter in circuit diagrams, neither power, nor ground, nor unused gates are conventionally shown.

Figure 3-11 shows some other common SSI chips. These chips belong to the 7400 TTL series developed by Texas Instruments and now produced by many semiconductor manufacturers. They are most appropriate for simple circuits that cannot be realized any other way. The circuit of Fig. 3-3(b) could be constructed from one 7404, two 7411s, and one 7432. No four-input OR gate is available in the 7400 series, but the same function can be computed by ORing pairs of inputs and then ORing the results together. Symbolically, this can be represented as $A + B + C + D = (A + B) + (C + D)$, that is, A and B are ORED together as are C and D , with these two sums then being ORED. If the circuit of Fig. 3-3(b) were actually constructed this way, the printed circuit board on which they were mounted would contain conducting wire-like tracks to connect the appropriate pins. Some of the gates would not be used. The 7486 chip contains a gate we have not shown before, the EXCLUSIVE OR gate, which is a one-gate equivalent to the circuit of Fig. 3-8.

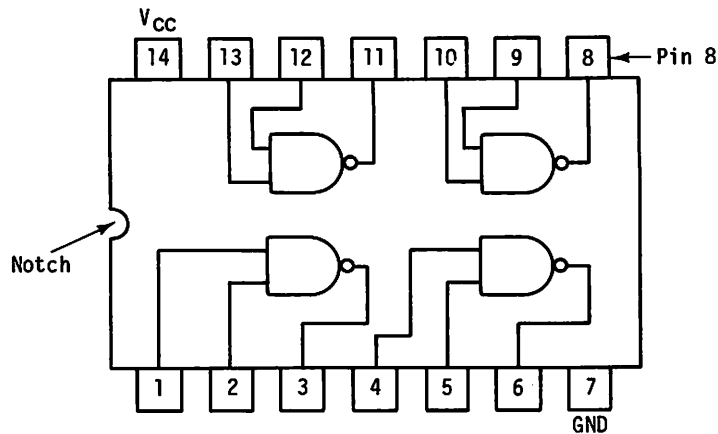


Fig. 3-10. An SSI chip containing four gates.

For our purposes, all gates are ideal in the sense that the output appears as soon as the input is applied. In reality, chips have a finite **gate delay**, which is the time it takes for the signal to propagate from the input to the output. Typical delays are 1 to 20 nsec. In schematics one often sees numbers like 74S00, 74L00, 74H00, and 74LS00. These represent functionally equivalent chips with different delay/power/price trade-offs. The 74C00 series consists of CMOS chips that are functionally identical to the corresponding 7400 TTL ones.

It is within the current state of the art to put nearly a million gates on a single chip. Because any circuit can be built up from NAND gates, you might think that a manufacturer could make a very general chip containing the equivalent of 250,000 chips like the 7400. Unfortunately, such a chip would need 3,000,002 pins. With the standard pin spacing of 0.1 inch, the chip would be over 2 miles long, which might have a negative effect on sales. Clearly, the only way to take advantage of the technology is to design circuits with a high gate/pin ratio. In the following sections we will look at simple MSI circuits that combine a number of gates internally to provide a useful function requiring only a limited number of external connections (pins). After that we will examine two applications requiring thousands of gates (LSI) yet only 20 to 40 pins—namely, memory chips and microprocessor chips. The dividing line between LSI and VLSI is vague, but a typical VLSI chip might have a large microprocessor *and* considerable memory on the same chip.

3.2.2. Combinational Circuits

Many applications of digital logic require a circuit with multiple inputs and multiple outputs in which the outputs are uniquely determined by the current inputs. Such a circuit is called a **combinational circuit**. Not all circuits have this property. For example, a circuit containing memory elements may generate outputs that depend on

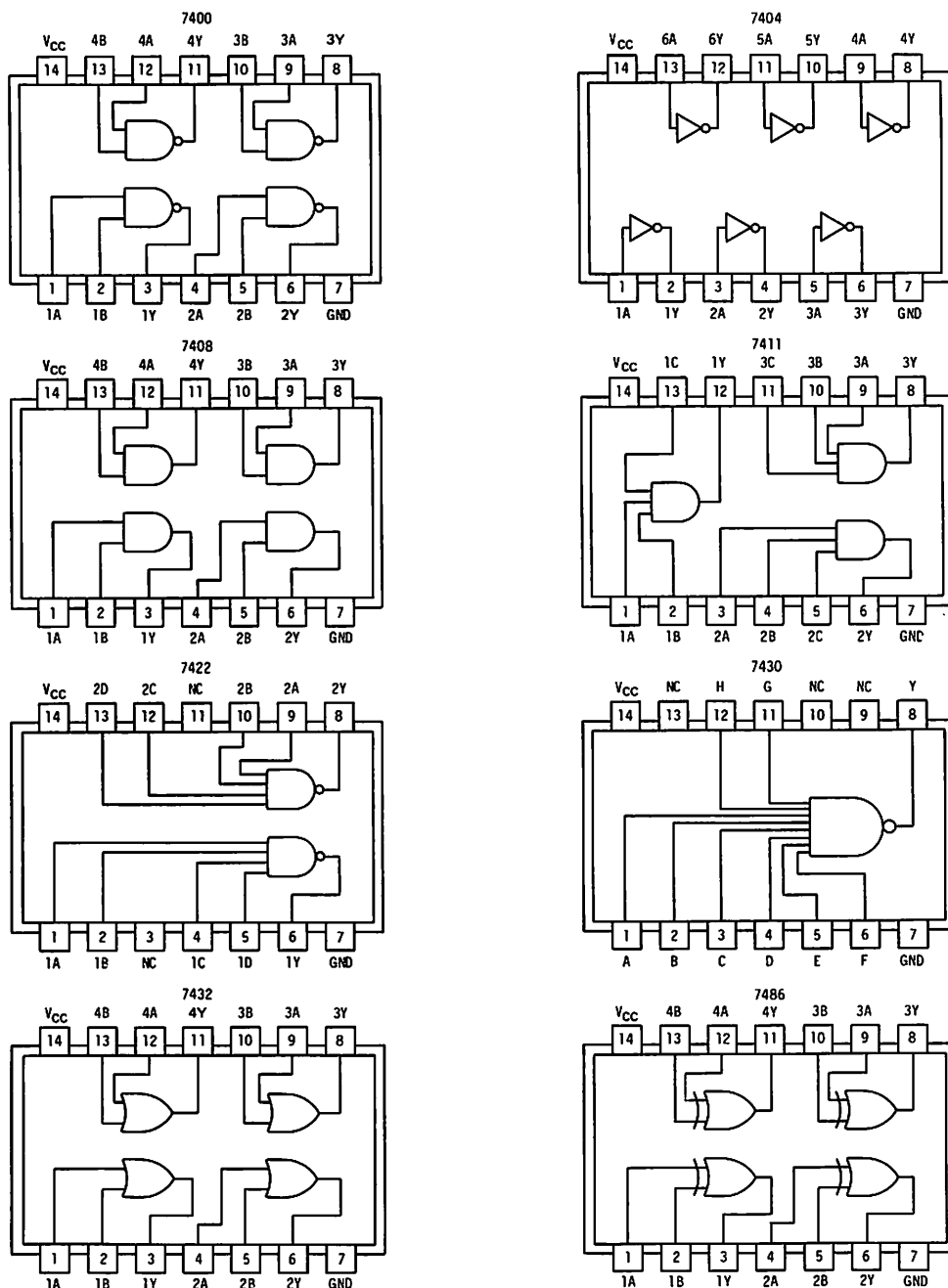


Fig. 3-11. Some SSI chips. Pin layouts from *The TTL Data Book for Design Engineers*, copyright © 1976 Texas Instruments Incorporated.

the stored values as well as the input variables. A circuit implementing a truth table, such as that of Fig. 3-3(a), is a typical example of a combinational circuit. In this section we will examine some frequently used combinational circuits, as examples of MSI chips.

Multiplexers

At the digital logic level, a **multiplexer** is a circuit with 2^n data inputs, one data output, and n control inputs that select one of the data inputs. The selected data input is “gated” (i.e., routed) to the output. Figure 3-12 is a schematic diagram for an eight-input multiplexer. The three control lines, A , B , and C , encode a 3-bit number that specifies which of the eight input lines is gated to the OR gate and thence to the output. No matter what value is on the control lines, seven of the AND gates will always output 0; the other one may output either 0 or 1, depending on the value of the selected input line. Each AND gate is enabled by a different combination of the control inputs.

The circuit of Fig. 3-12 is an ideal candidate for implementation as an MSI chip, as illustrated in Fig. 3-13(a). Such a chip has eight data inputs, three control inputs, and one output. When power and ground are included, the chip could be implemented in a 14-pin package. Using this MSI multiplexer chip, we can implement the circuit of Fig. 3-3(b) on a single chip, as shown in Fig. 3-13(b). For each combination of A , B , and C , one of the data input lines is selected. Each input is wired to either V_{cc} (logical 1) or ground (logical 0). The algorithm for wiring the inputs is simple: input D_i is the same as the value in row i of the truth table. In Fig. 3-3(a), rows 0, 1, 2, and 4 are 0, so the corresponding inputs are grounded; the remaining rows are 1, so they are wired to logical 1. In this manner any truth table of three variables can be implemented using the chip of Fig. 3-13(a).

We have already seen how a multiplexer chip can be used to select one of several inputs and how it can implement a truth table. Another of its applications is as a parallel-to-serial data converter. By putting 8 bits of data on the input lines and then stepping the control lines sequentially from 000 to 111 (binary), the 8 bits are put onto the output line in series. A typical use for parallel-to-serial conversion is in a keyboard, where each keystroke implicitly defines a 7- or 8-bit number that must be output serially over a telephone line.

The MSI chip of Fig. 3-13(a) has eight data inputs. Chips similar to it are commercially available, as are multiplexer chips with 16 data inputs. Furthermore, chips with two independent four-input multiplexers exist, as do chips with four independent two-input multiplexers. Some of these chips provide both the selected output and its complement, and some have an additional input line that forces the output to 0, independent of the inputs (i.e., a chip enable/disable pin).

The inverse of a multiplexer is a **demultiplexer**, which routes its single input signal to one of 2^n outputs, depending on the values of the n control lines. If the binary value on the control lines is k , output k is selected.

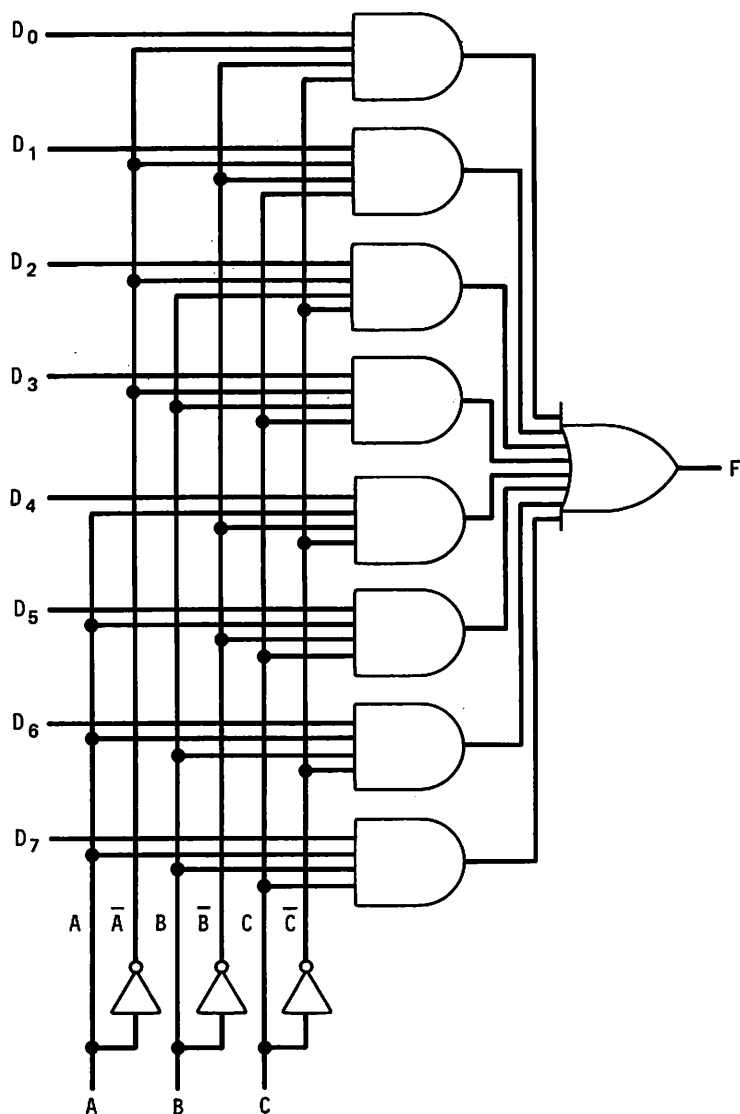


Fig. 3-12. An eight-input multiplexer circuit.

Decoders

As a second example of an MSI chip, we will now look at a circuit that takes an n -bit number as input and uses it to select (i.e., set to 1) exactly one of the 2^n output lines. Such a circuit, illustrated for $n = 3$ in Fig. 3-14, is called a **decoder**.

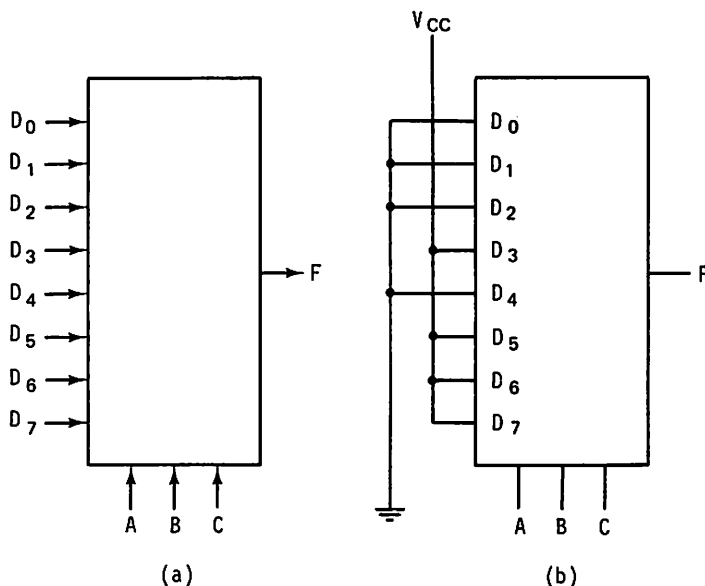


Fig. 3-13. (a) An MSI multiplexer chip. (b) The same chip wired to compute the majority function.

To see where a decoder might be useful, imagine a memory consisting of eight chips, each containing 8K bytes. Chip 0 has addresses 0 to 8191, chip 1 has addresses 8192 to 16383, and so on. When an address is presented to the memory, the high-order 3 bits are used to select one of the eight chips. Using the circuit of Fig. 3-14, these 3 bits are the three inputs, A , B , and C . Depending on the inputs, exactly one of the eight output lines, D_0 , ..., D_7 , is 1; the rest are 0. Each output line enables one of the eight memory chips. Because only one output line is set to 1, only one chip is enabled.

The operation of the circuit of Fig. 3-14 is straightforward. Each AND gate has three inputs, of which the first is either A or $\bar{A}$, the second is either B or $\bar{B}$, and the third is either C or $\bar{C}$. Each gate is enabled by a different combination of inputs: D_0 by $\bar{A}\bar{B}\bar{C}$, D_1 by $\bar{A}\bar{B}C$, and so on.

Commercially available MSI decoder chips include 4-to-16, 3-to-8 and dual 2-to-4 (i.e., two 2-to-4 decoders packaged together in a single chip). Furthermore, 4-to-10 decoders for decoding binary-coded decimal numbers are also common.

Comparators

Another useful MSI chip is the **comparator**, which compares two input words. The simple comparator of Fig. 3-15 takes two inputs, A , and B , each of length 4 bits, and produces 1 if they are equal and a 0 if they are not equal. The circuit is based on

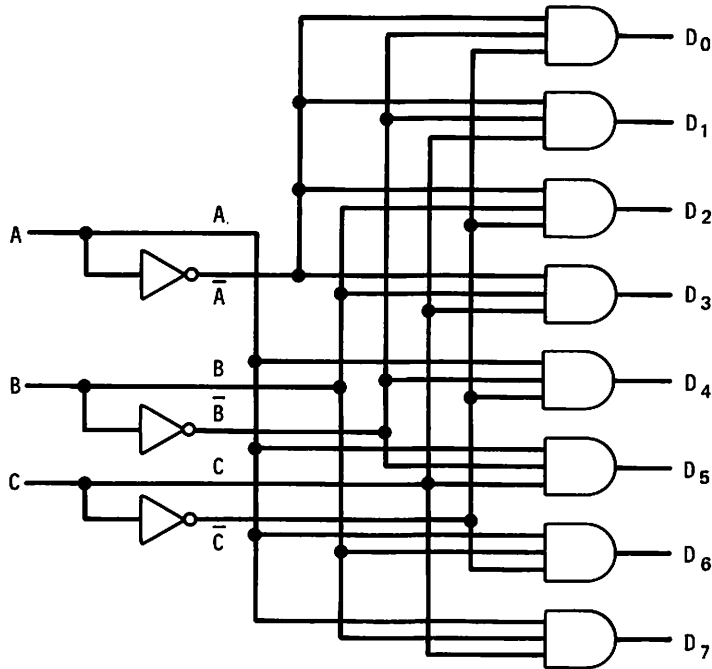


Fig. 3-14. A 3-to-8 decoder circuit.

the EXCLUSIVE OR gate, which puts out a 0 if its inputs are equal and a 1 if they are unequal. If the two input words are equal, all four of the EXCLUSIVE OR gates must output 0. These four signals can then be ORED together; if the result is 0, the input words are equal, otherwise not. In our example we have used a NOR gate as the final stage to reverse the sense of the test: 1 means equal, 0 means unequal. Commercially available MSI comparator chips not only have a pin for $A = B$, but also pins for $A < B$ and $A > B$.

Programmed Logic Arrays

We saw earlier that arbitrary functions (truth tables) can be constructed by computing product terms with AND gates and then ORing the products together. A very general chip for forming sums of products is the **programmed logic array** or **PLA**, which is shown in Fig. 3-16. This chip, the 74S330, has input lines for 12 variables. The complement of each input is generated internally, making 24 input signals in all. The heart of the circuit is an array of 50 AND gates, each of which can potentially have any subset of the 24 input signals as an input. Which input signal goes to which AND gate is determined by a 24×50 bit matrix supplied by the user. Each input line to the 50 AND gates contains a fuse. When shipped from the factory, all 1200 fuses

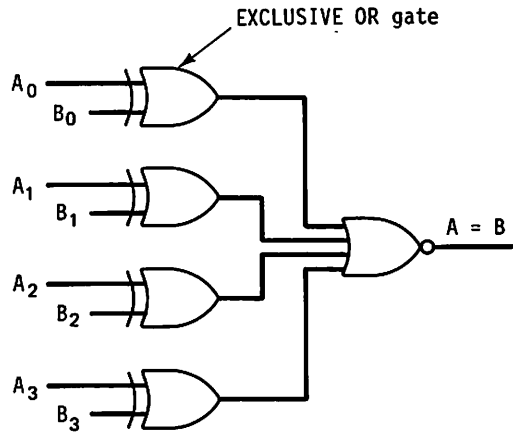


Fig. 3-15. A simple 4-bit comparator.

are intact. To program the matrix the user burns out selected fuses by applying a high voltage to the chip.

The output part of the circuit consists of six OR gates, each of which has up to 50 inputs, corresponding to the 50 outputs of the AND gates. Again here, a user-supplied (50×6) matrix tells which of the potential connections actually exist. The chip has 12 input pins, 6 output pins, power, and ground, and comes in a 20-pin package.

As an example of how a PLA can be used, let us reconsider the circuit of Fig. 3-3(b) once more. It has three inputs, four AND gates, one OR gate, and three inverters. With the appropriate internal connections made, our PLA can compute this function using three of its 12 inputs, four of its 50 AND gates, and one of its six OR gates. (The four AND gates should compute $\bar{A}BC$, $A\bar{B}C$, ABC , and ABC , respectively; the OR gate takes these four product terms as input.) In fact, the same PLA could be wired up to compute simultaneously a total of four functions of similar complexity. For these simple functions the number of input variables is the limiting factor; for more complicated ones it might be the AND or OR gates.

We can now compare the three different ways we have discussed for implementing the truth table of Fig. 3-3(a). Using SSI components, we need four chips. Alternatively, we could suffice with one MSI multiplexer chip, as shown in Fig. 3-13(b). Finally, we could use a quarter of one PLA chip. Obviously, if many functions are needed, the PLA is more efficient than the other two methods.

3.2.3. Arithmetic Circuits

It is now time to move on from the general-purpose MSI circuits discussed above to MSI combinational circuits used for doing arithmetic. We will begin with a simple 8-bit shifter, then look at how adders are constructed, and finally examine arithmetic logical units, which play a central role in any computer.

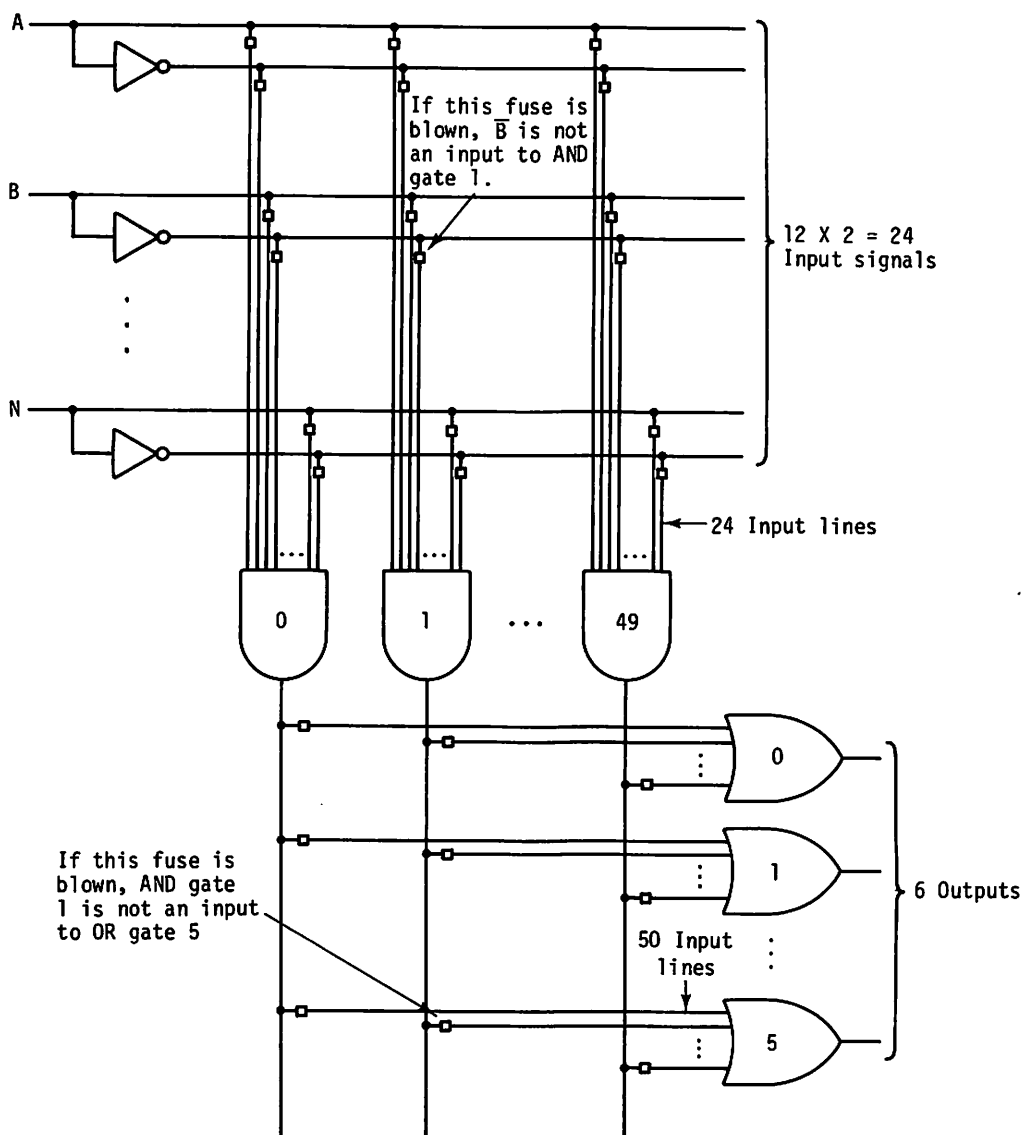


Fig. 3-16. A 12-input, 6-output programmed logic array. The little squares represent fuses that can be burned out to determine the function to be computed. The fuses are arranged in two matrices: the upper one for the AND gates and the lower one for the OR gates.

Shifters

Our first arithmetic MSI circuit is an eight-input, eight-output shifter (see Fig. 3-17). Eight bits of input are presented on lines $D_0, \dots, D_7$. The output, which is just the input shifted one bit, is available on lines $S_0, \dots, S_7$. The control line, C , determines the direction of the shift, 0 for left and 1 for right.

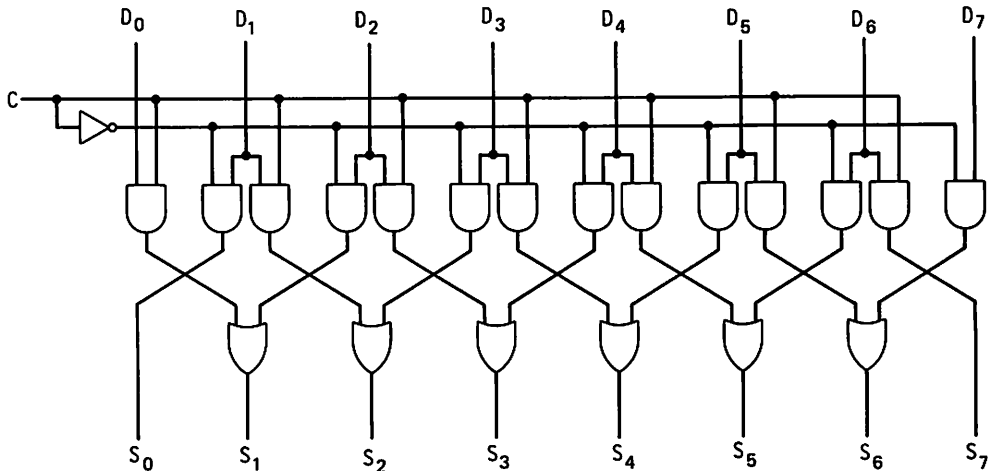


Fig. 3-17. A 1-bit left/right shifter.

To see how the circuit works, notice the pairs of AND gates for all the bits except the gates on the end. When $C = 1$, the right member of each pair is turned on, passing the corresponding input bit to output. Because the right AND gate is wired to the input of the OR gate to its right, a right shift is performed. When $C = 0$, it is the left member of the AND gate pair that turns on, effecting a left shift.

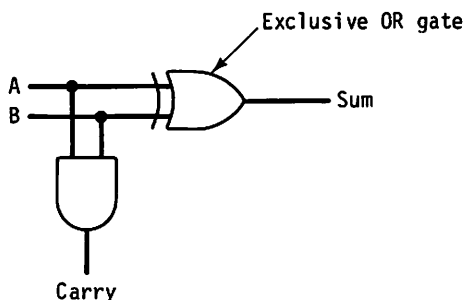
Adders

A computer that cannot add integers is unthinkable. Consequently, circuits for performing addition are an essential part of every CPU. The truth table for 1-bit addition is shown in Fig. 3-18(a). Two outputs are present, the sum of the inputs, A and B , and the carry to the next (leftward) position. A circuit for computing both the sum and the carry is given in Fig. 3-18(b). This circuit is known as a **half adder**.

Although a half adder is adequate for summing the low-order bits of two multibit input words, it will not do for a bit position in the middle of the word because it does not handle the carry into the position from the right. Instead, the **full adder** of Fig. 3-19 is needed. From inspection of the circuit it should be clear that a full adder is built up from two half adders. The *Sum* output line is 1 if an odd number of A , B , and the *Carry in* are 1. The *Carry out* is 1 if either A and B are both 1 (left input to the OR gate) or exactly one of them is 1 and the *Carry in* bit is also 1.

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(a)

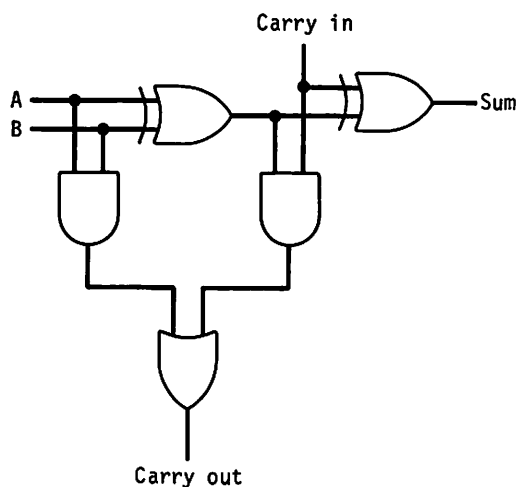


(b)

Fig. 3-18. (a) Truth table for 1-bit addition. (b) A circuit for a half adder.

A	B	Carry in	Sum	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(a)



(b)

Fig. 3-19. (a) Truth table for full adder. (b) Circuit for a full adder.

To build an adder for, say, two 16-bit words, one just replicates the circuit of Fig. 3-19(b) 16 times. The carry out of a bit is used as the carry into its left neighbor. The carry into the rightmost bit is wired to 0. This type of adder is called a **ripple carry adder**, because in the worst case, adding 1 to 111...111 (binary), the addition cannot complete until the carry has rippled all the way from the rightmost bit to the leftmost bit. Adders that do not have this delay, and hence are faster, also exist (Blaauw, 1976).

Arithmetic Logical Units

Most computers contain a single circuit for performing at least the basic operations of AND, OR, and sum of two machine words. Typically, such a circuit for n -bit words is built up of n identical circuits for the individual bit positions. Figure 3-20 is a simple example of such a circuit, called an **arithmetic logical unit** or **ALU**. It can compute any one of four functions—namely, $A \text{ AND } B$, $A \text{ OR } B$, $\bar{B}$, or $A + B$, depending on whether the function-select input lines F_0 and F_1 contain 00, 01, 10, or 11 (binary), respectively.

The lower left-hand corner of our ALU contains a 2-bit decoder to generate enable lines for the four operations, based on F_0 and F_1 . The upper left-hand corner has the logic to compute $A \text{ AND } B$, $A \text{ OR } B$, and $\bar{B}$, but at most one of these results is passed onto the final OR gate, depending on the enable lines coming out of the decoder. Because exactly one of the decoder outputs will be 1, exactly one of the four AND gates driving the OR gate will be enabled; the other three will output 0, independent of A and B .

The lower right-hand corner of the ALU contains a full adder for computing the sum of A and B , including handling the carries, because it is likely that several of these circuits will eventually be wired in parallel to perform full-word operations. Chips like Fig. 3-20 are actually available and are known as **bit slices**. They allow the computer designer to build an ALU of any desired width. Chips containing four or eight slices on one chip are also available to simplify the design and reduce chip count.

3.2.4. Clocks

In many digital circuits the order in which events happen is critical. Sometimes one event must precede another, sometimes two events must occur simultaneously. To allow designers to achieve the required timing relations, many digital circuits use clocks to provide synchronization. A **clock** in this context is a circuit that emits a series of pulses with a precise pulse width and precise interval between consecutive pulses. The interval between corresponding edges of two consecutive pulses is called the **clock cycle time**. Pulse frequencies are commonly between 1 and 100 MHz, corresponding to clock cycles of 1000 nsec to 10 nsec. To achieve high accuracy, the clock frequency is usually controlled by a crystal oscillator.

In a computer, many events may happen during a single clock cycle. If these events must occur in a specific order, the clock cycle must be divided into subcycles. A common way of providing finer resolution than the basic clock is to tap the primary clock line and insert a circuit with a known delay in it, thus generating a secondary clock signal that is phase-shifted from the primary, as shown in Fig. 3-21(a). The timing diagram of Fig. 3-21(b) provides four time references for discrete events:

1. Rising edge of C1
2. Falling edge of C1

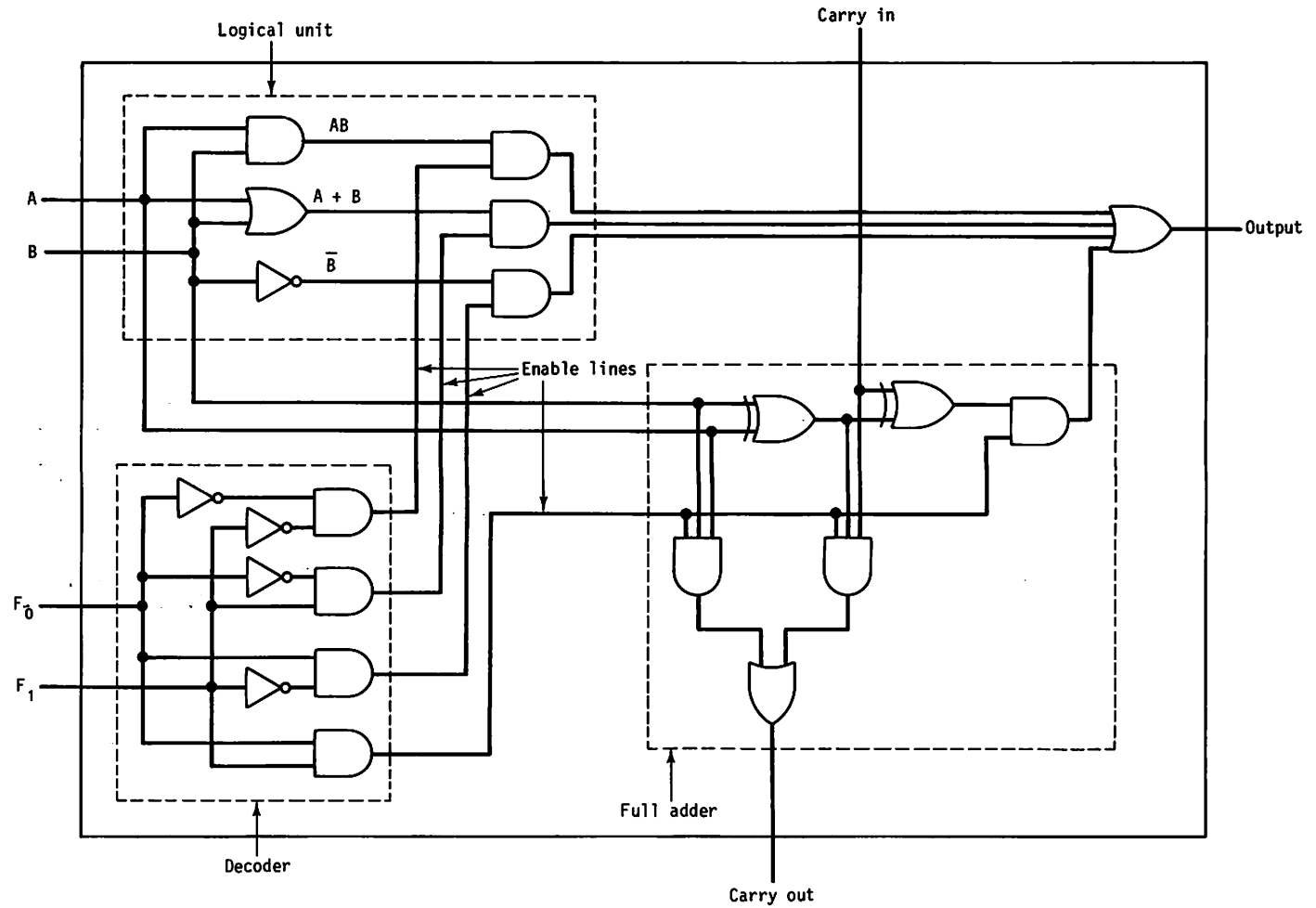


Fig. 3-20. A 1-bit ALU.

3. Rising edge of C2
4. Falling edge of C2

By tying different events to the various edges, the required sequencing can be achieved. If more than four time references are needed within a clock cycle, more secondary lines can be tapped from the primary, with different delays.

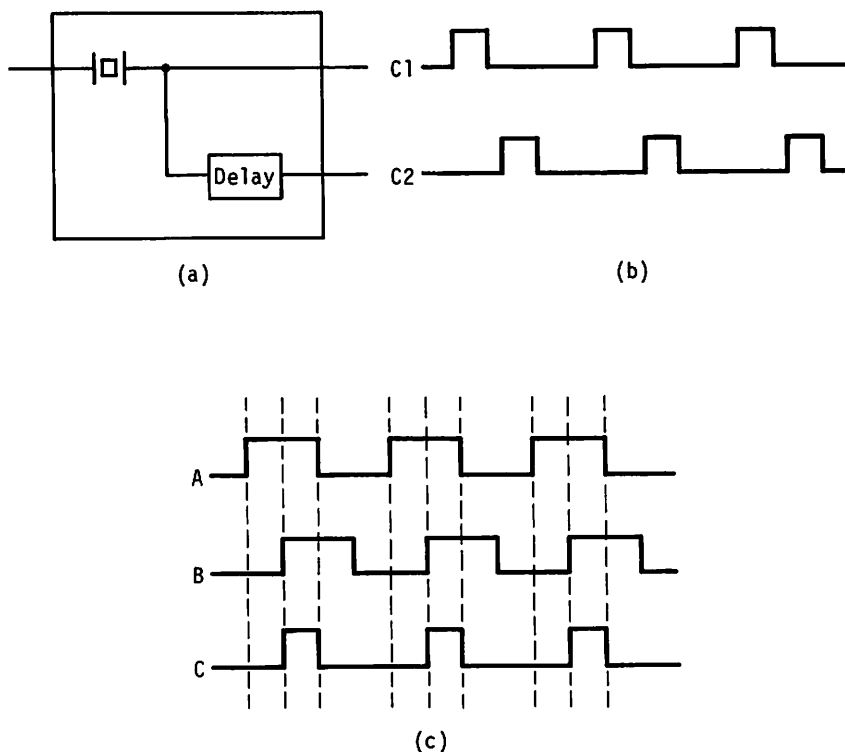


Fig. 3-21. (a) A clock. (b) The timing diagram for the clock. (c) Generation of an asymmetric clock.

In some circuits it is necessary to divide time into intervals rather than define points in time. This effect can be easily achieved by causing some events to happen when C1 is high and other events to happen when C2 is high. If more than two intervals are needed, more clock lines can be provided or the high states of the two clocks can be made to overlap partially in time. In the latter case four intervals can be distinguished: $\overline{C1}$ AND $\overline{C2}$, $\overline{C1}$ AND C2, C1 AND $\overline{C2}$, and C1 AND C2.

As an aside, it probably should be pointed out that Fig. 3-21(a)-(b) is really something of a simplification. Real clocks are symmetric, with time spent in the high state equal to the time spent in the low state, as shown by A in Fig. 3-21(c). To generate

an asymmetric pulse train, the basic clock is shifted using a delay circuit, as shown by B in Fig. 3-21(c). Finally, these two are ANDed together, to get C in Fig. 3-21(c). It is this signal that is shown as C1 in Fig. 3-21(b). C2 can be generated by delaying C1.

3.3. MEMORY

An essential component of every computer is its memory. Without memory there could be no computers as we now know them. Memory is used for storing both instructions to be executed and data. In the following sections we will examine the basic components of a memory system starting at the gate level to see how they work and how they are combined to produce large memories.

3.3.1. Latches

To create a 1-bit memory, we need a circuit that somehow “remembers” previous input values. Such a circuit can be constructed from two NOR gates, as illustrated in Fig. 3-22(a). Analogous circuits can be built from NAND gates. We will not mention these further, however, because they are conceptually identical to the NOR versions.

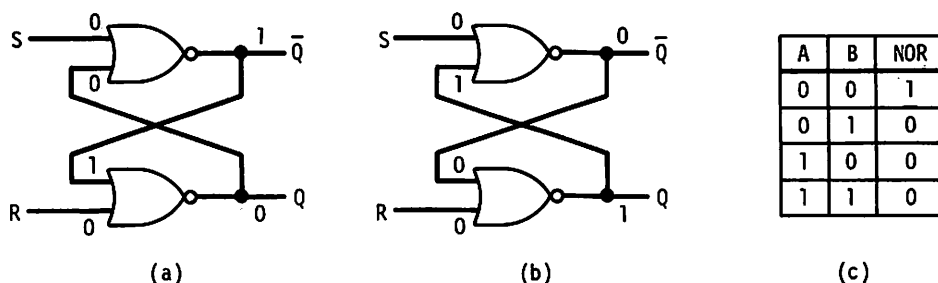


Fig. 3-22. (a) A NOR latch in state 0. (b) A NOR latch in state 1. (c) Truth table for NOR.

The circuit of Fig. 3-22(a) is called an **SR latch**. It has two inputs, S, for Setting the latch, and R, for Resetting (i.e., clearing) it. It also has two outputs, Q and $\bar{Q}$, which are complementary, as we will see shortly. Unlike a combinational circuit, the outputs of the latch are not uniquely determined by the current inputs.

To see how this comes about, let us assume that both S and R are 0, which they are most of the time. For argument's sake, let us further assume that $Q = 0$. Because Q is fed back into the upper NOR gate, both of its inputs are 0, so its output, $\bar{Q}$, is 1. The 1 is fed back into the lower gate, which then has inputs 1 and 0, yielding $Q = 0$. This state is at least consistent and is depicted in Fig. 3-22(a).

Now let us imagine that Q is not 0 but 1, with R and S still 0. The upper gate has inputs of 0 and 1, and an output, $\bar{Q}$, of 0, which is fed back to the lower gate.

This state, shown in Fig. 3-22(b), is also consistent. A state with both outputs equal to 0 is inconsistent, because it forces both gates to have two 0s as input, which, if true, would produce 1, not 0, as output. Similarly, it is impossible to have both outputs equal to 1, because that would force the inputs to 0 and 1, which yields 0, not 1. Our conclusion is simple: For $R = S = 0$, the latch has two stable states, which we will refer to as 0 and 1, depending on Q .

Now let us examine the effect of the inputs on the state of the latch. Suppose that S becomes 1 while $Q = 0$. The inputs to the upper gate are then 1 and 0, forcing the output to 0. This change makes both inputs to the lower gate 0, forcing the output to 1. Thus setting S (i.e., making it 1) switches the state from 0 to 1. Setting R in state 0 has no effect because the output of the lower gate is the same for inputs of 10 and 11.

Using similar reasoning, it is easy to see that setting S in state 1 has no effect but that setting R drives the latch to state 0. In summary, when S is turned on momentarily, the latch ends up in state 1, regardless of what state it was previously in. Likewise, setting R forces the latch to state 0. The circuit “remembers” whether S or R was last on. Using this property we can build computer memories.

Clocked SR Latches

It is often convenient to prevent the latch from changing state except at certain specified times. To achieve this goal, we modify the basic circuit slightly, as shown in Fig. 3-23, to get a **clocked SR latch**. This circuit has an additional input, the clock, which is normally 0. With the clock 0, both AND gates output 0, independent of S and R , and the latch does not change state. When the clock is 1, the effect of the AND gates vanishes and the latch becomes sensitive to S and R . Despite its name, the clock signal need not be driven by a clock. The terms **enable** and **strobe** are also widely used to mean that the clock input is 1; that is, the circuit is sensitive to the state of S and R .

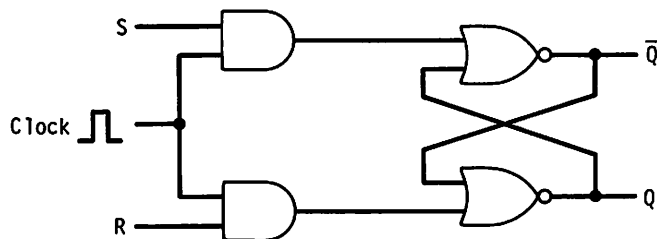


Fig. 3-23. A clocked SR latch.

Up until now we have carefully swept the problem of what happens when both S and R are 1 under the rug. And for good reason: the circuit becomes nondeterministic when both R and S finally return to 0. The only consistent state for $S = R = 1$ is

$Q = \bar{Q} = 0$, but as soon as both inputs return to 0, the latch must jump to one of its two stable states. If either input drops back to 0 before the other, the one remaining 1 longest wins, because when just one input is 1, it forces the state. If both inputs return to 0 simultaneously (which is very unlikely), the latch jumps to one of its stable states at random.

Clocked JK Latches

To prevent this ambiguity, we can modify the circuit again to get the **clocked JK latch** of Fig. 3-24. Here the outputs have been fed back into the AND gates. Because one of the outputs is always 0, setting both inputs to 1 only causes one of the AND gates to put out a 1—namely, the one attached to the output in 1 state. Let us consider the effect of $J = K = 1$ on each of the stable states. In state 0, the upper gate will be enabled and the lower one disabled, causing the latch to go to state 1. In state 1, the lower gate will be enabled and the upper one disabled, causing the latch to go to state 0. In both cases the latch switches state. The ambiguity has been removed. The letters J and K are universally used to distinguish JK latches from SR latches but they do not stand for anything in particular.

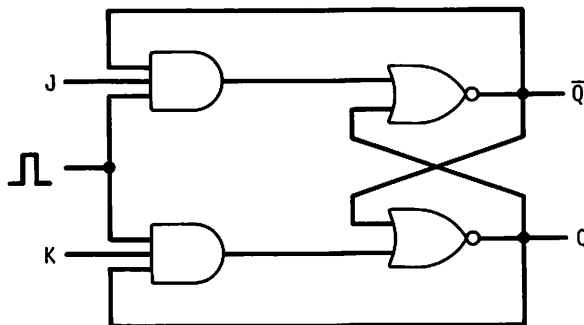


Fig. 3-24. A clocked JK latch.

Clocked D Latches

Another way to resolve the SR latch's ambiguity is to make sure that it cannot occur. Figure 3-25 gives a latch circuit with only one input, D . Because the input to the lower AND gate is always the complement of the input to the upper one, the problem of both inputs being 1 never arises. When $D = 1$ and the clock is 1, the latch is driven into state logical 1. When $D = 0$ and the clock is 1, the latch is driven into state logical 0. Put in other terms, when the clock is 1, the current value of D is sampled and stored in the latch. This circuit, called a **clocked D latch**, is a true 1-bit memory. The value stored is always available at Q . To load the current value of D into the memory, a positive pulse is put on the clock line.

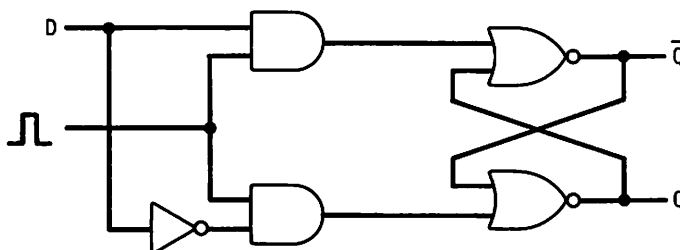


Fig. 3-25. A clocked D latch.

3.3.2. Flip-Flops and Registers

In many circuits it is necessary to sample the value on a certain line at a particular instant in time and store it. In principle, one way to achieve this goal is to use a clocked D latch with a very short pulse on the clock line around the time the sample is to be made. In practice, generating very short pulses can be difficult, so a variation on the latch circuit has been developed. In this variant, called a **flip-flop**, the state transition does not occur when the clock is 1 but during the clock transition from 0 to 1 or from 1 to 0 instead. Thus the length of the clock pulse is unimportant, as long as the transitions occur fast.

For emphasis, we will repeat the difference between a flip-flop and a latch. A flip-flop is **edge triggered**, whereas a latch is **level triggered**. Be warned, however, that in the literature these terms are often confused. Many authors use “flip-flop” when they are referring to a latch, and vice versa.

The standard symbols for latches and flip-flops are shown in Fig. 3-26. Figure 3-26(a) is a latch whose state is loaded when the clock, CK , is 1, in contrast to Fig. 3-26(b) which is a latch whose clock is normally 1 but which drops to 0 momentarily to load the state from D . Figure 3-26(c) and (d) are flip-flops rather than latches, which is indicated by the pointy symbol on the clock inputs. Figure 3-26(c) changes state on the rising edge of the clock pulse (0 to 1 transition), whereas Fig. 3-26(d) changes state on the falling edge (1 to 0 transition). Many, but not all, latches and flip-flops also have $\bar{Q}$ as an output, and some have two additional inputs *Set* or *Preset* (force state to logical 1) and *Reset* or *Clear* (force state to logical 0). In addition to D flip-flops, JK flip-flops are also available.

Registers

Flip-flops are available in a variety of configurations. A simple one, containing two independent D flip-flops with clear and preset signals, is illustrated in Fig. 3-27(a). Although packaged together in the same 14-pin chip, the two flip-flops are unrelated. A quite different arrangement is the octal flip-flop of Fig. 3-27(b). Here the eight (hence the term “octal”) D flip-flops are not only missing the $\bar{Q}$ and preset

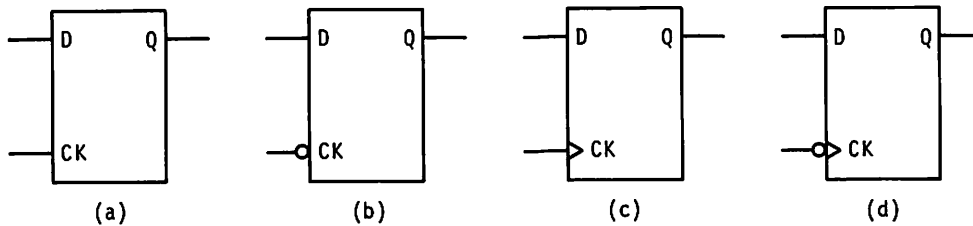


Fig. 3-26. D latches and flip-flops.

lines, but all the clock lines are ganged together and driven by pin 11. The flip-flops themselves are of the Fig. 3-26(d) type, but the inversion bubbles on the flip-flops are canceled by the inverter tied to pin 11, so the flip-flops are loaded on the rising transition. All eight clear signals are also ganged, so when pin 1 goes to 0, all the flip-flops are forced to their 0 state. In case you are wondering why pin 11 is inverted at the input and then inverted again at each CK signal, an input signal may not have enough current to drive all eight flip-flops; the input inverter is really being used as an amplifier.

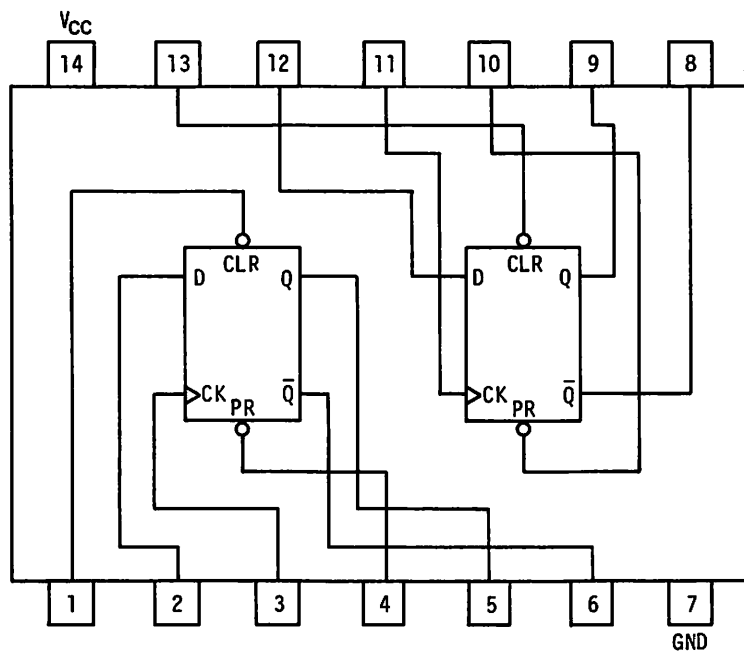
While the obvious reason for ganging the clock and clear lines of Fig. 3-27(b) is to save pins, in this configuration the chip is used in a different way from eight unrelated flip-flops. It is used as a single 8-bit register. Alternatively, two such chips can be used in parallel to form a 16-bit register by tying their respective pins 1 and 11 together. We will look at registers and their uses more closely in Chap. 4.

For the time being, the interesting thing to note is how the scarcity of pins drives chip design inexorably toward circuits with higher and higher gate-to-pin ratios. The simple D latch of Fig. 3-25 requires five gates, so an octal latch chip analogous to Fig. 3-27(b) needs 57 gates, including the eight inverters for CK and the eight inverters for CLR. An edge-triggered flip-flop is more complex than a latch internally, so Fig. 3-27(b) represents the equivalent of something like 100 gates, a far cry from the simple SSI chips of Fig. 3-11. In the next section we will study a different internal organization for memory chips, which allows a much higher gate-to-pin ratio.

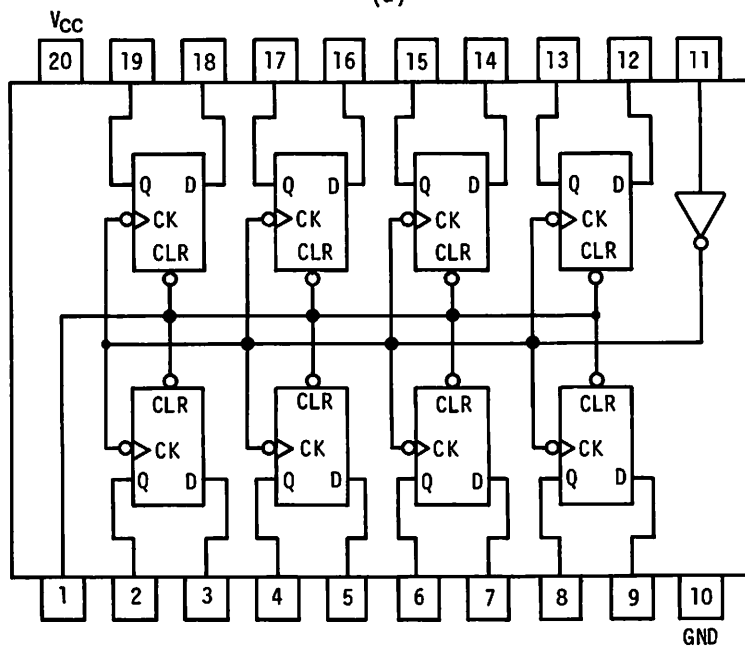
3.3.3. Memory Organization

Although we have now progressed from the simple 1-bit memory of Fig. 3-25 to the 8-bit memory of Fig. 3-27(b) we cannot continue this way due to lack of pins on the chip. Each flip-flop requires two pins, *D*, and *Q*, in addition to the control pins shared by all the flip-flops. To build large memories a different organization is required.

What we need is a design in which the number of pins grows logarithmically rather than linearly with the memory capacity. A widely used memory organization that meets this criterion is shown in Fig. 3-28. This example illustrates a memory



(a)



(b)

Fig. 3-27. (a) 7474 dual D flip-flop. (b) 74273 octal flip-flop.

with four 3-bit words. Each operation reads or writes a full 3-bit word. While the total memory capacity of 12 bits is hardly more than our octal flip-flop, it requires fewer pins and most important, the design extends easily to large memories.

While the memory of Fig. 3-28 may look complicated at first, it is really quite simple due to its regular structure. It has eight input lines and three output lines. Three inputs are data: I_0 , I_1 , and I_2 ; two are for the address: A_0 and A_1 ; and three are for control: CS for Chip Select, RD for distinguishing between read and write, and OE for Output Enable. The three outputs are for data: D_0 , D_1 , and D_2 . In principle this memory could be put into a 14-pin package, including power and ground versus 20 pins for the octal flip-flop.

To select this memory chip, external logic must set CS high and also set RD high (logical 1) for read and low (logical 0) for write. The two address lines must be set to indicate which of the four 3-bit words is to be read or written. For a read operation, the data input lines are not used, but the word selected is placed on the data output lines. For a write operation, the bits present on the data input lines are loaded into the selected memory word; the data output lines are not used.

Now let us look at Fig. 3-28 closely to see how it works. The four word-select AND gates at the left of the memory form a decoder. The input inverters have been placed so that each gate is enabled (output is high) by a different address. Each gate drives a word select line, from top to bottom for words 0, 1, 2, and 3. When the chip has been selected for a write, the vertical line labeled $CS \cdot \overline{RD}$ will be high, enabling one of the four write gates, depending on which word select line is high. The output of the write gate drives all the CK signals for the selected word, loading the input data into the flip-flops for that word. A write is only done if CS is high and RD is low and even then only the word selected by A_0 and A_1 is written; the other words are not changed.

Read is similar to write. The address decoding is exactly the same as for write. But now the $CS \cdot \overline{RD}$ line is low, so all the write gates are disabled and none of the flip-flops are modified. Instead, the word select line that is chosen enables the AND gates tied to the Q bits of the selected word. Thus one word outputs its data into the four-input OR gates at the bottom of the figure, while the other three words output 0. Consequently, the output of the OR gates is identical to the value stored in the word selected. The three words not selected make no contribution to the output.

Although we could have designed a circuit in which the three OR gates were just fed into the three output data lines, doing so sometimes causes problems. In particular, we have shown the data input lines and the data output lines as being different, but in actual memories the same lines are used. If we had tied the OR gates to the data output lines, the chip would try to output data, that is, force each line to a specific value, even on writes, thus interfering with the input data. For this reason, it is desirable to have a way to connect the OR gates to the data output lines on reads but disconnect them completely on writes. What we need is an electronic switch that can make or break a connection in a few nanoseconds.

Fortunately, such switches exist. Figure 3-29(a) shows the symbol for what is called a **noninverting buffer**. It has a data input, a data output, and a control input.

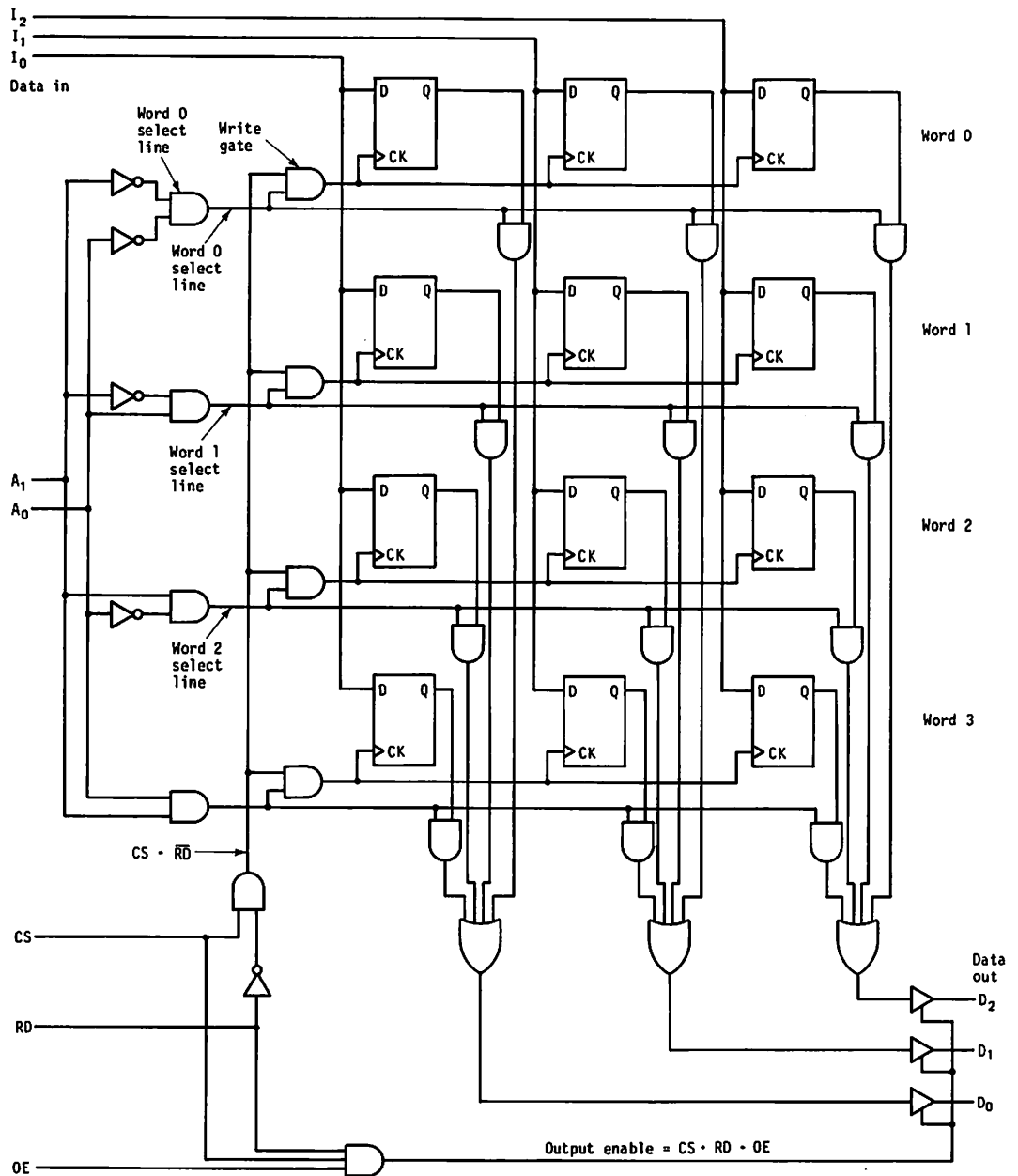


Fig. 3-28. Logic diagram for a 4×3 memory. Each row is one of the four 3-bit words. A read or write operation always reads or writes a complete word.

When the control input is high, the buffer acts like a wire, as shown in Fig. 3-29(b). When the control input is low, the buffer acts like an open circuit, as shown in Fig. 3-29(c); it is as though someone detached the data output from the rest of the circuit with a wirecutter. However, in contrast to the wirecutter approach, the connection can be subsequently restored in a few nanoseconds by just making the control signal high again.

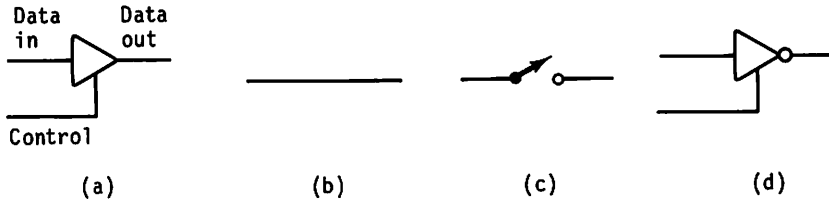


Fig. 3-29. (a) A noninverting buffer. (b) Effect of (a) when control is high. (c) Effect of (a) when control is low. (d) An inverting buffer.

Figure 3-29(d) shows an **inverting buffer**, which acts like a normal inverter when control is high and disconnects the output from the circuit when control is low. Both kinds of buffers are **tri-state devices**, because they can output 0, 1, or none of the above (open circuit). Buffers also amplify signals, so they can drive many inputs simultaneously. They are sometimes used in circuits for this reason, even when their switching properties are not needed.

Getting back to the memory circuit, it should now be clear what the three noninverting buffers on the data output lines are for. When CS, RD, and OE are all high, the output enable signal is also high enabling the buffers and putting a word onto the output lines. When any one of CS, RD, or OE is low, the data outputs are disconnected from the rest of the circuit.

As an aside, in Chap. 2 we discussed how a memory appears from the outside—namely, an MAR for the memory address and an MBR for the data. From Fig. 3-28 it is apparent that A_0 and A_1 are taken from a 2-bit MAR, while the three data inputs and three data outputs correspond to the MBR. Because memories usually have only one MBR, the importance of disconnecting D_0 , D_1 , and D_2 from the MBR on writes to memory to avoid interfering with I_0 , I_1 , and I_2 should be obvious now.

3.3.4. Memory Properties

The nice thing about the memory of Fig. 3-28 is that it extends easily to larger sizes. As we drew it, the memory is 4×3 , that is, four words of 3 bits each. To extend it to 4×8 we need only add five more columns of four flip-flops each, as well as five more input lines and five more output lines. To go from 4×3 to 8×3 we must add four more rows of three flip-flops each, as well as an address line A_2 . With this kind of structure, the number of words in the memory should be a power of 2 for maximum efficiency, but the number of bits in a word can be anything.

Because integrated circuit technology is well suited to making chips whose internal structure is a repetitive two-dimensional pattern, memory chips are an ideal application for it. As the technology improves, the number of bits that can be put on a chip keeps increasing, typically by a factor of 4 every 3 or 4 years. In the early 1970s, chips had 1K bits; in later years, they had 4K, 16K, 64K, 256K, and so on. By 1990, chips with millions of bits may be commonplace. The larger chips do not always render the smaller ones obsolete due to different trade-offs in capacity, speed, power, price, and interfacing convenience.

For any given memory size, there are various ways of organizing the chip. Figure 3-30 shows two possible organizations for a 16K-bit chip, $2K \times 8$ and $16K \times 1$. In the former, 11 address lines are needed to address the selected *byte*, and eight data lines are needed for loading and storing data. In the latter, 14 address lines are needed to address the selected *bit*, but only one line is needed for data in and one for data out. In both cases lines are needed for distinguishing reads from writes and for chip select. The $\overline{OE}$ line is Output Enable; when it is 0 the output is present, when it is 1 it is not present (i.e., the chip output is disconnected from the circuit). The $\overline{WE}$ line is for Write Enable. The bars over $\overline{OE}$, $\overline{WE}$, and $\overline{CS}$ mean that the signal is enabled in the low voltage state (logical 0) rather than the high-voltage state (logical 1). This convention will be discussed in more detail later.

Note that building a memory with a 16-bit MBR from $16K \times 1$ chips requires 16 chips in parallel and gives a total capacity of at least 32K bytes, whereas using $2K \times 8$ chips requires only two chips in parallel and allows memories as small as 4K bytes.

The memories we have studied so far can all be read and written. Such memories are called **RAMs** (Random Access Memories), which is a misnomer because all memory chips are randomly accessible, but the term is well established. RAMs come in two varieties, static and dynamic. **Static RAMs** are constructed internally using circuits similar to our basic D latch. These memories have the property that their contents are retained as long as the power is kept on: seconds, minutes, hours, even days.

Dynamic RAMs, in contrast, do not use latch-like circuits. Instead, a dynamic RAM is an array of tiny capacitors, each of which can be charged or discharged, allowing 0 and 1 to be stored. Because the electric charge tends to leak out, each bit in a dynamic RAM must be **refreshed** every few milliseconds to prevent the data from leaking away. Because external logic must take care of the refreshing, dynamic RAMs require more complex interfacing than static ones, although in many applications this disadvantage is compensated for by their larger capacities. Some dynamic RAMs have on-chip refresh logic, providing both high capacity and simple interfacing. Such chips are said to be **quasi-static**.

RAMs are not the only kind of memory chips. In many applications, such as toys, appliances, and cars, the program and some of the data must remain stored even when the power is turned off. Furthermore, once installed, neither the program nor the data are ever changed. These requirements have led to the development of **ROMs** (Read-Only Memories), which cannot be changed or erased, intentionally or otherwise. The data in a ROM are inserted during its manufacture, essentially by exposing

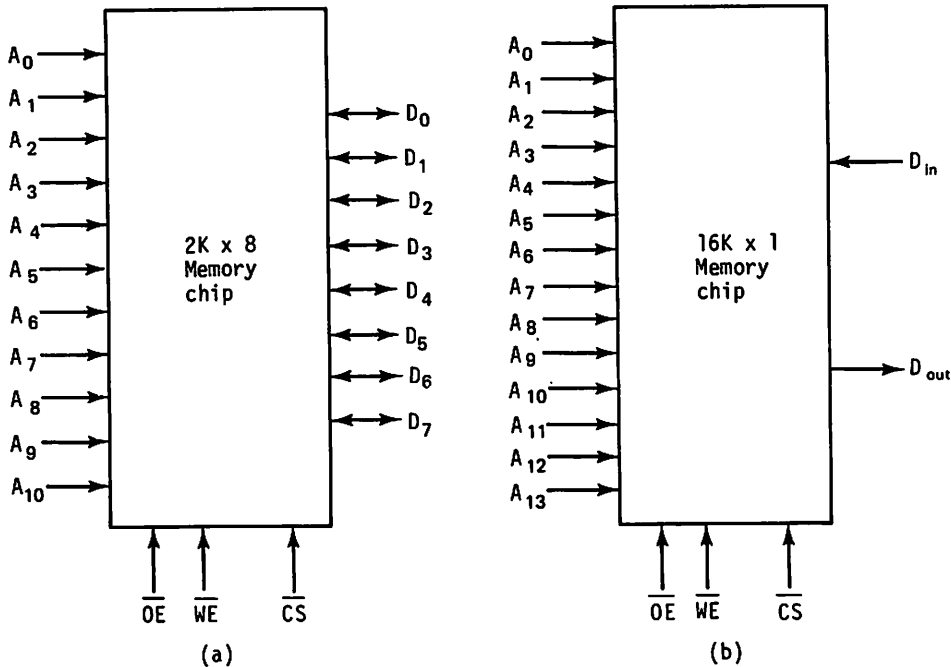


Fig. 3-30. Two ways of organizing a 16K-bit memory chip.

a photosensitive material through a mask containing the desired bit pattern and then etching away the exposed (or unexposed) surface. The only way to change the program in a ROM is to replace the entire chip.

ROMs are much cheaper than RAMs when ordered in large volumes to defray the cost of making the mask. However, they are inflexible, because they cannot be changed after manufacture and the turnaround time between placing an order and receiving the ROMs may be many weeks. To make it easier for companies to develop new ROM-based products, the **PROM** (Programmable ROM) was invented. This chip is like a ROM, except that it can be programmed (once) in the field, greatly reducing turnaround time.

The next development in this line was the **EPROM** (Erasable PROM), which can not only be field-programmed but also field-erased. When the quartz window in an EPROM is exposed to a strong ultraviolet light for 15 minutes, all the bits are set to 1. If many changes are expected during the design cycle, EPROMs are far more economical than PROMs because they can be reused.

Even better than the EPROM is the **EEPROM** (Electrically Erasable PROM), also called an **EAROM** (Electrically Alterable ROM), which can be erased by applying pulses to it instead of requiring it to put in a special chamber for exposure to ultraviolet light. EEPROMs differ from RAMs in that both writing a byte and erasing

a byte take thousands of times longer, although access times for reading ROMs, PROMs, EPROMs, EEPROMs, and RAMs are comparable (a few hundred nanoseconds at most).

3.4. MICROPROCESSORS AND MICROCOMPUTERS

Armed with information about SSI chips, MSI chips, and LSI memory chips, we can now tackle the main topic of this chapter—microprocessors and microcomputers. First, we will look at some general aspects of microprocessors as viewed from the digital logic level, including **pinout** (what the signals on the various pins mean) and buses. Then we will examine two popular microprocessors, the Zilog Z80 and the Motorola MC68000 in considerable detail.

3.4.1. Microprocessor Chips

For the purposes of this book we will use the term “microprocessor” to mean any CPU that is contained on a single chip and “microcomputer” to mean a system containing a microprocessor, memory, and some I/O equipment. With this definition it is clear that some microcomputers are more powerful than some minicomputers. Our definition is based on packaging, which makes it appropriate for the digital logic level we are now studying.

We have chosen to focus on CPUs contained on a single chip for a good reason: their interface with the rest of the system is well defined. A typical microprocessor chip has 40, 48, or 64 pins, through which all its communication with the outside world must take place. By understanding the function of all the pins, we can learn how the CPU interacts with the memory and I/O devices at the digital logic level. Although the following material specifically relates to microprocessors, the basic ideas, for example, how CPUs reference memory, how I/O devices are interfaced, and so on, also hold for minicomputers and to some extent for mainframes, albeit in slightly different form.

The pins on a microprocessor chip can be divided into three types: address, data, and control. Every CPU has an address space that it can read and write, commonly bytes numbered 0 to $2^n - 1$. Such a microprocessor needs n pins on which to transmit addresses to memory.

It also needs some pins to send data to memory and receive data from memory. Normally, the microprocessor will have as many data pins as it has bits in its word, in order to read or write a word in one memory cycle. Thus a microprocessor with an 8-bit word, 8-bit registers, and so on, will need eight data pins, whereas a microprocessor with a 16-bit word, 16-bit registers, and so on, usually has 16 data pins. Although it makes little sense to have more data pins than the word length, some microprocessors have only half as many pins as there are bits in a word. Consequently, they need two memory cycles to read or write a word. Although this slows down the machine, it also saves pins and makes the machine cheaper.

The control pins regulate the flow and timing of data to and from the microprocessor and have other miscellaneous uses. Their number and function vary considerably among microprocessors. Some pins regulate memory access, others deal with I/O, and some are needed for power, ground, and clock signals. We will describe the control pins in more detail below, after having described how the various components of a microcomputer are interconnected.

3.4.2. Microcomputer Buses

The components of a microcomputer, the microprocessor, memory, and I/O controller chips, are connected by a collection of parallel wires called a **bus**. The lines (wires) in the bus can be classified as address, data, or control. They correspond closely but not exactly, to the microprocessor's pins. Several buses have been standardized by IEEE and other organizations to facilitate interconnection of processors, memories, and other devices from different vendors (Boberg, 1980; Burr et al., 1979; Elmquist et al., 1979; Gilbert, 1982).

Figure 3-31 illustrates a simple microcomputer with a $2K \times 8$ EPROM for the program, a $2K \times 8$ RAM for data, and an I/O chip. In this figure, the 16 address lines (called the **address bus**), the 8 data lines (called the **data bus**), and the (unspecified number of) control lines (called the **control bus**) are shaded to indicate that more than one line is present.

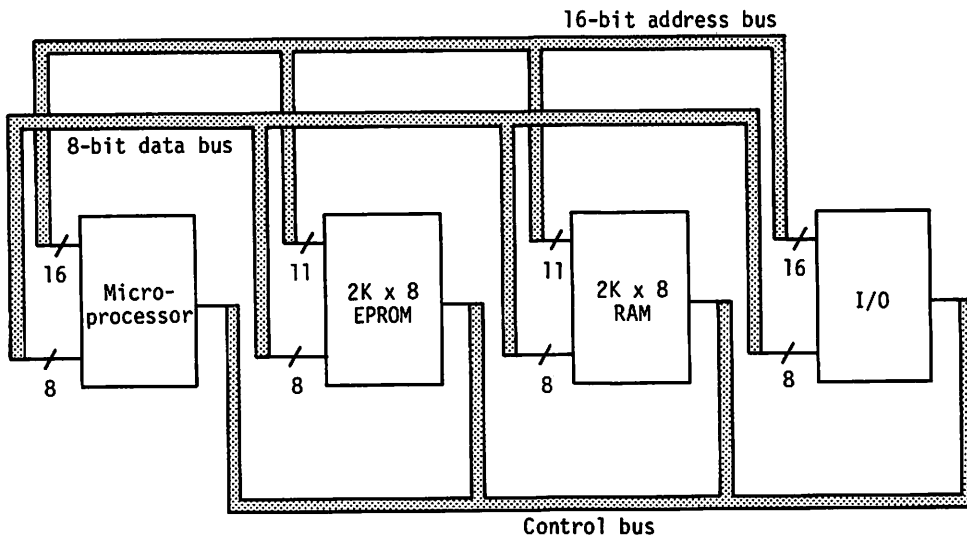


Fig. 3-31. A microcomputer with EPROM, RAM, and an I/O chip.

Another convention used in the figure is the short diagonal line with a number, telling how many bits are meant. For example, the microprocessor is connected to all

16 address lines, but the 2K EPROM only has 2048 bytes, so it only connects to 11 address lines, because $2048 = 2^{11}$. Which of the 16 lines are used and how the various chips are selected will be discussed later.

Communication between chips exhibits a master/slave relationship. One chip, the master, initiates a request to the other, the slave. Sometimes the slave replies, sometimes it just carries out the request silently. At any instant, at most one chip can be master, although many can be slaves. A typical bus transaction is the microprocessor, as master, requesting a byte from the RAM, as slave.

When a device is neither master nor slave, it must not interfere with transactions using the bus. In Fig. 3-28 we saw how using tri-state buffers on output lines could allow the chip to detach itself when not selected. In general, each chip in Fig. 3-31 that can output a signal onto a bus line will have an internal tri-state buffer to release the line when it is not being used, in order to allow other devices to use it. This process is called **floating** the bus. If one device tries to output 0 onto a bus line and another device tries to output a 1, one of the devices will burn out. Chips that OR their values onto the bus rather than trying to force the bus to their values do exist (open collector chips) but they have some drawbacks and are rarely used for microcomputer buses. Note that only *output* lines need be tristatable; a chip is free to look at its inputs all the time without causing conflicts. In Fig. 3-28, for example, the memory may read the address lines even when it is not selected, because it does no harm.

Just for the sake of completeness, we note that even if a device only reads a bus line, it does have an effect because it draws some current from the line. The chip driving the bus can output only a limited amount, which means that its **fanout** (the number of chips that can simultaneously read the output) is limited. For 74xx series TTL chips, the fanout is 10; for 74LSxx series it is 40; for MOS devices, such as microprocessor chips, there may be enough current to drive several other MOS chips but not enough to drive more than one or two standard 74xx TTL chips. High fanout can be achieved by putting tri-state buffers on the output lines, because they not only disconnect the chip when disabled but they normally provide enough current to drive 10 standard TTL chips when enabled.

Control Lines

When the microcomputer of Fig. 3-31 wants to read a byte from memory, it must output some information onto the buses. It puts the address it needs on the address bus and releases the data bus (using its tri-state buffers), but it must also indicate somehow that it is requesting a memory read. How it makes this request depends on the control lines available. One common arrangement is to have two control lines READ and WRITE. By setting the appropriate line to 1, it can tell the memory what to do.

Although we will study memory interfacing in detail later, for the time being we just note that normally a microcomputer has a combinational circuit attached to the address bus to determine which memory chip to select, based on the current address.

For example, if the EPROM is positioned from 0 to 2K in the address space, addresses containing all zeros in the high-order 5 bits should select the EPROM.

Many memory chips react when chip select is low rather than when it is high. To indicate that a signal is active (i.e., enabled) low instead of high, by convention a bar is drawn over its name. Thus a signal called READ is normally low (e.g., 0 to 1 volts), and goes high (e.g., 2 to 5 volts) to indicate that a read cycle is required. In contrast, a signal called $\overline{\text{READ}}$ would normally be high, and would be driven low by the microprocessor to initiate a read cycle. The choice between active high and active low is made by the designers of the microprocessor.

To avoid confusion, we will often use the term **asserted** to mean that a signal is in its active state, whatever that may be. Similarly, **negated** means that the signal is in its inactive (normal) state. Thus for a microprocessor with a pin called READ we would say: READ is asserted to start a memory cycle. For a different microprocessor with a pin called $\overline{\text{READ}}$, we would say: $\overline{\text{READ}}$ is asserted to start a memory cycle. For the former machine, asserted means set high; for the latter it means set low.

An alternative design to having signals (i.e., microprocessor pins and bus lines) called READ and WRITE is to have signals MEMREQ and R/W. The former is asserted when memory request is needed, and the latter is used to distinguish reads from writes.

Numerous other control lines are conceivable. Rather than trying to enumerate them in some abstract way, we have chosen to present two popular microprocessors, the Zilog Z80 and the Motorola MC68000, in some detail. These machines differ in many ways, and by the end of the discussion of both of them you should have a much better idea of what control lines are needed in a bus and why.

3.4.3. The Z80 Microprocessor

The Z80 is a complete 8-bit CPU in a 40-pin package. It has 158 instructions and can address 64K of memory. Its instructions are a superset of the Intel 8080 microprocessor, but the two chips are not pin compatible. We have chosen the Z80 as our first example because it is simple, powerful, and widely used in innumerable applications and products. Throughout our discussion we will use the term "Z80" to mean not only the original Z80 but also the Z80A, Z80B, and so on, which are functionally identical but faster.

Z80 Pinout

The Z80 has 16 address lines, 8 data lines, 13 control lines, power, ground, and clock input, as shown in Fig. 3-32. Both the address lines and the data lines are asserted high, but all the control signals are asserted low, as indicated by the bars. All the address and data lines and some of the control lines can be floated to allow other devices to become bus master. Figure 3-32 is a logical rather than a physical pinout; the actual signals are not grouped together so conveniently.

The $\overline{\text{MREQ}}$, $\overline{\text{IORQ}}$, $\overline{\text{RD}}$, $\overline{\text{WR}}$, and $\overline{\text{WAIT}}$ lines are for reading and writing memory and

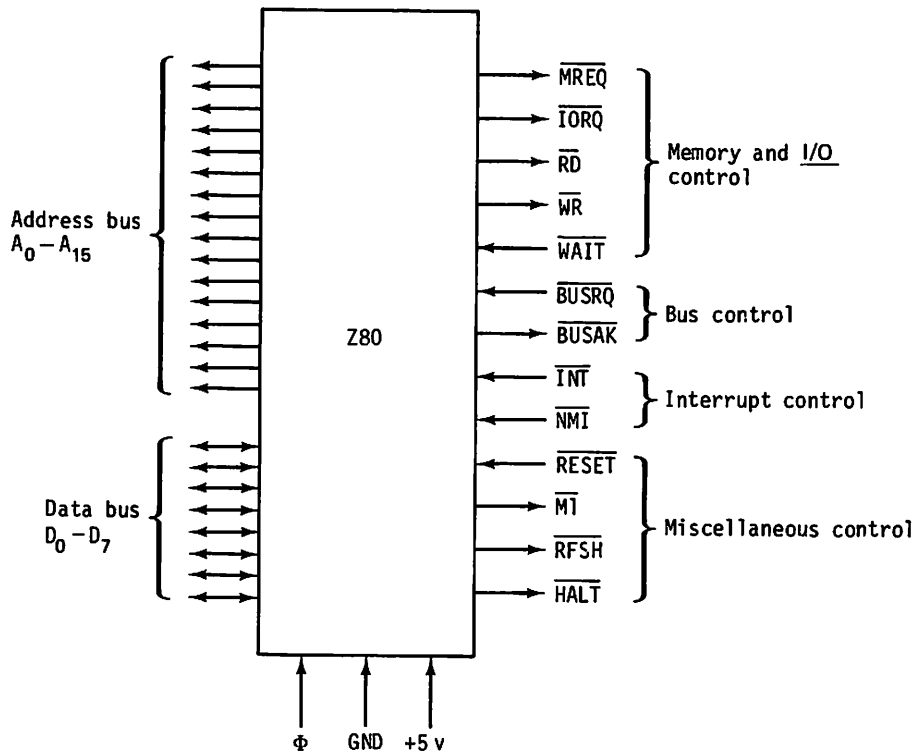


Fig. 3-32. The Z80 address, data, and control signals. The arrows designate input, output, and bidirectional lines.

I/O devices. To read from memory, the Z80 puts the desired address on the address lines and then asserts both $\overline{MREQ}$ and $\overline{RD}$. After waiting a certain period of time for the memory to respond, the Z80 then strobes (loads) the byte in from the data lines. A write consists of setting up both the address and data lines, then asserting both $\overline{MREQ}$ and $\overline{WR}$.

To read or write an I/O device, the same procedure is followed, except that $\overline{IORQ}$ is used instead of $\overline{MREQ}$. Thus memory and the I/O devices occupy two distinct address spaces, with the choice between $\overline{MREQ}$ and $\overline{IORQ}$ indicating which one is being referenced. Most memory chips have a chip select line that is asserted low, so $\overline{MREQ}$ can be used to drive it, possibly ANDed with the output of an address comparator. By wiring $\overline{MREQ}$ to the chip select lines of the memory chips and $\overline{IORQ}$ to the chip select lines of the I/O chips, no conflict arises when the same address corresponds to both memory and an I/O device.

The Z80 expects the memory to put the requested byte on the data lines within a certain fixed time. However, the Z80 designers made a provision to allow the use of

memories too slow to meet the basic requirements. These slow memories assert $\overline{\text{WAIT}}$ after being selected and keep it asserted until they have placed the data on the bus. The assertion of this signal tells the Z80 to just wait until it is negated.

In many microcomputers I/O devices can also become bus master. When a device other than the CPU is bus master, the CPU must refrain from using the bus—in effect do nothing—until the other device is finished. To achieve the necessary synchronization, the signals $\overline{\text{BUSRQ}}$ and $\overline{\text{BUSAk}}$ are provided. To request permission to become bus master, an I/O device asserts $\overline{\text{BUSRQ}}$. When the CPU is finished with its current cycle, it floats the address lines, data lines, and those control lines that an I/O device might need ($\overline{\text{MREQ}}$, $\overline{\text{IORQ}}$, $\overline{\text{RD}}$, and $\overline{\text{WR}}$) and then asserts $\overline{\text{BUSAk}}$ to grant permission to use the bus. Recognition and response to $\overline{\text{BUSRQ}}$ is done entirely in hardware; the Z80 programmer is completely unaware of it. If multiple I/O devices are competing for the bus, external arbitration logic is needed.

An I/O device may need to signal the CPU, for example, to announce that a data transfer has been completed. To do this, it only needs to assert $\overline{\text{INT}}$, which causes a program interrupt at the end of the current instruction, if interrupts are currently enabled. $\overline{\text{NMI}}$ (NonMaskable Interrupt) can also be used for causing interrupts. However, unlike $\overline{\text{INT}}$, it cannot be disabled in software. It is therefore typically used to report errors or potentially disastrous external conditions.

$\overline{\text{RESET}}$ is an external signal used to reset the internal status of the CPU and make it jump to location 0. It is normally used to start the microcomputer after the power has come on, or to reboot after a crash. Typically, $\overline{\text{RESET}}$ is just connected to a button on the front panel.

$\overline{\text{MI}}$ is output by the Z80 at the start of each instruction read. It is provided for compatibility with the 8080, where it plays an important role, and to allow other chips to distinguish between opcode reads and data reads, which have different properties. (The opcode is the first byte of the instruction; it identifies the operation to be performed.) In particular, opcode reads are longer than data reads to allow time for dynamic memories to be refreshed. During the second half of an opcode read cycle, the Z80 asserts $\overline{\text{RFSH}}$ and puts a 7-bit counter on the address lines to help keep track of which part of the memory is due for refreshing.

Because the Z80 plays an important role in refreshing the dynamic RAMs, it never stops, not even after a HALT instruction. Instead, it just keeps asserting $\overline{\text{RFSH}}$ while waiting for an interrupt. After a HALT instruction has been executed, however, the HALT line is asserted to tell the rest of the system that the CPU is “stopped” waiting for an interrupt.

The three remaining lines are power, ground, and clock (Φ). The Z80 is powered by a single 5-volt supply, and draws 1.5 watts. The clock is a 5-volt square wave at any frequency up to some maximum (2.5, 4, or 6 MHz) depending on the chip version. Because each instruction takes a specific number of clock cycles, the CPU speed is directly proportional to the clock frequency. Unlike many microprocessors, the Z80 has no lower limit on the clock frequency. Furthermore, successive clock periods need not be equally long. The Φ input can even be driven by a toggle switch to allow a system designer to step through a program bus cycle by bus cycle.

Z80 Memory Request Timing

When the Z80 needs to read a byte from memory, it puts the address of the byte on the address lines and then asserts $\overline{MREQ}$ and $\overline{RD}$, as mentioned above. In this section we are going to look at the precise timing of the various signals in detail because being able to read timing diagrams is essential to understanding how memory chips and I/O chips are interfaced to microprocessors.

A typical Z80 instruction is LD A,7, which loads the constant 7 into a Z80 register, the accumulator. The instruction occupies two bytes in memory, one for the opcode and one for the constant (7) to be loaded. Figure 3-33 shows the basic timing for this instruction. The instruction requires two machine cycles for its execution: one to read the opcode byte and one to read the operand byte. The opcode read takes four clock periods (which includes asserting signals for refreshing dynamic RAMs) and the operand read takes three clock periods. If the clock frequency is, say, 4 MHz, each clock period will be 250 nsec, so the full instruction execution will take 1750 nsec. Note that the rising and falling edges of the clock have not been drawn straight up and down, because no electrical signal can change in zero time.

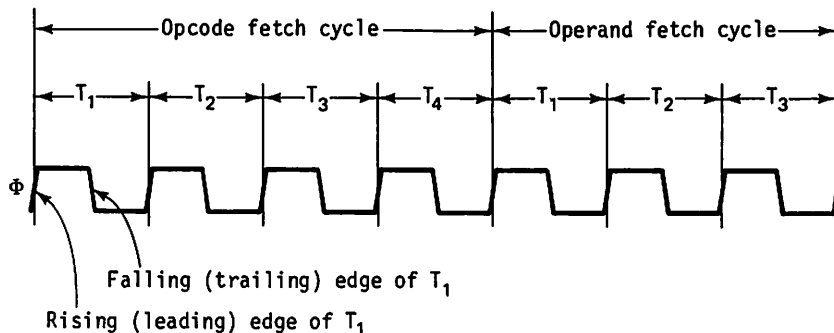
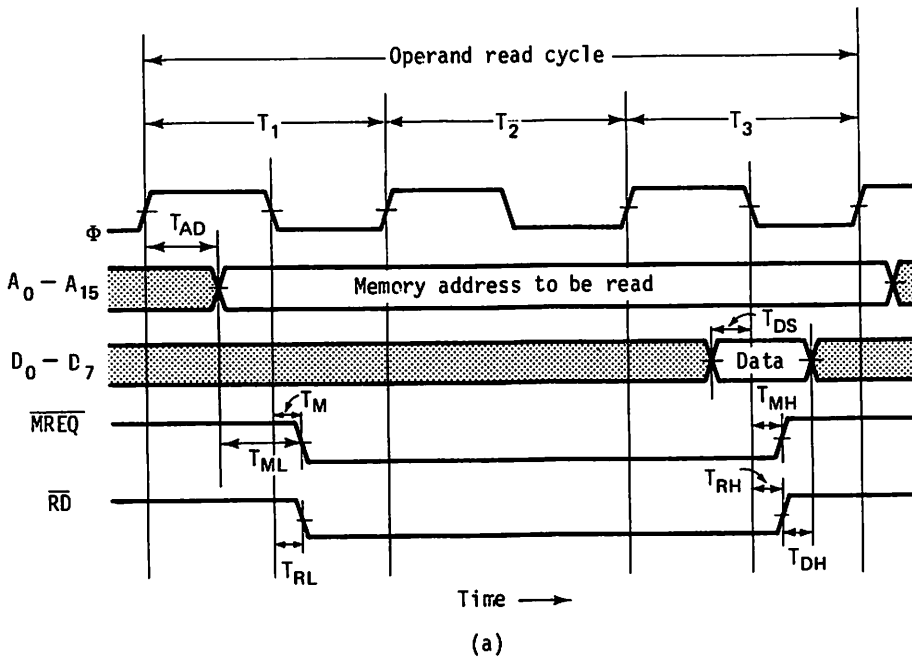


Fig. 3-33. Timing for LD A,7.

Figure 3-34(a) shows the operand read cycle of the load instruction in more detail, although slightly simplified from that of a real Z80. The clock, address, data, $\overline{MREQ}$, and $\overline{RD}$ lines are all shown on the same time scale. The start of T_1 is defined by the rising edge of the clock. Part way through T_1 the Z80 puts the address of the byte it wants on the address lines, A_0 to A_{15} . Because the address is not a single value, like the clock, we cannot show it as a single line in the figure; instead, it is shown as two lines, with a crossing at the time that the address changes. Furthermore, the shading prior to the crossing indicates that the shaded value is not important. Using the same shading convention, we see that the contents of the data lines are not significant until well into T_3 .

After the address lines have had a chance to settle down to their new values, $\overline{MREQ}$ and $\overline{RD}$ are asserted. Nothing happens during T_2 , to give the memory time to decode the address and put the data on the bus. At the falling edge of T_3 the Z80



Symbol	Parameter	Min	Max	Unit
T_{AD}	Address output delay		110	nsec
T_{ML}	Address stable prior to $\overline{MREQ}$	60		nsec
T_M	$\overline{MREQ}$ delay from falling edge of Φ in T_1		85	nsec
T_{RL}	$\overline{RD}$ delay from falling edge of Φ in T_1		85	nsec
T_{DS}	Data setup time prior to falling edge of Φ	50		nsec
T_{MH}	$\overline{MREQ}$ delay from falling edge of Φ in T_3		85	nsec
T_{RH}	$\overline{RD}$ delay from falling edge of Φ in T_3		85	nsec
T_{DH}	Data hold time from negation of $\overline{RD}$	0		nsec

(b)

Fig. 3-34. (a) Z80 operand read timing. (b) Specification of some critical times.

strokes the data lines, latching (i.e., storing) the value in an internal register. Having read the data, the Z80 negates $\overline{MREQ}$ and $\overline{RD}$. If need be, another memory cycle can begin at the next rising edge of the clock.

In the timing specification of Fig. 3-34(b), eight symbols that occur in the timing diagram are further clarified. T_{AD} , for example, is the time interval between the rising edge of the T_1 clock and the address lines being set. According to the timing specification, $T_{AD} \leq 110$ nsec. (For this example we assume a 250 nsec clock period, with 10 nsec rise and fall times.) This means that the chip manufacturer guarantees that during any operand read cycle, the Z80 will output the address to be read within 110 nsec of the T_1 rising clock.

The timing specifications also require that the data be available on the data lines at least 50 nsec before the falling edge of T_3 , to give it time to settle down before the Z80 strobes (i.e., reads) it in. The combination of the constraints on T_{AD} and T_{DS} means that in the worst case, the memory will have only $250 + 250 + 125 - 110 - 50 = 465$ nsec from the time the address appears until it must produce the data. If memory cannot respond fast enough, it must assert $\overline{WAIT}$ prior to the falling edge of T_2 when $\overline{WAIT}$ is strobed.

The timing specification further guarantees that the address will be set up at least 60 nsec prior to $\overline{MREQ}$ being asserted. This time can be important if $\overline{MREQ}$ drives chip select on the memory chip because some memories require an address setup time prior to chip select. Clearly, the microcomputer designer should not choose a memory chip with a 75-nsec setup time.

The constraint on T_M and T_{RL} mean that $\overline{MREQ}$ and $\overline{RD}$ will both be asserted within 85 nsec from the T_1 falling clock. In the worst case, the memory chip will have only $250 + 250 - 85 - 50 = 365$ nsec after the assertion of $\overline{MREQ}$ and $\overline{RD}$ to get its data onto the bus. This constraint is in addition to the address constraint.

T_{MH} and T_{RH} tell how long it takes $\overline{MREQ}$ and $\overline{RD}$ to be negated after the data have been strobed in. Finally, T_{DH} tells how long the memory must hold the data on the bus after $\overline{RD}$ has been negated. As far as the Z80 is concerned, the memory can remove the data from the bus as soon as $\overline{RD}$ has been negated; on some other microprocessors, the data must be kept stable for a little while after a request.

We would like to point out that Fig. 3-34 is a simplified version of the true Z80 timing. In reality, 49 different critical times are specified (not just for read, however). Nevertheless, Fig. 3-34 gives a good flavor for microprocessor timing requirements. Figures like this one are part of every microprocessor product specification and that of other chips as well.

3.4.4. The 68000 Microprocessor

As a second example of a microprocessor chip, we will now look at the Motorola MC68000 (Stritter and Gunter, 1979), or just 68000 for short. By any measure, the 68000 is more powerful (and more expensive) than the Z80. It can address 16M bytes of memory (versus 64K for the Z80), read and write data 16 bits at a time (versus 8 for the Z80) and is much faster. It also has many instructions that operate on 8-, 16-, or 32-bit quantities (versus mostly 8-bit and a few 16-bit instructions for the Z80). The 68000 is really a family of CPUs including the 68010, 68020, and so on, but the differences between the members will not concern us here.

In fact, from the programmer's point of view, it can be argued that the 68000 is a 32-bit word machine because it has 32-bit registers and so many instructions to use them. However, as seen from the digital logic level, it appears to be a 16-bit word machine because it reads and writes memory in 16-bit units. (A good way to see if a computer scientist is a hardware or software person is to ask if the 68000 is a 16-bit machine or a 32-bit machine.) If one were to categorize machines as micros, minis, or mainframes based on their architecture, the 68000 would be a large mini or a small mainframe.

68000 Pinout

The 68000 has 23 address lines, 16 data lines, 20 control lines, power ($2 \times$), ground ($2 \times$), and clock, as shown in Fig. 3-35. Addresses are actually 24 bits but because word addresses must be even, bit A_0 is always 0, and is not output. The 16 data lines are normally used to read or write 16-bit words but provision has also been made for reading and writing bytes. Reading and writing 32-bit quantities requires two memory cycles.

Memory reads and writes are done differently on the 68000 than on the Z80. To read a word, the 68000 puts the address on the bus, sets $R/\overline{W}$ (high for a read and low for a write), and then asserts $\overline{AS}$ (Address Strobe). This scheme is more efficient than the Z80's, because only two lines are needed ($\overline{AS}$, and $R/\overline{W}$), instead of three ($\overline{MREQ}$, $\overline{RD}$, and $\overline{WR}$). Unlike the Z80, the 68000 does not have a separate address space for I/O devices, so there is no signal analogous to $\overline{IORQ}$.

The $\overline{UDS}$ (Upper Data Strobe) and $\overline{LDS}$ (Lower Data Strobe) signals distinguish byte operations from word operations. If only one of them is asserted, only the corresponding byte is read or written. If both are asserted, both bytes (i.e., a full 16-bit word) are read or written.

In addition to the differences with the Z80 just described, there is also a more fundamental difference. On a read, the Z80 expects the memory to put the data on the bus 50 nsec before the falling edge of T_3 , or failing to do so, to assert $\overline{WAIT}$ during T_2 . Because memory transfers are closely tied to the clock they are called **synchronous**. The 68000 has no such expectations. The memory can take as long as it wants to. When it is finished, the memory just asserts $\overline{DTACK}$ to say it is done. Similarly, on a write, the memory just asserts $\overline{DTACK}$ when it has latched the data. The 68000 is **asynchronous**. A system like the 68000 that asserts a signal and then waits for another signal to be asserted as a reply is said to use **handshaking**.

The $\overline{BR}$ (Bus Request) line is used by I/O devices to request the bus. The $\overline{BG}$ (Bus Grant) line is asserted by the 68000 to announce that the bus has indeed been floated. This protocol is the same as on the Z80. A difference with the Z80, however, is the presence of the $\overline{BGACK}$ line, which is asserted by the requesting device after it sees the bus grant. In principle, a device could request and be granted the bus and then assert $\overline{BGACK}$. At this point $\overline{BR}$ and $\overline{BG}$ are available for another device to request and be granted the bus while the previous device was still using it for transferring data. The second device could not, of course, actually start using the bus until the first one indi-

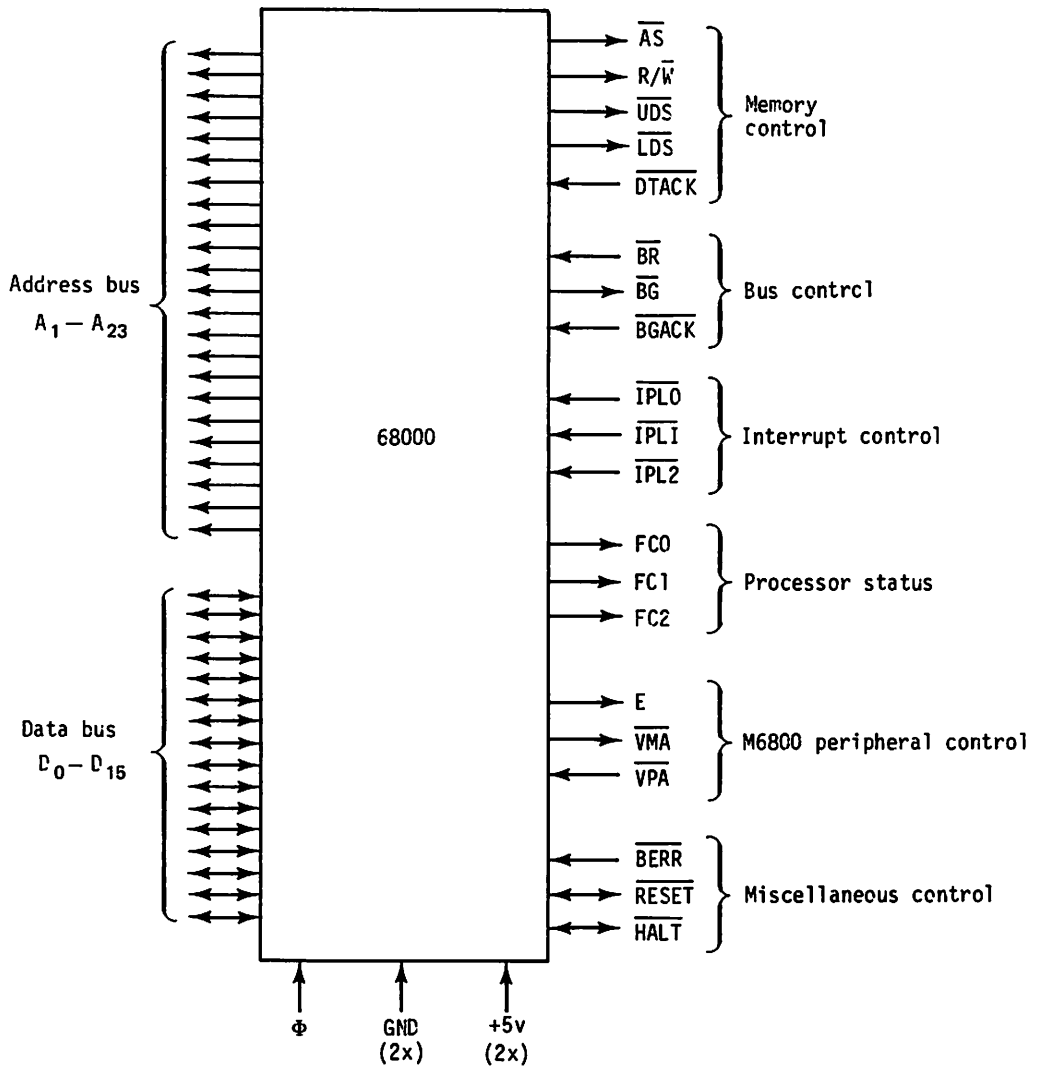


Fig. 3-35. The 68000 address, data and control signals. The arrows designate input, output, and bidirectional lines.

cated that it was finished by negating $\overline{BGACK}$. By allowing the bus grant protocol to run in parallel with data transfer, bus cycles are saved.

The three $\overline{IPL}$ pins are used to encode the priority of an I/O device requesting an interrupt. If the device has a higher priority than the CPU currently has, the interrupt occurs; otherwise, it must wait. This technique is more flexible than on the Z80, where interrupts are either enabled or disabled.

The three $\overline{FC}$ lines are used by the 68000 to output certain status information. $\overline{E}$, $\overline{VMA}$, and $\overline{VPA}$ are signals present on the M6800 microprocessor chip (a predecessor of the 68000), and are provided here to allow the 68000 to use peripheral chips designed for the M6800, which is synchronous, like the Z80.

$\overline{BERR}$ (Bus ERRor) is used to inform the 68000 that something is wrong, for example, a device does not respond. If no such signal was available, an attempt to read from nonexistent memory would cause the 68000 to hang forever waiting for $\overline{DTACK}$. Typically, an external timer asserts $\overline{BERR}$ if no response to a read or write request is forthcoming within a reasonable interval. The Z80 does not need such protection because an attempt to read nonexistent memory will just cause it to strobe in nonexistent data in T_3 but not to hang.

$\overline{RESET}$ can be used to reset the CPU, as on the Z80 but it can also be asserted under program control to reset I/O devices. $\overline{HALT}$ is also a bidirectional line. As an output, it indicates that the 68000 has stopped, just as on the Z80. As an input, it causes the CPU to stop after the current bus cycle, to allow programs to be run one bus cycle at a time. On the Z80 single cycling is accomplished by slowing down the clock, something the 68000 cannot tolerate.

Power, ground, and the clock serve the same function as on the Z80. The only difference is that power and ground are each needed on two pins. If future versions of the chip need two more pins, these two can always be reclaimed later.

3.5. INTERFACING

A microcomputer system consists of a microprocessor, some memory chips, and some I/O chips connected in such a way that they work together. We have already studied memories and microprocessors in some detail. After a brief introduction to I/O chips, we will see how the pieces can be put together, concluding with an examination of a small but complete microcomputer.

3.5.1. I/O Chips

Numerous I/O chips are already available and new ones are being introduced all the time. Common chips include UARTs, USARTs, CRT controllers, floppy disk controllers, and PIOs. A **UART** (Universal Asynchronous Receiver Transmitter) is a chip that can read a byte from the data bus and output it a bit at a time on a serial line for a terminal, or input data from a terminal. UARTs usually allow speeds from 50 bps to 19.2 kbps; character widths from 5 to 8 bits; 1, 1.5, or 2 stop bits; and provide even, odd, or no parity, all under program control. **USARTs** (Universal Synchronous Asynchronous Receiver Transmitters) can handle synchronous transmission using a variety of protocols as well as performing all the UART functions. As an example of how I/O chips work and are interfaced, we will now look in some detail at parallel I/O.

PIO Chips

A typical **PIO** (Parallel Input/Output) chip is the Intel 8255A, shown in Fig. 3-36. It has 24 I/O lines that can interface to any TTL-compatible device, for example, keyboards, switches, lights, printers, disk controllers, or paper tape readers. In a nutshell, the CPU program can write a 0 or 1 to any line, or read the input status of any line, providing great flexibility. A small microprocessor-based system using a PIO can often replace a complete board full of SSI or MSI chips.

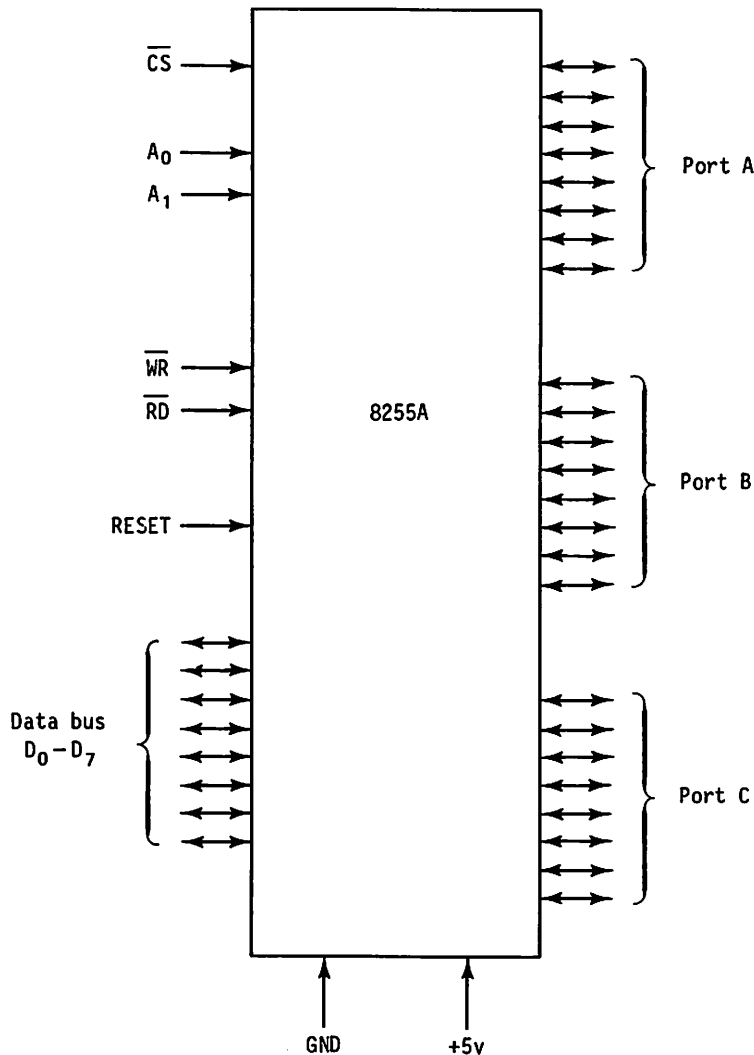


Fig. 3-36. An 8255A PIO chip.

Although the CPU can configure the 8255A in many ways by loading status registers within the chip, we will concentrate on some of the simpler modes of operation. The simplest way of using the 8255A is as three independent 8-bit ports, A, B, and C. Associated with each port is an 8-bit latch register. To set the lines on a port, the CPU just writes an 8-bit number into the corresponding register, and the 8-bit number appears on the output lines and stays there until the register is rewritten. To use a port for input, the CPU just reads the corresponding register.

Other operating modes provide for handshaking with external devices. For example, to output to a device that is not always ready to accept data, the 8255A can present data on an output port and wait for the device to send a pulse back saying that it has accepted the data and wants more. The necessary logic for latching such pulses and making them available to the CPU is included in the 8255A but we will not consider it further.

From the functional diagram of the 8255A we can see that in addition to 24 pins for the three ports, it has eight lines that connect directly to the data bus, a chip select line, read and write lines (that correspond exactly to the signals of the same name on the Intel 8080 and Zilog Z80), two address lines, and a line for resetting the chip. The two address lines select one of the four internal registers, corresponding to ports A, B, C, and the control register, which has bits determining which ports are for input and which for output, and other functions. Normally, the two address lines are connected to the low-order bits of the address bus.

3.5.2. Address Decoding

Up until now we have been deliberately vague about how chip select is asserted on the memory and I/O chips we have looked at. It is now time to look more carefully at how this is done. Let us consider a simple microcomputer consisting of a Z80 CPU, a $2K \times 8$ EPROM for the program, a $2K \times 8$ RAM for the data, and a PIO. This four-chip (so far) system might be used as a prototype for the brain of a toy or appliance. Once it went into production, the EPROM would normally be replaced by a ROM.

The PIO can be selected in one of two ways, as a true I/O device or as part of memory. If we choose to use it as an I/O device, then we must select it using $\overline{IORQ}$. If we use the other approach, **memory-mapped I/O**, then we must assign it four bytes of the address space for the three ports and the control register. The choice is somewhat arbitrary. We will choose memory-mapped I/O because it illustrates some issues in address decoding.

The EPROM needs 2K of address space, the RAM also needs 2K of address space, and the PIO needs 4 bytes. Because the Z80 address space is 64K, we must make a choice about where to put the three devices. One possible choice is shown in Fig. 3-37. The EPROM occupies addresses 000000 to 003777, the RAM occupies addresses 100000 to 103777, and the PIO occupies the highest four bytes of the address space, 177774 to 177777 (all addresses are in octal). From the programmer's point of view, it makes no difference which addresses are used; however, for

interfacing it does matter. If we had chosen to address the PIO via the I/O space, it would not need any memory addresses (but it would need four I/O space addresses).

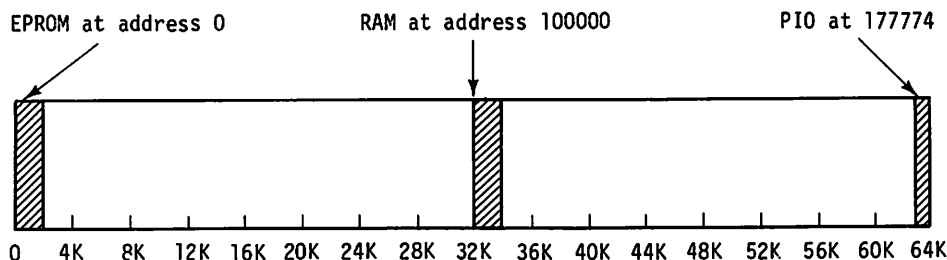


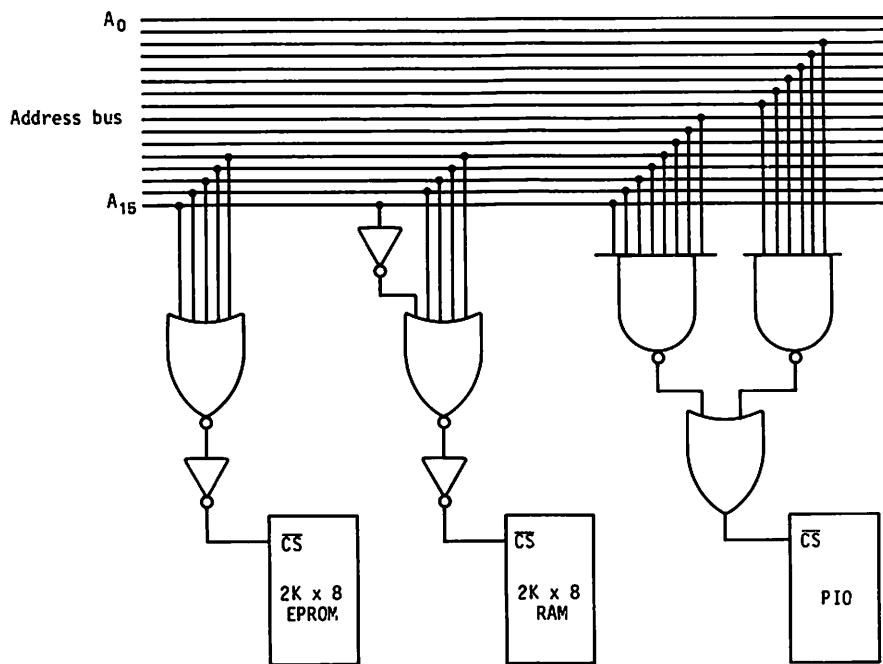
Fig. 3-37. Location of the EPROM, RAM, and PIO in the Z80's 64K address space. The upper addresses are in octal, the lower ones are in decimal.

With the address assignments of Fig. 3-37, the EPROM should be selected by any 16-bit memory address of the form 00000xxxxxxxxx. In other words, any address whose 5 high-order bits are all 0s falls in the bottom 2K of memory, hence in the EPROM. Thus the EPROM's chip select could be wired to a 5-bit comparator one of whose inputs was permanently wired to 00000.

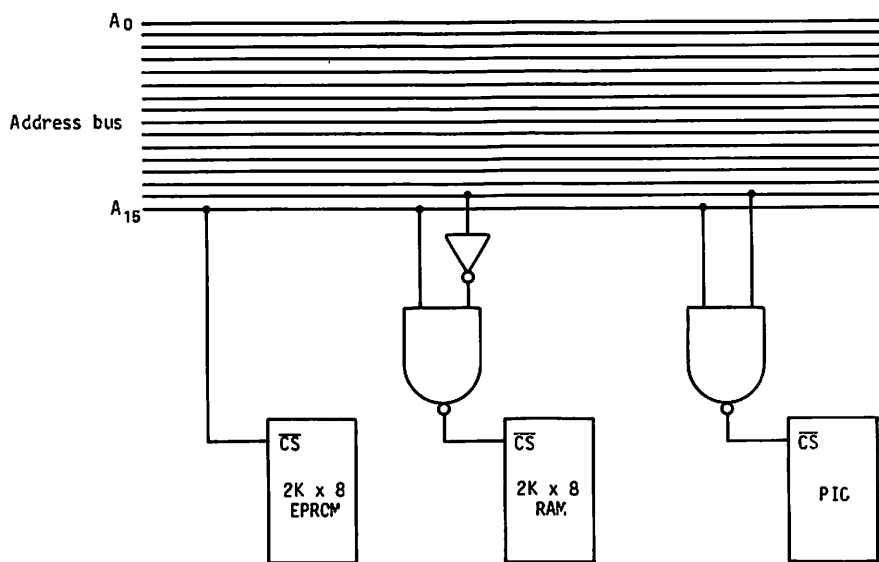
A better way to achieve the same effect is to use a five-input OR gate, with the five inputs attached to address lines A_{11} to A_{15} . If and only if all five lines are 0 will the output be 0, thus asserting $\overline{CS}$ (which is asserted low). Unfortunately, no five-input OR gate exists in the standard SSI series. The closest we can come is an eight-input NOR gate. By grounding three inputs and inverting the output we can nevertheless produce the correct signal, as shown in Fig. 3-38(a). By convention, unused inputs are not shown in circuit diagrams.

The same principle can be used for the RAM. However, the RAM should respond to binary addresses of the form 10000xxxxxxxxx, so an additional inverter is needed as shown in the figure. The PIO address decoding is more complicated, because it is selected by the four addresses of the form 111111111111xx. A possible circuit that asserts $\overline{CS}$ only when the correct address appears on the address bus is shown in the figure. It uses two eight-input NAND gates to feed an OR gate. To build the address decoding logic of Fig. 3-38(a) using SSI requires six chips—the four eight-input chips, an OR gate, and a chip with three inverters. The address decoding requires more chips than the CPU, memory, and I/O device combined!

However, if the computer really consists of only the CPU, two memory chips, and the PIO, we can use a trick to simplify greatly the address decoding. The trick is based on the fact that all EPROM addresses, and only EPROM addresses, have a 0 in the high-order bit, A_{15} . Therefore, we can just wire $\overline{CS}$ to A_{15} directly, as shown in Fig. 3-38(b). At this point the decision to put the RAM at 100000 (octal) may seem much less arbitrary. The RAM decoding can be done by noting that the only valid addresses of the form 10xxxxxxxxxxxx are in the RAM, so 2 bits of decoding is



(a)



(b)

Fig. 3-38. (a) Full address decoding. (b) Partial address decoding.

sufficient. Similarly, any address starting with 11 must be a PIO address. The complete decoding logic is now two NAND gates and an inverter. Because an inverter can be made from a NAND gate by just tying the two inputs together, a single quad NAND chip is now more than sufficient.

The address decoding logic of Fig. 3-38(b) is called **partial address decoding**, because the full addresses are not used. It has the property that a read from addresses 004000, 010000, or 014000 will give the same result. In fact, every address in the bottom half of the address space will select the EPROM. Because the extra addresses are not used, no harm is done, but if one is designing a computer that may be expanded in the future (an unlikely occurrence in a toy), partial decoding should be avoided because it ties up too much address space.

Another common address decoding technique is to use a decoder chip, such as that shown in Fig. 3-14. By connecting the three inputs to the three high-order address lines, we get eight outputs, corresponding to addresses in the first 8K, second 8K, and so on. For a computer with eight RAMs, each $8K \times 8$, one such chip provides the complete decoding. For a computer with eight $2K \times 8$ memory chips, a single decoder is also sufficient, provided that the memory chips are each located in distinct 8K chunks of address space. (Remember our earlier remark that the position of the memory and I/O chips within the address space matters.)

3.5.3. An Example Microcomputer

As the culmination of our study of the digital logic level, we will now combine all the pieces to get a small but complete microcomputer. Our six-chip system is shown in Fig. 3-39. It consists of a Z80 CPU, a $2K \times 8$ EPROM, a $2K \times 8$ static RAM, a PIO chip, a quad NAND gate chip for partial address decoding, the same as in Fig. 3-38(b), and a three-input NAND gate for driving the PIO's chip select pin. We have already studied the Z80 (Fig. 3-32), the memory chips [Fig. 3-30(a)], and the PIO (Fig. 3-36). All that is left to do is look at the interconnection. In fact, we have even seen an outline of that in Fig. 3-31. (A slightly simpler version of the computer using the I/O space instead of memory mapping for the PIO is the subject of Prob. 26.)

Let us first look at the CPU. The eight data lines are fed to the other three LSI chips in a straightforward way via the data bus, so we have not shown the detail here. Thirteen of the address lines are also used. The other three lines, A_{11} to A_{13} , are not used. Similarly, $\overline{MI}$, $\overline{IORQ}$, $\overline{RFSH}$, $\overline{HALT}$, and $\overline{BUSA\overline{K}}$ are all output signals that are not used, so no connection to them is needed. The four input signals that are not used, $\overline{INT}$, $\overline{NMI}$, $\overline{WAIT}$, and $\overline{BUSRQ}$ are wired high to keep them negated.

$\overline{MREQ}$ is used to enable the output lines on both memory chips. Note that the two memory chips only generate output when both $\overline{CS}$ and $\overline{OE}$ are asserted. $\overline{WR}$ and $\overline{RD}$ are needed for the corresponding PIO signals, and $\overline{WR}$ is also needed for the RAM. Because the EPROM cannot be written into while sitting in the computer, it does not need a signal to distinguish read from write.

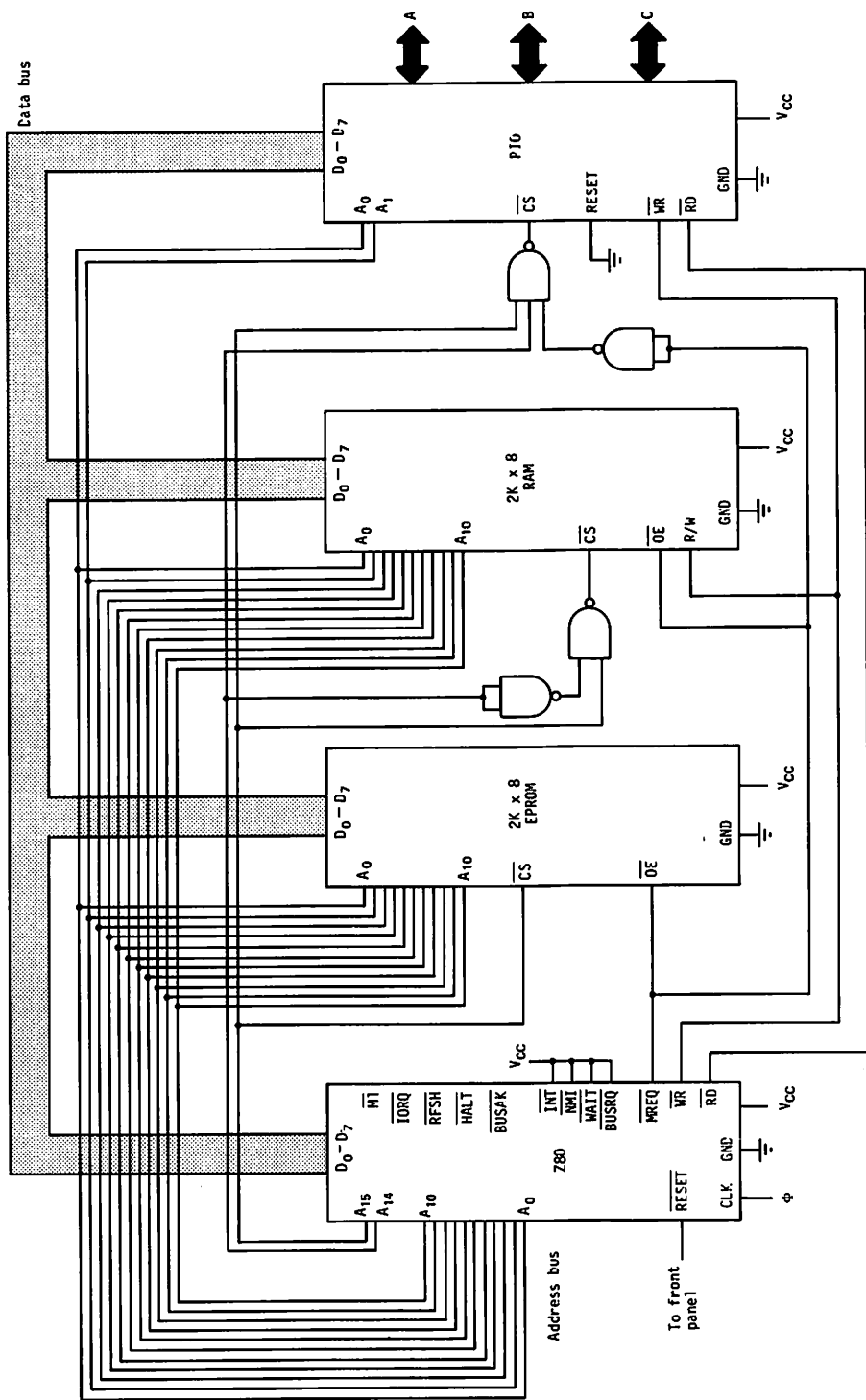


Fig. 3-39. A Z80-based microcomputer.

The **RESET** signal is wired to a button on the front panel to allow the microcomputer to be reset manually.

To make our microcomputer actually work, two more items are needed: a power supply and a clock. Depending on the specific chips chosen, the microcomputer will consume about 3.5 watts of power. For the clock, any square wave source up to the maximum frequency of the Z80 chip chosen will do.

We have come a long way from the basic transistor inverter to a complete computer system. Hopefully it is clear by now that there is nothing mysterious about how a computer works. We have described the operation of static RAMs down to the gate level. In the next chapter we will examine CPUs down to the gate level as well.

3.6. SUMMARY

Computers are constructed from integrated circuit chips containing tiny switching elements called gates. The most common gates are AND, OR, NAND, NOR, and NOT. Simple circuits can be built up by directly combining individual gates using SSI chips.

More complex circuits are constructed using standard MSI components such as multiplexers, demultiplexers, encoders, decoders, shifters, and ALUs. Arbitrary Boolean functions can be programmed using a PLA. If many Boolean functions are needed, PLAs are much more efficient (i.e., require fewer chips), than using SSI chips.

The components of (static) memories are latches and flip-flops, each of which can store 1-bit of information. These can be combined linearly into octal latches and flip-flops or logarithmically into full-scale word-oriented memories. Memories are available as RAM, ROM, PROM, EPROM, and EEPROM. Static memories need not be refreshed; they keep their values as long as the power remains on. Dynamic memories, on the other hand, must be refreshed periodically to compensate for leakage from the little capacitors on the chip.

The components of a computer system are connected by buses. Most of the pins on a typical microprocessor chip directly drive one bus line. The bus lines can be divided into address, data, and control lines. The Z80 and 68000 were discussed as examples, and a small but complete, Z80 system was given as an example and described in detail. The Z80 system contained the microprocessor chip, an EPROM chip for the program, a RAM chip for the data, and a PIO chip for interfacing with the outside world. In addition, two SSI chips were needed for address decoding.

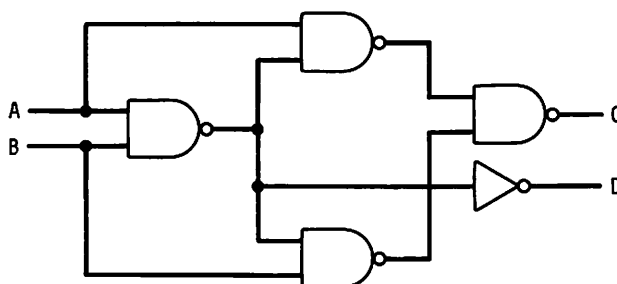
PROBLEMS

1. A logician drives into a drive-in restaurant and says, "I want a hamburger or a hot dog and french fries." The cook flunked out of sixth grade and doesn't know (or care) whether "and" has precedence over "or." As far as he is concerned, one interpretation is as good

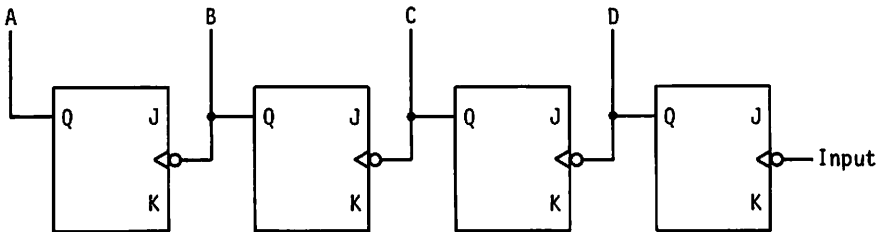
as the other. Which of the following cases are valid interpretations of the order? (Note that in English “or” means “exclusive or.”)

- a. Just a hamburger.
 - b. Just a hot dog.
 - c. Just french fries.
 - d. A hot dog and french fries.
 - e. A hamburger and french fries.
 - f. A hot dog and a hamburger.
 - g. All three.
 - h. Nothing—the logician goes hungry for being a wiseguy.
2. A missionary lost in Southern California stops at a fork in the road. He knows that two motorcycle gangs inhabit the area, one of which always tells the truth and one of which always lies. He wants to know which road leads to Disneyland. What question should he ask?
 3. There exist four Boolean functions of a single variable and 16 functions of two variables. How many functions of three variables are there? Of n variables?
 4. Use a truth table to show that $P = (P \text{ AND } Q) \text{ OR } (P \text{ AND NOT } Q)$.
 5. Use de Morgan’s law to find the complement of $A\bar{B}$.
 6. Using the three-variable multiplexer chip of Fig. 3-13, implement a function whose output is the parity of the inputs, that is, the output is 1 if and only if an odd number of inputs are 1.
 7. Put on your thinking cap. The three-variable multiplexer chip of Fig. 3-13 is actually capable of computing an arbitrary function of *four* Boolean variables. Describe how, and as an example, draw the logic diagram for the function that is 0 if the English word for the truth table row has an even number of letters, 1 if it has an odd number of letters (e.g., 0000 = zero = four letters $\rightarrow$ 0; 0111 = seven = five letters $\rightarrow$ 1; 1101 = thirteen = eight letters $\rightarrow$ 0). *Hint:* If we call the fourth input variable D , the eight input lines may be wired to V_{cc} , ground, D , or $\bar{D}$.
 8. Draw the logic diagram of a 2-bit demultiplexer, a circuit whose single input line is steered to one of the four output lines depending on the state of the two control lines.
 9. Draw the logic diagram of a 2-bit encoder, a circuit with four input lines, exactly one of which is high at any instant, and two output lines whose 2-bit binary value tells which input is high.
 10. Redraw the PLA of Fig. 3-16 in enough detail to show how the majority logic function of Fig. 3-3 can be implemented. In particular, be sure to show which connections are present in both matrices.

11. What does this circuit do?



12. A common MSI chip is a 4-bit adder. Four of these chips can be hooked up to form a 16-bit adder. How many pins would you expect the 4-bit adder chip to have? Why?
13. An n -bit adder can be constructed by cascading n full adders in series, with the carry into stage i , C_i , coming from the output of stage $i - 1$. The carry into stage 0, C_0 , is 0. If each stage takes T nsec to produce its sum and carry, the carry into stage i will not be valid until iT nsec after the start of the addition. For large n the time required for the carry to ripple through to the high-order stage may be unacceptably long. Design an adder that works faster. *Hint:* Each C_i can be expressed in terms of the operand bits A_{i-1} and B_{i-1} as well as the carry C_{i-1} . Using this relation it is possible to express C_i as a function of the inputs to stages 0 to $i - 1$, so all the carries can be generated simultaneously.
14. If all the gates in Fig. 3-20 have a propagation delay of 10 nsec, and all other delays can be ignored, what is the earliest time a circuit using this design can be sure of having a valid output bit?
15. What is the quiescent state of the S and R inputs to an SR latch built of two NAND gates?
16. Given a JK flip-flop packaged in a chip, is it possible to convert it to a D flip-flop using external logic? If so, how? If not, why not? What about the reverse, that is, converting a D flip-flop into a JK one?
17. To help meet the payments on your new personal computer, you have taken up consulting for fledgling SSI chip manufacturers. One of your clients is thinking about putting out a chip containing four D flip-flops, each containing both Q and $\bar{Q}$, on request of a potentially important customer. The proposed design has all four clock signals ganged together, also on request. Neither preset nor clear is present. Your assignment is to give a professional evaluation of the design.
18. The circuit depicted below consists of four JK flip-flops, each triggered by the 1-to-0 transition on its clock input. None of the J or K inputs are connected, which means that they are all logical 1. The value stored in the circuit can be seen as a 4-bit number, $ABCD$. The input line consists of rectangular pulses 1 msec wide, spaced 100 msec apart. If $ABCD$ is initially 0000, what will it be after 13 input pulses?



19. The 4×3 memory of Fig. 3-28 uses 22 AND gates and three OR gates. If the circuit were to be expanded to 256×8 , how many of each would be needed?
20. As more and more memory is squeezed onto a single chip, the number of pins needed to address it also increases. It is often inconvenient to have large numbers of address pins on a chip. Devise a way to address 2^n words of memory using fewer than n pins.
21. A finite-state machine has 2^n possible states. At each clock pulse, it reads in a k -bit symbol, emits a k -bit output symbol, and switches to a new state. The symbol emitted and the new state depend on the current state and the input symbol. Describe how the machine can be implemented using a ROM, and tell how big the ROM must be.
22. A computer with a 32-bit wide data bus uses $4K \times 8$ static RAM memory chips. What is the smallest memory (in bytes) that this computer can have?
23. Referring to the timing diagram of Fig. 3-34, suppose that you slowed the clock down to a period of 400 nsec instead of 250 nsec as shown but the timing constraints remained unchanged. How much time would the memory have to get the data onto the bus after $\overline{MREQ}$ was asserted, in the worst case?
24. The Z80's jump instruction occupies 3 bytes, one for the opcode and two for the address. If the internal processing time within the CPU chip were zero, how long would the instruction take on a system with a 2.5-MHz clock?
25. In Fig. 3-31(b), T_{ML} is specified to be at least 60 nsec. Can you envision a chip in which it is negative? In other words, could the CPU assert $\overline{MREQ}$ before the address was stable? Why or why not?
26. What changes must be made to the circuit of Fig. 3-39 to have the PIO selected as input/output ports 0 to 3, instead of as memory addresses in the top quadrant of memory?
27. Imagine that you have built the system of Fig. 3-39, programmed it to play tic-tac-toe and lose, and have started selling it in volume to the incoming computer science freshman class as an ego booster. Suddenly your marketing manager comes running in waving a computer printout claiming that it is time to bring out a new version. You are too lazy to change the hardware, so you decide the simplest change is to move the RAM within the address space. To what address(es), if any, can you move it without invalidating the program?
28. Write a program to simulate the behavior of an $m \times n$ array of two-input NAND gates. This circuit, contained on a chip, has j input pins and k output pins. The values of j , k , m , and n are compile-time parameters of the simulation. The program should start off by

reading in a "wiring list," each wire of which specifies an input and an output. An input is either one of the j input pins or the output of some NAND gate. An output is either one of the k output pins or an input to some NAND gate. Unused inputs are logical 1. After reading in the wiring list, the program should print the output for each of the 2^j possible inputs. Gate array chips like this one are widely used for putting custom circuits on a chip because most of the work (depositing the gate array on the chip) is independent of the circuit to be implemented. Only the wiring is specific to each design.

29. Write a program to read in two arbitrary Boolean expressions and see if they represent the same function.
30. Write a program to read in a collection of Boolean expressions and compute the 24×50 and 50×6 matrices needed to implement them with the PLA of Fig. 3-16. Print the matrices on the line printer.