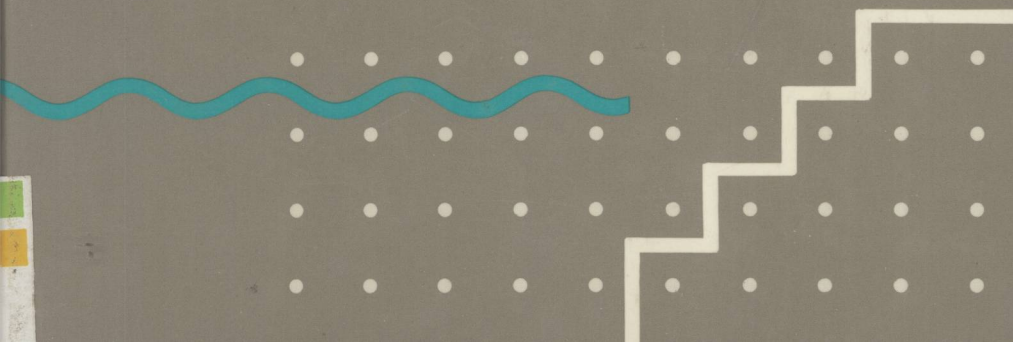


J.N. Demas • S.E. Demas

Interfacing and Scientific Computing on Personal Computers

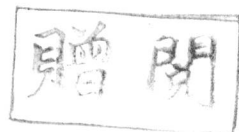


Science

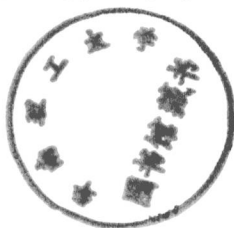


TP31
D372

9761513



Interfacing and Scientific Computing on Personal Computers



J. N. Demas

University of Virginia

S. E. Demas



E9761513

Allyn and Bacon

Boston London Sydney Toronto



Copyright © 1990 by Allyn and Bacon
A Division of Simon & Schuster, Inc.
160 Gould Street
Needham Heights, Massachusetts 02194

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without written permission from the copyright owner.

Library of Congress Cataloging-in-Publication Data

Demas, J. N.

Interfacing and scientific computing on personal computers / James
N. Demas, Susan E. Demas.

p. cm.

Includes bibliographical references.

ISBN 0-205-12368-6

1. Microcomputers--Programming. 2. Pascal (Computer program
language) 3. Turbo Pascal (Computer program) 4. Computer
interfaces. I. Demas, Susan E. II. Title.

QA76.6.D455 1990

005.26--dc20

89-49413

CIP

ISBN 0-205-12368-6

ISBN 0-205-12551-4 (International)

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1—95 94 93 92 91 90

GRAPHER and SURFER are registered trademarks of Golden Software, Inc.
IBM XT, IBM PC, IBM AT, PC-DOS, and PS-2 are registered trademarks of
International Business Machines Corporation.

Intel, i386, i386sx, i387, i387sx, and i486 are registered trademarks of
Intel Corporation. Selected material reprinted by permission of Intel
Corporation, Copyright/Intel Corporation.

Pizazz is a registered trademark of Application Techniques, Inc.

PROCComm is a registered trademark of Datastorm Technologies
Incorporated.

ProKey is a registered trademark of RoseSoft, Incorporated.

QuickBASIC, OS/2, MS-DOS, GWBASIC, and Microsoft are registered
trademarks of Microsoft Corporation. Selected material reprinted with
permission from Microsoft Corporation.

Turbo Pascal, SideKick, Superkey, and Turbo Assembler are registered
trademarks of Borland International. Portions copyright © 1988, Borland
International, Inc. Used by permission.

UNIX is a registered trademark of AT&T.

WordPerfect is a registered Trademark of WordPerfect, Corporation.

WordStar is a registered trademark of MicroPro International Corporation.

In Memory of

Elsie F. Demas

and

John W. Conlon

Preface

The microelectronics revolution has given low cost, very high performance computing power to virtually anyone. Scientists and engineers in all areas are in the position to use these general purpose microcomputers to control experiments, gather and reduce data, and optimize measurements—if they can determine what to buy, how to interface it, and how to acquire and manipulate data.

In *Interfacing and Scientific Computing on Personal Computers*, we present a highly applied, experimental approach to computer interfacing and data acquisition and reduction. We demonstrate basic concepts, clarify jargon, provide powerful software subroutines and complete programs, and present problems and experiments designed to develop skills for efficiently interfacing and automating many instruments and systems. This material is designed for those who are reasonably familiar with any computer language. We assume no knowledge of electronics, and sophisticated electronic hardware descriptions are avoided, although enough digital and analog background is provided to understand important hardware interfaces. This book forms the basis of our upper division undergraduate and graduate interfacing course that is taken by physicists, chemists, biochemists, physiologists, and engineers.

We generally utilize commercial high-performance, low-cost hardware and software. We build around software and hardware that is compatible with IBM Personal Computer (PCs) and work-alike clones because of their high performance-cost ratio, and the ready availability of scientifically useful peripherals and software.

Our main software language is Pascal, which is a structured high-level programming language that teaches good programming skills and prepares students for languages such as the government standard Ada. We use Borland's Turbo Pascal which is inexpensive, powerful, and convenient. Its wide availability makes Turbo Pascal the *de facto* microcomputer Pascal. Turbo Pascal compiles and executes extremely quickly, and for interfacing applications has extensions for direct hardware control and graphics.

Interfacing the new equipment is made easier with our detailed sections on input and output devices such as analog-to-digital converters (ADCs), digital-to-analog converters (DACs), parallel interfaces, RS-232 serial interfaces, keyboards, video displays, digitization tablets, and real time clocks (RTCs). Concepts are introduced in the context of laboratory automation and on-line data reduction of scientifically useful experiments. Emphasis is placed on understanding fundamental principles and avoiding the numerous subtle problems that can arise.

In addition to showing how to acquire data, we demonstrate how to improve its accuracy and precision using a variety of signal-to-noise (S/N) enhancement techniques, and data reduction and fitting techniques, especially nonlinear fitting. Pitfalls such as false minima and the blind use of smoothing in many commercial instruments are explained.

For those interfacing problems that require the exquisite control and speed of machine language programming, we devote several chapters to assembly language. In particular, we show how to marry high-level and assembly language routines to gain the benefits of both.

We emphasize understanding underlying principles, avoiding pitfalls, and using experiments and programming to develop skills. Thus, even though readers might not end up using specific hardware or software, they will understand the principles of commercial instrumentation and be able to probe the manufacturer to find out what the instrument can actually do rather than blindly accept any exaggerated or misleading claims.

Acknowledgments

We thank the many students who have contributed their time and energy to this course. Their comments, unfavorable as well as favorable, were considered heavily in the evolution of this material. I especially thank my teaching assistants, Nelson Ayala, Elizabeth R. Carraway, and Seth Snyder for invaluable assistance—indeed, the first year of the PC version of this course would have been an unmitigated disaster without their unflagging efforts and abilities. We also thank Richard Ballew for his software skills (which made many of our projects possible) as well as a number of students who patiently debugged many experiments. We thank our son, David, for writing and debugging some of the software, for reading and commenting on the manuscript, and for checking many of the experiments. Our daughter, Stacy, receives humble gratitude for giving up some of a well-earned summer vacation to decipher handwritten additions. Dee Irwin in Academic Computing was the UNIX troff guru who smoothed the rough spots on the typesetting of this book. Karen Mason's able editing of this book clarified some points and imposed a consistent format. We thank the University of Virginia's Division of Academic Computing for a grant that allowed us to upgrade the course to its current level, and the Chemistry Department for its continued support. We thank Borland International for permission to use part of a copyrighted program. Finally, special thanks to David G. Taylor who originally got one (J.N.D.) of us interested in microcomputers in the 1970s—that led to our purchase of one of the first Altair 8800 microcomputers, which, as it turned out, was the beginning of the end!

CONTENTS

Preface	xi
1. Logic, Numbers, Arithmetic, Software, and Computer Architecture	1
1. INTRODUCTION	1
2. LOGIC	2
3. BITS, NIBBLES, BYTES, AND WORDS	13
4. NUMBER BASES AND NUMBER REPRESENTATIONS	14
5. COMPUTER ARCHITECTURE	22
6. SOFTWARE	25
7. LANGUAGES	28
8. REFERENCES	30
Problems	30
2. DOS, BASIC, Interpreters, Compilers, and Math Coprocessors	32
1. INTRODUCTION	32
2. MICROSOFT DISK OPERATING SYSTEM (DOS)	33
3. BASIC INTERPRETER	45
4. INTEGRATED QuickBASIC 4.0 COMPILER	48
5. QuickBASIC COMMAND LINE COMPILER (BC)	52
6. RAM DISKS	54
7. BATCH FILES	55
8. FORTRAN	59
9. HARD OR FIXED DISKS	60
10. REFERENCES	65
Problems	66
3. Timing, Serial/Parallel I/O, and Pipes and Filters	70
1. INTRODUCTION	70
2. TIMING AND UNITS	71
3. ASCII CODES FOR CHARACTER REPRESENTATIONS	74
4. CHARACTER CONVERSIONS AND VIDEO OUTPUT	76
5. KEYBOARD INPUT, VIDEO OUTPUT	77
6. SERIAL DATA TRANSMISSION	79
7. PARALLEL I/O	84
8. FILTERS, PIPES, AND STANDARD INPUT AND OUTPUT	94
9. REFERENCES	105
Problems	105
4. Video Graphics and Digitizers	110
1. INTRODUCTION	110
2. VIDEOGRAPHICS	110

3. DIGITIZERS	127
4. REFERENCES	134
Problems	134
5. Analog Considerations	135
1. INTRODUCTION	135
2. OPERATIONAL AMPLIFIERS	135
3. ANALOG-TO-DIGITAL CONVERSIONS	144
4. ANALOG SWITCHES AND MULTIPLEXERS	149
5. NYQUIST SAMPLING THEOREM	152
6. REFERENCES	154
Problems	154
6. Data Translation Analog-Digital I/O System	156
1. INTRODUCTION	156
2. DT 2801 ORGANIZATION	159
3. IMMEDIATE COMMANDS	167
4. BLOCK COMMANDS	169
5. DMA BLOCK MODES	171
6. DTUtil UNIT	174
7. REFERENCES	187
Problems	187
7. Signal-to-Noise (S/N) Enhancement	188
1. INTRODUCTION	188
2. RC FILTERS	192
3. GATED FILTERS	195
4. LOCK-IN AMPLIFIERS	196
5. ENSEMBLE AVERAGING	206
6. QUANTIZATION NOISE	209
7. BOXCAR INTEGRATOR	213
8. DIGITAL FILTERS	218
9. MONTE CARLO SIMULATIONS	221
10. REFERENCES	225
Problems	226
8. Least Squares Data Reduction	228
1. INTRODUCTION	228
2. LINEAR LEAST SQUARES FITTING	229
3. NONLINEAR LEAST SQUARES	234
4. JUDGING THE FIT	248
5. ERROR ESTIMATION	252
6. TESTING/APPLYING NONLINEAR LEAST SQUARES	254

7. REFERENCES	254
Problems	255
9. Communications on the RS-232 Serial Port	259
1. INTRODUCTION	259
2. RS-232 INTERCONNECTIONS	260
3. IDEAL CASE	262
4. IT DOESN'T WORK!	265
5. SOFTWARE HANDSHAKES	266
6. GETTING INTO THE GUTS OF THE PC	267
7. COMMERCIAL INSTRUMENTS	269
8. REFERENCES	271
Problems	271
10. CPU Architecture and System Calls	272
1. INTRODUCTION	272
2. SYSTEM ARCHITECTURE	274
3. REFERENCING MEMORY	277
4. RECORDS	279
5. INTERRUPTS	281
6. REFERENCES	295
Problems	295
11. Assembly Language	298
1. INTRODUCTION	298
2. ASSEMBLY LANGUAGE NOMENCLATURE	299
3. MOVE OR COPY INSTRUCTIONS	302
4. ADDRESSING MODES	302
5. INTERRUPTS	307
6. FLAGS	308
7. REGISTER INCREMENTATION AND DECREMENTATION	308
8. LOOP INSTRUCTION	309
9. ARITHMETIC AND LOGICAL OPERATIONS	310
10. JUMPS	313
11. ANATOMY OF AN ASSEMBLY LANGUAGE PROGRAM	315
12. INPUT AND OUTPUT	317
13. SPECIAL MEMORY INSTRUCTIONS	319
14. SUBROUTINE CALLS AND RETURNS	320
15. REFERENCES	325
Problems	325
12. Coupling Pascal and Assembly Language	327
1. INTRODUCTION	327

2. SHARING INFORMATION WITH PASCAL (PUBLIC/EXTERN)	328
3. PARAMETER PASSING AND RETURNING RESULTS	331
4. INLINE MACHINE STATEMENTS	339
5. INLINE PROCEDURES AND FUNCTIONS	348
6. REFERENCES	349
Problems	350
13. 80X87 Math Coprocessor	352
1. INTRODUCTION	352
2. 80X87 HARDWARE ARCHITECTURE	353
3. 80X87 INSTRUCTION SET	354
4. EXAMPLES	361
5. REFERENCES	365
Problems	365
14. Miscellaneous	368
1. INTRODUCTION	368
2. SOFTWARE	369
3. IEEE-488 INTERFACE BUS	372
4. PLOTTERS	373
5. SENSORS AND MECHANICAL MOVEMENT	373
APPENDIX A. Experiments	375
A.1. Chapter 2 Experiments: DOS and BASIC	375
A.2. Async Serial Communication Unit	376
A.3. Chapter 3 Experiments: Serial-Parallel Interfacing	379
A.4. Chapter 4 Experiments: Graphics and Digitizers	382
A.5. Chapter 5 Experiments: Analog Considerations	387
A.6. Chapter 6 Experiments: Analog-Digital I/O System	388
A.7. Chapter 7 Experiments: Signal-to-Noise Enhancement	391
A.8. Chapter 8 Experiment: Nonlinear Data Fitting	401
A.9. Chapter 9 Experiments: Serial Interfacing	410
A.10. Chapter 10 Experiments: Timer Acquisition and Fitting	410
A.11. Chapter 11 Experiments: Using Turbo Assembler	413
APPENDIX B. Editing in QuickBASIC and Turbo Pascal	419
B.1. QuickBASIC	419
B.2. TurboPascal	419
APPENDIX C. Software Availability	423
References	425
Index	429

1

Logic, Numbers, Arithmetic, Software, and Computer Architecture

1.1 INTRODUCTION

In this chapter we introduce a number of important concepts and supply an overview of computer hardware and software. While much of what we say applies to computers in general, this text is based on IBM Personal Computers (PCs) and their clones or work-alikes.

We begin with basic digital logic because all computer operations are built on these logic elements and many computer interfaces utilize such logic. Then follows a description of the basic ways of representing numbers in computers, different number bases, and conversions between bases. This information is necessary to appreciate the resolution and range of numbers as well as to understand how they are stored in memory.

A brief overview of microcomputer architecture gives an idea of some of the capabilities and limitations of these machines. This is followed by a discussion of the driving force behind computers, software. Finally, there is a section on different types of computer languages with a description of the advantages and weaknesses of the more common languages.

1.2 LOGIC

In order to understand how a computer works and how many interfaces are constructed, we must understand basic logic and the corresponding electronic symbols. We describe here the basic building blocks of any computer system. These consist of NOT or invert, AND, OR, and exclusive OR functions as well as the basic counting element—the flip-flop. In addition, we show how the essential shift register can be built from flip-flops. We will not go into the electronic construction of the logic elements.

We first establish a convention. Symbolic logic has two states: true and false. However, in electronic logic, two other conventions are used. These are "0" and "1" or "L" (low) and "H" (high). Both alternatives are widely used but we prefer the more common 0–1 convention.

H and 1, being true states, correspond to "positive true" logic. Positive true is the most likely. The other convention is "negative true" logic where the true and false states of positive true logic are reversed. That is, a true is low or 0 and a false is high or 1. Certain computer (DEC 11 series) and communications buses (HPIB or IEEE-488 instrument bus) use negative true logic. Unfortunately, some systems mix positive and negative true logic and do not tell you which is used where, or bury this information in an appendix.

One widely used logic family is Transistor-Transistor-Logic (TTL). The high state is 2–5 V, while the low state range is 0–0.8 V. Any voltage in the 0.8–2.0 V range is ambiguous to the logic devices. Many devices that are built with other technologies are TTL compatible; that is, they recognize and generate TTL level voltages. Most logic devices in PCs are TTL compatible.

Logic functions are commonly described by a truth table, which is a complete list of the output states for all possible input states. We turn now to different logic functions.

1.2.1 NOT Gate or Invert Function

The NOT function is the inverse of the input. The NOT function of input A is written as \bar{A} and is read NOT A. The truth table and logic symbols are shown in Figure 1–1.

A	\bar{A}
0	1
1	0

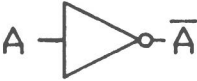


Figure 1–1. NOT function truth table and logic symbol.

1.2.2 AND Function

The AND function is true if the inputs are all true. For a two-input AND function of A and B, it is written as AB and read as A AND B. The truth table and symbol are shown in Figure 1–2.

A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1

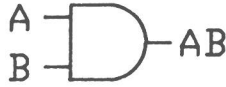


Figure 1–2. AND function truth table and logic symbol.

Logic devices such as the AND are called gates because they can be used to control or "gate" the flow of information. Consider a two-input AND. If B is the control input, then the A input can be transmitted or blocked by the setting of B. If B=1 then A is transmitted directly to the output, or the gate is transmitting. Conversely, if B=0, the gate is off and the input is fixed at 0 regardless of what A is.

A related function is the NAND or NOT AND denoted by \overline{AB} and read NOT the quantity A AND B. The symbol and truth table are shown in Figure 1–3. An "o" at an input or an output of a logic symbol stands for the NOT or invert function.

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

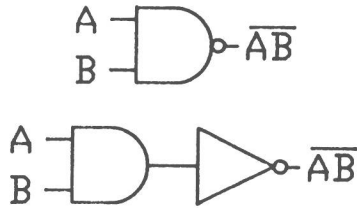


Figure 1–3. NAND truth table and logic symbol.

1.2.3 OR Function

The OR function is true if any input is true. The OR of two inputs is denoted by $A+B$ and read A OR B. Similarly, there is a NOR or NOT OR function. The truth tables and symbols for the OR and NOR are shown in Figure 1–4.

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

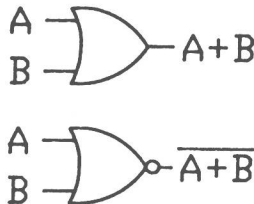


Figure 1–4. OR and NOR truth tables and logic symbols.

1.2.4 Exclusive OR Function

The exclusive OR (XOR) is a two-input function that is true if and only if one or the other input is true. The truth table and logic symbol are shown in Figure 1–5.

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0



Figure 1–5. Exclusive OR truth table and logic symbol.

An exclusive OR can function as a controllable inverter. If $A=0$, the gate transmits B unchanged. If $A=1$, then B is inverted at the output.

1.2.5 Flip-Flops (FF)

Flip-flops are the basic counting elements of digital logic. Their name arises since their output can flip-flop or toggle between different output states. In contrast to the static logic devices described thus far, flip-flops change state in response to a clock pulse. The changes are dependent on the states of the control inputs. The clock pulse can be a complete $0 \rightarrow 1 \rightarrow 0$ transition or the edges of a $0 \rightarrow 1$ or negative $1 \rightarrow 0$ transition. The first is called level triggered while the latter two are positive and negative edge-triggered respectively.

There are numerous types of flip-flops and we restrict our current discussion to a positive edge-triggered JK flip-flop. The truth table and symbols are shown in Figure 1–6. Q_n is the state of the Q output before the clock pulse and Q_{n+1} is the state after the trigger clock. The flip-flop has two outputs Q and \bar{Q} where \bar{Q} is the complement of Q (i.e., NOT Q). As we will see, having the complement available greatly simplifies certain types of logic design.

To read the table we consider different state changes. If $J=K=0$ and there is a clock transition, the outputs remain unchanged. That is, Q at time $n+1$ is equal to Q at n or $Q_{n+1}=Q_n$. If $J=1$ and $K=0$, Q goes to a 1 after the clock transition regardless of its original state. If $J=K=1$, then Q is complemented or toggled after the clock pulse. This last

J	K	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	\bar{Q}_n (toggle)

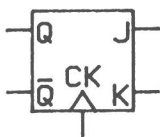


Figure 1–6. JK flip-flop truth table and logic symbol.

mode is the normal flip-flop mode where Q will toggle between two possible states for each clock transition.

To make a divide by 4 counter, consider the circuit shown in Figure 1-7A where Q_A and Q_B are the outputs. Q_A and Q_B respond to the clock as shown in Figure 1-7B.

Note that after four clock edges, Q_A and Q_B return to the original state. Further, Q_A changes at exactly half the rate as the input clock and always changes on a positive clock edge; this is the simple divide by two or flip-flop mode. Q_B changes at half the rate of the Q_A output or one fourth the clock rate. Thus, we get a divide by four. The divide by four works because the outputs of the first FF are tied to the inputs of the B FF. The necessary condition for the B FF to change state occurs only on every other output state of FF A.

One subtle point requires comment. On clock edge 2, Q_B goes to a 1, but its input seems to be changing at the same time. Why does it not become confused? In reality, the JK inputs of B are derived from Q_A . Flip-flop A sees the clock pulse at the same instant as B does, but it takes a finite length of time for Q_A to change. Thus, B changes states based on the original inputs. It is for this reason that counters of this design must use edge-triggered flip-flops.

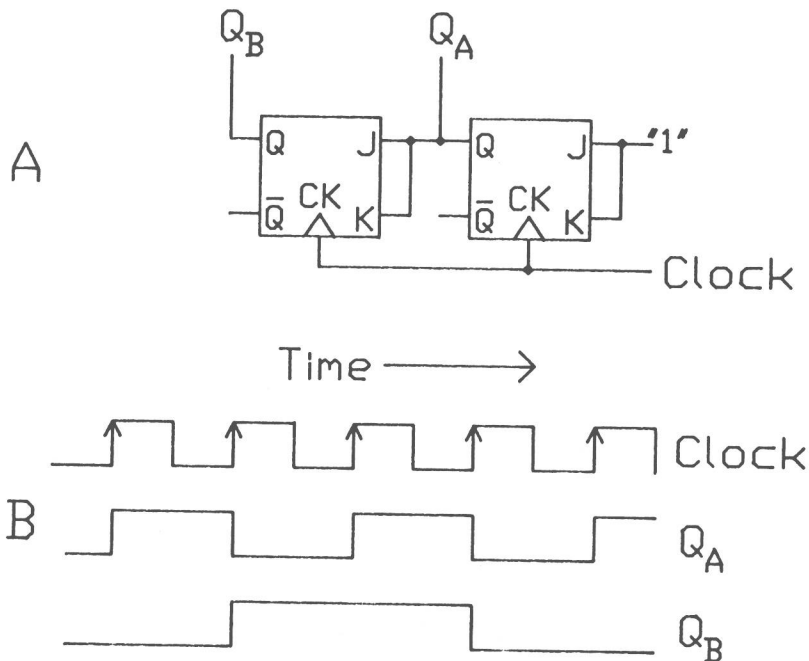


Figure 1-7. Schematic representation of a divide by 4 counter built on flip-flops.
 A) Circuit. B) Waveforms in response to clock pulse.

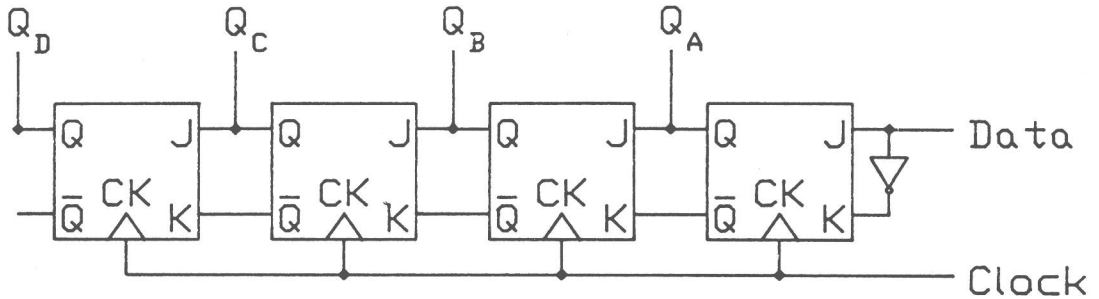


Figure 1–8. Four-bit shift register built from JK flip-flops.

1.2.6 Shift Registers

Shifting strings of binary numbers is an essential arithmetic and logic operation. In particular, computers (and humans) do multiplication and division by repeated shift-add or shift-subtract operations.

A 4-bit shift register is shown in Figure 1–8. While seemingly complex, its operation is straightforward. Q and \bar{Q} from the previous stage are applied to the JK inputs of the next. For the first stage, the input datum is applied directly to the J input, then inverted and applied to the K input. The FF truth table shows that on each clock transition Q and \bar{Q} information from the previous stage gets loaded or shifted into the next FF.

To load four bits of information, apply each bit in turn to the data input and clock it in by a single clock pulse. Thus, four clock pulses are needed to fully load a 4-bit shift register. This is called a serial input or serial load shift register. For very high speed circuits, the time overhead of a serial load can seriously degrade system performance.

To overcome the slowness of a serial load, there are parallel load shift registers where all bits are presented in parallel. A single load pulse loads all the FFs at once. Because of the large number of external connections involved, parallel load devices are usually limited to 4 or 8 bits of data.

1.2.7 Memory

Memory is necessary for storage of program and data. It comes in two basic forms: random access (RAM) or read only (ROM). Random access memory can be written to (altered) as well as read back, while ROM has permanent or semipermanent data or programs in it and can only be read. Actually, both RAM and ROM are random access in that you can access any location in any order by changing the address. A more accurate term for RAM is read/write memory, but this is not in common use.

ROM comes in several versions. There is real ROM, which comes from the factory with the information built directly on the chip. Only if the information is cast in concrete and you plan to use thousands of this program is ROM attractive; the semiconductor manufacturers will only make these in quantities of hundreds to thousands. However, for small users there is PROM or programmable ROM, which is field programmable. Some of these are erasable either electrically or with ultraviolet (uv) light; you can tell the latter by the clear window over the chip. You will see PROM or ROM within most computers

and on many peripheral boards that use dedicated microprocessors to implement their function and where the microprocessor must know what to do as soon as the system is turned on.

In its simplest form, RAM consists of a two-dimensional array of flip-flops with a means for selectively storing data in a specific flip-flop and reading it back later. This configuration is called static since the information is statically held in the flip-flop and will stay there until you explicitly change it or turn the power off.

Much more prevalent is dynamic memory in which the data are stored as microcharges on capacitors; charged corresponds to one state while uncharged is the other. Since the charge on a capacitor will leak off and disappear after a while, you must refresh the charge periodically. Typically, this refreshing must be done a thousand times a second, which is generally a problem for the designer, not the user.

The prevalence of dynamic memory over static arises from its much lower power consumption, higher memory densities, and lower cost. Dynamic RAM is generally not as fast as static memory; this is getting to be a problem in many of the very fast new microcomputers in which computation is being limited by how fast the data can be fetched or stored in memory.

Memory chips come in a variety of sizes, but with increasingly memory-intensive software, the bigger the better. About the smallest size useful chip is 256K, with 1M chips rapidly coming into prominence. Four megabyte chips are in production. A K in computer notation is $1024 (2^{10})$ and M or meg is $1,048,576 (2^{20})$.

1.2.8 Three-State and Open Collector Logic

There are two additional types of logic that are used in computers. These are three-state logic and open collector logic. Three-state logic is also known as Tri-State logic, which is the copyrighted name from National Semiconductors. Other manufacturers are limited to the term three-state logic, although in spoken usage Tri-State and three-state are used interchangeably.

Three-state and open collector logic are what make modern computer architecture possible. While open collector has been replaced by three-state logic for new computer construction, open collector devices still find wide use in the IEEE-488 instrumentation interface bus.

As the term implies, three-state logic has three possible output states. These are high, low, and "not there." That is, the device can have the usual high and low or it can be disconnected from the circuit. The not there state, which is referred to as the high impedance or off state, is controlled by an additional input to the device. This control is called the enable. The truth table and schematic of a simple, straight through, noninverting driver is shown in Figure 1-9. The Z in the output indicates the high impedance or not there state. The enable input in this case is active high; the device is turned on (i.e., connects the circuitry to the output) with a high enable signal. Active low circuitry is also common. The notation on the IC is the invert or circle symbol on the input.

Of what use is this three-state function? The answer is that it allows multiple devices to sit on a common line, but have only one of them active at a time. Consider Figure 1-10. For simplicity we show only two inputs on a common line, but in a computer many devices may share a common line. Assume that there are two input signals that