FEYNMAN LECTURES ON COMPUTATION

RICHARD P. FEYNMAN

EDITED BY

ANTHONY J.G. HEY ▪ ROBIN W. ALLEN

Department of Electronics and Computer Science
University of Southampton
England

Addison-Wesley Publishing Company, Inc.
The Advanced Book Program

Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Harlow, England Amsterdam Bonn
Sydney Singapore Tokyo Madrid San Juan
Paris Seoul Milan Mexico City Taipei

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial capital letters.

Library of Congress Cataloging-in-Publication Data

Feynman, Richard Phillips.
Feynman lectures on computation / Richard P. Feynman ; edited by Anthony J.G. Hey and Robin W. Allen.
p. cm.
Includes bibliographical references and index.
ISBN 0-201-48991-0
1. Electronic data processing. I. Hey, Anthony J.G. II. Allen, Robin W. III. Title.
QA76.F45. 1996
004'.01—dc20 96–25127
CIP

Copyright © 1996 by Carl R. Feynman and Michelle Feynman

Foreword and Afterword copyright © 1996 by Anthony J.G. Hey and Robin W. Allen

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America.

Jacket design by Lynne Reed
Text design and typesetting by Tony Hey

2 3 4 5 6 7 8 9 10-MA-0100999897
Second Printing January 1997

Photograph courtesy of Michelle Feynman

CONTENTS

Foreword viii

Preface (Richard Feynman) xiii

1 Introduction to Computers 1

1.1 The File Clerk Model 5
1.2 Instruction Sets 8
1.3 Summary 17

2 Computer Organization 20

2.1 Gates and Combinational Logic 20
2.2 The Binary Decoder 30
2.3 More on Gates: Reversible Gates 34
2.4 Complete Sets of Operators 39
2.5 Flip-Flops and Computer Memory 42
2.6 Timing and Shift Registers 46

3 The Theory of Computation 52

3.1 Effective Procedures and Computability 52
3.2 Finite State Machines 55
3.3 The Limitations of Finite State Machines 60
3.4 Turing Machines 66
3.5 More on Turing Machines 75
3.6 Universal Turing Machines and the Halting Problem 80
3.7 Computability 88

4	**Coding and Information Theory**	94
	4.1 Computing and Communication Theory	95
	4.2 Error Detecting and Correcting Codes	95
	4.3 Shannon's Theorem	106
	4.4 The Geometry of Message Space	110
	4.5 Data Compression and Information	115
	4.6 Information Theory	120
	4.7 Further Coding Techniques	123
	4.8 Analogue Signal Transmission	129
5	**Reversible Computation and the Thermodynamics of Computing**	137
	5.1 The Physics of Information	137
	5.2 Reversible Computation and the Thermodynamics of Computing	151
	5.3 Computation: Energy Cost versus Speed	167
	5.4 The General Reversible Computer	172
	5.5 The Billiard Ball Computer	176
	5.6 Quantum Computation	182
6	**Quantum Mechanical Computers** (Reprinted from *Optics News*, February 1985)	185
	6.1 Introduction	185
	6.2 Computation with a Reversible Machine	187
	6.3 A Quantum Mechanical Computer	191
	6.4 Imperfections and Irreversible Free Energy Loss	199
	6.5 Simplifying the Implementation	202
	6.6 Conclusions	210
	6.7 References	211

7 Physical Aspects of Computation 212

A Caveat from the Editors 212

7.1 The Physics of Semiconductor Devices 213
7.2 Energy Use and Heat Loss in Computers 238
7.3 VLSI Circuit Construction 257
7.4 Further Limitations on Machine Design 274

Afterword: Memories of Richard Feynman 284

Suggested Reading 294
Index 297

Editor's Foreword

Since it is now some eight years since Feynman died I feel it necessary to explain the genesis of these 'Feynman Lectures on Computation'. In November 1987 I received a call from Helen Tuck, Feynman's secretary of many years, saying that Feynman wanted me to write up his lecture notes on computation for publication. Sixteen years earlier, as a post-doc at CalTech I had declined the opportunity to edit his 'Parton' lectures on the grounds that it would be a distraction from my research. I had often regretted this decision so I did not take much persuading to give it a try this time around. At CalTech that first time, I was a particle physicist, but ten years later, on a sabbatical visit to CalTech in 1981, I became interested in computational physics problems – playing with variational approaches that (I later found out) were similar to techniques Feynman had used many years before. The stimulus of a CalTech colloquium on 'The Future of VLSI' by Carver Mead then began my move towards parallel computing and computer science.

Feynman had an interest in computing for many years, dating back to the Manhattan project and the modeling of the plutonium implosion bomb. In 'Los Alamos from Below', published in 'Surely You're Joking, Mr. Feynman!', Feynman recounts how he was put in charge of the 'IBM group' to calculate the energy release during implosion. Even in those days before the advent of the digital computer, Feynman and his team worked out ways to do bomb calculations in parallel. The official record at CalTech lists Feynman as joining with John Hopfield and Carver Mead in 1981 to give an interdisciplinary course entitled 'The Physics of Computation'. The course was given for two years and John Hopfield remembers that all three of them never managed to give the course together in the same year: one year Feynman was ill, and the second year Mead was on leave. A handout from the course of 1982/3 reveals the flavor of the course: a basic primer on computation, computability and information theory followed by a section entitled 'Limits on computation arising in the physical world and "fundamental" limits on computation'. The lectures that year were given by Feynman and Hopfield with guest lectures from experts such as Marvin Minsky, John Cocke and Charles Bennett. In the spring of 1983, through his connection with MIT and his son Carl, Feynman worked as a consultant for Danny Hillis at Thinking Machines, an ambitious, new parallel computer company.

In the fall of 1983, Feynman first gave a course on computing by himself, listed in the CalTech record as being called 'Potentialities and Limitations of

Computing Machines'. In the years 1984/85 and 1985/86, the lectures were taped and it is from these tapes and Feynman's notebooks that these lecture notes have been reconstructed. In reply to Helen Tuck, I told her I was visiting CalTech in January of 1988 to talk at the 'Hypercube Conference'. This was a parallel computing conference that originated from the pioneering work at CalTech by Geoffrey Fox and Chuck Seitz on their 'Cosmic Cube' parallel computer. I talked with Feynman in January and he was very keen that his lectures on computation should see the light of day. I agreed to take on the project and returned to Southampton with an agreement to keep in touch. Alas, Feynman died not long after this meeting and we had no chance for a more detailed dialogue about the proposed content of his published lectures.

Helen Tuck had forwarded to me both a copy of the tapes and a copy of Feynman's notes for the course. It proved to be a lot of work to put his lectures in a form suitable for publication. Like the earlier course with Hopfield and Mead, there were several guest lecturers giving one or more lectures on topics ranging from the programming language 'Scheme' to physics applications on the 'Cosmic Cube'. I also discovered that several people had attempted the task before me! However, the basic core of Feynman's contribution to the course rapidly became clear — an introductory section on computers, followed by five sections exploring the limitations of computers arising from the structure of logic gates, from mathematical logic, from the unreliability of their components, from the thermodynamics of computing and from the physics of semiconductor technology. In a sixth section, Feynman discussed the limitations of computers due to quantum mechanics. His analysis of quantum mechanical computers was presented at a meeting in Anaheim in June of 1984 and subsequently published in the journal 'Optics News' in February 1985. These sections were followed by lectures by invited speakers on a wide range of 'advanced applications' of computers — robotics, AI, vision, parallel architectures and many other topics which varied from year to year.

As advertised, Feynman's lecture course set out to explore the limitations and potentialities of computers. Although the lectures were given some ten years ago, much of the material is relatively 'timeless' and represents a Feynmanesque overview of some standard topics in computer science. Taken as a whole, however, the course is unusual and genuinely interdisciplinary. Besides giving the 'Feynman treatment' to subjects such as computability, Turing machines (or as Feynman says, 'Mr. Turing's machines'), Shannon's theorem and information theory, Feynman also discusses reversible computation, thermodynamics and quantum computation. Such a wide-ranging discussion of the fundamental basis of computers is undoubtedly unique and a 'sideways', Feynman-type view of the

whole of computing. This does not mean to say that all aspects of computing are discussed in these lectures and there are many omissions, programming languages and operating systems, to name but two. Nevertheless, the lectures do represent a summary of our knowledge of the truly fundamental limitations of digital computers. Feynman was not a professional computer scientist and he covers a large amount of material very rapidly, emphasizing the essentials rather than exploring details. Nevertheless, his approach to the subject is resolutely practical and this is underlined in his treatment of computability theory by his decision to approach the subject via a discussion of Turing machines. Feynman takes obvious pleasure in explaining how something apparently so simple as a Turing machine can arrive at such momentous conclusions. His philosophy of learning and discovery also comes through strongly in these lectures. Feynman constantly emphasizes the importance of working things out for yourself, trying things out and playing around before looking in the book to see how the 'experts' have done things. The lectures provide a unique insight into Feynman's way of working.

I have used editorial license here and there in ways I should now explain. In some places there are footnotes labeled 'RPF' which are asides that Feynman gave in the lecture that in a text are best relegated to a footnote. Other footnotes are labeled 'Editors', referring to comments inserted by me and my co-editor Robin Allen. I have also changed Feynman's notation in a few places to conform to current practice, for example, in his representation of MOS transistors.

Feynman did not learn subjects in a conventional way. Typically, a colleague would tell him something that interested him and he would go off and work out the details for himself. Sometimes, by this process of working things out for himself, Feynman was able to shed new light on a subject. His analysis of quantum computation is a case in point but it also illustrates the drawback of this method for others. In the paper on quantum computation there is a footnote after the references that is typically Feynman. It says: 'I would like to thank T. Toffoli for his help with the references'. With his unique insight and clarity of thinking Feynman was often able not only to make some real progress but also to clarify the basis of the whole problem. As a result Feynman's paper on quantum computation is widely quoted to the exclusion of other lesser mortals who had made important contributions along the way. In this case, Charles Bennett is referred to frequently, since Feynman first heard about the problem from Bennett, but other pioneers such as Rolf Landauer and Paul Benioff are omitted. Since I firmly believe that Feynman had no wish to take credit from [illegible] the historical record in a few places

and refer the reader, in a footnote, to more complete histories of the subject. The plain truth was that Feynman was not interested in the history of a subject but only the actual problem to be solved!

I have exercised my editorial prerogative in one other way, namely in omitting a few lectures on topics that had become dated or superseded since the mid 1980's. However, in order to give a more accurate impression of the course, there will be a companion volume to these lectures which contains articles on 'advanced topics' written by the self-same 'experts' who participated in these courses at CalTech. This complementary volume will address the advances made over the past ten years and will provide a fitting memorial to Feynman's explorations of computers.

There are many acknowledgements necessary in the successful completion of a project such as this. Not least I should thank Sandy Frey and Eric Mjolness, who both tried to bring some order to these notes before me. I am grateful to Geoffrey Fox, for trying to track down students who had taken the courses, and to Rod van Meter and Takako Matoba for sending copies of their notes. I would also like to thank Gerry Sussman, and to place on record my gratitude to the late Jan van de Sneepscheut, for their initial encouragement to me to undertake this task. Gerry had been at CalTech, on leave from MIT, when Feynman decided to go it alone, and he assisted Feynman in planning the course.

I have tried to ensure that all errors of (my) understanding have been eliminated from the final version of these lectures. In this task I have been helped by many individuals. Rolf Landauer kindly read and improved Chapter 5 on reversible computation and thermodynamics and guided me patiently through the history of the subject. Steve Furber, designer of the ARM RISC processor and now a professor at the University of Manchester, read and commented in detail on Chapter 7 on VLSI — a topic of which I have little first-hand knowledge. Several colleagues of mine at Southampton also helped me greatly with the text: Adrian Pickering and Ed Zaluska on Chapters 1 and 2; Andy Gravell on Chapter 3; Lajos Hanzo on Chapter 4; Chris Anthony on Chapter 5; and Peter Ashburn, John Hamel, Greg Parker and Ed Zaluska on Chapter 7. David Barron, Nick Barron and Mike Quinn, at Southampton, and Tom Knight at MIT, were kind enough to read through the entire manuscript and, thanks to their comments, many errors and obscurities have been removed. Needless to say, I take full responsibility for any remaining errors or confusions! I must also thank Bob Churchhouse of Cardiff University for information on Baconian ciphers, Bob Nesbitt of Southampton University for enlightening me about the geologist William Smith, and James Davenport of Bath University for

help on references pertaining to the algorithmic solution of integrals. I am also grateful to the Optical Society of America for permission to reproduce, in slightly modified form, Feynman's classic 1985 'Optics News' paper on Quantum Mechanical Computing as Chapter 6 of these lectures.

After Feynman died, I was greatly assisted by his wife Gweneth and a Feynman family friend, Dudley Wright, who supported me in several ways, not least by helping pay for the lecture tapes to be transcribed. I must also pay tribute to my co-editor, Robin Allen, who helped me restart the project after the long legal wrangling about ownership of the Feynman archive had been decided, and without whom this project would never have seen the light of day. Gratitude is also due to Michelle Feynman, and to Carl Feynman and his wife Paula, who have constantly supported this project through the long years of legal stalemate and who have offered me every help. A word of thanks is due to Allan Wylde, then Director of the Advanced Book Program at Addison-Wesley, who showed great faith in the project in its early stages. Latterly, Jeff Robbins and Heather Mimnaugh at Addison-Wesley Advanced Books have shown exemplary patience with the inevitable delays and my irritating persistence with seemingly unimportant details. Lastly, I must record my gratitude to Helen Tuck for her faith in me and her conviction that I would finish the job — a belief I have not always shared! I hope she likes the result.

Tony Hey

Electronics and Computer Science Department
University of Southampton
England

May 1996

FEYNMAN'S PREFACE

When I produced the *Lectures on Physics*, some thirty years ago now, I saw them as an aid to students who were intending to go into physics. I also lamented the difficulties of cramming several hundred years' worth of science into just three volumes. With these *Lectures on Computation*, matters are somewhat easier, but only just. Firstly, the lectures are not aimed solely at students in computer science, which liberates me from the shackles of exam syllabuses and allows me to cover areas of the subject for no more reason than that they are interesting. Secondly, computer science is not as old as physics; it lags by a couple of hundred years. However, this does not mean that there is significantly less on the computer scientist's plate than on the physicist's: younger it may be, but it has had a far more intense upbringing! So there is still plenty for us to cover.

Computer science also differs from physics in that it is not actually a science. It does not study natural objects. Neither is it, as you might think, mathematics; although it does use mathematical reasoning pretty extensively. Rather, computer science is like engineering — it is all about getting something to do something, rather than just dealing with abstractions as in pre-Smith geology[1]. Today in computer science we also need to "go down into the mines" — later we can generalize. It does no harm to look at details first.

But this is not to say that computer science is all practical, down to earth bridge-building. Far from it. Computer science touches on a variety of deep issues. It has illuminated the nature of language, which we thought we understood: early attempts at machine translation failed because the old fashioned notions about grammar failed to capture all the essentials of language It naturally encourages us to ask questions about the limits of computability about what we can and cannot know about the world around us. Computer science people spend a lot of their time talking about whether or not man is merely a machine, whether his brain is just a powerful computer that might one day be copied; and the field of 'artificial intelligence' — I prefer the term 'advanced applications' — might have a lot to say about the nature of 'rea

---

[1] William Smith was the father of modern geology; in his work as a canal and mining engineer observed the systematic layering of the rocks, and recognized the significance of fossils as a means of determining the age of the strata in which they occur. Thus was he led to formulate [illegible] principle in which rocks are successively laid down upon older layers. Prior to Smith [illegible] than an empirical scien

intelligence, and mind. Of course, we might get useful ideas from studying how the brain works, but we must remember that automobiles do not have legs like cheetahs nor do airplanes flap their wings! We do not need to study the neurologic minutiae of living things to produce useful technologies; but even wrong theories may help in designing machines. Anyway, you can see that computer science has more than just technical interest.

These lectures are about what we can and can't do with machines today, and why. I have attempted to deliver them in a spirit that should be recommended to all students embarking on the writing of their PhD theses: imagine that you are explaining your ideas to your former smart, but ignorant, self, at the beginning of your studies! In very broad outline, after a brief introduction to some of the fundamental ideas, the next five chapters explore the limitations of computers – from logic gates to quantum mechanics! The second part consists of lectures by invited experts on what I've called advanced applications – vision, robots, expert systems, chess machines and so on[2].

[2]A companion volume to these lectures is in preparation. As far as is possible, this second volume will contain articles on 'advanced applications' by the same experts who contributed to Feynman's course but updated to reflect the present state of the art. [Editors]

ONE

INTRODUCTION TO COMPUTERS

Computers can do lots of things. They can add millions of numbers in the twinkling of an eye. They can outwit chess grandmasters. They can guide weapons to their targets. They can book you onto a plane between a guitar-strumming nun and a non-smoking physics professor. Some can even play the bongoes. That's quite a variety! So if we're going to talk about computers, we'd better decide right now which of them we're going to look at, and how.

In fact, we're not going to spend much of our time looking at individual machines. The reason for this is that once you get down to the guts of computers you find that, like people, they tend to be more or less alike. They can differ in their functions, and in the nature of their inputs and outputs — one can produce music, another a picture, while one can be set running from a keyboard, another by the torque from the wheels of an automobile — but at heart they are very similar. We will hence dwell only on their innards. Furthermore, we will not assume anything about their specific Input/Output (I/O) structure, about how information gets into and out of the machine; all we care is that, however the input gets in, it is in digital form, and whatever happens to the output, the last the innards see of it, it's digital too; by digital, I mean binary numbers: 1's and 0's.

What does the inside of a computer look like? Crudely, it will be built out of a set of simple, basic elements. These elements are nothing special — they could be control valves, for example, or beads on an abacus wire — and there are many possible choices for the basic set. All that matters is that they can be used to build everything we want. How are they arranged? Again, there will be many possible choices; the relevant structure is likely to be determined by considerations such as speed, energy dissipation, aesthetics and what have you. Viewed this way, the variety in computers is a bit like the variety in houses: a Beverly Hills condo might seem entirely different from a garage in Yonkers, but both are built from the same things — bricks, mortar, wood, sweat — only the condo has more of them, and arranged differently according to the needs of the owners. At heart they are very similar.

Let us get a little abstract for the moment and ask: *how* do you connect up *which* set of elements to do the *most* things? It's a deep question. The answer again is that, up to a point, it doesn't matter. Once you have a computer that can

do a few things — strictly speaking, one that has a certain "sufficient set" of basic procedures — it can do basically anything any other computer can do. This, loosely, is the basis of the great principle of "Universality". Whoa! You cry. My pocket calculator can't simulate the red spot on Jupiter like a bank of Cray supercomputers! Well, yes it can: it would need rewiring, and we would need to soup up its memory, and it would be damned slow, but if it had long enough it could reproduce anything the Crays do. Generally, suppose we have two computers **A** and **B**, and we know all about **A** — the way it works, its "state transition rules" and what-not. Assume that machine **B** is capable of merely *describing* the state of **A**. We can then use **B** to simulate the running of **A** by describing its successive transitions; **B** will, in other words, be mimicking **A**. It could take an eternity to do this if **B** is very crude and **A** very sophisticated, but **B** will be able to do whatever **A** can, eventually. We will prove this later in the course by designing such a **B** computer, known as a Turing machine.

Let us look at universality another way. Language provides a useful source of analogy. Let me ask you this: which is the *best* language for describing something? Say: a four-wheeled gas-driven vehicle. Of course, most languages, at least in the West, have a simple word for this; we have "automobile", the English say "car", the French "voiture", and so on. However, there will be some languages which have not evolved a word for "automobile", and speakers of such tongues would have to invent some, possibly long and complex, description for what they see, in terms of their basic linguistic elements. Yet none of these descriptions is inherently "better" than any of the others: they all do their job, and will only differ in efficiency. We needn't introduce democracy just at the level of words. We can go down to the level of alphabets. What, for example, is the best alphabet for English? That is, why stick with our usual 26 letters? Everything we can do with these, we can do with three symbols — the Morse code, dot, dash and space; or two — a Baconian cipher, with *A* through *Z* represented by five-digit binary numbers. So we see that we can choose our basic set of elements with a lot of freedom, and all this choice really affects is the efficiency of our language, and hence the sizes of our books: there is no "best" language or alphabet — each is logically universal, and each can model any other. Going back to computing, universality in fact states that the set of complex tasks that can be performed using a "sufficient" set of basic procedures is independent of the specific, detailed structure of the basic set.

For today's computers to perform a complex task, we need a precise and complete description of how to do that task in terms of a sequence of simple basic procedures — the "software" — and we need a machine to carry out these

procedures in a specifiable order — this is the "hardware". This instructing has to be exact and unambiguous. In life, of course, we never tell each other *exactly* what we want to say; we never need to, as context, body language, familiarity with the speaker, and so on, enable us to "fill in the gaps" and resolve any ambiguities in what is said. Computers, however, can't yet "catch on" to what is being said, the way a person does. They need to be told in excruciating detail exactly what to do. Perhaps one day we will have machines that can cope with approximate task descriptions, but in the meantime we have to be very prissy about how we tell computers to do things.

Let us examine how we might build complex instructions from a set of rudimentary elements. Obviously, if an instruction set *B* (say) is very simple, then a complex process is going to take an awful lot of description, and the resulting "programs" will be very long and complicated. We may, for instance, want our computer to carry out all manner of numerical calculations, but find ourselves with a set *B* which doesn't include multiplication as a distinct operation. If we tell our machine to multiply 3 by 35, it says "what?" But suppose *B* does have addition; if you think about it, you'll see that we can get it to multiply by adding lots of times — in this case, add 35 to itself twice. However, it will clearly clarify the writing of *B*-programs if we augment the set *B* with a separate "multiply" instruction, *defined* by the chunk of basic *B* instructions that go to make up multiplication. Then when we want to multiply two numbers, we say "computer, 3 times 35", and it now recognizes the word "times" — it is just a lot of adding, which it goes off and does. The machine breaks these compound instructions down into their basic components, saving us from getting bogged down in low level concepts all the time. Complex procedures are thus built up stage by stage. A very similar process takes place in everyday life; one replaces with one word a set of ideas and the connections between them. In referring to these ideas and their interconnections we can then use just a single word, and avoid having to go back and work through all the lower level concepts. Computers are such complicated objects that simplifying ideas like this are usually necessary, and good design is essential if you want to avoid getting completely lost in details.

We shall begin by constructing a set of primitive procedures, and examine how to perform operations such as adding two numbers or transferring two numbers from one memory store to another. We will then go up a level, to the next order of complexity, and use these instructions to produce operations like multiply and so on. We shall not go very far in this hierarchy. If you want to see how far you can go, the article on Operating Systems by P.J. Denning and

R.L. Brown (*Scientific American*, September 1984, pp. 96-104) identifies thirteen levels! This goes from level 1, that of electronic circuitry — registers, gates, buses — to number 13, the Operating System Shell, which manipulates the user programming environment. By a hierarchical compounding of instructions, basic transfers of 1's and 0's on level one are transformed, by the time we get to thirteen, into commands to land aircraft in a simulation or check whether a forty digit number is prime. We will jump into this hierarchy at a fairly low level, but one from which we can go up or down.

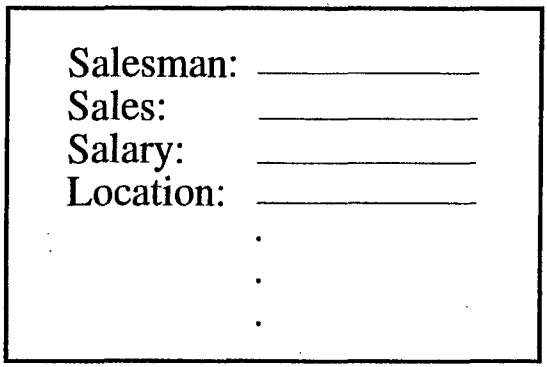
Also, our discussion will be restricted to computers with the so-called "Von Neumann architecture". Don't be put off by the word "architecture"; it's just a big word for how we arrange things, only we're arranging electronic components rather than bricks and columns. Von Neumann was a famous mathematician who, besides making important contributions to the foundations of quantum mechanics, also was the first to set out clearly the basic principles of modern computers[1]. We will also have occasion to examine the behavior of several computers working on the same problem, and when we do, we will restrict ourselves to computers that work in sequence, rather than in parallel; that is, ones that take turns to solve parts of a problem rather than work simultaneously. All we would lose by the omission of "parallel processing" is speed, nothing fundamental.

We talked earlier about computer science not being a real science. Now we have to disown the word "computer" too! You see, "computer" makes us think of arithmetic — add, subtract, multiply, and so on — and it's easy to assume that this is all a computer does. In fact, conventional computers typically have one place where they do their basic math, and the rest of the machine is for the computer's main task, which is shuffling bits of paper around — only in this case the paper notes are digital electrical signals. In many ways, a computer is reminiscent of a bureaucracy of file clerks, dashing back and forth to their filing cabinets, taking files out and putting them back, scribbling on bits of paper, passing notes to one another, and so on; and this metaphor, of a clerk shuffling paper around in an office, will be a good place to start to get some of the basic ideas of computer structure across. We will go into this in some detail, and the impatient among you might think too much detail, but it is a perfect model for communicating the essentials of what a computer does, and is hence worth spending some time on.

[1] Actually, there is currently a lot of interest in designing "non-Von Neumann" machines. These will be discussed by invited "experts" in a companion volume. [Editors]

1.1: The File Clerk Model

Let's suppose we have a big company, employing a lot of salesmen. An awful lot of information about these salesmen is stored in a big filing system somewhere, and this is all administered by a clerk. We begin with the idea that the clerk knows how to get the information out of the filing system. The data is stored on cards, and each card has the name of the salesman, his location, the number and type of sales he has made, his salary, and so on and so forth.



Now suppose we are after the answer to a specific question: "What are the total sales in California?" Pretty dull and simple, and that's why I chose it: you must start with simple questions in order to understand difficult ones later. So how does our file clerk find the total sales in California? Here's one way he could do it:

Take out a card
If the "location" says *California*, then
    Add the number under "sales" to a running count called "total"
Put "sales" card back
Take next card and repeat.

Obviously you have to keep this up until you've gone through all the cards. Now let's suppose we've been unfortunate enough to hire particularly stupid clerks, who can read, but for whom the above instructions assume too much: say, they don't know how to keep a running count. We need to help them a little bit more. Let us invent a "total" card for our clerk to use. He will use this to keep a running total in the following way:

Take out next "sales" card
If *California*, then
    Take out "total" card
    Add sales number to number on card
    Put "total" card back
Put "sales" card back
Take out next "sales" card and repeat.

This is a very mechanical rendering of how a crude computer could solve this adding problem. Obviously, the data would not be stored on cards, and the machine wouldn't have to "take out a card" — it would read the stored information from a register. It could also write from a register to a "card" without physically putting something back.

Now we're going to stretch our clerk! Let's assume that each salesman receives not only a basic salary from the company, but also gets a little on commission from sales. To find out how much, we multiply his sales by the appropriate percentage. We want our clerk to allow for this. Now he is cheap and fast, but unfortunately too dumb to multiply[2]. If we tell him to multiply 5 by 7 he says "what?" So we have to teach him to multiply. To do this, we will exploit the fact that there is one thing he does well: he can get cards very, very quickly.

We'll work in base two. As you all probably know, the rules for binary arithmetic are easier than those for base ten; the multiplication table is so small it will easily fit on one card. We will assume that even our clerk can remember these; all he needs are "shift" and "carry" operations, as the following example makes clear:

In decimal: 22 x 5 = 110

```
In binary:    10110          In decimal:  22
               101                        5
              10110
             10110   (shift twice)
             -------                     ---
             1101110                     110
```

[2]As an aside, although our dense file clerk is assumed in these examples to be a man, no sexist implications are intended! [RPF]

So as long as our clerk can shift and carry he can, in effect, multiply. He does it very stupidly, but he also does it very quickly, and that's the point of all this: the inside of a computer is as dumb as hell but it goes like mad! It can perform very many millions of simple operations a second and is just like a very fast dumb file clerk. It is only because it is able to do things so fast that we do not notice that it is doing things very stupidly. (Interestingly enough, neurons in the brain characteristically take milliseconds to perform elementary operations, which leaves us with the puzzle of why is the brain so smart? Computers may be able to leave brains standing when it comes to multiplication, but they have trouble with things even small children find simple, like recognizing people or manipulating objects.)

To go further, we need to specify more precisely our basic set of operations. One of the most elementary is the business of transferring information from the cards our clerk reads to some sort of scratch pad on which he can do his arithmetic:

Transfer operations

"Take Card X" = Information on card X written to pad
"Replace Card Y" = Information on pad written on card Y

All we have done is to define the instruction "take card *X*" to mean copying the information on card *X* onto the pad, and similarly with "replace card *Y*". Next, we want to be able to instruct the clerk to check if the location on card *X* was "California". He has to do this for each card, so the first thing he has to do is be able to remember "California" from one card to the next. One way to help him do this is to have *California* written on yet another card *C* so that his instructions are now:

Take card X (from store to pad)
Take card C (from store to pad)
Compare what is on card X with what is on card C.

We then tell him that if the contents match, do so and so, and if they don't, put the cards back and take the next ones. Keeping on taking out and putting back the California card seems to be a bit inefficient, and indeed, you don't have to do that; you can keep it on the pad for a while instead. This would be better, but it all depends on how much room the clerk has on his pad and how many pieces of information he needs to keep. If there isn't much room, then there will have

to be a lot of shuffling cards back in and out. We have to worry about such things!

We can keep on breaking the clerk's complex tasks down into simpler, more fundamental ones. How, for example, do we get him to look at the "location" part of a card from the store? One way would be to burden the poor guy with yet another card, on which is written something like this:

0000 0000 0000 0000 0000 1111 0000 0000 0000 0000...

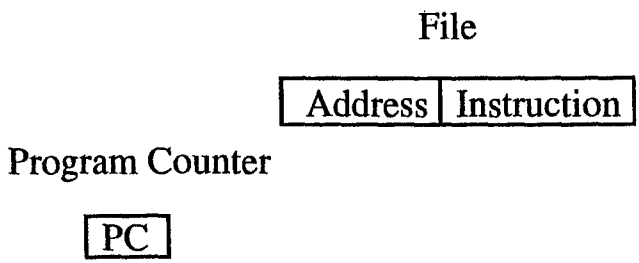
Each sequence of digits is associated with a particular piece of information on the card: the first set of zeroes is "lined up" with the salesman's name, the next with his age, say, and so on. The clerk zips through this numeric list until he hits a set of 1's, and then reads the information next to them. In our case, the 1111 is lined up with California. This sort of location procedure is actually used in computers, where you might use a so-called "bitwise AND" operation (we'll discuss this later). This little diversion was just to impress upon you the fact that we need not take any of our clerk's skills for granted — we can get him to do things increasingly stupidly.

1.2: Instruction sets

Let's take a look at the clerk's scratch pad. We haven't yet taught the clerk how to use this, so we'll do that now. We will assume that we can break down the instructions he can carry out into two groups. Firstly, there is a core "instruction set" of simple procedures that comes with the pad — add, transfer, etc. These are in the hardware: they do not change when we change the problem. If you like, they reflect the clerk's basic abilities. Then we have a set which is specific to the task, say calculating a salesman's commission. The elements of this set are built out of the instructions in the core set in ways we have discussed, and represent the combinations of the clerk's talents that will be required for him to carry out the task at hand.

The first thing we need to get the clerk to do is do things in the right order, that is, to follow a succession of instructions. We do this by designating one of the storage areas on the pad as a "program counter". This will have a number on it, which indicates whereabouts in the calculational procedure the clerk is. As far as the clerk is concerned, the number is an address — he knows that buried in the filing system is a special "instruction file" cabinet, and the number in the counter labels a card in that file which he has to go and get; on

the card is the instruction as to what he is to do next. So he gets the instruction and stores it on his pad in an area which we call the "instruction register".



Before he carries out the instruction, however, he prepares for the next one by incrementing the program counter; he does this simply by adding one to it. Then he does whatever the instruction in the register tells him to do. Using a bracketed notation where () means "contents of" — remember this, as we will be using it a lot — we can write this sequence of actions as follows[3]:

Fetch instruction from address PC
PC ← (PC) + 1
Do instruction

The second line is a fancy way of saying that the counter *PC* "gets" the new value (*PC*)+1. The clerk will also need some temporary storage areas on the pad; to enable him to do arithmetic, for example. These are called registers, and give him a place to store something while he goes and finds some other number. Even if you are only adding two numbers you need to remember the first until you have fetched the second! Everything must be done in sequence and the registers allow us to organize things. They usually have names; in our case we will have four, which we call registers *A*, *B* and *X*, and the fourth, *C*, which is special — it can only store one bit of data, and we will refer to it as the "carry" register. We could have more or fewer registers — generally, the more you have, the easier a program is to write — but four will suffice for our purposes.

[3]The conventions adopted for such "Register Transfer Language" vary according to the whim of the author. We choose to follow the so-called "right to left" convention utilized in standard programming languages. [Editors]

So our clerk knows how to find out what he has to do, and when. Let's now look at the core instruction set for his pad. The first kind of instruction concerns the transfer of data from one card to another. For example, suppose we have a memory location *M* on the pad. We want to have an instruction that transfers the contents of register *A* into *M*:

$$\text{Transfer (A) into M} \qquad \text{or} \qquad \text{M} \leftarrow \text{(A)}$$

Similarly, we might want to go the other way, and write the contents of *M* into *A*:

$$\text{Transfer (M) into A} \qquad \text{or} \qquad \text{A} \leftarrow \text{(M)}$$

M, incidentally, is not necessarily designed for temporary storage like *A*. We must also have analogous instructions for register *B*:

$$\begin{array}{lcl} \text{Transfer (B) to M} & \text{or} & \text{M} \leftarrow \text{(B)} \\ \text{Transfer (M) to B} & \text{or} & \text{B} \leftarrow \text{(M)} \end{array}$$

Register *X* we will use a little differently. We shall allow transfers from *B* to *X* and *X* to *B*:

$$\text{X} \leftarrow \text{(B)} \qquad \text{and} \qquad \text{B} \leftarrow \text{(X)}.$$

In addition, we need to be able to keep tabs on, and manipulate, our program counter *PC*. This is obviously necessary: if the clerk shoots off to execute some multiplication, say, when he comes back he has to know what to do next — he has to remember the number in *PC*. In fact, we'll keep it in register *X*. Thus we add the transfer instructions:

$$\text{PC} \leftarrow \text{(X)} \qquad \text{and} \qquad \text{X} \leftarrow \text{(PC)}.$$

Next, we need arithmetical and logical operations. The most basic of these is a "clear" instruction:

$$\text{Clear A, or A} \leftarrow 0.$$

This means, whatever is in *A*, forget it, wipe it out. Then we need an Add operation:

$$\text{Add B to A, or } A \leftarrow (A) + (B)$$

This means that register *A* receives the sum of the contents of *B* and the previous contents of *A*. We also have a shift operation, which will enable us to do multiplication without having to introduce a core instruction for it:

Shiftleft A and Shiftright A

The first merely moves all the bits in *A* one place to the left. If this shift causes the leftmost bit to overflow we store it in the carry register *C*. We can also shift our number to the right; I have no use for this in mind, but it could come in handy!

The next instructions are logical ones. We will be looking at these in greater detail in the next chapter, but I will mention them here for completeness. There are three that will interest us: AND, OR and XOR. Each is a function of two digital "inputs" x and y. If *both* inputs are 1, then AND gives you 1; otherwise it gives you zero. As we will see, the AND operation turns up in binary addition, and hence multiplication; if we view x and y as two digits we are adding, then (x AND y) is the carry bit: it's only one if both digits are one. In terms of our registers, x and y are (*A*) and (*B*), and AND operates on these:

$$\text{AND: } A \leftarrow (A) \wedge (B),$$

where we have used the logical symbol $\wedge$ for the AND operation. The result of acting on a pair of variables with an operator such as AND is often summarized in a "truth table" (Table 1.1.):

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

$$X = A \wedge B$$

Table 1.1 The Truth Table for the AND Operator

Our other two operators can be described in similar terms. The OR also operates on (*A*) and (*B*); it gives a one unless both (*A*) and (*B*) are zero — (*x* OR *y*) is one if *x or y* is one. XOR, or the "exclusive or", is similar to OR, except it gives zero if both (*A*) and (*B*) are one; in the binary addition of *x* and *y*, it corresponds to what you get if you add *x* to *y* and ignore any carry bits. A binary add of 1 and 1 is 10, which is zero if you forget the carry. We can introduce the relevant logical symbols:

$$\text{OR} \quad A \leftarrow (A) \vee (B)$$
$$\text{XOR} \quad A \leftarrow (A) \oplus (B)$$

The actions of OR and XOR can also be summarized by truth tables:

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

$X = A \vee B$

OR

A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

$X = A \oplus B$

XOR

Table 1.2 The Truth Tables for the OR and XOR Operators

Two more operations that it turns out are convenient to have are the instructions to increment or decrement the contents of *A* by one:

$$\text{Increment } A, \text{ or } A \leftarrow (A) + 1$$
$$\text{Decrement } A, \text{ or } A \leftarrow (A) - 1$$

Obviously, one can go on adding instructions that may or may not turn out to be very convenient. Here, we already have more than the minimum number necessary to be able to do some useful calculations. However, we want to be able to do as much as possible, so we can bring in other instructions. One other that will be useful is one that allows us to put a data item directly into a register. For example, rather than writing *California* on a card and then transferring from card to pad, it would be convenient to be able to write *California* on the pad directly. Thus we introduce the "Direct Load" instruction:

Direct Load: $B \leftarrow N$,

where N is any constant.

There is one class of instructions that it is vital we add: that of branches, or jumps. A "jump to *Z*" is basically an instruction for the clerk to look in (instruction) location *Z*; that is, it involves a change in the value of the program counter by more than the usual increment of one. This enables our clerk to leap from one part of a program to another. There are two kinds of jumps, "unconditional" and "conditional". The unconditional jump we have touched on above:

Jump to (Z) or $PC \leftarrow (Z)$

The really new thing is the conditional jump:

Jump to (Z) *if C=1*

With this instruction, the jump to location (*Z*) is only made if the carry register *C* contains a carry bit. The freedom given by this conditional instruction will be vital to the whole design of any interesting machines.

There are many other kinds of jump we can add. Sometimes it turns out to be convenient to be able to jump not only to a definite location but to one a specific number of steps further on in the program. We can therefore introduce jump instructions that add this number of steps to the program counter:

Jump to (PC) + (Z) or $PC \leftarrow (PC) + (Z)$

Jump to (PC) + (Z) if C=1

Finally, there is one more command that we need; namely, an instruction that tells our clerk to quit:

Halt.

With these instructions, we can now do anything we want and I will suggest some problems for you to practice on below. Before we do that, let us summarize where we are and what we're trying to do. The idea has been to

outline the basic computer operations and methods and indicate what is actually in a computer (I haven't been describing an actual design, but I've come close). In a simple computer there are only a few registers; more complex ones have more registers, but the concepts are basically the same, just scaled up a bit.

It is worth looking at how we represent the instructions we considered above. In our particular case the instructions contain two pieces: an instruction address and an instruction number, or "opcode":

Instruction address	Instruction opcode/number

For example, one of the instructions was "put the contents of memory M into register A". The computer doesn't speak English, so we have to encode this command into a form it can understand; in other words, into a binary string. This is the opcode, or instruction number, and its length clearly determines how many different instructions we can have. If the opcode is a four-digit binary number, then we can have $2^4 = 16$ different instructions, of which loading the contents of a memory address into A is just one. The second part of the instruction is the instruction address, which tells the computer where to go to find what it has to load into A; that is, memory address M. Some instructions, such as "clear A", don't require an address direction.

Details such as how the instruction opcodes are represented or exactly how things are set out in memory are not needed to use the instructions. This is the first and most elementary step in a series of hierarchies. We want to be able to maintain such ignorance consistently. In other words, we only want to have to think about the lower details once and then design things so that the next guy who comes along and wants to use your structure does not have to worry about the lower level details.

There is one feature that we have so far ignored completely. Our machine as described so far would not work because we have no way of getting numbers in and out. We must consider input and output. One quick way to go about things would be to assign a particular place in memory, say address 17642, to be the input, and attach it to a keyboard so that someone from outside the machine could change its contents. Similarly, another location, say 17644, might be the output, which would be connected to a TV monitor or some other device,

so that the results of a calculation can reach the outside world.

Now there are two ways in which you can increase your understanding of these issues. One way is to remember the general ideas and then go home and try to figure out what commands you need and make sure you don't leave one out. Make the set shorter or longer for convenience and try to understand the tradeoffs by trying to do problems with your choice. This is the way I would do it because I have that kind of personality! It's the way I study — to understand something by trying to work it out or, in other words, to understand something by creating it. Not creating it one hundred percent, of course; but taking a hint as to which direction to go but not remembering the details. These you work out for yourself.

The other way, which is also valuable, is to read carefully how someone else did it. I find the first method best for me, once I have understood the basic idea. If I get stuck I look at a book that tells me how someone else did it. I turn the pages and then I say "Oh, I forgot that bit", then close the book and carry on. Finally, after you've figured out how to do it you read how they did it and find out how dumb your solution is and how much more clever and efficient theirs is! But this way you can understand the cleverness of their ideas and have a framework in which to think about the problem. When I start straight off to read someone else's solution I find it boring and uninteresting, with no way of putting the whole picture together. At least, that's the way it works for me!

Throughout the book, I will suggest some problems for you to play with. You might feel tempted to skip them. If they're too hard, fine. Some of them are pretty difficult! But you might skip them thinking that, well, they've probably already been done by somebody else; so what's the point? Well, of *course* they've been done! But so what? Do them for the *fun* of it. That's how to learn the knack of doing things when you have to do them. Let me give you an example. Suppose I wanted to add up a series of numbers,

$$1 + 2 + 3 + 4 + 5 + 6 + 7 \ldots$$

up to, say, 62. No doubt you know how to do it; but when you play with this sort of problem as a kid, and you haven't been shown the answer . . . it's *fun* trying to figure out how to do it. Then, as you go into adulthood, you develop a certain confidence that you can discover things; but if they've already been discovered, that shouldn't bother you at all. What one fool can do, so can another, and the fact that some other fool beat you to it shouldn't disturb you:

you should get a kick out of having discovered something. Most of the problems I give you in this book have been worked over many times, and many ingenious solutions have been devised for them. But if you keep proving stuff that others have done, getting confidence, increasing the complexities of your solutions — for the fun of it — then one day you'll turn around and discover that *nobody actually did that one!* And that's the way to become a computer scientist.

I'll give you an example of this from my own experience. Above, I mentioned summing up the integers. Now, many years ago, I got interested in the generalization of such a problem: I wanted to figure out formulae for the sums of squares, cubes, and higher powers, trying to find the sum of m things each up to the nth power. And I cracked it, finding a whole lot of nice relations. When I'd finished, I had a formula for each sum in terms of a number, one for each n, that I couldn't find a formula for. I wrote these numbers down, but I couldn't find a general rule for getting them. What was interesting was that they were integers, until you got to n=13 — when it wasn't (it was something just over 691)! Very shocking! And fun.

Anyway, I discovered later that these numbers had actually been discovered back in 1746. So I had made it up to 1746! They were called "Bernoulli Numbers". The formula for them is quite complicated, and unknown in a simple sense. I had a "recursion relation" to get the next one from the one before, but I couldn't find an arbitrary one. So I went through life like this, discovering next something that had first been discovered in 1889, then something from 1921 . . . and finally I discovered something that had the same date as when I discovered it. But I get so much fun out of doing it that I figure there must be others out there who do too, so I am giving you these problems to enjoy yourselves with. (Of course, everyone enjoys themselves in different ways.) I would just urge you not to be intimidated by them, nor put off by the fact that they've been done. You're unlikely to discover something new without a lot of practice on old stuff, but further, you should get a heck of a lot of fun out of working out funny relations and interesting things. Also, if you read what the other fool did, you can appreciate how hard it was to do (or not), what he was trying to do, what his problems were, and so forth. It's much easier to understand things after you've fiddled with them before you read the solution. So for all these reasons, I suggest you have a go.

Problem 1.1: (a) Go back to our dumb file clerk and the problem of finding out the total number of sales in California. Would you advise the management to hire two clerks to do the job quicker? If so, how would you use them, and could you speed up the calculation by a factor of two? You have to think about

how the clerks get their instructions. Can you generalize your solution to K, or even 2^K clerks?

(b) What kinds of problems can K clerks actually speed up? What kinds can they apparently not?

(c) Most present-day computers only have one central processor — to use our analogy, one clerk. This single file clerk sits there all day long working away like a fiend, taking cards in and out of the store like mad. Ultimately, the speed of the whole machine is determined by the speed at which the clerk — that is, the central processor — can do these operations. Let's see how we can maybe improve the machine's performance. Suppose we want to compare two n-bit numbers, where n is a large number like 1024; we want to see if they're the same. The easiest way for a single file clerk to do this would be to work through the numbers, comparing each digit in sequence. Obviously, this will take a total time proportional to n, the number of digits needing checking. But suppose we can hire n file clerks, or $2n$ or perhaps $3n$: it's up to us to decide how many, but the number must be proportional to n. Now, it turns out that by increasing the number of file clerks we can get the comparison-time down to be proportional to $\log_2 n$. Can you see how?

(d) If you can do this compare problem, you might like to try a harder one. See if you can figure out a way of adding two n-bit numbers in "log n" time. This is more difficult because you have to worry about the carries!

Problem 1.2: The second problem concerns getting the clerk to multiply (multiplication, remember, is not included in his basic instruction set). The problem comes in two parts. First, find the appropriate set of basic instructions required to perform multiplication. Having these, let's assume we save them some place in the machine so that we don't have to duplicate them every time we want to multiply; put them, say, in locations m to $m+k$. Show how we can give the clerk instructions to use this set-up to do a multiplication and return to the right place in the program.

1.3: Summary

We have now covered enough stuff for us to go on to understand any particular machine design. But instead of looking at any particular machine in detail we are going to do something rather different. From where we are now we can go

up, down or sideways. What do I mean by this? Well, "up" means hiding more details of the workings of the machine from the user — introducing more levels of abstraction. We have already seen some examples of this; for example, building up new operations such as multiplication from operations in our basic set. Every time we want to multiply we just use this multiply "subroutine". Another example worth discussing is the ability to talk about algebraic variables rather than locations in memory. Suppose you want to take the sum of X and Y, and call it Z:

$$Z = X + Y$$

X and Y are already known to the computer and stored at specific locations in memory. The first thing we have to do is assign some place in memory to store the value of Z and then ensure that this location holds the sum of the contents of the X and Y memory cells. Now we know all about Z and can use it in other expressions, such as $Z+X$. It is clearly much simpler talking about algebraic variables rather than memory locations all the time although it is quite a job to set this up. However, up to now we have had to know exactly where a number is located in order to make a transfer. We can now introduce a new number Z, and say to the computer "I want a number Z — find a place to put it and don't bother telling me where it is!" This is what I mean by moving "up".

Of course, we already went "up" a bit when we summarized operations by instructions such as "Clear A", and so on. This sort of shorthand is introduced for our benefit, and programs written in it cannot be understood directly by the machine itself. Such "assembly language" programs have to be translated into a "machine language" that the computer can understand, and this is done by a program called an "assembler". The next level up, where we have multiplication and variables and so on, needs another program to translate these "high-level" programs into assembly language. These translation programs are called "compilers" or "interpreters". The difference between them is in when the translation is done. An interpreter works out what to do step by step, as the program runs, interpreting each successive instruction in terms of the cruder language. A compiler takes the program as a whole and converts it all into assembly or machine language before the program is run. Compilers have the advantage that, in some cases, looking at the whole "code" it is possible for them to find clever ways to simplify the required operations. This is the nub of the important field of "compiler optimization" and is becoming of increasing importance for the new types of "non-Von Neumann" parallel computers.

Clearly, one can keep going up in level, putting together new algorithms, programming languages, adding the ability to manipulate "files" containing programs and data, and so on. Nowadays it is possible for most people to happily work at these higher levels using high-level languages to program their machines. Imagine how tedious it was — and is, for modern computer designers — to work solely in machine code!

That was "up"; now it's time to go down. How can anything be simpler than our dumb file clerk model and our simple list of instructions? What we have not considered is what our file clerk is *made of*; to be more realistic, we have not looked at how we would actually build electronic circuits to perform the various operations we have discussed. This is where we are going to go next, but before we do, let me say what I mean by moving "sideways". Sideways means looking at something entirely different from our Von-Neumann architecture, which is distinguished by having a single Central Processing Unit (CPU) and everything coming in and going out through the "fetch and execute" cycle. Many other more exotic computer architectures are now being experimented with, and some are being marketed as machines people can buy. Going "sideways" therefore means remaining at the same level of detail but examining how calculations would be performed by machines with differing core structures. We already invited you to think of such "parallel" computers with the problem of organizing several file clerks to work together on the same problem.

TWO

COMPUTER ORGANIZATION

2.1: Gates and Combinational Logic

We shall begin our trip downwards by looking at what we need to be able to perform our various simple operations — adds, transfers, control decisions, and so forth. We will see that we will need very little to do *all* of these things! To get an idea of what's involved, let's start with the "add" operation. Our first, important, decision is to restrict ourselves to working in base 2, the binary system: the only digits are 1 and 0, and as we shall see, these can easily fit into a computer framework: we represent them electronically by a simple "on/off" state of a component. In the meantime, we shall adopt a somewhat picturesque, and simpler, technique for depicting binary numbers: rather than just write out strings of 1's and 0's, we will envisage a binary number to be a compartmentalized strip of plastic, rather like an ice tray, with each compartment corresponding to a digit; if the compartment is empty, that means the digit is 0, but if the digit is 1 we put a pebble there. Now let us take two such strips, and pretend these are the numbers to be added — the "summands". Underneath these two we have laid out one more, to hold the answer (Fig. 2.1):

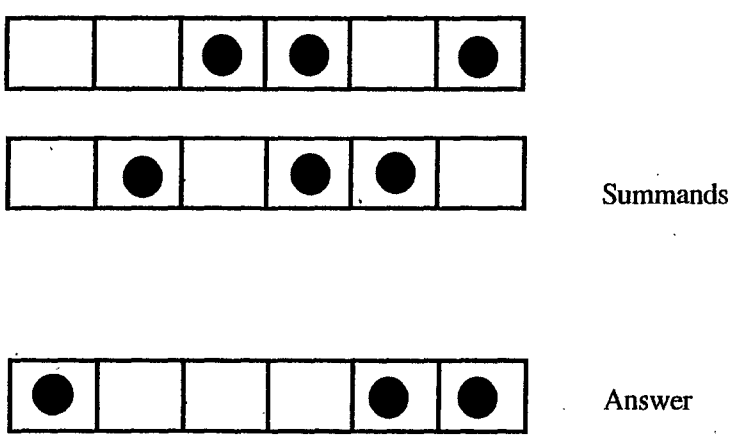


Fig. 2.1 A Pictorial Depiction of Binary Addition

This turns our abstract mathematical problem into a matter of real world "mechanics". All we need to do the addition is a simple set of rules for moving the pebbles. Now instead of pebbles, which are slow and hard to handle, we

could use anything else, say, wires with either a high voltage for 1 and low voltage for 0. The basic problem is the same: what are the rules for combining pebbles or voltages? For binary addition the basic rules are:

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 0 \text{ plus a carry}$$

So now you can imagine giving instructions on how to move the pebbles to someone who is a complete idiot: if you have two pebbles here, one above the other, you put no pebble in the sum space beneath them, but carry one over one space to the left — and so on. The marvellous thing is, with sufficiently detailed rules this "idiot" is able to add two numbers of any size! With a slightly more detailed set, he can graduate to multiplication. He can even, eventually, do very complicated things involving hypergeometric functions and what have you. What you tell an apparent idiot, who can do no more than shuffle pebbles around, is enough for him to tackle the evaluation of hypergeometric functions and the like. If he shifts the pebbles quickly enough, he could even do this quicker than you — in that respect, he is justified in thinking himself smarter than you!

Of course, real machines do not calculate by fiddling with pebbles (although don't forget the abacus of old!). They manipulate electronic signals. So, if we are going to implement all of our notions about operations, we have to start thinking about electric circuits. Let us ditch our ice trays and stones and look at the problem of building a real, physical adder to add two binary digits *A* and *B*. This process will result in a sum, *S*, and a carry, *C*; we set this out in a table as follows:

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 2.1 A "Truth Table" for Binary Addition

Let us represent our adder as a black box with two wires going in — A and B — and two coming out — S and C[1] (Fig. 2.2):

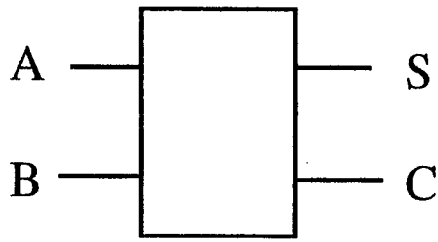


Fig. 2.2 A Black Box Adder

We will detail the actual nature of this box shortly. For the moment, let us take it for granted that it works. (As an aside, let us ask how many such adders we would need to add two r-bit numbers? You should be able to convince yourself that $(2r\text{-}1)$ single-bit adders are required. This again illustrates our general principle of systematically building complicated things from simpler units.)

Let us go back to our black box, single-bit adder. Suppose we just look at the carry bit: this is only non-zero if both A and B are one. This corresponds precisely to the behavior of the so-called AND gate from Boolean logic. Such a gate is itself no more than a black box, with two inputs and one output, and a "truth table" which tells us how the output depends on the inputs. This truth table, and the usual pictorial symbol for the AND gate are given below:

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

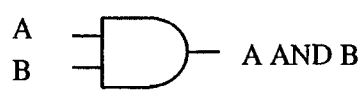


Fig. 2.3 The AND Gate

[1]This box is sometimes known as a "half adder". We will encounter a "full adder" later in this chapter. [RPF]

Simple enough: *A* AND *B* is 1 if, and only if, *A* is 1 *and B* is 1. Thus, carry and "and" are really the same thing, and the carry bit for our adder may be obtained by feeding the *A* and *B* wires into an and gate. Although I have described the gate as a black box, we do in fact know exactly how to build one using real materials, with real electronic signals acting as values for *A*, *B* and *C*, so we are well on the way to implementing the adder. The sum bit of the adder, *S*, is given by another kind of logic gate, the "exclusive or" or XOR gate. Like the AND, this has a defining truth table and a pretty symbol (Fig. 2.4):

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

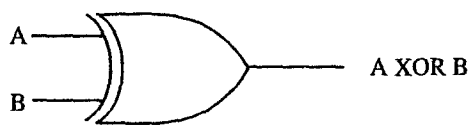


Fig. 2.4 The XOR Gate

A XOR *B* is 1 if *A or B* is 1, but *not both*. XOR is to be distinguished from a similar type of gate, the conventional OR gate, which has truth table and symbol shown in Figure 2.5:

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

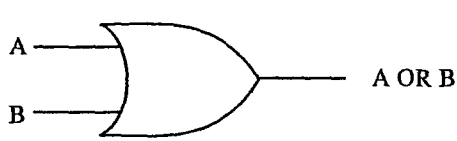


Fig. 2.5 The OR Gate

All of these gates are examples of "switching functions", which take as input some binary-valued variables and compute some binary function. Claude

Shannon was the first to apply the rules of Boolean algebra to switching networks in his MIT Master's thesis in 1937. Such switching functions can be implemented electronically with basic circuits called, appropriately enough, "gates". The presence of an electronic signal on a wire is a "1" (or "true"), the absence a "0" (or "false"). Let us continue going down in level and look in more detail at these basic gates.

The simplest operation of all is an "identity" or "do-nothing" operation. This is just a wire coming into a box and then out again, with the same signal on it. This just represents a wire (Fig. 2.6):

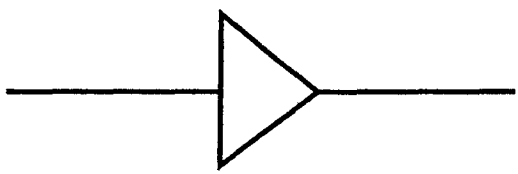


Fig. 2.6 The Identity

In a real computer, this element would be considered a "delay": as we will see in Chapter Seven, electric current actually takes time to move along wires, and this finite travel time — or delay — is something which must be taken into consideration when designing machines; with computers, even elements that do nothing on paper can do something when we build them! But let us skip this operation and look at the next simplest, namely, a box which "negates" the incoming signal. If the input is a 1, then the output will be 0, and vice versa. This is the NOT operation, with the obvious truth table (Fig. 2.7):

A	NOT A
0	1
1	0

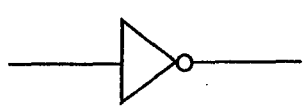


Fig. 2.7 The NOT Gate

Diagrammatically, the NOT is just the delay with a circle at its tip. Now with a little thought, one can see that there is a relationship between OR and AND,

using NOT. By playing with the truth tables you should be able to convince yourself that *A* OR *B* is the same as NOT{(NOT *A*) AND (NOT *B*)}. This is just one example of an equivalence between operators; there are many more[2]. Of course, one need not express OR in terms of AND and NOT; one could express AND in terms of NOT and OR, and so on. One of the nice games you can play with logic gates is trying to find out which is the best set to use for a specific purpose, and how to express other operators in terms of this best set. A question that naturally arises when thinking of this stuff is whether it's possible to assemble a basic set with which you could, in principle, build *all possible* logic functions: that is, if you invent *any* black box whatsoever (defined by assigning an output state to each possible input state), could you actually *build* it using just the gates in the basic set? We will not consider this matter of "completeness" of a set of operators in any detail here; the actual proof is pretty tough, and way beyond the level of this course. We will content ourselves with a hand-waving proof in section 2.4, later in this chapter. Suffice it to say that the set AND, OR and NOT *is* complete; with these operators, one can build absolutely any switching function. To tempt you to go further with all this cute stuff, I will note that there exist *single* operators that are complete!

We now have pretty much all of the symbols used by engineers to depict the various gates. They're a useful tool for illustrating the links between their physical counterparts. For example, we can diagrammatically depict our relationship between AND, OR and NOT as follows (Fig. 2.8):

Fig. 2.8 The Relationship Between And, Or and Not

Note that we have adopted the common convention of writing the NOTs as circles directly on the relevant wires; we don't need the triangles.

Let's play with these awhile. How do we make an XOR gate out of them?

[2]These relationships are actually specific instances of a general and venerable old law known as de Morgan's Theorem. [Editors]

Now XOR only gives 1 if A=1 and B=0, or A=0 and B=1. The general rule for constructing novel gates like this is to write out the truth tables for A AND B, A OR B, A AND (NOT B) and so on, and see how you might turn the outputs of such gates into the inputs for another, in such a way that you get the desired result. For example, we can get a 1 from A=1 and B=0 if we feed A and B into an AND gate, with a NOT on the B line. Similarly, we use the same trick to get the second option, using an AND, but with the NOT on the A line. If we then feed the outputs of these two gates through a third — an OR — we end up with a XOR (Fig. 2.9):

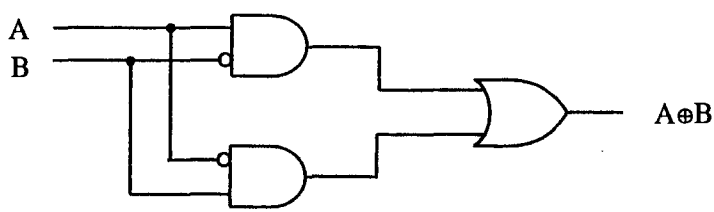


Fig. 2.9 XOR expressed in ANDs and ORs

(Notice the convention we are using: if two crossing wires are electrically connected, we place a dot on the crossing point. If the lines cross without connection, there is no dot.) Of course, you have to check that this combination works for the other two input sets of A and B; and indeed it does. If both A and B are 0, both AND gates give zero, and the OR gives zero; if both A and B are 1, again, both AND gates give zero, leading to zero as the final result. Note that this circuit is not unique. Another way of achieving an XOR switch is as follows (Fig. 2.10):

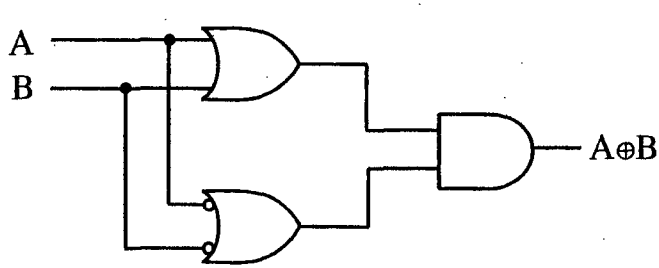


Fig. 2.10 An Alternative XOR

Which way should we make the XOR circuit in practice? It just depends on the details of the particular circumstance — the hardware, the semiconductor technology, and so on. We might also be interested in other issues, such as which method requires the fewest elements. As you can imagine, such stuff amounts to an interesting design problem, but we are not going to dwell on it here. All we care to note is that we can make any switch we like as long as we have a big bag of ANDs, ORs and NOTs. We have already seen how to make a single-bit adder — the carry bit comes from an AND gate, and the sum bit from an XOR gate, which we now know how to build from our basic gates. Let us look at another example: a multiple AND, with four inputs *A,B,C,D*. This has four inputs but still just one output, and by extension from the two-input case, we declare that this gate only "goes off" — that is, gives an output of one — when all four inputs are 1. Sometimes people like to write this problem symbolically thus:

$$A \wedge B \wedge C \wedge D$$

where the symbol $\wedge$ means "AND" in propositional logic (as we mentioned earlier). Of course, when logicians write something like this they have no particular circuit in mind which can perform the operation. We, however, can design such a circuit, built up from our primitive black box gates: to be precise, three AND gates as in Figure 2.11:

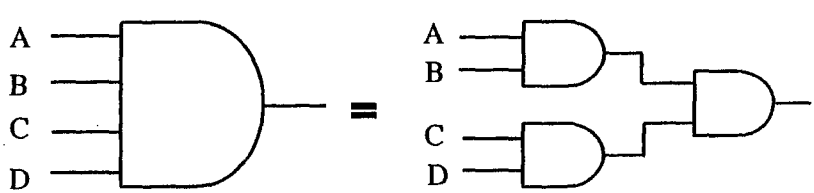


Fig. 2.11 A Multiple AND Gate

In a similar way, one can build up a multiple AND of any size.

Now the time has come to hit nearly rock-bottom in our hierarchy by looking at the actual electronic components one would use to construct logic gates. We *will* actually hit rock bottom, by which I mean discussing the physics

of semiconductors and the motion of actual electrons through machines, later in the course (in Chapter Seven). For now, I will give some quick pointers to gate construction that should be intelligible to those of you with some grasp of electronics.

Central to the construction of all gates is the *transistor*. This is arguably the most important of all electronic components, and played a critical role in the development and growth of the industry. Few electronic devices contain no transistors, and an understanding of the basic properties of these elements is essential for understanding computers, in which they are used as switches. Let us see how a transistor can be used to construct a NOT gate. Consider the following circuit (Fig. 2.12):

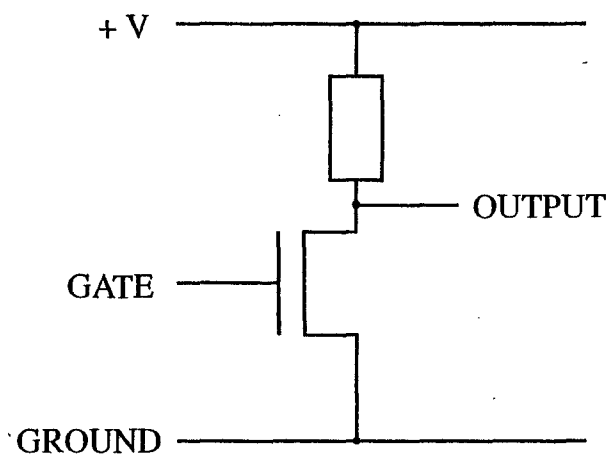


Fig. 2.12 The Transistor Inverter, or NOT Gate

A transistor is a three-connection device: one input is connected to the gate signal, one to ground, and the other to a positive voltage via a resistor. The central property of the transistor is that if the gate has a distinctly positive voltage the component conducts, but if the gate is zero or distinctly negative, it does not. Now look at the behavior of the output voltage as we input a voltage to the gate. If we input a positive voltage, which by convention we label a 1, the transistor conducts: a current flows through it, and the output voltage becomes zero, or binary 0. On the other hand, if the gate was a little bit negative, or zero, no current flows, and the output is the same as $+V$, or 1. Thus, the output is the

opposite of the input, and we have a NOT gate[3].

What about an AND gate? Due to the nature of the transistor, it actually turns out to be more convenient to use a NAND gate as our starting point for this. Such a gate is easier to make in a MOS environment than an AND gate, and if we can make the former, we can obtain the latter from it by using one of de Morgan's rules: that AND ≡ NOT {NAND}. So consider the following simple circuit (Fig. 2.13):

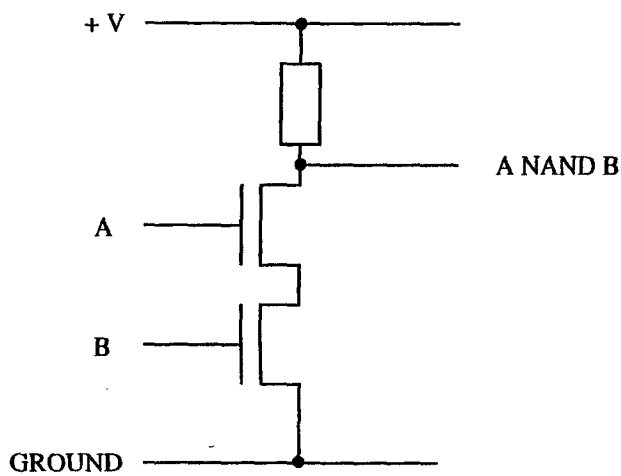


Fig. 2.13 A Transistor NAND Gate

In order for the output voltage to be zero here, we need to have current flow through both *A* and *B*, which we can clearly only achieve if both *A* and *B* are positive. Hence, this circuit is indeed a "NOT AND" or NAND gate. To get an AND gate, we simply take the NAND output from Figure 2.13 and feed it in as input to the NOT gate illustrated in Figure 2.12. The resultant output is our AND.

What about an OR gate? Well, we have seen how to make an OR from ANDs and NOTs, and we could proceed this way if we wished, combining the transistor circuits above; however, an easier option (both conceptually and from

[3]As a technical aside, we have assumed that our circuits are fabricated using MOS (Metal Oxide Semiconductor) technology. Resistors are hard to implement in this type of silicon technology, and in practice the resistor would actually be replaced by another type of MOS transistor (see Chapter Seven). [RPF]

the viewpoint of manufacture) results from consideration of the following, parallel circuit (Fig. 2.14):

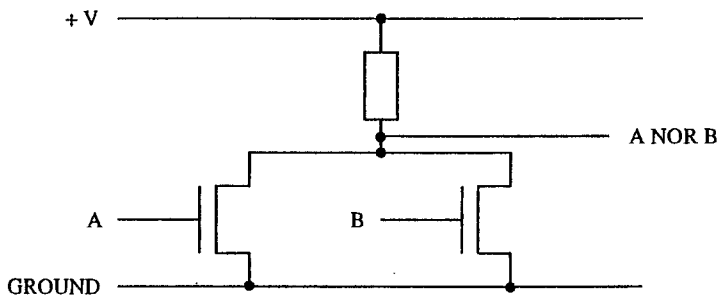


Fig. 2.14 A Transistor NOR Gate

If either A or B is positive, or both positive, current flows and the output is zero. If both A and B are zero, it is $+V$, or 1. So again, we have the opposite of what we want: this is a "NOT OR" or NOR gate. All we do now is send our output through a NOT, and all is well.

Hopefully this has convinced you that we can make electrical circuits which function as do the basic gates. We are now going to go back up a level and look at some more elaborate devices that we can build from our basic building blocks.

2.2: The Binary Decoder

The first device that we shall look at is called a "binary decoder". It works like this. Suppose we have four wires, A, B, C, D coming into the device. These wires could bring in any input. However, if the signals on the wires are a specific set, say 1011, we want to know this: we want to receive a signal from the decoder telling us that 1011 has occurred. It is as if we have some demon scanning the four bits coming into the decoder and, if they turn out to be 1011, he sends us a signal! This is easy to arrange using a modified AND gate (and much cheaper than hiring a demon). The following device (Fig. 2.15) clearly only gives us an output of 1 when A, C, D are 1 and B is 0:

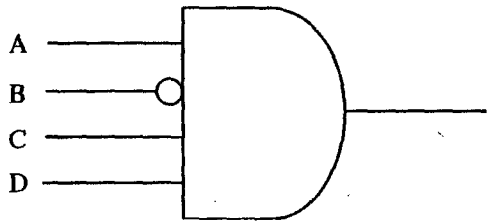


Fig. 2.15 A Simple Decoder

This is a very special type of decoder. Suppose we want a more general one, with lots of demons each looking for their own particular number amidst the many possible input combinations. Such a decoder is easy to make by connecting individual decoders in parallel. A full decoder is one that will decode every possible input number. Let us see how this works with a three-to-eight binary decoder. Here, we have three input bits on wires *A*, *B*, *C* giving $2^3 = 8$ combinations. We therefore have eight output wires, and we want to build a gate that will assign each input combination to a distinct output line, giving a 1 on just one of these eight wires, so that we can tell at a glance what input was fed into the decoder. We can organize the decoder as follows (Fig. 2.16):

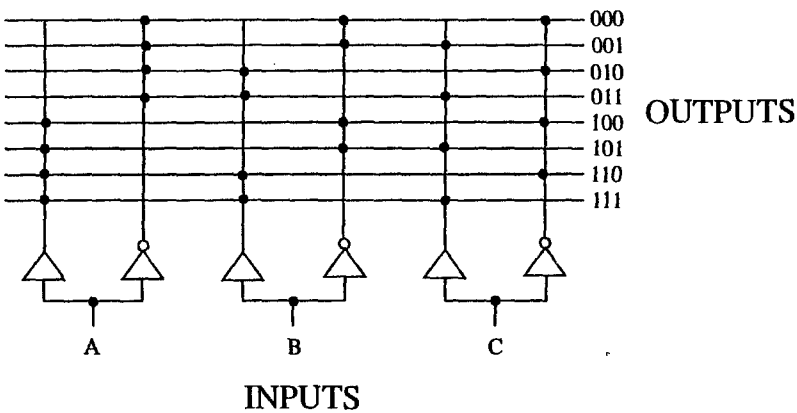
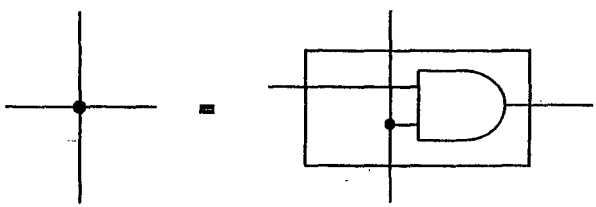


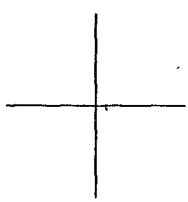
Fig. 2.16 A Binary Decoder

We have introduced the pictorial convention that three dots on a horizontal line implies a triple AND gate (see the discussion surrounding Figure 2.11). Notice that each input wire branches into an A and NOT A signal and so on. As we have arranged things, only the bottom four wires can go off if A is one, and the top four if A is zero. The dots on the wires for B and C (and NOT B and NOT C) similarly show us immediately which of the eight output wires can go off: we have labeled each output line with its corresponding input state. Thus, we have explicitly constructed a three-to-eight binary decoder.

Now, there is a profound use to which we can put the device in Fig. 2.16; one which reveals the decoder to be an absolutely essential part of the machine designer's arsenal. Suppose we feed 1's from the left into all of the horizontal input wires of the decoder. Now interpret each dot on an intersection as a two-way AND:



and a simple crossing as no connection:



In order for the 1 input from the left to get past the first dot, the correct signal A=1 or NOT A=1, depending on the wire, must be present. Similarly for B and C. So we still have a binary decoder; nothing has changed in this regard. However, we have also invented something else, which a little thought should show you is indispensable in a functioning computer: *this device can serve as a multiple switch to connect you to a selected input wire.* The original input lines of the decoder, A, B, C now serve as "address" lines to select which output wire gives a signal (which may be 1 or 0). This is very close to something called a "multiplexer": multiplexing is the technique of selecting and

transmitting signals from multiple input lines to a single output line. In our example, we can make our device into a true multiplexer by adding an eight-way OR gate to the eight output lines (Fig. 2.17):

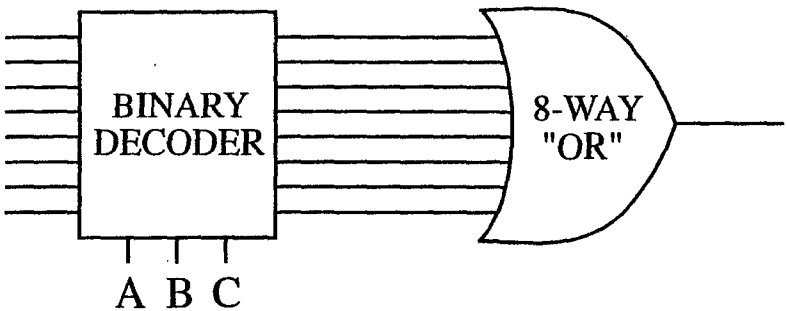


Fig. 2.17 The Multiplexer

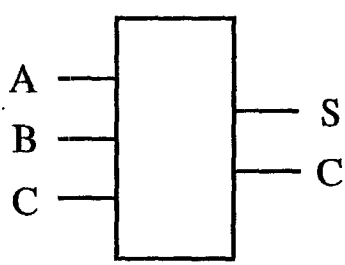
This rather neat composite device clearly selects which of the eight input lines on the left is transmitted, using the 3-bit address code. Multiplexers are used in computers to read and write into memory, and for a whole host of other tasks.

Let me give you some problems to play with.

Problem 2.1: Design an 8 to 3 encoder. In other words, solve the reverse problem to that considered earlier: 8 input wires, only one of which has a signal on at any given time; 3 output wires which "encode" which wire had the signal on.

Problem 2.2: Design a simple adder using AND, OR and NOT gates.

Problem 2.3: Design a 1-bit full adder:



Problem 2.4: Make an r-bit full adder using r 1-bit full adders. How many

simple adders would be needed?

2.3: More on Gates: Reversible Gates

We stated earlier, without proof, that the combinational circuits for AND and NOT are sufficient building blocks to realize any switching function.

AND NOT

Actually, there are two other elements that we added without noticing. These are the "fanout" and "exchange" operations (Fig. 2.18):

FANOUT EXCHANGE

Fig. 2.18 FANOUT and EXCHANGE

In the case of wires and pulses of 1's and 0's the presence of these "gates" is an obvious assumption; FANOUT just splits a wire into two or more and EXCHANGE just swaps over a pair of connections. If, on the other hand, the information were carried by pebbles, then a fanout into two means that one pebble has become two, so it is quite a special operation. Similarly, if the information were stored in separate boxes in distinct locations, then exchange is also a definite operation. We are emphasizing the logical necessity of including these two "obvious" operations since we will be needing them in our discussion of reversibility. The other thing we will assume we have is an endless supply of 0's and 1's; a store somewhere into which we can stick wires and get signals for as long as we want. This can have unforeseen uses. For example, we have already noted that one can in fact replace the AND and NOT set of gates

by a single NAND gate (Fig. 2.19):

A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

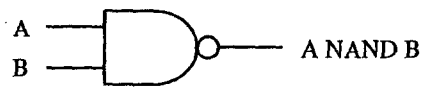
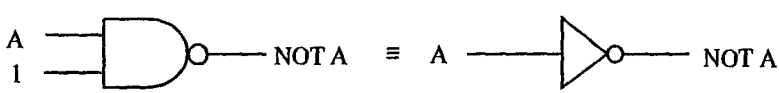


Fig. 2.19 The NAND Gate

It is easy to see that this single gate is as good as having both AND and NOT. To get a NOT operation from a NAND, all we do is turn to our storehouse of 1s and connect one of the NAND inputs up to it. Now, whatever the other input *A*, the output will be NOT *A*:



Now that we have a NOT and an AND, we can clearly construct a NAND, and we have demonstrated their equivalence as a set of operators.

We want to discuss a rather different problem, which will enable us to look at some rather more exotic logic gates. Both the AND and the NAND operation — and the OR and XOR — are *irreversible* operations. By this I mean simply that from the output of the gate you cannot reconstruct the input: information is irreversibly lost. If the output of an AND gate with four inputs is zero, it could have resulted from any one of fifteen input sets, and you have no idea which (although you obviously know about the inputs if the output is one!). We would like to introduce the concept of a *reversible* operation as one with enough information in the output to enable you to deduce the input. We will need such a concept when we come to study the thermodynamics of

computation later. It will make it possible for us to make calculations about the free energy — or, if you like, the physical efficiency — of computation.

The problem of reversible computers has been studied independently by Bennett and Fredkin. Our basic constructs will be three gates: NOT (N), CONTROLLED NOT (CN) and a CONTROLLED CONTROLLED NOT (CCN). Let us explain what these are. A NOT is just a NOT as before, a one element object. A CONTROLLED NOT is a two-wire input gadget that, unlike the AND and NAND gates, has two outputs as well. It works in the following way. We have two wires, on one of which we write a circle, representing a control, and on the other a cross (Fig. 2.20):

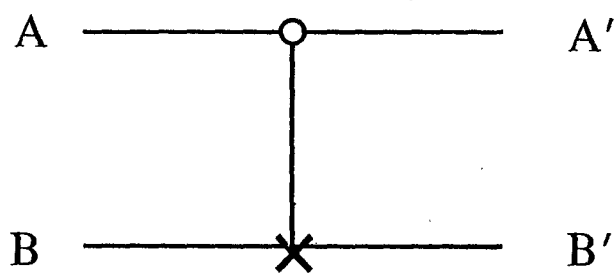


Fig. 2.20 The CN Gate

The "X" denotes a NOT operation: however, this NOT is not a conventional one; it is *controlled by the input to the O-wire*. Specifically, if the input to the O-wire is 1, then the input to the X-wire is inverted; if the O-input is zero, then the NOT gate does not work, and the signal on the X-wire goes through unchanged. In other words, the input to the O-line activates the NOT gate on the lower line. The O-output, however, is always the same as the O-input — the upper line is the identity. The truth table for this gate is simple enough:

A	B	A′	B′
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

Table 2.2 Truth Table for the CN Gate

Note that we can interpret B' as the output of an XOR gate with inputs A and B: $B' = \text{XOR}(A,B)$.

One of the most important properties of this CN gate is that it is reversible — from what comes out we can deduce what went in. Notice that we can actually reverse the operation of the gate by merely repeating it:

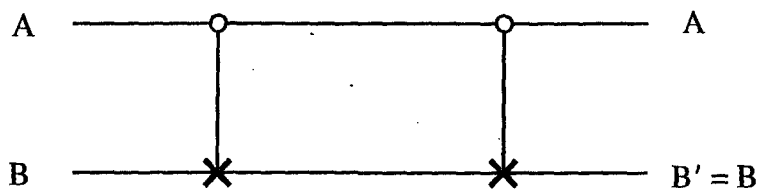


Fig. 2.21 The Identity Utilizing CN Gates

We can use a CN gate to build a fanout circuit. If we set B=0, then we have B'=A and A'=A. As an exercise, you might like to show how CN gates can be connected up to make an exchange operator (Hint: it takes several).

Sadly, we cannot do everything with just N and CN gates. Something more is needed, for example, a CCN, or CONTROLLED CONTROLLED NOT gate (Fig. 2.22):

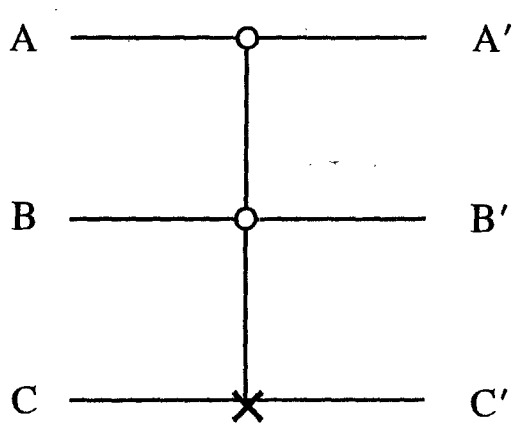
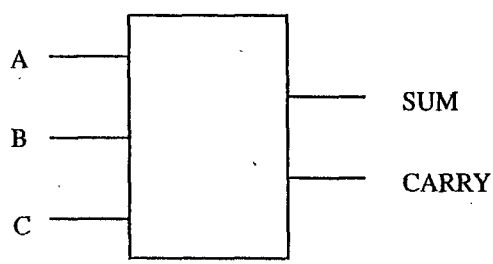


Fig. 2.22 The CCN Gate

In this gate, we have two control lines A and B, each marked by an **O**, and as

with the CN gate, the signals on this line are unchanged on passage through the gate: $A'=A$, $B'=B$. The remaining line, once again, has a NOT on it, but this is only activated if both $A=1$ and $B=1$: then, C'=NOT C. Notice that this single gate is very powerful. If we keep both A and B equal to one, then the CCN gate is just an N, a NOT. If we keep just $A=1$, then the gate is just a CN gate with B and C as inputs. So if we have a CCN gate and a source of 1s and 0s, we can junk both the CN and N gates. But things are even better: with this CCN gate we can do everything! We have already seen how a CN gate can be used to produce an XOR output. We know that throwing in a NOT or two enables us to get an AND gate. So clearly, we can generate any gate we like with just a CCN gate: by itself, it forms a complete operator set. As an example, the AND gate can be made by holding $C=0$, and taking the inputs to be A and B. The output, A AND B is then C', which is clearly 1 only when the NOT gate is activated to invert $C=0$, which in turn is only the case – by the property of the CCN gate – when $A=B=1$.

The next thing we must do is show that we can do something useful with only these reversible operations. This is not difficult, as we have just shown that we can do anything with them that we can do with a complete operator set! However, we would like whatever we build to be itself reversible. Consider the problem of making a full adder:



We need to add A, B and C and obtain the sum and carry. Now as it stands, this operation is not reversible – one cannot, in general, reconstruct the three inputs from the sum and carry. We have decided that we want to have a reversible adder, so we need more information at the output than at present. As you can see with a little thought, reversible gates have the general property that "lines in = lines out" – this is the only way that all possible inputs can be separately "counted" at the output – and so we need another line coming out of our adder. In fact, it turns out that we need two extra lines coming out of the gate, and one

extra going in, which you set to 0, say. Using N, CN and CCN (or just the latter) we can get AND, OR and XOR operators, and we can clearly use these to build an adder: the trick of making it reversible lies in using the redundancy of the extra outputs to arrange things such that the two extra output lines, on top of the sum and carry ones, are just the inputs *A* and *B*. It is a worthwhile exercise to work this out in detail.

Fredkin added an extra constraint on the outputs and inputs of the gates he considered. He demanded that not only must a gate be reversible, but the number of 1s and 0s should never change. There is no good reason for this, but he did it anyway. He introduced a gate performing a controlled exchange operation (Fig. 2.23):

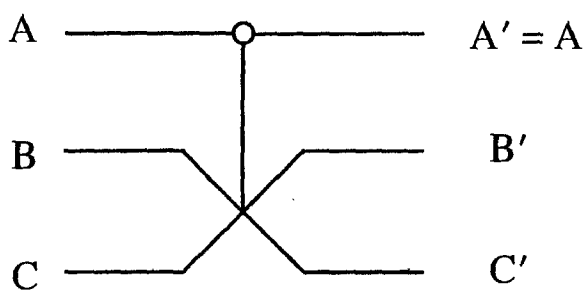


Fig. 2.23 The Fredkin Gate: A Controlled Exchange

In his honor, we will call this a Fredkin gate. You should be used to the notion of control lines by now; they just activate a more conventional operation on other inputs. In this case, the operation is exchange. Fredkin's gate works like this: if A=0, B and C are not exchanged; B'=B, and C'=C. However, if A=1 they are, and B'=C, C'=B. You can check that the numbers of 1s and 0s is conserved. As a further, and more demanding, exercise, you can try to show how this Fredkin gate can be used (perhaps surprisingly) to perform all logical operations instead of using the CCN gate.

2.4: Complete Sets of Operators

I have introduced you to the notion of reversible gates so that you can see that there is more to the subject of computer logic than just the standard AND, NOT and OR gates. We will return to these gates in chapter five. I want for the moment to leave the topic of reversible computation and return to the issue of

complete sets of operators. Now I've been very happy to say that with a so-called "complete set" of operators, you can do anything, that is, build any logical function. I will take as my complete set the operations AND, NOT, FANOUT and EXCHANGE. The problem I would like to address is how we can know that this set *is* complete. Suppose we have a bunch of n input wires, which we'll label $X_1, X_2, X_3, \ldots X_n$. For each pattern of inputs $\{X\}$, we will have some specific output pattern on a set of wires $Y_1, Y_2, \ldots, Y_m$, where m is not necessarily equal to n. The output on Y_i is a logical function of the X_i. Formally, we write

$$Y_i = F_i(\{X\}),\ i=1,\ldots,m \tag{2.1}$$

What we want to demonstrate is that for *any* set of functions F_i we can build a circuit to perform that function on the inputs using just our basic set of gates. Let us look at a particular example, namely, the sum of the input wires. We can see how in principle we can do this as follows. In our binary decoder, we had n input wires and 2^n output wires, and we arranged for a particular output wire to fire by using a bunch of AND gates. This time we want to arrange for that output to give rise to a specific signal on another set of output wires. In particular, we can then arrange for the signals on the output wires to be the binary number corresponding to the value of the sum of the particular input pattern.

Let us suppose that for a particular input set of Xs we have selected one wire. One wire only is "hot", and all the others "cold". When this wire is hot we want to generate a specific set of output signals. This is the opposite problem to the decoder. What we need now is an *en*coder. As you should have figured out from one of the problems you were set, this can be constructed from a bunch of OR gates. So you see, we have separated the problem into two parts. The first part that we looked at before was how to arrange for different wires to go off according to the input. The answer was our decoder. Our encoder must have a lot of input wires but only one goes off at a time. We want to be able to write the number of which wire went off in the binary system. A three-bit encoder may be built from OR gates as follows (Fig. 2.24):

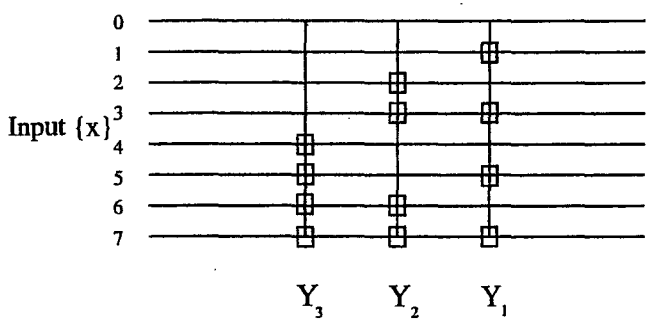
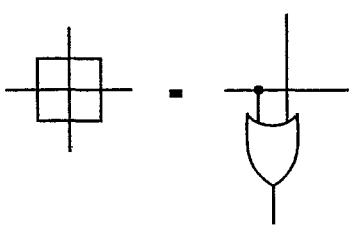


Fig. 2.24 The Three-bit Encoder

where we have used the following notation for the OR gates:



Thus, if we are not bothered about the proliferation of 2^n wires, then we can construct any logical function we wish. In general, we have an AND plane and an OR plane and a large number of wires connecting these two regions (Fig. 2.25):

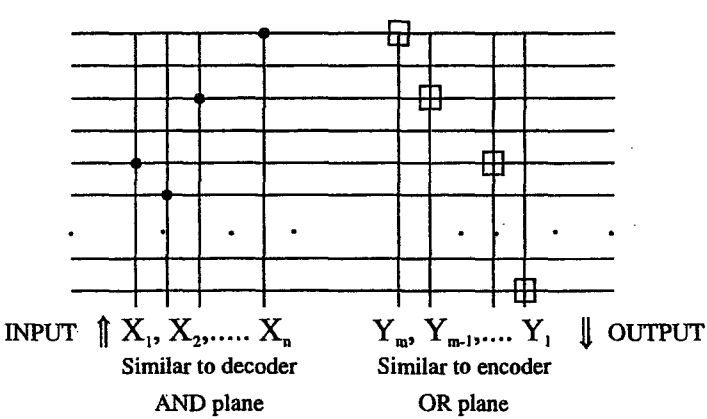


Fig. 2.25 Construction of a General Logical Function

where we have used the same notation for AND gates as in Figure 2.16. If you succeeded in solving any of the problems 2.2-2.4, which required you to construct a number of different adders, then you will have already seen simple examples of this principle at work.

Some of the logical functions we could construct in this way are so simple that using Boolean algebra we can simplify the design and use fewer gates. In the past people used to invest much effort in finding the simplest or smallest system for particular logical functions. However, the approach described here is so simple and general that it does not need an expert in logic to design it! Moreover, it is also a standard type of layout that can easily be laid out in silicon. Thus this type of design is usually used for Programmable Logic Arrays, or PLAs. These are often used to produce custom-made chips for which relatively few copies are needed. The customer only has to specify which ANDs and which ORs are connected to get the desired functionality. For mass-produced chips it is worthwhile investing the extra effort to do the layout more efficiently.

2.5: Flip-Flops and Computer Memory

Now I want to come onto something different, which is not only central to the functioning of useful computers, but should also be fun to look at. We start with a simple question: can we store numbers? That is, can we build a computer's memory from the gates and tidbits we've assembled so far? A useful memory store will allow us to alter what we store; to erase and rewrite the contents of a memory location. Let's look at the simplest possible memory store, one which holds just one bit (a 1 or 0), and see how we might tinker with it. As a reasonable first guess at building a workable memory device, consider the following black box arrangement:

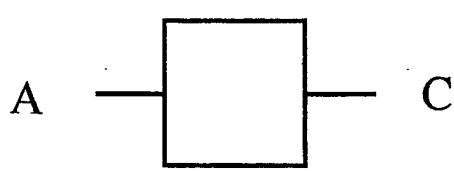


Fig. 2.26 A Black Box Memory Store

We take the signal on line C to represent what is in our memory. The input A is a control line, with the following properties: as long as A is 0, i.e. we are

feeding nothing into our box, *C* remains the same. However, if we switch *A* to 1, then we change *C*: it goes from 0 to 1 or vice versa. We can write a kind of "truth table" for this:

A	Present C	Next C
0	0	0
0	1	1
1	0	1
1	1	0

Table 2.3 "Truth Table" for the Memory Device

It is easily noticed from this table that "Next *C*" is the XOR of *A* and the present *C*. So it might seem that if we get clever and replace our black box by an XOR gate with *feedback* from *C*, we may have a possible memory unit (Fig. 2.27):

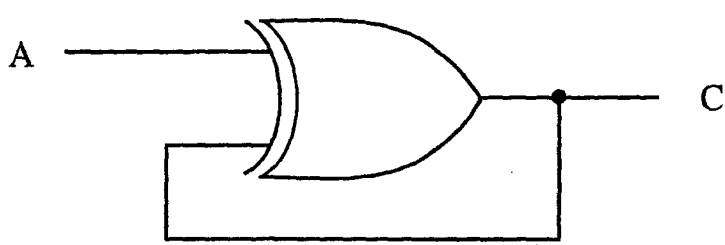


Fig. 2.27 A Plausible (but Non-Workable) Memory Device

Will this work? Well, it all depends on the timing! We have to interrupt our abstract algebra and take note of the limitations on devices imposed by the physical world. Let's suppose that *A* is 0 and *C* is 1. Then everything is stable: so far, so good. Now change the input *A* to 1. What happens? *C* changes to 0, by definition, which is what we want. But this value is then fed back into the XOR gate, where with *A*=1 it gives an output of 1 — so *C* changes back to 1. This then goes back into the XOR, where with *A*=1 it now gives an output *C* = 0. We then start all over again. Our gate oscillates horribly, and is of no use whatsoever.

However, if you think about it, you can see that we can salvage the gate somewhat by building in delays to the various stages of its operation; for example, we can make the XOR take a certain amount of time to produce its output. However, we cannot stop it oscillating. Even if we were prepared to build a short-term memory bank, the physical volatility of electronic components would introduce extra instabilities leading to unforeseen oscillations that make this gate pretty useless for practical purposes. Out of interest, note what happens if we build the circuit with an OR rather than an XOR?

Clearly, the crucial troublesome feature in this device is the element of *feedback*. Can we not just dispense with it? The answer is yes, but this would be at quite a cost. For reasons of economy and space, one thing we would like our computer to be able to do is repeated calculations with the same pieces of operating equipment. For example, if we used a certain adder to do part of a calculation, we would like to use the same adder to do another subsequent part of the calculation, which might involve using its earlier output. We would not want to keep piling more and more adders into our architecture for each new stage of a program: yet without feedback, we would have no choice. So we will want to crack this problem!

What we want is a circuit that can hold a value, 0 or 1, until we decide to reset it with a signal on a wire. The circuit that turns out to do the job for us is called a *flip-flop*, schematically drawn as shown in Figure 2.28:

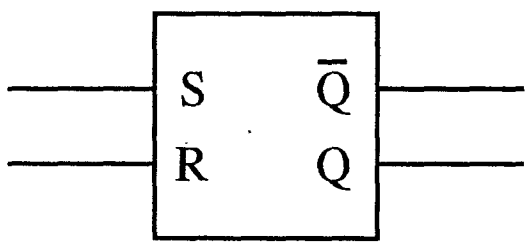


Fig. 2.28 A Flip-Flop

The flip-flop has two input wires — the "set" (S) and "reset" (R) wires — and two outputs, which we call Q and $\overline{Q}$. This latter labeling reflects the fact that one is always the logical complement — the inverse — of the other. They are sometimes misleadingly referred to as the 0 and 1 lines; misleading, because each can take either value, as long as the other is its inverse.

We can actually use NOR gates (for example) to build a circuit that functions as a flip-flop:

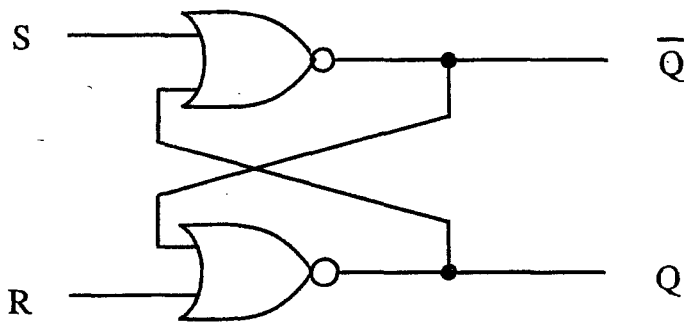


Fig. 2.29 Gate Structure of a Simple Flip-Flop

Note that the device incorporates feedback! Despite this, it is possible to arrange things so that the flip-flop does not oscillate, as happened with our naive XOR store. It is important to ensure that S and R are never simultaneously 1, something which we can arrange the architecture of our machine to ensure. The device then has just two output states, both of which are stable: Q=1 (hence $\overline{Q}$=0), and Q=0 (hence $\overline{Q}$=1). How does this help us with memory storage?

The way the thing works is best seen by examining its truth table:

Present Q	S	R	Next Q
0	0	0	0
0	0	1	0
0	1	0	1
1	0	0	1
1	1	0	1
1	0	1	0

Table 2.4 Truth Table for a Simple Flip-Flop

The signal on the Q-line is interpreted as the contents of the flip-flop, and this stays the same whenever S and R are both 0. Let us first consider the case when the reset line, R, carries no signal. Then we find that, if the contents Q of the flip-flop are initially 0, setting S=1 changes this to 1; otherwise, the S-line has no effect. In other words, the S-line sets the contents of the flip-flop to 1, but subsequently manipulating S does nothing; if the flip-flop is already at 1, it will stay that way even if we switch S. Now look at the effect of the reset line, R. If the flip-flop is at 0, it will stay that way if we set R=1; however, if it is at 1, then setting R=1 resets it to 0. So the R line clears the contents of the flip-flop. This is pretty confusing upon first exposure, and I would recommend that you study this set-up until you understand it fully. We will now examine how we can use this flip-flop to solve our timing problems.

2.6: Timing and Shift Registers

We have now designed a device — a flip-flop — which incorporates feedback, and doesn't suffer from the oscillations of naive structures. However, there is a subtle and interesting problem concerning this gadget. As I pointed out in the last lecture, the signals traveling between the various components take differing times to arrive and be processed, and sometimes the physical volatility of the components you use to build your equipment will give you freaky variations in these times in addition, which you wouldn't allow for if you assumed technology to be ideal. This means that often you will find signals arriving at gates later than they are supposed to, and doing the wrong job! We have to be aware of the possible effects of this. For the flip-flop, for example, what would happen if both the outputs turned out to be the same? We have assumed, as an idealization, that they would be complementary, but things can go wrong! You can see that if this happens, then the whole business of the set and reset would go out the window.

The way around this is to introduce into the system a *clock*, and have this send an "enable" signal to the flip-flop at regular intervals. We then contrive to have the flip-flop do nothing until it receives a clock signal. These signals are spaced far enough apart to allow everything to settle down before operations are executed. We implement this idea by placing an AND gate on each input wire, and also feeding into each gate the clock signal:

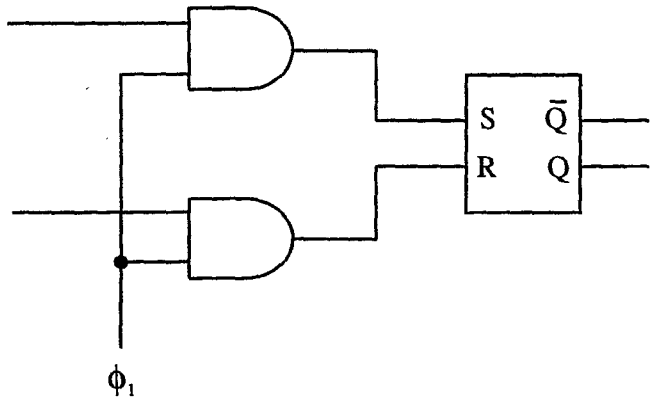


Fig. 2.30 A Clocked RS Flip-Flop

This is sometimes called a *transparent latch* since all the time the clock is asserted any change of input is transmitted through the device.

We represent the signal ϕ_1 from the clock as a series of pulses (Fig. 2.31):

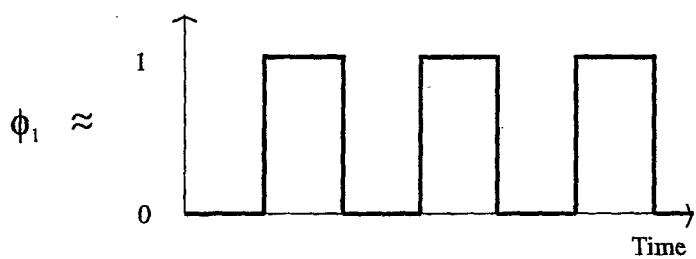


Fig. 2.31 The Clock Pulse

Clearly, whatever the input to the AND gates, it will only get through to S and R when the signal from the clock ϕ_1 is 1. So as long as we get the timing of the clock right, and we can be sure it does not switch the gate on until there is no chance of the inputs playing up, we have cleared up the problem. But of course, we have created another one! We have merely deferred the difficulty: the output of this gate will shoot off to another, or more than one, and we will have the same problems with travel times, and so on, all over again. It will not help to connect everything up to our clock ϕ_1 – far from it; one part of the system may be turning on just as another is changing its outputs. We still have delays. So

we might think, to get around this, to try to build a machine with great precision, calculating delay times and making sure that everything comes out right. It can be done, and the resultant system is fast and efficient, but it's also very expensive and difficult to design. The best way to get around the problem is to introduce another clock, ϕ_2, and not allow the next gate in the chain to accept input from the first until *this* clock is asserted. This arrangement is the basis for a special type of flip-flop called a *Master-Slave Flip-Flop* (Fig. 2.32):

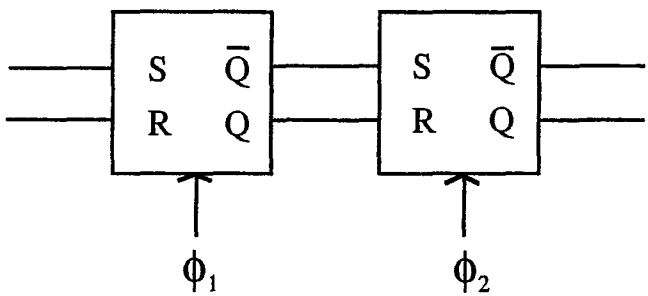
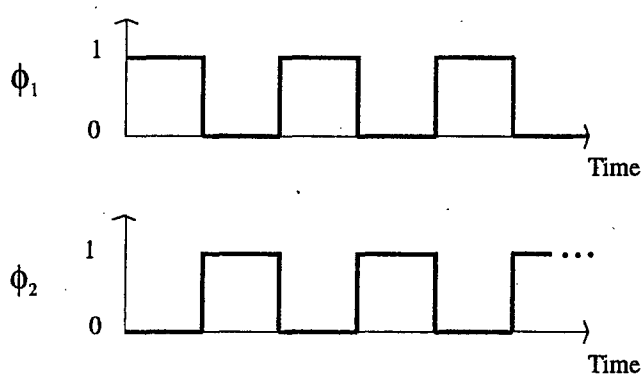


Fig. 2.32 The Master-Slave Flip-Flop

The signals from the two clocks should be complementary:



The easiest way to ensure this is to get ϕ_2 from NOT ϕ_1. We also note that we need our logical operations to be fast in comparison with the clock pulse-length. Don't forget that in all this we are using the abstractions that (1) all levels are 0 or 1 (not true: they are always changing with time. They are never exactly one or zero, but they are near saturation), and (2) there is a definite, uniform delay time between pulses: we can say that *this* happens, then *that* happens, and so on. This is a good idealization, and we can get closer to it by introducing more

clock signals if we like.

It is possible to design a variety of flip-flop devices, and learning how and why they work is a valuable exercise. One such device is the *D-type* flip-flop, which has the structure shown in Figure 2.33:

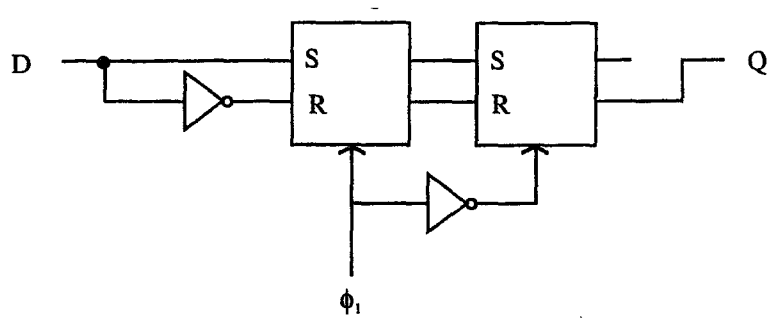


Fig. 2.33 A "D-type" Flip-Flop

It is unclear why this device is labeled a "D-type" flip-flop. One plausible suggestion is that the "D" derives from the "delaying" property of the device: basically, the output is the same as the input, but only becomes so after a clock pulse.

Let us introduce the following shorthand notation for the D-type flip-flop:

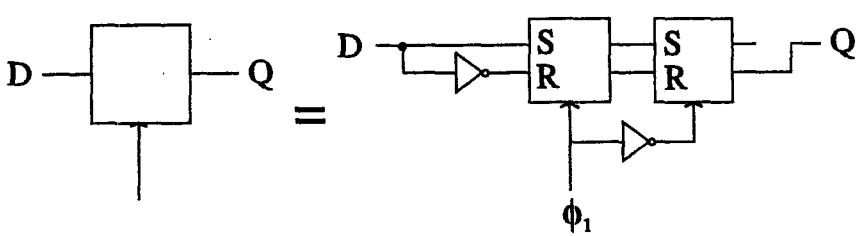


Fig. 2.34 Simplified Notation for the D-type Flip-Flop

A very useful device that may be built from flip-flops, and one which we shall take the trouble to examine, is a *shift register*. This is a device which can, amongst other things, store arbitrary binary numbers — sequences of bits — rather than just one bit. It comprises a number of flip-flops, connected sequentially, into which we feed our binary number one bit at a time. We will just use our basic S-R's, with a delay built in. The basic structure of a shift register is as follows:

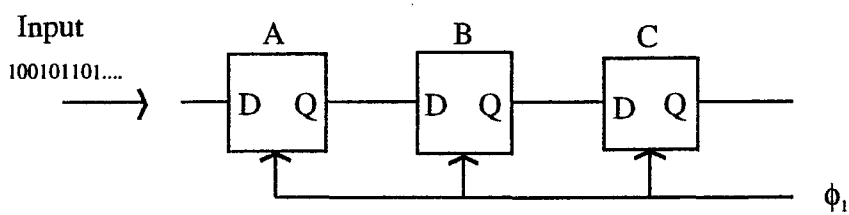


Fig. 2.35 A Shift Register

Each unit of this register is essentially a stable delay device of the kind I described earlier. Note that each flip-flop in the array is clocked by the same clock ϕ_1.

The reader should have little difficulty in seeing how the device works. We start with the assumption (not necessary, but a simplifying one) that all of the flip-flops are set to zero. Suppose we wish to feed the number 101 into the device. What will happen? We feed the number in lowest digit first, so we stick a 1 into the left hand S-R, which I've labeled **A**, and wait until the clock pulse arrives to get things moving. After the next clock pulse, the output of **A** becomes 1. We now feed the next bit, 0, into **A**. Nothing happens until the next clock pulse. After this arrives, the next S-R in the sequence, **B**, gets a 1 on its output (the original 0 has been reset). However, the output of **A** switches to 0, reflecting its earlier input. Meanwhile, we have fed into **A** the next bit of our number which is 1. Again, we wait for the next clock pulse. Now we find that **A** has an output of 1, **B** of 0 and **C** of 1 — in other words, reading from left to right, the very number we fed into it! Generalizing to larger binary strings is straightforward (note that each flip-flop can hold just the one bit, so a register containing n flip-flops can only store up to 2^n). So you can see that a register like this takes a sequential piece of information and turns it into parallel

information; shifting it along bit by bit and storing it for our later examination. It is not necessary to go any further with them; the reader should be able to see that registers clearly have uses as memory stores for numbers and as shifting devices for binary arithmetical operations, and that they can therefore be built into adders and other circuits.

THREE

THE THEORY OF COMPUTATION

Thus far, we have discussed the limitations on computing imposed by the structure of logic gates. We now come on to address an issue that is far more fundamental: is there a limit to what we can, in principle, compute? It is easy to imagine that if we built a big enough computer, then it could compute anything we wanted it to. Is this true? Or are there some questions that it could never answer for us, however beautifully made it might be?

Ironically, it turns out that all this was discussed long before computers were built! Computer science, in a sense, existed before the computer. It was a very big topic for logicians and mathematicians in the thirties. There was a lot of ferment at court in those days about this very question — what can be computed *in principle*? Mathematicians were in the habit of playing a particular game, involving setting up mathematical systems of axioms and elements — like those of Euclid, for example — and seeing what they could deduce from them. An assumption that was routinely made was that any statement you might care to make in one of these mathematical languages could be proved or disproved, in principle. Mathematicians were used to struggling vainly with the proof of apparently quite simple statements — like Fermat's Last Theorem, or Goldbach's Conjecture — but always figured that, sooner or later, some smart guy would come along and figure them out[1]. However, the question eventually arose as to whether such statements, or others, might be inherently unprovable. The question became acute after the logician Kurt Gödel proved the astonishing result — in "Gödel's Theorem" — that arithmetic was incomplete.

3.1: Effective Procedures and Computability

The struggle to define what could and could not be proved, and what numbers could be calculated, led to the concept of what I will call an *effective procedure*. If you like, an effective procedure is a set of rules telling you, moment by moment, what to do to achieve a particular end; it is an algorithm. Let me

---

[1]In the case of Fermat's Last Theorem, some smart guy *did* come along and solve it! Fermat's Theorem, which states that the equation

$$x^n + y^n = z^n \ (n \text{ an integer}, n \geq 3)$$

has no solutions for which x, y and z are integers, has always been one of the outstanding problems of number theory. The proof, long believed impossible to derive (mathematical societies even offered rewards for it!), was finally arrived at in 1994 by the mathematicians Andrew Wiles and Richard Taylor, after many, many years' work (and after a false alarm in 1993). [Editors]

explain roughly what this means, by example. Suppose you wanted to calculate the exponential function of a number x, e^x. There is a very direct way of doing this: you use the Taylor series

$$e^x = 1 + x + (x^2/2!) + (x^3/3!) + \ldots. \tag{3.1}$$

Plug in the value of x, add up the individual terms, and you have e^x. As the number of terms you include in your sum increases, the value you have for e^x gets closer to the actual value. So if the task you have set yourself is to compute e^x to a certain degree of accuracy, I can tell you how to do it — it might be slow and laborious, and there might be techniques which are more efficient, but we don't care: it works. It is an example of what I call an effective procedure.

Another example of an effective procedure in mathematics is the process of differentiation. It doesn't matter what function of a variable x I choose to give you, if you have learned the basic rules of differential calculus you can differentiate it. Things might get a little messy, but they are straightforward. This is in contrast to the inverse operation, integration. As you all know, integration is something of an art; for any given integrand, you might have to make a lot of guesses before you can integrate it: should I change variables? Do we have the derivative of a function divided by the function itself? Is integration by parts the way to go? In that we none of us have a hotline to the correct answer, it is fair to say that we do not possess an effective procedure for integration. However, this is not to say that such a procedure does not exist: one of the most interesting discoveries in this area of the past twenty years has been that there *is* such a procedure! Specifically, any integral which can be expressed in terms of a pre-defined list of elementary functions — sines, exponentials, error functions and so forth — can be evaluated by an effective procedure. This means, among other things, that machines can do integrals. We have to thank a guy named Risch for this ("The Problem of Integration in Finite Terms", *Trans. A.M.S.* 139(1969) pp. 167-189).

There are other examples in mathematics where we lack effective procedures; factoring general algebraic expressions, for example: there are effective procedures for expressions up to the fourth degree, but not fifth and over. An interesting example of a discipline in which every school kid would give his eye-teeth for an effective procedure is geometry. Geometrical proof, like integration, strikes most of us as more art than science, requiring considerable ingenuity. It is ironic that, like integration, there is an effective procedure for

geometry! It is, in fact, Cartesian analytic geometry. We label points by coordinates, (x,y), and we determine all lengths and angles by using Pythagoras' Theorem and various other formulae. Analytic geometry reduces the geometry of Euclid to a branch of algebra, at a level where effective procedures exist.

I have already pointed out that converting questions to effective procedures is pretty much equivalent to getting them into a form whereby computers can handle them, and this is one of the reasons why the topic has attracted so much attention of late (and why, for example, the notion of effective procedures in integration has only recently been addressed and solved). Now when mathematicians first addressed these problems, their interest was more general than the practical limits of computation; they were interested in principle with what could be proved. The question spawned a variety of approaches. Alan Turing, a British mathematician, equated the concept of "computability" with the ability of a certain type of machine to perform a computation. Church defined a system of logic and propositions and called it effective calculability. Kleene defined certain so-called "general recursive propositions" and worked in terms of these. Post had yet another approach (see the problem at the end of this chapter), and there were still other ways of examining the problem. All of these workers started off with a mathematical language of sorts and attempted to define a concept of "effective calculability" within that language. Thankfully for us, it can be shown that all of these apparently disparate approaches are equivalent, which means that we will only need to look at one of them. We choose the commonest method, that of Turing.

Turing's idea was to make a machine that was kind of an analogue of a mathematician who has to follow a set of rules. The idea is that the mathematician has a long strip of paper broken up into squares, in each of which he can write and read, one at a time. He looks at a square, and what he sees puts him in some state of mind which determines what he writes in the next square. So imagine the guy's brain having lots of different possible states which are mixed up and changed by looking at the strip of paper. After thinking along these lines and abstracting a bit, Turing came up with a kind of machine which is referred to as — surprise, surprise — a Turing machine. We will see that these machines are horribly inefficient and slow — so much so that no one would ever waste their time building one except for amusement — but that, if we are patient with them, they can do wonderful things.

Now Turing invented all manner of Turing machines, but he eventually discovered one — the so-called Universal Turing Machine (UTM) — which was the best of the bunch. Anything that any specific special-purpose Turing

machine could do, the UTM could do. But further, Turing asserted that *if anything could be done by an effective procedure, it could be done by his Universal machine, and vice versa*: if the UTM could not solve a problem, there was no effective procedure for that problem. Although just a conjecture, this belief about the UTM and effective procedures is widely held, and has received much theoretical support. No one has yet been able to design a machine that can outdo the UTM in principle. I will actually give you the plans for a UTM later. First, we will take a closer look at its simpler brother — the finite state machine.

3.2: Finite State Machines

A typical Turing machine consists of two parts; a tape, which must be of potentially unlimited size, and the machine itself, which moves over the tape and manipulates its contents. It would be a mistake to think that the tape is a minor addition to a very clever machine; without the tape, the machine is really quite dumb (try solving a complex integral in your head). We will begin our examination of Turing machines and what they can do by looking at a Turing machine without its tape; this is called a *finite state machine.*

Although we are chiefly interested in finite state machines (FSMs) as component parts of Turing machines, they are of some interest in their own right. What kinds of problems can such machines do, or not do? It turns out that there are some questions that FSMs cannot answer but that Turing machines can. Why this should be the case is naturally of interest to us. We will take all of our machines to be black boxes, whose inner mechanical workings are hidden from us; we have no interest in these details. We are only interested in their behavior. To familiarize you with the relevant concepts, let me give an example of a finite state machine (Fig. 3.1):

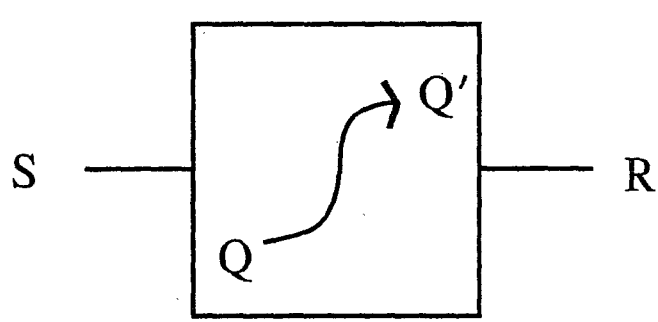


Fig. 3.1 A Generic Finite State Machine

The basic idea is as follows. The machine starts off in a certain *internal state*, Q. This might, for example, simply be holding a number in memory. It then receives an *input*, or *stimulus*, S — you can either imagine the machine reading a bit of information off a (finite) tape or having it fed in in some other way. The machine reacts to this input by *changing to another state*, Q', and spitting something out — a *response* to the input, R. The state it changes to and its response are determined by both the initial state and the input. The machine then repeats this cycle, reading another input, changing state again, and again issuing some response.

To make contact with real machines, we will introduce a discrete time variable, which sets the pace at which everything happens. At a given time t, we have the machine in a state $Q(t)$ receiving a symbol $S(t)$. We arrange things so that the response to this state of affairs comes one pulse later, at time $(t+1)$. Let us, for notational purposes, introduce two functions F and G, to describe the FSM and write:

$$\begin{aligned} R[t+1] &= F[S(t), Q(t)] \\ Q[t+1] &= G[S(t), Q(t)] \end{aligned} \tag{3.2}$$

We can depict the behaviour of FSMs in a neat diagrammatic way. Suppose a machine has a set of possible states $\{Q_j\}$. We represent the basic transition of this machine from a state Q_j to a state Q_k upon reception of a stimulus S, and resulting in a response R, as follows:

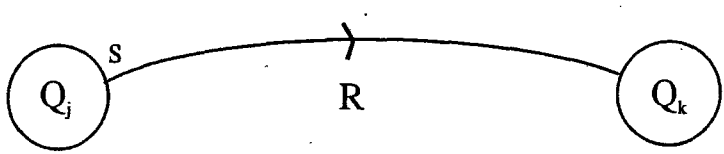


Fig. 3.2 A Graphical Depiction of a State Transition

This graphical technique comes into its own when we have the possibility of multiple stimuli, responses and state changes. For example, we might have the system shown below in Fig. 3.3:

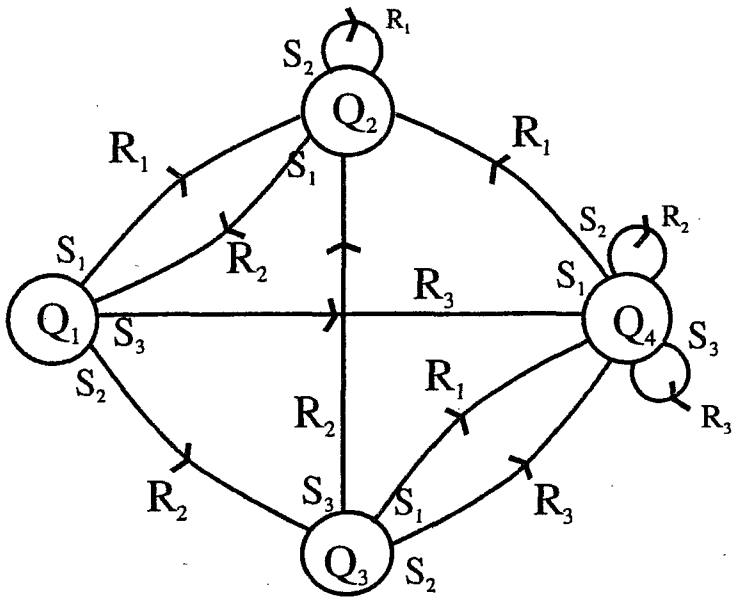


Fig. 3.3 A Complex Finite State Machine

This FSM behaves as follows: if it is in state Q_1 and it receives a stimulus S_1, it spits out R_1 and goes into state Q_2. If, however, it receives a stimulus S_2, it spits out R_2 and changes to state Q_3. Getting S_3, it switches to state Q_4 and produces R_3. Once in state Q_2, if it receives a stimulus S_1, it returns to state Q_1, responding with R_2, whilst if it receives a stimulus S_2 it stays where it is and spits out R_1. The reader can figure out what happens when the machine is in states Q_3 and Q_4, and construct more complex examples for himself.

One feature of this example is that the machine was able to react to three distinct stimuli. It will suit our purposes from here on to restrict the possible stimuli to just two — the binary one and zero. This doesn't actually affect what we can do with FSMs, only how quickly we can do it; we can allow for the possibility of multiple input stimuli by feeding in a sequence of 1's and 0's, which is clearly equivalent to feeding in an arbitrary number, only in binary format. Simplifications of this kind are common in the study of FSMs and Turing machines where we are not concerned with their speed of operation.

Let me now give a specific example of an FSM that actually does something, albeit something pretty trivial — a delay machine. You feed it a

stimulus and, after a pause, it responds with the same stimulus. That's all it does. Figure 3.4 shows the "state diagram" of such a delay machine.

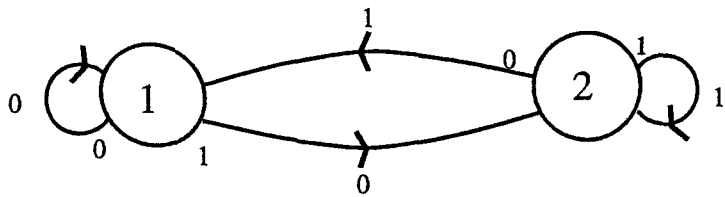


Fig. 3.4 A Delay Machine

You can hardly get a simpler machine than this! It has only two internal states, and acts as a delay machine solely because we are using pulsed time and demanding that the machine's response to a stimulus at time t comes at time t+1. If we tell our machine to spit out whatever we put in, we will have a delay time of one unit. It is possible to increase this delay time, but it requires more complicated machines. As an exercise, try to design a delay FSM that remembers *two* units into the past: the stimulus we put in at time t is fed back to us at time t+2. (Incidentally, there is a sense in which such a machine can be taken as having a memory of only one time unit: if we realize that the *state* at time t+1 tells us the input at time t. It is often convenient to examine the state of an FSM rather than its response.)

Another way of describing the operation of FSMs is by tabulating the functions F and G we described earlier. Understanding the operation of an FSM from such a table is harder than from its state diagram, and becomes hopeless for very complex machines, but we will include it for the sake of completeness:

G	Q_0	Q_1
S_0	Q_0	Q_0
S_1	Q_1	Q_1

F	Q_0	Q_1
S_0	R_0	R_1
S_1	R_0	R_1

Table 3.1 State Table for a Generic FSM

Now it is surprising what you can do with these things, and it is worth getting used to deciphering state diagrams so that you can appreciate this. I am going to give you a few more examples, a little more demanding than our delay machine. First up is a so-called "toggle" or "parity" machine. You feed this machine a string of 0's and 1's, and it keeps track of whether the number of 1's it has received is even or odd; that is, the parity of the string.

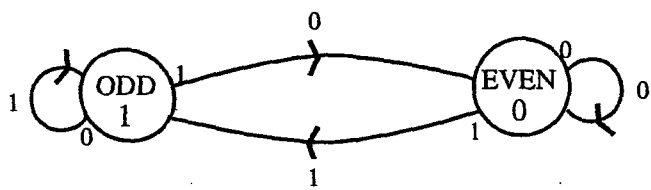


Fig. 3.5 The Parity Machine

From the diagram in Figure 3.5, you can see that, one unit of time after you feed in the last digit, the response of the FSM tells you the parity. If it is a 1, the parity is odd — you have fed in an odd number of 1's. A 0 tells you that you have fed in an even number. Note that, as an alternative, the parity can be read off from the state of the machine; which I have flagged by labeling the two possible states as "odd" and "even".

Let me give you some simple problems to think about.

Problem 3.1: Suppose we feed a sequence of 1's and 0's — a binary number — into a machine. Design a machine which performs a pairwise sum of the digits, that is, one which takes the incoming digits two at a time and adds them, spitting the result out in two steps. So, if two digits come in as 00, it spits out 00; a 10 or 01 results in a 01 (1+0 = 0+1!); but a 11 results in binary 10: 1+1 = 2, in decimal, 10 in binary. I will give you a hint: the machine will require four states.

Problem 3.2: Another question you might like to address is the design of another delay machine, but this time one which remembers and returns *two* input stimuli. You can see that such a device needs four states — corresponding to the four possible inputs 00, 01, 10 and 11.

Problem 3.3: Finally, if you are feeling particularly energetic, design a two-

input binary adder. I want the full works! I feed in two binary numbers, simultaneously, one bit from each at a time, *with the least significant bits first*, and the FSM, after some delay, feeds me the sum. I'm not interested in it telling me the carry, just the sum. We can schematically depict the desired behaviour of the machine as follows:

Time ⟶		
Inputs	1 0 1 0 1 1	
	0 1 1 0 1 0	
Output = sum	1 1 0 1 0 0	(Carrying 1 into the next column)

3.3: The Limitations of Finite State Machines

If you have succeeded in designing an adder, then you have created a little wonder — a simple machine that can add two numbers of any size. It is slow and inefficient, but it does its job. This is usually the case with FSMs. However, it is important to appreciate the limitations of such machines; specifically, there are many tasks that they cannot perform. It is interesting to take a look at what they are. For example, it turns out that one cannot build a FSM that will multiply any two arbitrary binary numbers together. The reason for this will become clear in just a moment, after we have examined a simpler example. Suppose we want to build a *parenthesis checker*. What is one of these? Imagine you have a stream of parentheses, as follows:

(((( )) ((( ) ( ) ((( )) ( ) ( ) (( ))) ( ))

The task of a parenthesis checker is to ascertain whether such an expression is "balanced": that the brackets open and close consistently throughout the expression. This is not the same as just counting the number of left and right brackets and making sure they are equal! They have to match in the correct order. This is a common problem in arithmetic and algebra, whenever we have operations nested within others. The above example, incidentally, is invalid; this one:

(( ) (( ) ( )) ((( ) ( ) (( ) ( )))))

is valid. You might like to check in each case.

On the face of it, building a parenthesis checker seems a pretty straightforward thing to do. In many ways it is, but anything you get to implement the check would not be an FSM. Here is one way you could proceed. Starting from the left of the string, you count open brackets until you reach a close bracket. You "cancel" the close bracket with the rightmost open bracket, then move one space to the right. If you hit a close bracket, cancel it with another open bracket; if you hit an open bracket, add one to the number of open brackets you have uncanceled and move onto the next one. It is a very simple mechanism, and it will tell you whether or not your parenthesis string is OK: if you have any brackets left over after you process the rightmost one, then your string is inconsistent. So why cannot an FSM do something this simple?

The answer is that the parenthesis checker we want has to cope with *arbitrary* strings. That means, in principle, strings of arbitrary length which might contain arbitrarily large numbers of "open" brackets. Now recall that an essential feature of the machine is that it must keep track of how many open brackets remain uncancelled by closed ones at each stage of its operation; yet to do this, in the terminology of FSMs, it will need a distinct state for each distinct number of open brackets. Here lies the problem. An *arbitrary* string requires a machine with an arbitrary — that is, ultimately, infinite — number of states. Hence, no *finite* state machine will do. What will do, as we shall see, is a Turing machine — for a Turing machine is, essentially, an FSM with infinite memory capacity.

For those who think I am nitpicking, it is important to reiterate that I am discussing matters of principle. From a practical viewpoint, we can clearly build a finite state machine to check the consistency of any bracket string we are likely to encounter. Once we have set its number of states, we can ensure that we only feed it strings of an acceptable size. If we label each of its states by 32 32-bit binary numbers we can enumerate over 2^{1000} states, and hence deal with strings 2^{1000} brackets long. This is far more than we are ever likely to encounter in practice: by comparison, note that current estimates place the number of protons in the universe only of the order of 2^{200}. But from a mathematical and theoretical standpoint, this is a very different thing from having a universal parenthesis checker: it is, of course, the difference between the finite and the infinite, and when we are discussing academic matters this is important. We *can* build an FSM to add together two arbitrarily large numbers; we *cannot* build a parenthesis checking FSM to check any string we choose. Incidentally, it is the need for an infinite memory that debars the construction of an FSM for binary multiplication.

Before getting onto Mr. Turing and his machines, I would like to say one or two more things about those with a finite number of states. One thing we looked at in detail in previous chapters was the extent to which complicated logic functions could be built out of simple, basic logic units — such as gates. A similar question arises here: is there a core set of FSMs with which all others can be built? To examine this question, we will need to examine the ways in which FSMs can be combined.

Figure 3.6 shows two machines, which I call **A** and **B**. I have linked them up in something of a crazy way, with feedback and whatnot. Don't worry if you can't see at a glance what is going on!

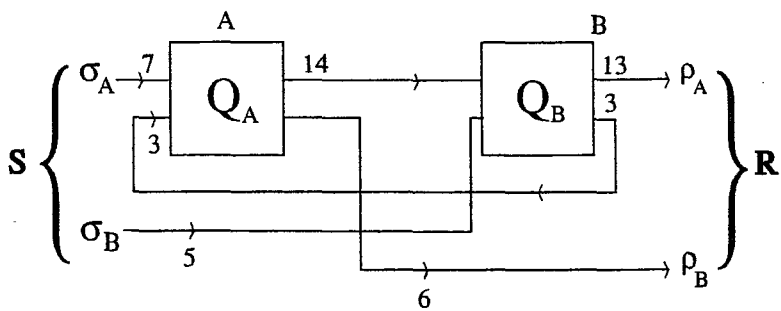
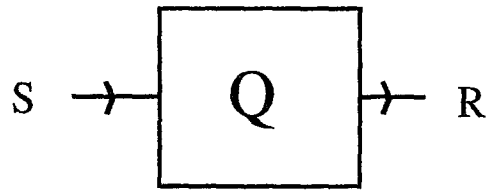


Fig. 3.6 A Composite FSM

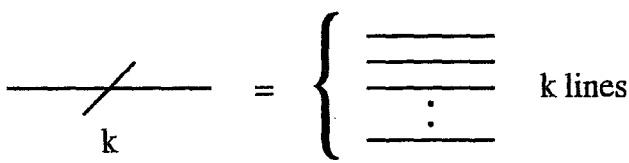
Let me describe what the diagram represents. In a general FSM, the input stimulus can be any binary number, as can its response. Whether the stimulus is fed in sequentially, or in parallel (e.g. on a lot of on/off lines), we can split it up into two sets. Suppose the stimulus for **A** has ten bits. We split this up into, say, a 7-bit and a 3-bit stimulus. Now comes the tricky part: we take the 7-bit input to be external, fed in from outside on wire σ_A, but the 3-bit input we take from the *response of machine B* — which we have also split up. In the case of **B**, we take the response to have, say, 16 bits, and 3 of these we re-route to **A**, the other 13 we take as output. Bear with me! What about the response of **A**? Again, we split this up: suppose it is 20 bits. We choose (this is all arbitrary) to feed 14 into **B** as input, and with the remaining 6 we bypass **B** and take them as output. The remainder of **B**'s input — whatever it may be — is fed in from outside, on wire σ_B. Let's say σ_B carries 5 bits.

The point of all these shenanigans is that this composite system can be represented as a single finite state machine:



where the input stimulus is the combined binary input on wires σ_A and σ_B, and the output is the partial responses from **A** and **B**, again combined. Clearly, the machine has an input stimulus of 7+5=12 bits, and a response of 13+6=19 bits. Exactly what the thing does depends on the properties of **A** and **B**; it seems feasible that the number of internal states of this combined machine is the product of the number of states of **A** and **B**, but one must be careful about the extent to which things can be affected by feedback and the information running around the wires. What I wanted to show was how you could build an FSM from smaller ones by tying up the loose wires appropriately. You might like to see what happens if you arrange things differently – by forgetting feedback, for example. You will find that feedback is essential if you want as few constraints as possible on the size of the overall input and output bit sizes: connecting up two machines by, say, directly linking output to input not only fixes the sizes of the overall stimulus/response but also requires the component FSMs to match up in their respective outputs and inputs.

Let me return to my question: can we build any FSM out of a core set of basic FSMs? The answer turns out to be yes: in fact, we find ourselves going right back to our friends AND and NOT, which can be viewed as finite state machines themselves, and which we can actually use to build any other FSM. Let me show roughly how this is done. We will first need a bit of new notation. Let us represent a set of k signal-bearing wires by a single wire crossed with a slash, next to which we write the number k:



With this convenient diagrammatic tool, we can draw a schematic diagram of

a general finite state machine (Fig. 3.7):

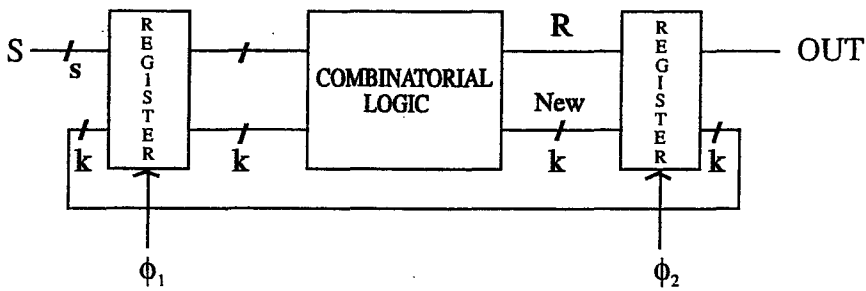


Fig. 3.7 The General FSM

The operation of this rather complicated-looking device is quite straightforward. It comprises two registers (such as those we constructed in Chapter 2 from clocked flip-flops) and a black box that performs certain logical functions. The input to the first register has two pieces, the stimulus S to the FSM and the state the machine is in, Q: central to our design is the fact that we can label the internal states by a binary number. In this case, the stimulus has s bits, and is fed in on s wires, while the state has k bits, fed in on k wires. (The FSM has therefore up to 2^k internal states). Subject to timing, which I will come back to, the register passes these two inputs into the logic unit. Here is the trick. An FSM, in response to a given stimulus and being in a given state, produces a response and goes into a (possibly) new state. In terms of our current description, this simply amounts to our black box receiving two binary strings as input, and producing two as output — one representing the response, the other the new state. The new state information is then fed back into the first register to prime the machine for its next stimulus. Ensuring that the FSM works is then just a matter of building a logic unit which gives the right outputs for each input, which we know is just a matter of combining ANDs and NOTs in the right way.

A quick word about timing. As we have discussed, the practicalities of circuit design mean that we have to clock the inputs and outputs of logic

devices; we have to allow for the various delays in signals arriving because of finite travel times. Our FSM is no exception, and we have to connect the component registers up to two clocks as usual; the way these work is essentially the same as with standard logic circuits. The first register is clocked by ϕ_1, the second by ϕ_2, and we arrange things such that when one is on, the other is off – which we do by letting ϕ_2 = NOT ϕ_1 and hooking both up to a standard clock – and ensuring that the length of time for which each is on is more than enough to let the signals on the wires settle down. The crucial thing is to ensure that ϕ_2 is *off* whilst ϕ_1 is *on*, to prevent the second register sending information about the change of state to the first while it is still processing the initial state information.

Problem 3.4: Before turning to Turing machines, I will introduce you to a nice FSM problem that you might like to think about. It is called the "Firing Squad" problem. We have an arbitrarily long line of identical finite state machines that I call "soldiers". Let us say there are N of them. At one end of the line is a "general", another FSM. Here is what happens. The general shouts *"Fire"*. The puzzle is to get all of the soldiers to fire simultaneously, in the shortest possible time, subject to the following constraints: firstly, time goes in units; secondly, the state of each FSM at time T+1 can only depend on the state of its next-door neighbors at time T; thirdly, the method you come up with must be independent of N, the number of soldiers. At the beginning, each FSM is quiescent. Then the general spits out a pulse, *"fire"*, and this acts as an input for the soldier immediately next to him. This soldier reacts in some way, enters a new state, and this in turn affects the soldier next to him, and so on down the line. All the soldiers interact in some way, yack yack yack, and at some point they become synchronized and spit out a pulse representing their "firing". (The general, incidentally, does nothing on his own initiative after starting things off.)

There are different ways of doing this, and the time between the general issuing his order and the soldiers firing is usually found to be between $3N$ and $8N$. It is possible to prove that the soldiers cannot fire earlier than T=2N-2 since there would not be enough time for all the required information to move around. Somebody has actually found a solution with this minimum time. That is very difficult though, and you should not be so ambitious. It is a nice problem, however, and I often spend time on airplanes trying to figure it out. I haven't cracked it yet.

3.4: Turing Machines

Finally, we come to Turing machines. Turing's idea was to conceive of himself, or any other mathematician, as a machine, having a finite state machine in his head, and an unlimited amount of paper at his disposal to write on. It is the unlimited paper — hence effectively unbounded memory — that distinguishes a Turing machine from an FSM. Remember that some problems — parenthesis checking, multiplication — cannot be done by finite state machines, because, by definition, they lack an unlimited memory capacity. This restriction does not apply to Turing machines. Note that we are not saying that the amount of paper attached to such a machine *is* infinite; at any given stage it will be finite, but we have the option of adding to the pile whenever we need more. Hence our use of the word "unlimited".

Turing machines can be described in many ways, but we will adopt the picture that is perhaps most common. We envisage a little machine, with a finite number of internal states, that moves over a length of tape. This tape is how we choose to arrange our paper. It is sectioned off into cells, in each of which might be found a symbol. The action of the machine is simple, and similar to that of an FSM: it starts off in a certain state, looking at the contents of a cell. Depending on the state, and the cell contents, it might erase the contents of the cell and write something new, or leave the cell as it is (to ensure uniformity of action, we view this as erasing the contents and writing them back in again). Whatever it does, it next moves one cell to the left or right, and changes to a new internal state. It might look something like Figure 3.8:

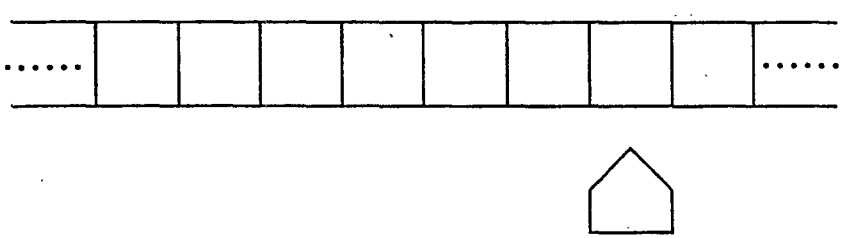


Fig. 3.8 A Turing Machine

We can see how similar the Turing machine is to an FSM. Like an FSM, it has internal states. Reading the contents of a cell is like a stimulus, and overwriting the contents is like a response, as is moving left or right. The restriction that the machine move only one square at a time is not essential; it just makes it more

primitive, which is what we want. One feature of a Turing machine that *is* essential is that it be able to move both left and right. You can show (although you might want to wait until you are more familiar with the ideas) that a Turing machine that can only move in one direction is just a finite state machine, with all its limitations.

Now we are going to start by insisting that only a finite part of the tape have any writing on it. On either side of this region, the tape is blank. We first tell the machine where to start, and this is at time T. Its later behavior, at a time T+1 say (Turing machines operate on pulsed time like FSMs), is specified by three functions, each of which depends on the state Q_i at time T and the symbol S_i it has just read: these are its new state, Q_j, the symbol it writes, S_j, and the direction of its subsequent motion, D. We can write:

$$\begin{aligned} Q_j &= F(Q_i, S_i) \\ S_j &= G(Q_i, S_i) \\ D &= D(Q_i, S_i) \end{aligned} \tag{3.3}$$

This list is just like the specification of an FSM but with the extra function D. The complete machine is fully described by these functions, which you can view as one giant (and finite) look-up list of "quintuples" — a fancy name for the set of five functions we have defined, two at time T (Q_i and S_i), and three at T+1 (Q_j, S_j and D). All you do now is stick in some data — which you do by writing on the tape and letting the machine look at it — tell the machine where to start, and leave it to get on with it. The idea is that the machine will finish up by printing the result of its calculation somewhere on the tape for you to peruse at your leisure. Note that for it to do this, you have to give it instructions as to when it is to halt or stop. This seems pretty trivial, but as we will see later, matters of "halting" hide some very important, and very profound, issues in computation.

Before giving you some concrete examples of Turing machines, let me remind you of why we are looking at them. I have said that finding an effective procedure for doing a problem is equivalent to finding a Turing machine that could solve it. This does not seem much of an insight until we realize that among the list of all Turing machines, by which I mean all lists of quintuples, there exists a very special kind, a *Universal Turing machine* (UTM), which can do anything any other Turing machine can do! Specifically, a UTM is an

imitator, mimicking the problem-solving activities of simpler Turing machines. (I say "a" UTM, rather than "the" UTM since, while all UTMs are computationally equivalent, they can be built in many different ways). Suppose we have a Turing machine, defined by some list of quintuples, which computes a particular output when we give it a particular set of input data. We get a UTM to imitate this process by feeding it a description of the Turing machine — that is, telling the UTM about the machine's quintuple list — and the input data, both of which we do by writing them on the UTM's tape in some language it understands, in the same way we feed data into any Turing machine. We also tell the UTM where each begins and ends[2]. The UTM's internal program then takes this information and mimics the action of the original machine. Eventually, it spits out the result of the calculation: that is, the output of the original Turing machine. What is impressive about a UTM is that *all* we have to do is give it a list of quintuples and some initial data — its own set of defining quintuples suffice for it to mimic any other machine. We don't have to change them for specific cases[3]. Why such machines are important to us is because it turns out that, if you try to get a UTM to impersonate *itself*, you end up discovering that there are some problems that no Turing machine — and hence no mathematician — can solve!

Let us now look at a few real Turing machines. The first, and one of the simplest, is related to a finite state machine we have already examined — a parity counter. We feed the machine a binary string and we want it to tell us whether the number of 1's in the string is odd or even. Schematically we have (Fig. 3.9):

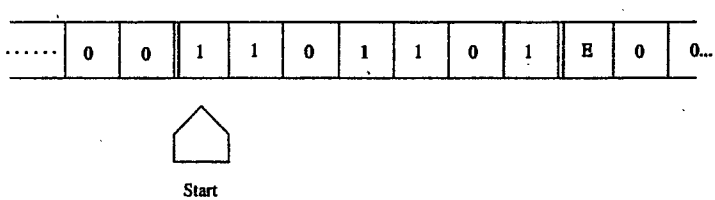


Fig. 3.9 Input Tape for the Parity Counter

We begin by writing the input data, the binary string, onto the tape as shown;

[2]The section of the UTM's tape containing information about the machine it is imitating is usually referred to as the "pseudotape". [RPF]

[3]We will actually construct a UTM later. [RPF]

each cell of the tape holds one digit. The "tape-head" of the machine rests at the far left of the string, on the first digit, and we define the machine to be in state Q_0. To the left of the string are nothing but zeroes, and to the right, more zeroes – although we separate these from the string with a letter E, for "end", so that the machine does not assume they are part of it.

The operation of the machine, which we will shortly translate into quintuples, is as follows. The state of the machine tells us the parity of the string. The machine starts off in state Q_0, equal to even parity, as it has not yet encountered any 1s. If it encounters a zero, it stays in state Q_0 and moves one space to the right. The state does not change because the parity does not change when it hits a zero. However, if it hits a 1, the machine erases it, replaces it with a zero, moves one space to the right, and enters a state Q_1. Now if it hits a zero, it stays in state Q_1 and moves a space to the right, as before. If it hits a 1, it erases it, putting a zero in its place, and moves to the right, this time reverting to state Q_0. You should now have an idea what is happening. The machine works its way across the string from left to right, changing its state whenever it encounters a 1, and leaving a string of 0s behind. If the machine is in state Q_0 when it kills the last digit of the string, then the string has even parity; if it is in state Q_1, it is odd. How does the machine tell us the parity? Simple – we include a rule telling the machine what to do if it reads an E. If it is in state Q_0 and reads E, it erases E and writes "0", meaning even parity. In state Q_1, it overwrites E with a "1", denoting odd parity. In both cases it then enters a new state Q_H, meaning "halt". It does not need to move to the right or left. We examine the tape, and the digit directly above the head is the answer to our question. We end up with the situation shown in Figure 3.10:

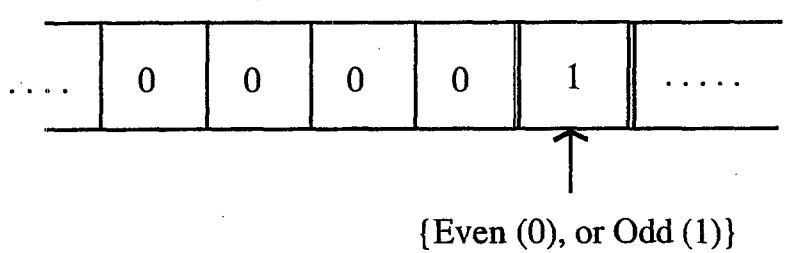


Fig. 3.10 Output Tape from the Parity Counter

The quintuples for this machine are straightforwardly written out (Table 3.2):

Initial State	Read	New State	Write	Direction of Move
0	0	0	0	R
0	1	1	0	R
1	0	1	0	R
1	1	0	0	R
0	E	H(alt)	0	-
1	E	H	1	-

Table 3.2 Quintuples for the Parity Couuter

Now this device is rather dumb, and we have already seen that we could solve the parity problem with a finite state machine (note here how our Turing machine has only moved in one direction!). We will shortly demonstrate the superiority of Mr. Turing's creations by building a parenthesis checker with them, something which we have seen cannot be done with an FSM, but first let me introduce some new diagrammatics which will make it easier for us to understand how these machines work without tying ourselves in knots wading through quintuple lists.

The idea is, unsurprisingly, similar to that we adopted with FSMs. In fact, the only real difference in the diagrams is that we have to somehow include the direction of motion of the head after it has overwritten a cell, and we have to build in start and halt conditions. In all other respects the diagrams resemble those for FSMs. Take a look at Figure 3.11, which describes our parity counter:

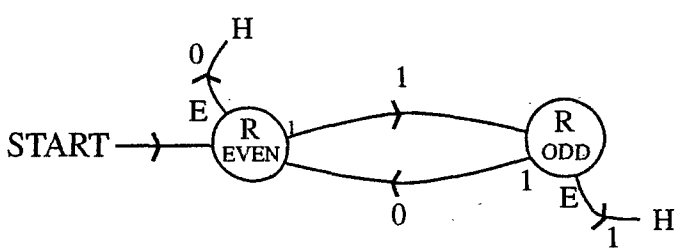


Fig. 3.11 A Turing Machiue Parity Counter

This is essentially the same as Figure 3.5, the FSM which does the same job. Where the FSM has a stimulus, the TM has the contents of a cell. In these diagrams, both are written at the point of contact of lines and circles. Where the FSM spits out a response, which we wrote on the arrow linking states, the TM overwrites the cell contents, what it writes being noted on the arrow. The state labels of both FSMs and TMs are written inside the circles. The major differences are that, firstly, we have to know where the machine starts, which we do by adding an external arrow as shown; and we have to show when it stops, which we do by attaching another arrow to each state to allow for the machine reading *E*, each arrow terminating in a "Halt". More subtly, we also have to describe the direction of its motion after each operation. It turns out that machines whose direction of motion depends *only* on their internal state — and not on the symbols they read — are not fundamentally less capable of carrying out computations than more general machines which allow the tape symbols to influence the direction of motion. I will thus restrict myself to machines where motion to the right or left depends solely on the internal state. This enables me to solve the diagrammatic problem with ease: just write *L* or *R*, as appropriate, inside the state box. In this case, both states are associated with movement to the right.

I have gone on at some length about the rather dumb parity machine as it is important that you familiarize yourself with the basic mechanics and notation of Turing machines. Let me now look at a more interesting problem, that of building a parenthesis checker. This will illustrate the superiority of Turing machines over finite state machines. Suppose we provide our Turing machine with a tape, in each cell of which is written a parenthesis (Fig. 3.12):

....E(())()(()))(()E....

Fig. 3.12 Input Tape to the Parenthesis Checker

Each end of the string is marked with a symbol *E*. This is obviously the simplest way of representing the string. How do we get the machine to check its validity? One way is as follows. I will describe things in words first, and come back to discuss states and diagrams and so forth in a moment. The machine starts at the far left end of the string. It runs through all the left brackets until it comes to a right bracket. It then overwrites this right bracket with an *X* — or any other symbol you choose — and then moves one square to

the left. It is now on a left bracket. It overwrites this with an X, too. It has now canceled a pair of brackets. The key property of the X's is that the machine doesn't care about them; they are invisible. After having canceled a pair in this way, the machine moves right again, passing through any X's and left brackets, until it hits a right bracket. It then does its stuff with the X again. As you can see, in this way the machine systematically cancels pairs of brackets. Sooner or later, the head of the machine will hit an E — it could be either one — and then comes the moment of truth. When this happens, the machine has to check whether the tape between the two Es contains only X's, or some uncanceled brackets too. If the former, the string is valid, and the machine prints (say) a 1 somewhere to tell us this; if the latter, the machine prints 0, telling us the string is invalid. Of course, after printing, the machine is told to halt.

If you think about it, this very simple procedure will check out any parenthesis string, irrespective of size. The functioning of this machine is encapsulated by the state diagram of Figure 3.13 (following Minsky [1967]):

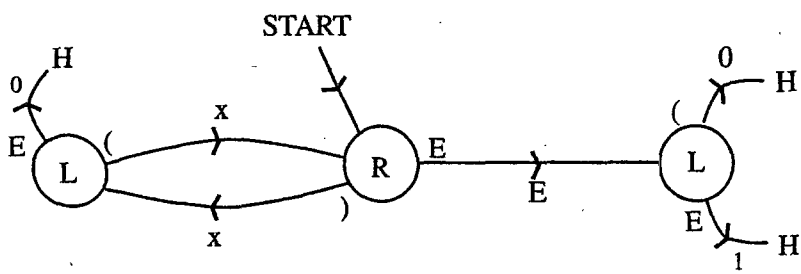


Fig. 3.13 The Parenthesis Checker State Diagram

Note how the diagram differs from that for an FSM: we have to include start and stop instructions, and also direction of motion indicators. In fact, this machine, unlike the parity counter, requires two different left-moving states.

Now that you have some grasp of the basic ideas, you might like to try and design a few Turing machines for yourself. Here are some example problems to get you thinking.

Problem 3.5: Design a unary multiplier. "Unary" numbers are numbers written in base 1, and are even more primitive than binary. In this base, we have only

the digit 1, and a number N is written as a string of N 1's: 1 = 1, 2(base 10) = 11(base 1), 3 = 111, 4 = 1111, and so on. I would like you to design a Turing machine to multiply together any two unary numbers. Start with the input string:

00...E 1111....1 B 1111 ....11 E...00
m n

which codes the numbers being multiplied, m and n and separates the two numbers with the symbol B. The goal is to end up with a tape that gives you mn. It might look something like this:

...00 E 0000...0 B XXX...X E YYYY...Y 00...
m n mn

where Y is some symbol distinct from 0, 1, X, E and B. You can consider the given tape structure a strong hint as one way in which you could solve the problem!

Problem 3.6: We have discussed binary adders before. I would now like you to design a Turing machine to add two binary numbers, but only for the case where they have the same number of bits (this makes it easier). You can start with the initial tape:

...00 A 1101..1 B 1001..0 C 000...
m n

for numbers m and n with the field of the two numbers delineated by the symbols A, B and C. I will leave it to you to decide where the machine starts, how it proceeds, what its final output looks like, where it appears, and so on.

Problem 3.7: If you're finding these problems too easy, here's one that is much harder: design a Turing machine for a binary multiplier!

Problem 3.8: This last problem is neat: design a unary to binary converter. That is, if you feed the machine a string of 1's representing a unary number, it gives you that number converted to binary. The secret to this problem lies in the mathematics of divisors and remainders. Consider what we mean when we talk of the binary form of an n-bit number $N = N_n N_{n-1} \ldots N_1 N_0$. By definition we have:

$$N = N_n.2^n + N_{n-1}.2^{n-1} + \ldots + N_1.2 + N_0$$

We start with N written in unary — i.e. a string of N 1's — and we want to find the coefficients N_i, the digits in binary. The rightmost digit, N_0, can be found by dividing N by two, and noting the remainder, since:

$$N = 2.X + N_0$$

with X easily ascertained. To find N_1, we get rid of N_0, and use the fact that:

$$X = 2.Y + N_1$$

That is, we divide X by two and note the remainder — N_1. We just keep doing this, shrinking the number down by dividing by two and noting the remainder, until we have the binary result. Note that, since N is an n-bit number, by definition N_n *must be 1*.

If we are given the number N in unary form, we can simulate the above procedure by grouping the 1's off pairwise and looking at what is left. Let us take a concrete example. Use the number nine in base ten, or 111111111 in unary. Pair up the 1's:

(11) (11) (11) (11) 1

Clearly, this is just like dividing by two. There is an isolated digit on the right. This tells us that N_0 is 1. To find N_1, we scratch the righthand 1 and pair up the pairs in the remaining string:

(11 11) (11 11).

This time, there is no remainder: N_1 is 0. Similarly, we find that N_2 is 0. We have now paired up all our pairs and pairs of pairs, and the only thing left to do is tag a 1, for N_3, to the left of the number, giving us 111111111 (unary) = 1001 (binary).

I will leave it up to you to implement this algorithm with a Turing machine. You have to get the thing to pair off digits, mark them as pairs and check the remainder; and then come back to the beginning and mark off pairs of pairs, and so on. Marking pairs is probably best done by starting at the left end of the string and going to the right, striking out every other digit and replacing it with an *X* symbol. When the machine gets to go through the string again, it ignores the *X*'s and strikes out every other 1 again. This method, suitably refined, will work! I leave it to you to figure out the details. Don't forget that you have to get the machine to start, perform the conversion, write its output and then stop.

3.5: More on Turing Machines

I would now like to take a look at a fairly complicated Turing machine that bears on a different aspect of computing. Earlier in these lectures I pointed out that computers were more paper pushers than calculators, and it would be nice to see if we can build a Turing machine that performs filing, rather than arithmetic, functions. The most primitive such function is looking up information in a file, and that is what we are going to examine next. We want a machine that first locates a file in a file system, then reads its contents, and finally relays these contents to us[4].

We will employ the following Turing "filing system", or tape (Fig. 3.14), which we are to feed into our machine:

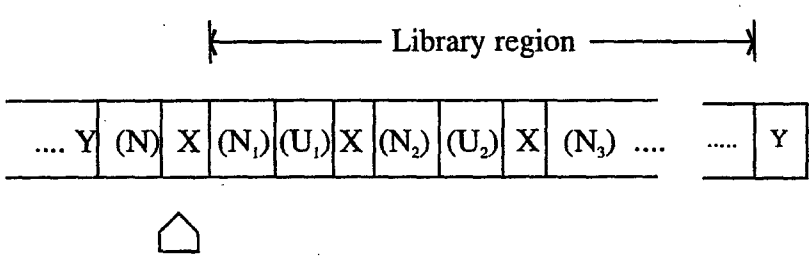


Fig. 3.14 Input Tape to the Locating Machine

[4]Our discussion closely follows Minsky [1967]. [RPF]

This is a bit schematic. The X-symbols this time play the role of segregating various file entries; there is one entry between each pair of X's. Each entry comprises a name (or address), "N", and contents, "U", both of which we take to be binary strings, one digit per tape square as usual. We have attached to the left hand end of the tape the name of a file which we want the machine to read for us, and denoted the left end of the tape by a symbol Y. To the left of this is a string of zeroes; the same is true at the right-hand end of the tape. The machine is to start where marked to the right of the name N of the file we want to find.

The first task confronting the machine is that of locating the right file. It does this by systematically comparing each file name in the list with the target name, working from left to right, until it finds the correct one. How should it do this? For ease of understanding, suppose we have the following filing tape (Fig. 3.15):

```
...000 Y 101 X 001 011 X 101 110 X 111 000 Y 000...
                 N   U
```

Fig. 3.15 A Sample Filing Tape

For convenience, we are taking both the name strings and the data strings to be of the same length, three bits. We want to read the contents of file (101) which we'll call the target file. Now it might seem that the best thing to do is the following: assign to each possible target a distinct state of the Turing machine. This will give us at most eight states. The machine starts in the state 101 dictated by the target file name and goes to the first file from the left, and looks at the name. If there is a match, all well and good. If not, it goes to the next file on the right, checks that, and so on. In this way, the machine smoothly moves from left to right until it hits the correct address. However, the problem with such a machine is that it has only eight states and will only be any good for three-bit filing systems: it has no universality of application. We want a single machine that can handle any size of filename. To achieve this, the machine must compare each filename with the target on a sequential, digit-by-digit basis, laboriously shuttling between the two until a mismatched digit is found, in which case it goes onto the next filename, or until a complete match is found, when (say) it returns to its starting point. To keep track of those parts of the

tape it has already considered, the machine would do the now-familiar trick of overwriting digits with symbols which it subsequently ignores, just as we did with the parenthesis checker. By assigning different symbols to 0's and 1's — *A*'s and *B*'s, say — we can keep track of which were 0's and 1's; if we wanted to come along tomorrow and use the file again, we could, only we would find it written in a different alphabet. We could then reconstruct the entire original file by overwriting the new symbols with 0's and 1's.

Minsky's solution for a locating Turing machine is shown in Figure 3.16:

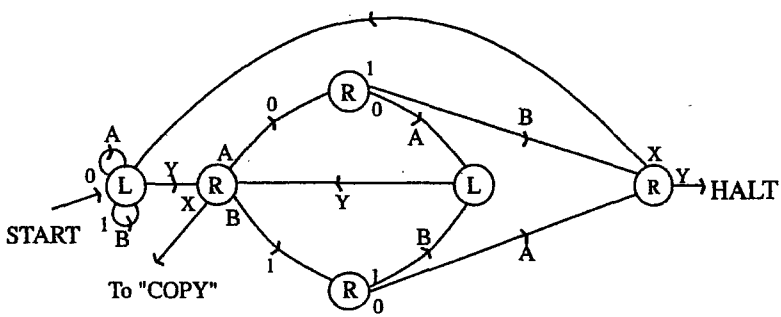


Fig. 3.16 The Locating Machine

There is a loose end in this diagram, pointing to "copy". This represents the stage at which the machine has located the correct filename and is wondering what to do next. We will shortly show how we are going to get it to copy the information in the file to a point of our choosing on the tape.

For the moment let us stick with our location machine and look in more detail at how it works. The head starts on the first *X* to the right of the target number. As the loop instructions show, the machine then heads left, changing the 0's and 1's in the target to *A*'s and *B*'s respectively. This may seem a little bizarre, but there is a point to it, as we will see. Eventually, the machine hits the *Y*. It then goes into a new state, and as is clear from the diagram, it will start moving right. It will first encounter one of the *A*'s or *B*'s it has just written: it overwrites this with the original digit (this *definitely* seems bizarre, but it will

make sense!), a 0 or 1, and moves right again. It now enters one of two states in which it will only recognise a 0 or 1: not an *A* or *B*. If it hits an *A* or *B*, it will ignore it, keeping on moving right – in other words, it is going to pass right through the remainder of the rewritten target string, having in a sense "noted" the first digit of the string. This is why we overwrote the 0's and 1's of the string with *A*'s and *B*'s. It will also pass straight through the *X* it encounters and go on to the first filename to be checked.

Now comes the crucial sequence of operations. The machine is going to hit either a 1 or a 0, and how it reacts depends on how it has been primed – i.e. on the state it is in as a result of reading the first target digit. There are two possibilities. Firstly, if the digit it hits is different from the first target digit, so the filenames do not match from the outset, the machine overwrites the digit as appropriate, and then moves to the right until it hits the next *X*, denoting the end of the file. It then starts to move to the left, overwriting the contents of the rest of the file with *A*'s and *B*'s. It passes through the leftmost *X*, zips through the target filename (*A*'s and *B*'s are invisible to it), changes the first digit to an *A* or *B*, and hits *Y*. This is a cue for the whole process to start again: only now it goes to the next filename. Sooner or later, the first target digit and that of the checked filename will match, and this is the second possibility we must consider.

When a match occurs, the machine overwrites the matching digit, and enters a state in which it moves back left until it encounters the *Y*. Then, it goes forwards, overwrites the second target symbol with the correct digit, and then moves on to the files. It checks the second digit for a match, and so it goes on. Working through the machine diagram, you should be able to convince yourself that the tape above would ultimately be converted into the tape of Figure 3.17:

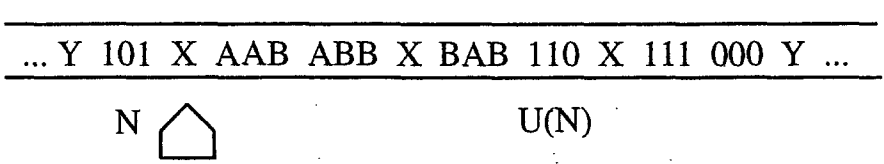


Fig. 3.17 Output Tape from the Locating Machine

Note that the head has returned to its starting point, and the effect of its activities has been to change all 0's and 1's between the start and the end of the

desired filename (but not the contents of the file) to *A*'s and *B*'s. (There is the important possibility that the target filename cannot be found, because we have typed it in wrongly, say, and in this case the machine head will end up on the *Y* at the far right; as the diagram indicates, at this juncture it is instructed to "Halt".)

As I have said, there is a "loose wire" on our diagram, representing a feed to a copy machine: we have our file, now we want to know what to do with it! True to the spirit of Turing machines, we are going to copy it slowly and laboriously to another part of the tape. That is, *you* are: the copying machine is shown in Figure 3.18, and its input tape is the output tape of the location machine. Have fun figuring out how it works!

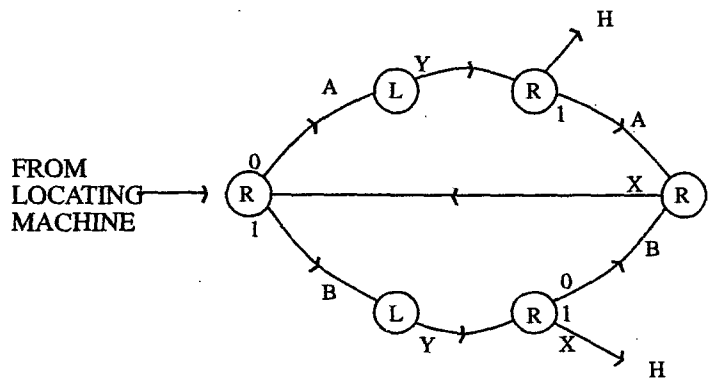


Fig. 3.18 The Copy Machine

A cute feature of this machine is that it copies the contents of the file into the block containing the target filename on the original tape; that is, the target string is overwritten. (We can do this because we chose to have filenames and contents the same size.) The end result of this machine operating on its tape is the tape of Figure 3.19:

... Y BBA X AAB ABB X BAB BBA X B11 000 Y ...

U

Fig. 3.19 Output Tape from the Copy Machine

I will finish this section by giving you a couple more Turing problems.

Problem 3.9: Make a Turing machine which starts with a blank tape and ends up with all the binary numbers written on it in succession, separated by "*Y*'s", with the restriction that after you write the terminating *Y*, you never change the number again. An additional restriction you might impose is that the machine does not even look at terminated numbers.

Problem 3.10: Design a machine which recognizes only, and all, sequences of the form

$$1011001110001111 0000.....1^n0^n.$$

That the machine has "accepted" such a tape is indicated by its halting and leaving the tape blank after its machinations. More generally, we define an arbitrary sequence as "acceptable" by a Turing machine if the machine eventually halts with a blank tape. We can extend this notion to cover finite state machines. Design a Turing machine that accepts exactly the set of sequences accepted by any FSM. (Hint: use the FSM functions *F* and *G* to make Turing quintuples.)

3.6: Universal Turing Machines and the Halting Problem

Let us return to the reason why we are studying Turing machines. I said earlier that if you had an effective procedure for doing some computation, then that was equivalent to it being possible in principle to find a Turing machine to do the same computation. It is useful to talk in terms of functions. Suppose we start with a variable x, and we take a function of that variable, $F(x)$. We say that $F(x)$ is *Turing computable* if we can find a Turing machine $\mathbf{T_F}$ which, if fed a tape on which x is written, in some representation — binary, unary, whatever — will eventually halt with $F(x)$ printed on the tape. Every other effective procedure that anyone else has been able to cook up has turned out to be equivalent to this — the general recursive functions are Turing computable, and vice versa — so we can take "Turing computable" to be an effective synonym for "computable".

Now it may be the case that for some values of x, the Turing machine might not halt. This is weird behavior, but it might happen. Many functions — such as x^2 — are called "complete", meaning that for all values of x we plug into our machine, it will halt with the value of the function written on the tape. Functions for which this is not true are called "partial". In such cases, we have

to alter our operational definition of the function as follows: if, for a value x, the machine stops, we define the value of the function to be $F(X)$; if the machine does not stop, we *define* the value of the function to be *zero*. This does *not* mean that if we put x into F we get zero, in the way that putting $x = 3$ in the function $(x\text{-}3)$ gives us zero. Here, "zero" is just a useful label we attach to $F(x)$ when our Turing machine does not quit its computing. This redefined function is complete in the sense that we can assign *some* numerical value to it for *any* x.

A question naturally arises: can we say, in advance, which values of x might cause our machine to hang up? In some cases, the answer is yes. For example, there may be times when the machine goes into a recognizable infinite loop, perhaps shuttling between a couple of states and not achieving anything, and we can then say for sure that it will never stop. But in general, we cannot say in advance when a particular value of x is going to give us trouble! *Put another way, it is not possible to construct a computable function which predicts whether or not the machine T_F halts with input x.* In seeing why this is so, we shall appreciate the power of Mr. Turing's little machines.

I have flagged what is to follow in the penultimate sentence of the previous paragraph. I have raised the question of whether there is a computable function which will tell us whether or not $\mathbf{T_F}$ halts for input x. But, if there is such a function, by definition it must be describable by a Turing machine. This concept, of Turing machines telling us about other Turing machines, is central to the topic of *Universal* Turing machines to which we now turn.

We can pose the question we have set ourselves in the following way. Suppose we have a machine which we call **D**. As input, **D** takes a tape which contains information about $\mathbf{T_F}$ and $\mathbf{T_F}$'s initial tape (that is, information about X). Machine **D** is required to tell us whether $\mathbf{T_F}$ will halt or not: yes or no. Importantly, **D** must always write the answer and halt, itself. What we now do is introduce another machine **Z**, which reacts to the output from **D** in the following way:

*If T_F halts (**D** says "yes"), then **Z** does not.*
*If T_F does not halt (**D** says "no"), then **Z** does.*

We then get **Z** to operate on itself and find a contradiction! Let us expand on this argument.

To begin our quest for **D**, we first need to look at how we get one Turing machine to understand the workings of another. We need to characterize a given machine **T**, and its tape *t*; there are several ways of doing this. We will choose a description in terms of quintuples (Table 3.3):

Initial		Final		
State	Read	State	Write	Move
Q	S	Q′	S′	d (= L or R)

Table 3.3 Quintuple Description of a Turing Machine

We want to build a universal machine that is capable of imitating any **T**. In other words if we feed it information about **T** and about **T**'s tape *t*, our universal machine spits out the result of **T** acting on *t*. We will characterize our universal machine – call it **U** – in terms of quintuples in similar fashion to T. Let these quintuples for **U** be written (q,s; q', s', d') and note that they must suffice for all possible machines **T** that we want **U** to imitate: q,s, etc. must not depend on the specifics of **T**. A constraint we shall impose on our machines is that the tape symbols S, S', s, s' must be binary numbers. An arbitrary Turing machine **T** will come with an arbitrary set of possible symbols, but with thought you should be able to see that we can always label the distinct symbols by binary numbers and work with these (e.g. if we had 8 symbols, each could be redescribed by a three-bit binary string)[5].

The basic behavior of **U** is simple enough to describe. (Our discussion again closely follows Minsky [1967].) We need **U** to imitate **T** step by step, keeping a record of the state of **T**'s tape at each stage. It must note the state of **T** at each point, and by examining its simulated **T**-tape it can inform itself what **T** would read at any given stage. By looking at the description it has of **T**, **U** can find out what **T** is supposed to do next. Minsky nicely relates this process to what *you* would do when using a quintuple list and a tape to figure out what a Turing machine does. The universal Turing machine **U** is just a slower version of you!

[5]In fact, as an exercise, examine how you would reprogram a Turing machine **T** that operated with 2^n symbols to become a machine T′ operating on 0 and 1. Hint: where **T** had to read one

Let us supply **U** with the tape shown in Figure 3.20:

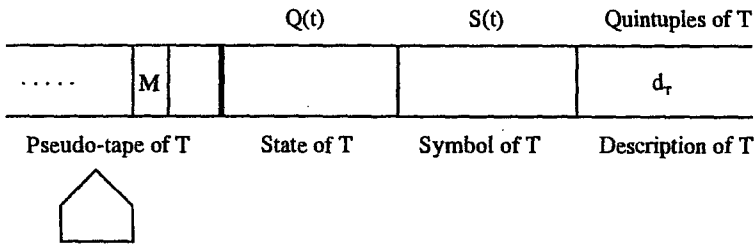


Fig. 3.20 Input Tape to the Universal Turing Machine

The infinite "pseudo-tape" on the left is **U**'s working space, where **U** keeps track of what **T**'s tape looks like at each stage of its simulation. Choosing to have it infinite only to the left is not essential, but simplifies things. The marker *M* tells **U** where the tape head of **T** currently is on *t*. To the right of this working space is a segment of tape containing the state of **T**; then, next right is a segment containing the symbol just read by **T**; and finally, to the right again, is a region containing the description of **T**. This description of **T**, which we denote as d_T, comprises a sequential listing of the quintuples of **T**, written as a binary sequence (Fig. 3.21):

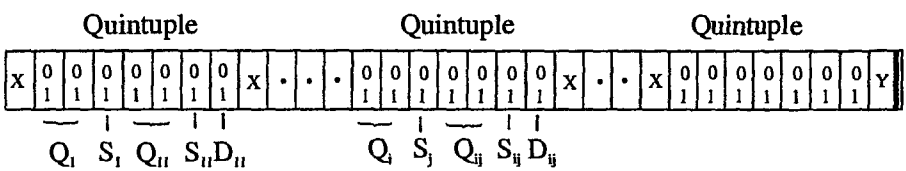


Fig. 3.21 The Description d_T of T for U's Tape

Each quintuple is segregated from the next by the symbol *X*. To start **U** off, we need to tell it **T**'s initial state Q_0 and the symbol S_0 it reads first. Let us assume that **U**'s tape head is initially over the leftmost *X* as shown in Figure 3.22:

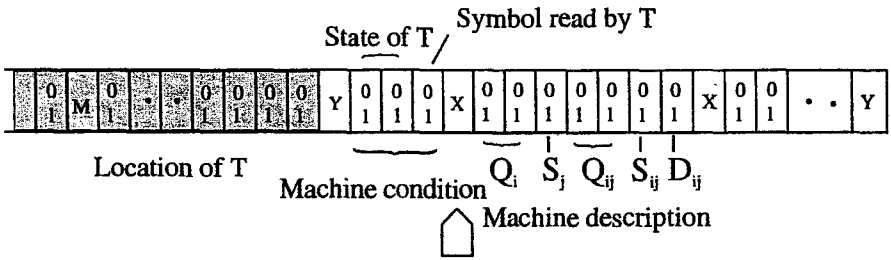


Fig. 3.22 Starting Position for the Tapehead of U

Essential to the operation of **U** are the locating and copying operations we described earlier. In general terms **U** operates as follows. First, **U** looks in the section of the tape describing the action of **T** for a given Q and S, exactly as we did with the locating machine: the set (Q,S) can be regarded as the filename of the file containing the relevant quintuple. As with the locator, on its way this operation changes all the 0's and 1's it encounters to A's and B's. After it finds the relevant pair (and changes them to A's and B's), it returns to the leftmost X.

The next stage involves the copy machine. **U** moves to the right until it hits the first set of 0's and 1's; because of the way we have set up the tape, these represent the three remaining parts of the quintuple specified by Q and S. These are the new state of **T**, the symbol it writes (on the pseudo-tape in position M) and its subsequent direction of motion. The machine then copies A's and B's representing both the new Q and new S into the machine condition region in the middle of the tape. It remembers the direction of motion d (L or R, represented as A or B). The machine now heads left until it reaches M. Once there, it erases M and temporarily overwrites it with the direction d (A or B). It then moves right, changing all A's and B's to 0's and 1's on the way (leaving an A or B in M's old location). Finally, it moves to the immediate left of the leftmost X, erases the symbol S that is there (but remembers it) and prints the special symbol V in its place (this is all that V is used for).

The machine now enters its final phase. It shifts left until it encounters the A or B that we stored in M; this represents the direction d in which **T** should next move. The machine overwrites the A or B with the S it has remembered, and then moves left or right depending on the instruction d. It reads the symbol of the square it is now on, remembers it and prints an M in its place. It then shifts right until it reaches the V, which it replaces with the remembered symbol. Now the sequence starts all over again.

What the machine has done is simulate one cycle of **T**'s operation: it has started off in a certain state Q and a given input symbol S; it has then changed state, written a new symbol and moved on to the next symbol dictated by **T**. **U** continues like this until it has mimicked **T** completely. Importantly, **U** has a halt state: it recognizes when **T** has halted, and proceeds to stop itself.

The description of **U** given above, due to Minsky, requires **U** to have 8 symbols and 23 states. So that you can appreciate the beauty of his machine, we reproduce it in full in Figure 3.23. You should not find it too hard to break it down into its constituent sub-machines.

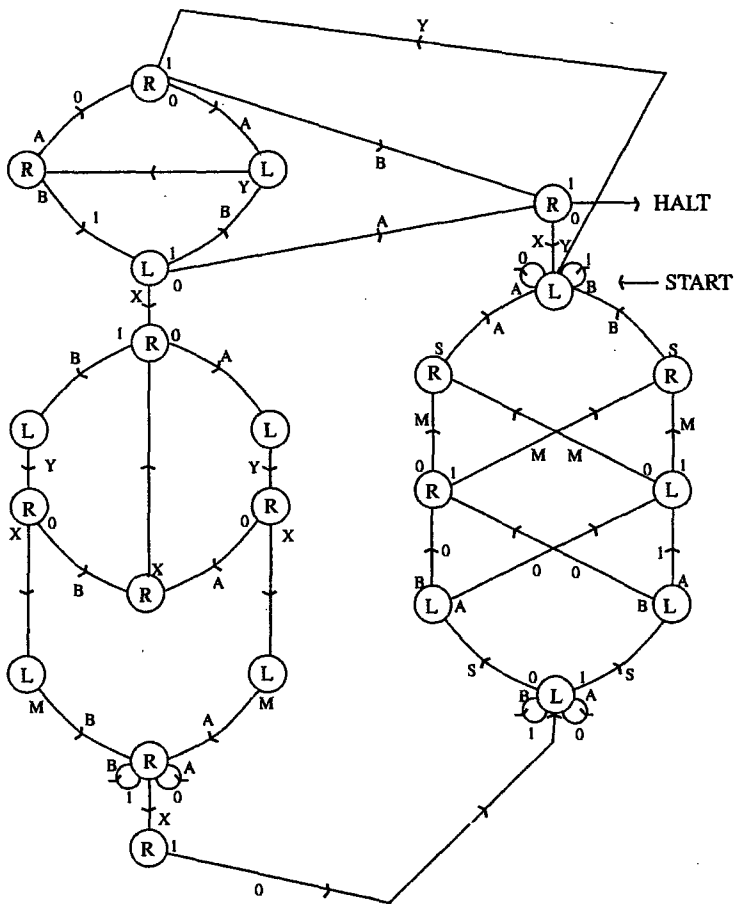


Fig. 3.23 A Universal Turing Machine

It is possible to build a UTM with the same number of symbols but just 6 states. If one wants to get tricky, there are ways of using the same state for more than one purpose, to minimize the number of states required. A UTM can be built with just two states and lots of symbols, or two symbols and lots of states. It is surprising that such a general purpose machine should require so few parts for its description; surely a machine that can do *everything* should be enormously complicated? The surprising answer is that it's not! How efficient one can make a UTM is an entertaining question, but has no deep significance.

Let us now turn to the real reason why we have been interested in demonstrating the existence of a UTM. We have asked whether it is possible to build a machine that will tell us whether a Turing machine **T** with tape *t* will halt, for all **T** and *t*. We can clearly rephrase this as a halting problem for a universal machine **U**. Let us define a new machine **D**, which is just **U** with the added property that it tells us whether or not **T** halts with tape *t*, and that it can do this for all machines **T** and all tapes *t* (Fig. 3.24):

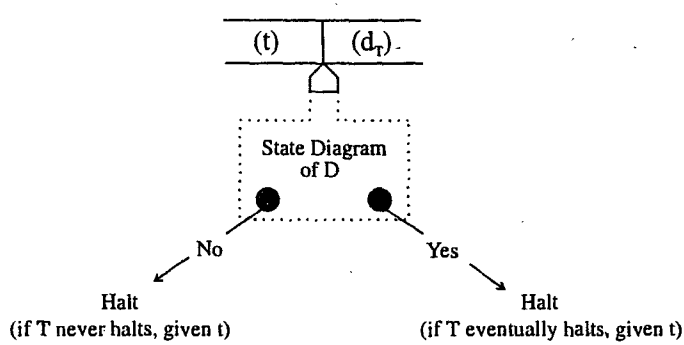


Fig. 3.24 Universal Machine D with tape t and d_T

In other words, **D** *always* halts with an answer. Can such a machine exist? The answer is no! We can actually show that **D** is an impossible dream, and we do this by picking a machine **T** and a tape *t*, for which **D** cannot do what it is supposed to.

Information about **T** and *t* are fed into a universal machine in the form d_T, the quintuple description of **T**, and the information on the tape *t* (see Fig. 3.24). Now for no apparent reason, let us see what happens if we let the tape *t* contain the description d_T. We now enhance our machine **D** slightly and introduce another machine **E**. This new machine only requires as input a tape containing d_T; it then copies d_T onto a blank part of the tape and now behaves like machine

D with an input tape containing $t = d_T$ and d_T. **E** will now behave the same way as **D**, and halt giving the answer "yes" if **T** halts when reading its own description: otherwise, **E** will answer "no" (Fig. 3.25). Whatever the case, **E** always halts.

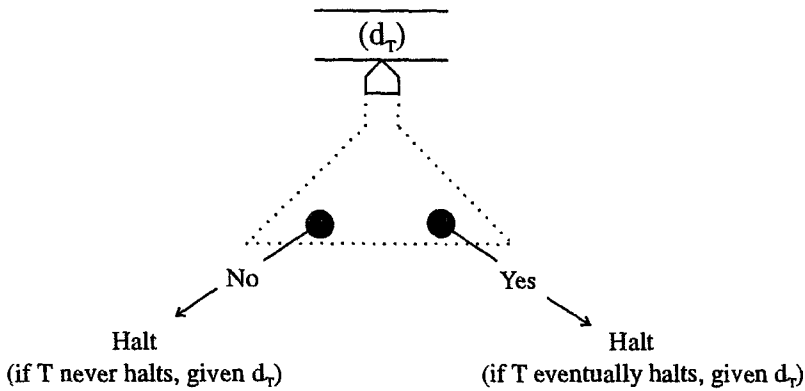


Fig. 3.25 Universal Machine E with input tape d_T

Now we introduce a modified version of **E** which we shall call **Z**. Our new machine **Z** has two extra states that are used to prevent **Z** from halting if **E** takes the "yes" route (Fig. 3.26):

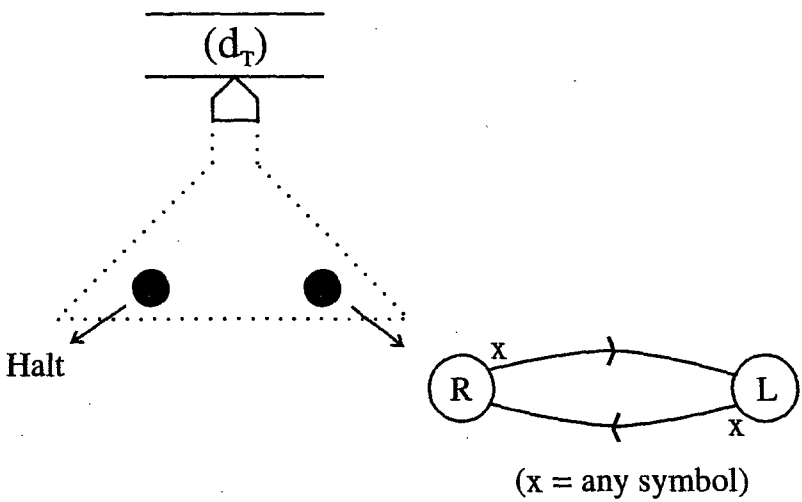


Fig. 3.26 Universal Machine Z

Thus **Z** has the property that, if **E** spits out the answer "yes", it does not halt; whereas if **E** spits out "no", it also gives us a "no" and does halt (i.e. **Z** = **E** in this case). So, **Z** halts when we feed it d_T, if **T** applied to d_T does *not* halt, but does not halt if **T** applied to d_T *does*. Now comes the crucial step. Let us write a description d_Z for **Z**, and substitute **Z** for **T** in the foregoing argument. We then deduce that:

Z applied to d_Z halts if and only if **Z** applied to d_Z does not halt.

This is a clear contradiction! Going back through our argument, we find that it is our assumption that **D** exists that is wrong. *So there are some computational problems (e.g. determining whether a UTM will halt) that cannot be solved by any Turing machine.* This is Turing's main result.

3.7 Computability

There must be many uncomputable functions. How many are there? We can gain some insight into this by considering a counting argument. Consider computable real numbers: by which we mean those whose binary expansions can be printed on a tape, whether the machine halts or not. We can show that there are many more real numbers than computable real numbers since the latter are countable, while the former are not. We call a set "countable" if we can put its elements in one-to-one correspondence with elements of the set of positive integers; that is, if we can label each set member by a unique integer. Two examples of countable sets are the even and rational numbers:

Even numbers	0	2	4	6	8	10....	
	0	1	2	3	4	5	
Rational numbers	(1/2)	(1/3	2/3)	(1/4	2/4	3/4)	....
	1	2	3	4	5	6	

The real numbers, however, are not countable. We can supply a neat proof of this as follows. Let us suppose the opposite. Then we would be able to pair off the reals with the integers in some way, say as follows:

Integer	Real
1	0.124
2	0.015
3	0.53692
4	0.8003444
5	0.334105011
6	0.3425.....

The exact assignment of real numbers to integers, and we have chosen a weird one here, is arbitrary; as long as we have one real number per integer, and all the reals are accounted for, we are OK. However, this cannot be so! To see why, we will find a real number that cannot be on our list. In the above list, I have underlined certain digits: the first digit of the first number, the second digit of the second, the third of the third, and so on. We define a new number using these: all we require is that the nth digit of this number *differs* from the nth digit in our list. The real number:

0.22741....

going on forever, is just such a number. We have obtained this by adding one to each of the underlined digits. (We can include the rule "9+1=0" to make this a consistent procedure or we can use other procedures entirely to generate new real numbers.) What have we achieved? By construction, the above number differs from the mth number in our correspondence list in its *m*th digit, and this is true for all m — that is, for all integers. Hence, we have found a real number that cannot appear on our list. So by "diagonalization" as it is called (referring to the "diagonal" line we can draw through all of the underlined numbers above) we have shown that the real numbers are not countable.

Turing machines, however, *are* countable. To see this, consider the tape description d_T of a machine **T**. We can consider this to be a string of binary symbols unique to the machine if we ignore the spacings between quintuple listings. The resulting binary number serves to uniquely label the machine by an element of the set of integers. On the other hand, if we define a function $f(n)$ to be 1 if the nth Turing machine halts and 0 otherwise, then clearly this function is not computable, as we have seen from the Halting Problem. There are many other examples.

Let us return to the subject of effective procedures and make a few comments. Although we have tended to portray effective procedures as

algorithms that enable us to calculate things, in reality many such procedures are of little practical use — they might require too much tape for their execution, for example, or some other extravagant use of resources. A procedure might take the age of the Universe to complete yet still be technically "effective". In practice we want procedures that are not just effective but also efficient. The word "efficient", of course, is not easy to define precisely and so we end up leaving the clean and unambiguous world of logic and entering that of the real world of the comparatively dirty and vague — or exciting and interesting — depending on your viewpoint! Many problems in "artificial intelligence", such as face recognition, involve effective procedures that are not efficient — and in some cases, they are not even very effective!

Sometimes we do not strictly need effective procedures at all. It might be the case, for example, that you can ask a question and, while I cannot give you a sure answer, I can answer it with a probability of correctness of $(1 - 10^{-20})$. You might be quite happy with such good odds. There is nothing particularly bad about uncertainty. An obvious, and rather uninteresting, example of this would be if you asked me whether a given number x was divisible by some other number y. I could simply say "no", and if y is big enough, the odds are in my favor that I am right: to be precise, the odds are 1 in y that a randomly chosen number is divisible by y. The principle here is that you can know a lot more than you can prove! Unfortunately, it is also possible to think you know a lot more than you actually know. Hence the frequent need for proof.

A related, but more interesting problem, is the question of whether or not a given number n is prime. An effective procedure for this might involve taking all prime numbers up to $n^{1/2}$ and seeing if any divide n; if not, n is prime. This is fine, and rather neat, for small n, but when we get to the big numbers it becomes impractical. A better test is a probabilistic one. This uses one of Fermat's famous theorems:

$$a^p = a \bmod p \tag{3.4}$$

What this means is that, for any number a and prime p, if we divide a^p by p, we get the remainder a. So for example, we write:

$$3^5 = 243 = (48 \times 5) + 3$$

The idea behind the method is to take a large value of a, and calculate a mod

p. For large p, the odds are good that p is not a prime and that this quantity does not equal a since there are so many possible remainders. (The actual odds are not simple to calculate, but you get the idea.) However, if p is huge — something of the order of 10^{200}, say — how do we calculate a^p? Well, we don't actually need this number: we only need the remainder after division by p. Why this is so I will leave as an exercise for you! (Don't worry about the general case: do it for a nine-digit p.)

Another similar problem deals with factorization: I give you a number m, and tell you that it is the product of two primes, $m=pq$. You have to find p and q. This problem does not have an effective procedure as yet, and in fact forms the basis of a coding system. It is possible to build our ignorance of the general solution of this mathematical problem into ciphering a message. The moment some clever guy cracks it — and people have gotten up to 72 digit m's so far — the code is useless, and we'd better find another one.

Before leaving the subject of computability, I want to make some remarks about the related topic of "grammars". In mathematics, as in linguistics, a grammar is basically a set of rules for combining the elements of a language, only the language is a mathematical one (such as arithmetic or algebra). It is possible to misapply these rules. Consider the following statements:

$$(a + b)\ c \qquad\qquad a + b(\ c$$

Within the context of arithmetic, only the first of these makes sense. The second, however, does not: the parenthesis is wrongly, even meaninglessly, placed. An interesting general question in computing is whether we can build machines that will test mathematical (and other) expressions for their grammatical correctness. We have seen one example: the parenthesis checker. This checked a very simple grammar involving) and (and the only grammatical rule was that strings of parentheses balanced. But remember it took a Turing machine to do this: a finite state machine was not up to it. Now there are certain classes of grammar that FSMs can check — for example, strings of ones, 1111111.... where valid strings have to have even numbers of digits, for example — but the abilities of this type of machine are limited. We can actually draw up a table relating types of grammars to the machines required for their analysis (Table 3.4):

Language	**Description**	**Example**	**Machine required**
Finite enumerable	A list of acceptable expressions	*ab*, *abc*	Memory (table look-up)
Regular language	Regular expressions built with *, ∨, ∧, ( )	ab^*c, *=any no. of repetitions, incl. none. $a(b \vee d)^*c$	Finite state machines (a theorem)
Context free	Language generated by production rules which admit recursion	$a^n b^n$ (not $a^n b^m$ where $n \neq m$)	An in-between machine: a push-down automaton. Has one "stack" inside — a pile of paper with a spring underneath, can only take off the top one
General recursive	Computable functions	$a^n b^m c^q$	Turing

Table 3.4 General Grammars and Their Machine Implementation

It is sad that Turing machines are so easy to make that we have to leap over all this pretty theory. Nevertheless, in the design of compilers (which involve the interpretation of languages) the use of such theory is so fundamental that you might find further study of it worthwhile.

We will finish our look at computability with an interesting problem discovered by Post as a graduate student in 1921. Consider a binary string, say 10010. It is arbitrary. Given the string, play with it according to the following rules: read the first three digits; if the first is 0, delete all three and add 00 to the end of the string; if the first is 1, delete all three and add 1101 to the end. So with our string we would have

```
10010
--- 101101
  ---    1011101

         -----
```

The question is this: does this process go on forever, stop, or go on periodically? The last I heard, all tested sequences had either stopped or gone into a loop, but that this should be so generally had not been proved. It is an interesting issue because it has been shown that a so-called "Post machine" — one which takes a string g and writes a result $h(g)$ depending on the first digit g_i of the string — can act as a Universal machine and do anything a Turing machine can do!

FOUR

CODING AND INFORMATION THEORY

In this chapter we move from abstract considerations of computation to the more concrete realities of computer structure. I want to examine the limitations on machines resulting from the *unreliability* of their component parts. A typical machine may be built from millions of logic gates and other bits and pieces and if these components have a tendency to malfunction in some way, the operation of the machine could be seriously affected.

Components can let us down chiefly in two ways. Firstly, they may contain faults: these can arise during manufacture and are obviously of extreme importance. For example, when making a memory chip from silicon, flaws can be anywhere — where there was a bit of dirt in the material, or where the machine making it made a mistake — and the smallest fault can screw up an element of memory. If your memory system is such that all the cells have to work or the whole thing is useless, then just one tiny mistake can be very costly. A neat way to resolve this problem is to design systems which work around such flaws, spotting them and, perhaps, sealing them off from further usage. However, I will look at the physical structure of components later.

What I want to focus on now is a second way in which an element can let us down. This is when it fails *randomly*, once in a while, perhaps because of the random Brownian motion of atoms or just irregular noise in a circuit. Any such glitch can cause a component to fail, either temporarily or permanently. Now the odds against a particular element failing in such a way may be a million to one, but if we have billions of such elements in our machine, we will have thousands failing, all over the place, at any one time. When the earliest Von Neumann machines were in operation, they were constructed from relays and vacuum tubes whose failure rate was very high (of the order of one in a thousand), and the problem of unreliability was acute: with a million components one could expect a thousand of them to be acting up at any one time! Now it has turned out that as we have developed better and better systems with transistors, the failure rate has been going down for almost every machine we build. Indeed, until recently, the problem has ceased to be considered very serious. But as we manufacture computers with more and more parts, and get them to work faster and faster, and particularly as we miniaturize things more and more, this might not remain true. There are something like 10^{11} atoms in a modern transistor, but if we try to get this number down, to build switching devices with, say, a thousand atoms, the importance of noise and random failure

examine the matter of unreliability in some detail. Besides, it is an interesting subject, and that should be reason enough to study it!

4.1: Computing and Communication Theory

We begin our discussion of unreliability by considering the aspect of computers for which it is most problematic, that of memory storage. For example, suppose that we have some data stored somewhere for a long time, and at some point the system makes a mistake and switches a bit somewhere — a one gets changed to a zero, for example. This sort of error can occur elsewhere in the machine, in its CPU for example, but this is less likely than it happening in memory, where the number of transistors and elements is so much larger. To examine this situation I am going to draw a useful analogy with another area of engineering; namely, with *communication theory*. In a communication system you send out a bunch of bits at one end, the transmitter, and at the other end, the receiver, you take them in. This is just sending a message. In the process errors can creep in: noise could affect the message in transit, reception could be bad, we might get glitches. Any of these could mean that the message we receive differs from the one sent: this is the so-called "communication problem in the presence of noise". Now this isn't exactly the same situation as with memory — which is like sending a message through time rather than space — but you can see the similarities. We store something in memory and at a later time we read it back out — in the interim the stored "message" is subject to noise. When it comes to the reliability of stored memory and sent messages there are important practical differences. It is possible in principle, for example, to continually check on the contents of our memory, whereas, if NASA sends a radio communication to a Jupiter probe there is no way of checking its contents while in transit. Nevertheless, the analogy is strong enough to make a look at communication theory worthwhile. We will start with a look at how we might go about detecting errors, an essential step before we can correct them.

4.2: Error Detecting and Correcting Codes

From now on I am going to use the language of communications, and will generally leave it to the reader to make the connection with memory systems in machines. Let us suppose we have a transmitted message, which we take to be some sequence of symbols[1], and we are going to be doing the receiving.

[1]From here on, we restrict these symbols to be binary digits, 1s and 0s. [RPF]

Obviously, we would like to know how trustworthy the received message is, and this brings us to our first matter, that of error detection. Is there some way in which we could know whether the message we have received is correct? Clearly, all we have to work with is the message: calling up the sender for confirmation defeats the object! Is there some way of building a check into the message itself, that will enable us to confirm it? The answer is yes, as we will shortly see.

4.2.1: Parity Checking

We first assume that the probability of an error arising in the message is very small; in fact, so small that we never have to worry about more than one error turning up. Furthermore, we will only consider errors in individual bits and not, for example, errors spread out across several neighboring, or related bits (such as "error bursts" caused by scratches on disks). Suppose the chance of an error in a symbol is one in ten thousand, and our message is ten symbols long. The chances of an error in the message are about one in a thousand. However, the odds against two errors are of the order of a million to one, and we shall consider this negligible. We will only bother trying to detect single errors, assuming doubles are too rare to worry about.

Here is a very simple scheme for checking for single errors, known as a *parity checking* scheme. Suppose we are sending the following ten-bit message:

1101011001

What we do is tag another bit onto the end of this string, which tells us the parity of the string – the number of 1's it contains, or, the same thing, the sum of its digits modulo 2. In other words, the extra digit is a 1 if the original message has an odd number of 1's: otherwise, the above message would have a 0 attached. This is an example of *coding* a message; that is, amending its basic structure in some way to allow for the possibility of error correction (or, of course, for reasons of security). When we receive the message, we look at the parity digit and if the number of 1's is wrong, then we have obviously received a faulty message. Note that this simple check actually enables us to detect any odd number of errors, but not any even number (although as we have ascribed vanishing probability to anything more than a single error, these are assumed not to occur)[2].

[2]Note that a simple machine that could do this checking would be the parity FSM discussed in the previous chapter. [RPF]

There are two main shortcomings of this procedure which we should address. Firstly, we might get an error in the parity bit! We would then be mistaken if we thought the message itself was wrong. Clearly, the longer the message, the less likely the error is to occur in the parity bit itself. Secondly, at best the check only tells us whether an error exists — it does not tell us *where* that error might be. All we can do on finding a mistake is have the message sent again. In our case, where we are using a computer, we might simply reboot the machine and go back to square one. Another minor shortcoming of this particular approach is that it leads to a certain inefficiency of communication — in this case, a ten percent inefficiency, as we had to send eleven bits to communicate a message of ten.

The obvious next question to ask is: can we construct a method for not only detecting the existence of an error, but actually locating it? Again the answer is yes, and the method is quite ingenious. It is a generalization of the simple parity check we have just examined. What we do is imagine that the data in the message can be arranged into a rectangular array, of (say) m rows and n columns (Table 4.1):

$$
\begin{array}{c|ccccc}
 & & & n & & \\
\hline
 & 1 & 1 & 0 & 1 \ldots\ldots & 0\ 1 \\
 & 0 & 0 & 0 & 1 \ldots\ldots & 1\ 1 \\
m & \ldots & & \ldots\ldots & & \ldots \\
 & \ldots & & \ldots\ldots & & \ldots \\
 & 1 & 1 & 0 & 0 \ldots\ldots & 0\ 1
\end{array}
$$

Table 4.1 A Rectangular Data Array

Of course, the data would not be sent in this form: it would be sent as a binary sequence and then arranged according to some predefined rule, such as breaking the message into m blocks of n symbols and placing them one over the other. To check for errors, what we do is include at the end of each row a digit giving the parity of the row, and at the base of each column a parity digit for the column. These parity digits can be seeded into a sequential message without difficulty. An error in the array will then lead to a parity mismatch in both the row and the column in which the error appears, enabling us to pinpoint it precisely. In principle this scheme can detect any number of message errors, as long as they occur in different rows and columns.

We have to be careful, however, about errors occuring among the parity check bits. A particularly nasty instance would be a double error where a message digit and the parity digit for the row (say) both switch. We would know that there was an error, due to the column parity being wrong, but we might be inclined to think that it was a single column parity bit that was at fault — as we would have no confirming row parity error. However, we can safeguard against this ambiguity by placing another parity check in the array, this time at its lower right corner. This bit gives us the parity of the whole message (i.e. of the row and column totals), and using it we can detect — but not locate — such a double error. The end result of this double error detection may well be the same as with our single error detector — we go back to square one and send the message again — but it is still an improvement.

A useful way to quantify the efficiency of a coding method like this is by calculation of a quantity called the *redundancy R*:

$$R = \frac{\textit{no. of bits used in full}}{\textit{no. of bits in message}} \tag{4.1}$$

The bigger R is, the less efficient our code. The quantity $R-1$ is usually known as the "excess redundancy". For our first, single-error-detecting code, the redundancy is $(n+1)/n$. For the rectangular array above we are using mn bits to send a message that is only $(m-1)(n-1)$ bits long. So we have the following result:

$$R = \frac{mn}{(m-1)(n-1)} \tag{4.2}$$

This quantity is a minimum, and hence the code most efficient, when the array is a square, i.e. $m=n$. You might be tempted to say, "Well, I can get the redundancy down to near one by just taking m and n very large — let's just send our message in blocks and rows with not ten, but ten-thousand bits!" The problem with this is that there is a certain probability of each bit being in error, and if the number you are sending gets too big the chance of multiple errors begins to creep up.

4.2.2: Hamming Codes

I will now take a look at another single-error-correcting (and double-error-

detecting) coding method based on parity-checking, which is both more efficient and a lot more subtle than the rectangular type. Actually, it is a kind of higher-dimensional generalization of the array method. In any message we send, some of the bits will be defined by the message itself, and the rest will be coding symbols — parity bits and the like. For any given message, we can ask the question: "How many check bits do I need to not only *spot* a single error, but also to *correct* it?" One clever answer to this question was discovered by Hamming, whose basic idea was as follows. The message is broken down into a number of subsets of digits, which are not independent, over each of which we run a parity check. The presence of an error will result in some of these checks failing. We use a well-defined rule to construct a binary number, called the "syndrome", which is dependent on the outcome of the parity checks in some way. If the syndrome is zero, meaning all parity checks pass, there is no error; if it is non-zero, there is an error, and furthermore the *value of the syndrome tells us the precise location of this error.* For example, if the syndrome reads 101, that is decimal 5, then the error is in bit five of the message. If on the other hand it reads 110010, then the error is in the fiftieth bit. The trick is to implement this idea.

We can straightaway make some statements about how many check bits we will need. Suppose our syndrome is m bits long so that we have m check bits. If we decide that a vanishing syndrome is to represent no error, that leaves at most (2^m-1) message error positions that can be coded. However, errors can occur in the syndrome as well as the original message we are sending. Hence, if n is the length of the original message, we must have:

$$2^m - 1 \geq (n + m) \tag{4.3}$$

or

$$n \leq 2^m - m - 1. \tag{4.4}$$

For example, if we wanted to send a message 11 bits in length, we would have to include a syndrome of at least four bits, making the full message fifteen bits long. This does not seem particularly efficient (efficiency = 11/15 or about 70%). However, if the original message was, say, 1000 bits long, we would only need ten bits in our syndrome (2^{10}= 1024) which is a considerable improvement!

Let us now see precisely how this syndrome idea works. As an example,

let us continue with our problem of sending a message eleven bits long. As we saw, we will need four check bits. Each such bit will be a parity check run over a subset of the bits in the full fifteen-bit message. Just as with the simple parity check method, we will select a few specific bits in the message, calculate their overall parity, and adjust the corresponding check bit to make the total parity of the (subset plus check) zero. If there is an error in this subset, the parity check will fail. The clever thing about the Hamming code is that each message bit is in more than one subset and hence contributes to more than one parity check, but not to all of them. By seeing which parity checks fail and which pass, we can home in on the error uniquely. We assign to each parity check a one if it fails and a zero if it passes, and arrange the resulting bits into a binary number, the syndrome. This indicates directly the error position. It is pretty much arbitrary, but we will construct the syndrome by reading the parity checks from left to right in the message.

For the moment, we will assume that the parity check bits are placed in some order throughout the message, although we will not mind where for the moment. We will first identify the subsets each covers. To do this, it will help to list the four-digit binary representations of the positions within the message:

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

Let us look at the rightmost parity check, the far right digit of the syndrome. Suppose this is non-zero. Then there will be a parity failure in a position whose binary representation ends in a one: that is, one of positions 1, 3, 5, 7, 9, 11, 13 or 15. This is our first subset. To get the second, look at the second digit from the right. This can only be non-zero for numbers 2, 3, 6, 7, 10, 11, 14 and 15.

Note what is happening. Suppose we have found that both of these parity checks failed, i.e., we assign a 1 to each. The error must be in a position that is common to both sets, i.e. a binary number of the form *ab*11. This can only be 3, 7, 11 or 15; we have narrowed the possible location choices. To find out which, we have to do the remaining parity checks. The third check runs over digits 4-7, and 12-15. The final check covers digits 8 through 15. Suppose both of these are zero, that is, the parity checks out; we hence put two zeroes in our syndrome. Then there is no error in positions 4-7, 12-15, 8-15 (which obviously overlap). But there is an error somewhere in 3, 7, 11, 15. The only one of these latter four that is not excluded from the previous sets is position 3. That must be where the error lies. Of course, in binary 3 is 0011 — the syndrome calculated from the parity checks.

Let us pick a real example to illustrate these ideas in a more concrete manner. Suppose we want to send the eleven-bit message 10111011011. We first have to decide where to stick in our parity bits. There is nothing in what we have said so far that tells us whereabouts in the message these must go, and in fact we can put them anywhere. However, certain positions make the encoding easier than others, and we will use one of these. Specifically, we place our check bits at positions 1, 2, 4 and 8. We now have:

Codeword	*a*	b	1	*c*	0	1	1	*d*	1	0	1	1	0	1	1
Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Importantly, the check bits here, read from left to right, will give the *reverse* of the syndrome: that is, the first leftmost digit of the syndrome, when written out, would be read from the parity check of *d*, not *a*, and the last rightmost read from *a*, not *d*. Again, this is a matter of calculational simplicity.

We can now work out *a*, *b*, *c* and *d*. Bit *a* is the parity of the odd positions: 1, 3, 5, 7, 9... This is 1. Hence, *a* is 1. Bit *b* is found by summing the parities of positions 2, 3, 6, 7, 10, 11, 14 and 15. This gives zero. Bit *c* comes from the parity of positions 4 through 7 and 12 through 15, giving c=1. Finally, we get *d* by doing a check on 8 through 15, giving d=1. Note, incidentally, how this placing of the check bits leads to straightforward encoding, that is, calculation of *abcd*. If we had chosen an apparently more straightforward option, such as placing them all at the left end of the string occupying positions 1 to 4, we would have had to deal with a set of simultaneous equations in *a*,*b*,*c*,*d*. The important feature of our choice is that the parity positions 1, 2, 4, 8 each appear in only one subset, giving four independent equations. The completed message

is thus:

101101111011011.

Let us see what happens when an error occurs in transmission. Suppose we receive the message:

101101011011011.

Where is the mistake? The sum over the odd-placed digits is 1 — a failure. We assign a 1 to this in our syndrome, this being the rightmost digit. The second parity check also gives 1, another failure, and our syndrome is now $xy11$, with x and y to be determined. Bit y comes from our next check, and is 1 again — another failure! The syndrome is now $x111$. To find x we check the parity of places 8 through fifteen, and find that this is zero — a pass. We therefore assign a zero to x, giving our syndrome as 0111. This represents position 7, and indeed if you compare the original and corrupted messages, position 7 is the only place you will find any difference.

An interesting feature of the Hamming code is that the message and code bits are on the same footing — an error in a code bit is located the same way as in a message bit. We can extend this code to detect double errors quite simply. We tag on to the end of the message yet another check bit, this time representing the parity of the whole thing. For the (uncorrupted) message we gave above, the parity is 1, so we attach a 1 at the rightmost end, to give us zero overall parity as usual. Now, if there is a single error in the message, the parity of the 15-bit message will change, and this will show up as a mismatch with the sixteenth bit. However, if there is a double error, the parity of the 15-bit message will not change, and all will look normal in the sixteenth bit; yet the parity checks within the former will fail, and this indicates a double error. Observe that if the 15-bits check out, but the overall parity does not, this indicates an error in the overall check bit. Note that the cost of these benefits is almost a 50% inefficiency — five check bits for an eleven bit message. As we pointed out earlier however, the inefficiency drops considerably as we increase the message length. For a one-thousand bit message, the inefficiency is a tiny one percent or so.

It is worth examining the practical usefulness of error-detecting codes like this by looking at how the consequences of message failure become less drastic for quite small losses of efficiency. Let us suppose that we are sending a message in separately coded batches of about a thousand bits apiece, and the

probability of an error in a single one of the bits is 10^{-6}, or 10^{-3} per batch. We take these errors to be random and independent of each other. We can use Poisson's Law to get a handle on the probabilities of multiple errors occurring when we send our batches. If the mean number of errors expected is m, then the actual probability of k errors occurring is given by:

$$\frac{1}{k!} m^k \exp(-m). \tag{4.5}$$

The expected number of errors per batch is, as we have said, 10^{-3}. Hence, the probability of a double error in a batch is $(1/2).10^{-6}$ to high accuracy (we can ignore the exponential), and the probability of a triple error is $(1/6).10^{-9}$. Now suppose we have no error detection or correction, so we are expending no cash on insurance. If an error occurs, we get a dud message; the system fails. On average, we should only be able to send a thousand batches before this happens, which is pretty miserable. Suppose now that we have single error detection, but not correction, say a simple parity check. Now, when the system detects an error, it at least stops and tells me, and I can try to do something about it. This still happens once in every thousand or so batches, but it is an improvement that we gain at the cost of just one-tenth of a percent message inefficiency — one parity bit per thousand message bits. This system will fail whenever there are two errors, which occurs roughly once every two million batches. However, suppose we have our one percent gadget, with single error correction and double error detection. This system will take care of it itself for single errors, and only stop and let me know there is a problem if a double error occurs, once in every million or so batches. It will only fail with a triple error, which turns up in something like every six billion batches. Not a bad rate for a one percent investment!

Issues of efficiency and reliability are understandably central in computer engineering. An obvious question to ask is: "How long should our messages be?" The longer the better as regards efficiency of coding, but the more likely an error is to occur, and the longer we will have to wait to find out if an error has occurred. On the other hand, we might be prepared to sacrifice efficiency for security, sending heavily coded but brief messages so that we can examine them and regularly feel that we can trust what we receive. An example of the latter kind that is worth noting is in the field of communications with deep solar system spacecraft such as the *Voyager* series. When your spacecraft costs billions of dollars, and you have to send radio messages across millions of miles and be as certain as possible that it arrives uncorrupted — we don't want the

cameras pointing at the Sun when they should be looking at Jupiter or Saturn — efficiency goes out the window. Spacecraft communications rely on a kind of voting technique, referred to as "majority logic decisions". Here, each bit in the source message is sent an odd number of times, the idea being that most of these will arrive unchanged — i.e. correct — at the receiver. The receiver takes as the message bit whichever bit appears most in each bit-batch, in best democratic fashion. How many copies you send depends on the expected error rate. Anyway, this is just a little example from communication, so I won't dwell on it. It doesn't seem to have too much relevance for computing (except perhaps for those in the habit of backing up their files a dozen times).

4.2.3: An Aside On Memory

Let me briefly discuss one interesting way in which the Hamming coding technique can be applied to computer memory systems. With the advent of parallel processing, it has become necessary to load and download information at an increasingly rapid rate, as multiple machines gobble it up and spit it out faster and faster. This information is still stored on disks, but individual disks are simply not fast enough to handle the required influx and outflow of data. Consequently, it is common practice to use "gang-disk" systems, where lots of disks share the load, simultaneously taking in and spewing out data. Such systems are obviously sensitive to errors on individual disks: if just one disk screws up, the efforts of the whole bunch can be wasted. Every manufacturer would like to build the perfect disk, one that is error-free — sadly, this is impossible. In fact, it is fair to say that the probability of any given floppy, and certainly any hard disk, on the market being free of flaws (e.g. bits of dirt, scratches) is virtually nil[3]. The reason you do not usually notice this is that machines are designed to spot flaws and work around them: if a computer locates a bad sector on a disk it will typically seal it off and go on to the next good one. This all happens so fast that we don't notice it. However, when a disk is working alongside many others in a parallel processing environment, the momentary hang up as one disk attends to a flaw can screw everything up.

The Hamming method can come to our rescue. Let's suppose we have thirty-two disks working together. We take twenty-six of these to be loaded with information, and six to be fake. We only have twenty-six worth of data, but thirty-two lines coming into the system. In each click of the clock we get one

[3]Readers might like to contact disk manufacturers and try to get some figures on the flaw rate on their products. You will have a hard time getting anywhere! [RPF]

bit fed in from each disk. What we do is run a parity check for each input of thirty-two bits coming into the system, one per disk, according to Hamming's method — hence six parity bits for twenty-six of message — and correct the single errors as they come along. Note that in this sort of set-up the odds against double errors occurring are enormous; that would need two or more disks to have errors in the same disk locations. It is a possibility, of course, but even if it happens we can soup up our system to detect these double errors and have it grind to a halt temporarily so that we can fix things. The flaws we are talking about here, in any case, are not really random: they are permanent, fixed on the disk, and hence will turn up in the same place whenever the disk is operating. We can avoid double errors of this kind by running the system, debugging it, and throwing away any disks that have coincidentally the same error spots. We then buy new ones. We do this until no more double errors are found. The use of this Hamming coding method saved the whole idea of gang disks from going down the drain.

Here are some problems for you to look at.

Problem 4.1: Devise a Hamming-type code for a message alphabet with a number (a) of elements (for binary, a=2). Show that, if the number of code symbols is r, and the total message length is N (so the original message is $N-r$) we must have

$$1 + (a-1)N = a^r$$

Work out a simple example.

Problem 4.2: This is an interesting mathematical, but not overly practical, exercise. We define a *perfect* code to be one for which:
(a) each received coded message can be decoded into some purported message, and
(b) the purported message is correct if there are less than a specific number of errors.

Codes can correct up to e errors (thus far we have only considered e=1). Can you construct a perfect code for binary symbols (a=2) with e=3, i.e. triple error correction? Hint: stick to N=23, with 12 bits of message and 11 of code (or syndrome). There is also a solution to this problem for a tertiary alphabet (a=3) For this case, try N=11, with 6 data digits and 5 of code.

4.3: Shannon's Theorem

We have asked lots of fundamental questions so far in this book. Now it is time to ask another. In principle, how far can we go with error correction? Could we make a code that corrects two, three, four, five, six, seven... errors and so on, up to the point where the error rate is so low that there is no point in going any further? Let's set the acceptable chance of us getting a failure at 10^{-30}. If you don't like that, you can try 10^{-100} : any number will do, but it must be non-zero, or you'll get into trouble with what follows. I reckon 10^{-30} will do.

Suppose we're sending a message of length M_C, which is the length of the full coded message, containing original data and coding bits. As usual, we're working in binary. The length of the data message we call M. Let's assume that the probability of any single bit going wrong is q. We want to design a coding scheme that corrects single, double, triple errors and so on, until the chances of getting more errors in M_C is less than our chosen number, 10^{-30}. How many code bits are we going to need? How much of M_C is going to be message? What's left?

Claude Shannon has shown that the following inequality holds for M and M_C:

$$M/M_C \leq f(q) = 1 - q(\log_2[1/q] + [1-q]\log_2[1/(1-q)]) \qquad (4.6)$$

Given this, says Shannon, if no limit is placed on the length of the batches M_C, *the residual error rate can be made arbitrarily close to zero.* In other words, yes, we can construct a code to correct n-tuple errors to any accuracy we choose. The only restriction in principle is the inequality (4.6). In practice, however, it might require a large batch size; and a lot of ingenuity. However, in his extraordinarily powerful theorem, Shannon has told us we can do it. Unfortunately, he hasn't told us *how* to do it. That's a different matter!

We can construct a table with a few values for q to illustrate the upper limit Shannon has placed on coding efficiencies (Table 4.2):

q	M/M_c	M_c/M
1/2	0	∞
1/3	0.082	12.2
1/4	0.19	5.3
0.1	0.53	1.9
0.01	0.919	1.09
0.001	0.988	1.012

Table 4.2 Shannon's Coding Efficiency Limits

Note that if q is 0.5 — that is, there is a fifty-fifty chance that any bit we receive might be in error — then we can get no message through. This obviously makes sense. As the error rate drops, the upper limit on the efficiency increases, meaning that we need fewer codebits per data bit. For any given q, however, it is very difficult to reach Shannon's limit.

The actual proof of this theorem is not easy. I would like to first give a hand-waving justification of it which will give you some insight into where it comes from. Later I will follow a geometrical approach, and after that prove it in another way which is completely different and fun, and involves physics, and the definition of a quantity called information. But first, our hand-waving. Let us start with the assumption that M_C is very, very large. This will enable us to make some approximations. If the probability of a single bit error is q, then the average number of errors we would expect in a batch is:

$$k = qM_C. \tag{4.7}$$

This isn't exactly true - the actual number may be more or less — but this is the average error rate we expect, and it will do as a rough guess. We have to figure out how much coding we need to dispose of this number of errors. The number of ways this number of errors could be distributed through a batch is given by simple combinatorics:

$$\frac{M_C!}{k!\ (M_C - k)!} \tag{4.8}$$

Let us assume that we have m code bits. Such a number of bits can describe 2^m things. This number of bits must be able to describe at least the M bits of the data message plus the exact locations (to give us error *correction*, not just detection) of each possible distribution of errors, of which we are saying there are $M_C!/k!(M_C-k)!$ It is clear that $m \leq M_C - M$ (since some bits could be redundant), so we have the inequality:

$$2^{M_C-M} \geq \frac{M_C!}{k!\ (M_C - k)!} \tag{4.9}$$

We now take the logarithm of both sides. The right hand side we work out approximately, using Stirling's formula:

$$n! = \sqrt{(2\pi n)}\ n^n\ e^{-n} \exp[(1/12n) - (1/360n^3) + ...] \tag{4.10}$$

for large n. Hence:

$$\log n! \approx (1/2) \log n + n \log n - n + O(1/n) \tag{4.11}$$

(Here, $\log x = \log_e x$.) The last term simply represents a lot of junk that gets smaller as n gets bigger (tending to zero in the limit), plus terms like $\log 2\pi$. We can in fact get rid of the first term, namely $(1/2)\log n$, as this is small compared to the next two when n is large. We thus use:

$$\log n! \approx n \log n - n \tag{4.12}$$

and with this the right hand side of the inequality becomes:

$$M_C \log M_C - M_C \log(M_C-k) + k\log(M_C-k) - k\log k \tag{4.13}$$

which, using $k = qM_C$ and a little algebra, is:

$$M_C\ [q\log_2(1/q) + (1-q)\log_2(1/1-q)]. \tag{4.14}$$

We have here converted the natural logarithm to base two, which simply introduces a multiplicative factor on both sides, which cancels. Taking the logarithm to base two of the left hand side of the inequality, and dividing both sides by M_C, we end up with Shannon's inequality.

This inequality tells us that, if we want to code a message M, where the bit error rate is q, so that we can correct k errors, the efficiency of the result cannot exceed the bounds in (4.6). Of course, k is not arbitrary; we have taken it to be the mean number of errors, $k=qM_C$. The question we would like to have answered is whether we can code a message to be sure that the odds against more than a certain number of errors, say k', occurring is some number of our choosing; such as 10^{-30}. Shannon's actual Theorem says that we can do this; let us take our "proof" a little further to see why this might be so.

The number of errors that can occur in the message is not always going to be k, but will be k within some range and probability. In fact, the distribution of errors will follow a binomial distribution, with mean qM_C ($=k$) and standard deviation $\sigma = \sqrt{[M_C q(1-q)]}$. It is a standard result that, for MC large and q small, we can approximate this with a Gaussian (or Normal) distribution with mean qMC ($=k$) and standard deviation $\sigma = \sqrt{[M_C q(1-q)]}$; that is, the same as before. Now to ask that our error rate be less than a number N (e.g. 10^{-30}) is equivalent to demanding that the number of errors we have to correct be less than k', where:

$$k' = k + g\sigma \tag{4.15}$$

for some finite number g dependent on N. For the Gaussian distribution, the probability that the number of errors lies within one standard deviation from the mean k is 68%; within two it is 96%; within three 99%. So, for example, if we wanted to be 95% certain that there would be no errors in our message, we would have to demand not that we be able to correct k errors but:

$$k + 2\sigma \tag{4.16}$$

errors. In this case g=2. For any level of probability we pick, we can find a

value for g. As a rule, the probability of finding errors g standard deviations from k goes like:

$$\exp(-g^2/2) \tag{4.17}$$

and we can see how incredibly rare we can make errors for relatively small g. If g is twenty, for instance, this factor is exp(-200), or about 10^{-100}. A heck of a lot smaller than our 10^{-30}! For our choice of number, we get g to be about six.

To make use of this, we simply amend (4.9) by replacing k by k', the new number of errors we want to correct. If we can still find a code to do this, then we know that the odds of errors occurring in its transmission are less than our 10^{-30} or whatever. I leave it as an exercise for the reader to put:

$$k' = k + g\sigma = k + g\sqrt{k(1-q)} \tag{4.18}$$

into the inequality, and show that, in the limit, Shannon's result emerges as before.

What Shannon has given us is an upper limit on the efficiency. He hasn't told us whether or not we can find a coding method that reaches that limit. Can we? The answer is yes. I'll explain in more detail later, but the technique basically involves picking a random coding scheme and then letting M_C get larger and larger. It's a terrific mathematical problem. Provided M_C is big enough, we can reach the upper efficiency whatever the coding scheme. However, the message length might have to be enormous. A nice illustration of how big we have to take M_C in one case is from satellite communication. In sending messages from Earth to Jupiter or Saturn, it is not unusual for an error rate q of the order of a third to come through. The upper limit on the efficiency for this, from our table, is 8%; that is, we would have to send about 12 code bits for each data bit. However, to do this would require a prohibitively long M_C, so long that it is not practical. In fact, a scheme is used in which about *one hundred and fifty* code bits are sent for each data bit!

4.4: The Geometry of Message Space

I am now going to look at Shannon's Theorem from another angle, this time using geometry. In doing so I will introduce the useful idea of "message space".

Although this is primarily of importance in communications, and we are doing computing, I have found the idea interesting and useful, and you might too.

Message space, simply, is a space made up of the messages that we want to transmit. We are used to thinking of a space as something which can be many-dimensional, either continuous or discrete, and whose points can be labeled by coordinates. Message space is a multidimensional discrete space, some or all of whose points correspond to messages. To make matters a little more concrete, consider a three-bit binary code, with acceptable words:

000, 001, 010, 011, 100, 101, 110, 111.

These are just the binary numbers zero to seven. We can consider these numbers to be the coordinates of the vertices of a cube in three-dimensional space, as shown in Figure 4.1 below:

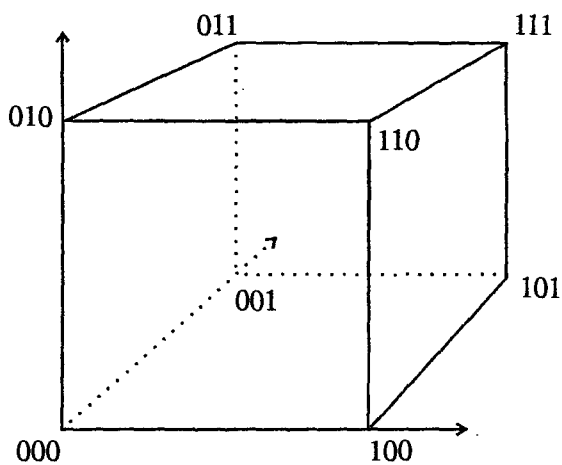


Fig. 4.1 A Simple Message Space

This cube is the message space corresponding to the three-bit messages. The only points in this space are the vertices of the cube — the space between them in the diagram, the edges, and whatnot are not part of it. This space is quite tightly packed, in that every point in it is an acceptable message; if we change one bit of a message, we end up with another. There is no wastage, everything is significant. We could easily generalize to a four-bit message, which would have a message space that was a 16-vertex "hypercube", which unfortunately our brains can't visualize! An m-bit message would require a 2^m-dimensional space.

What happens if there is an error in transmission? This will change the bits in the sent message, and correspond to moving us to some other point in the message space. Intuitively, it makes sense to think that the more errors there are, the "further" we move in message space; in the above diagram, (111) is "further" from (000) than is (001) or (100). This leads us to introduce a so-called "distance function" on the message space. The one we shall use is called the *Hamming distance*. The Hamming distance between two points is defined to be the number of bits in which they differ. So, the Hamming distance from 111 to 000 is 3, while from 001 to 000 it is just 1. According to this definition, in a 4-d space 1110 is as far from 1101 as is 0100, and so on. This makes sense. The notion of distance is useful for discussing errors. Clearly, a single error moves us from one point in message space to another a Hamming distance of one away; a double error puts us a Hamming distance of two away, and so on. For a given number of errors e we can draw about each point in our hypercubic message space a "sphere of error", of radius e, which is such that it encloses all of the other points in message space which could be reached from that point as a result of up to e errors occurring. This gives us a nice geometrical way of thinking about the coding process.

Whenever we code a message M, we rewrite it into a longer message M_C. We can build a message space for M_C just as we can for M; of course, the space for M_C will be bigger, having more dimensions and points. Clearly, not every point within this space can be associated one-on-one with points in the M-space; there is some redundancy. This redundancy is actually central to coding. e-Error correction involves designing a set of acceptable coded messages in M_C such that if, during the transmission process, any of them develops at most e errors, we can locate the original message with certainty. In our geometrical picture, acceptable messages correspond to certain points within the message space of M_C; errors make us move to other points, and to have error correction we must ensure that if we find ourselves at a point which does not correspond to an acceptable message, we must be able to backtrack, uniquely, to one that does. A straightforward way to ensure this is to make sure that, in M_C, all acceptable coded message points lie at least a Hamming distance of:

$$d = 2e + 1 \tag{4.19}$$

from each other. We can see why this works. Suppose we send an acceptable message M, and allow e errors to occur in transmission. The received message M' will lie at a point in M_C e units away from the original. How do we get back

from M' to M? Easy. Because of the separation of $d = 2e + 1$ we have demanded, M is the closest acceptable message to M'! All other acceptable messages must lie at a Hamming distance of at least $e+1$ from M'. Note that we can have simple error *detection* more cheaply; in this case, we can allow acceptable points in M_C to be within e of one another. The demand that points be $(2e+1)$ apart enables us to either correct e errors or detect $2e$.

Pictorially, we can envisage what we have done as mapping the message space of M into a space for M_C in such a way that each element of M is associated with a point in M_C such that no other acceptable points lie within a Hamming distance of $2e+1$ units. We can envisage the space for M_C as built out of packed spheres, each of radius e units, centered on acceptable coded message points (Fig. 4.2). If we find our received message to lie anywhere within one of these spheres, we know exactly which point corresponds to the original message.

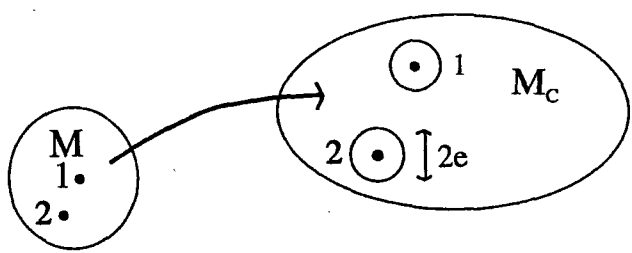
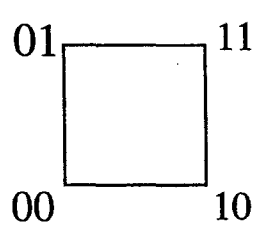


Fig. 4.2 A Message Space Mapping

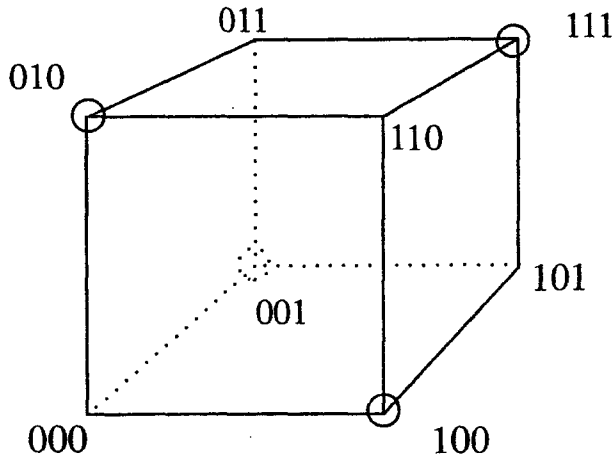
To make this idea a little more concrete, let us return to the three-bit cube we drew earlier. We can consider this the message space M_C corresponding to a parity-code that detects, but does not correct, single errors for the two-bit message system M comprising:

00, 01, 10, 11.

This system has the simple two-dimensional square message space:



The parity code simply tags an extra digit onto each message in M, clearly resulting in a 3-d cubic space for M_C. The acceptable messages are 000, 011, 101, and 110. This leaves four vertices that are redundant:



Any error in transmission will put us on one of these vertices, telling us that an error has occurred, but not where. Note that each false vertex lies within a Hamming distance of 1 from a genuine one. If we wanted single error correction for this system, we would have to use a space for M_C of four dimensions.

So if our coding system works, we should be able to move each of our message points into somewhere in the message space of M_C such that they are sufficiently separated. Every now and again we will be forced to allow some overlap between spheres of error, but this is not usually a problem[4]. We can now quickly see how this geometrical approach offers another proof of Shannon's Theorem. Use M and M_C to denote the dimensions of the original and coded message spaces respectively — this is just a fancy way of describing the lengths of the message strings. The number of points in M is 2^M, in M_C, 2^{M_C}. To correct k errors, we need to be able to pack M_C with spheres of error of radius k, one for each point in M. We do not want these to overlap. Using this, we can obtain an inequality relating the volume of M_C to that of the spheres. Now in a discrete space of the kind that is message space, the volume of a sphere is defined to be the number of points contained within it. It is possible to show

[4]"Perfect" codes, which we introduced in a problem earlier, are actually those for which the error spheres "fill" the message space without overlapping. If the spheres have radius e, then every point in the space lies within e units of one, and only one, message point. [RPF]

that, for an M_C-dimensional space, the number of points spaced a unit length apart that lie within a radius k units of a point is:

$$\frac{M_C!}{k!\ (M_C - k)!} \tag{4.20}$$

By noting that the volume of M_C must be greater than or equal to the number of points in each error sphere multiplied by the number of spheres, that is, the number of points in M, we recover the inequality (4.6). There is no need to go through the subsequent derivations again; you should be able to see how the proof works out.

Problem 4.3: Here is a nice problem you can try to solve using message space (that's how I did it). By now you are familiar with using a single parity bit at the end of a message to detect single errors. One feature of this technique is that you always need just one check bit irrespective of how long the message is; it is independent of M_C. The question is, can we also set up a method for detecting double errors that is M_C-independent? We are not interested in correcting, just detecting. We want a finite number of bits, always the same, and it would seem pretty likely that we could do it with only two bits! Recall that for the Hamming code we could correct a single error and detect doubles with a check bit for overall parity and the syndrome, but the number of check bits contributing to the syndrome depended on the message length. You should find that it is actually impossible to detect doubles without bringing in the length of the message.

4.5: Data Compression and Information

In a moment, I am going to look at Shannon's Theorem in yet another way, but first I would like you to let me wander a bit. The first direction I want to wander in is that of data compression, and I'd like to explain some of the ideas behind this. Consider a language like English. Now this has 26 letters, and if we add on commas, full stops, spaces and what-not, we have about thirty symbols for communication. So, how many things can I say in English if I have ten symbols at my disposal? You might say, well, thirty times ten. Not true. If I wrote the following string for you:

cpfajrarfw

it wouldn't be English. In real, interpretable English, you can't have everything; the number of acceptable words is limited, and the ordering of the letters within them is not random. If you have a "T", the odds are the next letter is an "H". It's not going to be an "X", and rarely a "J". Why? The letters are not being used uniformly, and there are very much fewer messages in English than seem available at first sight.

Perhaps, almost certainly, each one of you has parents back home who suffer from the fact that you never get around to writing letters. So they send you a card, all addressed and everything, which has on the back a lot of squares. And one of the squares says "I'm having a good time and enjoying CalTech. Yes/No." Next square says "I met a beautiful girl or handsome boy" or something, "Yes/No." The next message says "I have no more laundry to send", or "I'm sending it at such-and-such a time", and so on. What the poor parents are doing is converting long English sentences into single bits! They are actually trying to shame you into writing, but failing that, they are producing a method for you to communicate with them which is more efficient bitwise than your going to all that trouble. (Of course, you still have to post the card!) Another example of improving efficiency, and this is something you've all probably done, is to clip a long distance telephone company by prearranging to ring a friend and let the phone ring a set number of times if you're going to his party, another number if you're not, or maybe signal to him with three rings that you'll be at his place in your car in five minutes and he should go outside to wait, or what-have-you. You're calling long distance but getting the message through costs you nothing because he doesn't pick up the phone.

A related question is: how inefficient is English? Should I send my ten symbols directly, from the English alphabet? Or should I perhaps try a different basis? Suppose I had 32 rather than 30 symbols, then I could represent each element as a 5-bit string. Ten symbols hence becomes fifty bits, giving a possible 2^{50} messages. Of course, as I've said, most messages won't make sense. Whenever a Q appears, we expect it to be followed by a U. We therefore don't need to send the U; except for awkward words like Iraq, of course, which we could deal with by just sending a double Q. So we could exploit the structure of the language to send fewer symbols, and this packing, or compression, of messages is what we're going to look at now.

A good way to look at the notion of packing is to ask, if we receive a symbol in a message, how surprised should we be by the next symbol? I mean, if we receive a T, then we would not be surprised if we got an I next, but we would if we got an X, or a J. If you pick a T, the chances of a J next are very

small; in English, you don't have the freedom to just pick any letter. What we want to guess is how much freedom we have. Here is an interesting way to do it. Try this experiment with your friends. Take an English text, and read it up to a certain point. When you stop, your friend has to guess the next letter or symbol. Then you look at the next one and tell him whether he's right. The number of guesses he has to make to get the next letter is a handy estimate of the freedom, of the possibilities for the next letter. It won't necessarily be accurate — people's guesses will not be as good as a mechanical linguistic analysis — but it'll give you an idea. But in any case, people will be able to guess the next letter at a much better rate than one in thirty! In fact, the average number of possible letters following a letter in English is not 26 but about 5. You can work this out from your experiment by doing it often and averaging the number of guesses. And that gives you an idea of how English can be compacted. It also introduces us to the notion of how much *information* is in a message. We will return to this.

Another way of considering this problem of compression is to ask: if you had N symbols of English, with 32 possibilities for each, how many messages could you send? If you like, what is the greatest amount of information you could convey? As we have discussed, you could not send the full 32^N, as most would not make sense. Suppose the number of potentially sendable messages[5] is n. We can label each of these, code them if you like, by a number. We'll take this to be in binary. Now both guys at each end of the message can have a long list, telling them which number corresponds to which message, and instead of communicating by sending a full message, they just send numbers. This is exactly analogous to you with the cards your parents send — "Yes, I need a haircut"; "No, I didn't do my homework"; and so on. You cannot get a more efficient sending method than this, compressing a whole message down to one number. We can work out how many bits we will need to send to cover all the messages. If we need I bits, then we have:

$$2^I = n, \quad or \quad I = \log_2 n. \tag{4.21}$$

This number, the number of bits we minimally need to send to convey as much as we possibly could have conveyed in the N bits of English (or whatever other

[5]Strictly speaking, we mean the number of equally likely messages that can be sent. In reality, some messages will be more likely than others. However, we are not being rigorous in what follows, and will not worry about this for the time being. [RPF]

system being used) is called the *information* content, or just the *information* being sent. It is important to stress that the meaning of the word "information" here differs from that in ordinary usage — it is not totally distinct from this, but generally "information" in our sense tells us nothing about the usefulness of or interest in a message, and is strictly an academic term. There are lots of words like this in science: the meaning of the words "work" in physics and "function" in mathematics bears little relationship to their colloquial meanings. We will return to the concept of information and define it more rigorously later. For the moment, just bear in mind the fundamental idea: that we are coding messages into a binary system and looking at the bare minimum of bits we need to send to get our messages across.

It is possible to give a crude definition of the "average information per symbol" using the notions we have developed. Suppose we have a number of symbols that is not N, but twice N. What number of possible messages will correspond to this? For figurative purposes we can split the $2N$-symbol string into two N-symbol strings:

$$2N$$

$$\text{.....} \; / \; \text{.....}$$

$$N \qquad\qquad N$$

As a rough guess, we may expect that the number of potentially sendable messages will be the number of messages in each string multiplied together, or n^2. In general, of course, the precise answer will be horribly difficult to find. For example, there will be what we might call "edge effects" — the possibility of words being formable at the join of the two strings, crossing from one into the other — since the two N-symbol strings are not actually separated. There can also be "long-range correlations" where parts of the string influence others some distance away. This is true in English, where the presence of a word at one point in a message typically affects what other words can appear near it. As we are not being rigorous yet we will not worry about such problems. In fact, you can see that if we let N get bigger and bigger, our rough guess gets more accurate. If we have $2N$ symbols, we get about n^2 messages; if we have $3N$, about n^3; and generally if we have xN symbols, the number of messages will be about n^x. If we write the information content from N symbols as $I(N)$, we have:

$$I(xN) \approx \log_2(n^x) = x\log_2 n, \tag{4.22}$$

and we see that the ratio:

$$r = \frac{I(xN)}{xN} \approx \frac{\log_2 n}{N} = \frac{I(N)}{N} \tag{4.23}$$

is independent of x. So for large enough N, as long as our approximation gets better, it tends to a constant. We call this ratio the *information per symbol* of our system, an interpretation that seems clear from the right hand side of (4.23).

Let us return to the notion of information and try to get a better idea of what it means. In a sense, the amount of information in a message reflects how much surprise we feel at receiving it. Consider, say, receiving a printed communication from a bookshop, such as: "We are pleased to tell you that the book you ordered is in stock"; or its opposite: "We are sorry to inform you that ... is not in stock." These long messages contain many more symbols but no more information than the simple "Yes" or "No" you could elicit from a shopworker if you called the bookshop direct. Most of the symbols in the printed communications are redundant in any case: you only have to spot the words "pleased" and "sorry" to figure out what they are saying. In this respect, information is as much a property of your own knowledge as anything in the message.

To clarify this point, consider someone sending you two duplicate messages: a message, then a copy. Every time you receive a communication from him, you get it twice. (This is not for purposes of error detection; it's just a bad habit!) We might say, well, the information in the two messages must be the sum of that in each (remember, $I(n_1n_2) = \log_2(n_1n_2) = \log_2 n_1 + \log_2 n_2$). But this would be wrong. There is still only one message, the first, and the information only comes from this first half. This illustrates how "information" is not simply a physical property of a message: it is a property of the message and your knowledge about it.

Perhaps the best way to demonstrate the difference between our definition of information and the everyday term is to consider a *random* message, that is, an N-bit binary string with random bits. If all possible strings are allowable messages, and all are equally likely (which will happen if each bit is equally likely to be 0 or 1), then the information in such a message will be:

$$I = \log_2(2^N) = N \tag{4.24}$$

This is actually the most information you can get with this particular choice of symbols. No other type of message will reach $I=N$. Now surely this doesn't make sense – how can a random string contain *any* information, let alone the maximum amount? Surely we must be using the wrong definition of "information"? But if you think about it, the N-bit strings could each label a message, as we discussed earlier, and receiving a particular string singles out which of the 2^N possible messages we could get that we are actually getting. In this sense, the string contains a lot of "information". Receiving the message changes your circumstance from not knowing what it was to now knowing what it is; and the more possible messages you could have received, the more "surprised", or enlightened, you are when you get a specific one. If you like, the difference between your initial uncertainty and final certainty is very great, and this is what matters.

4.6: Information Theory

We have defined the information in a message to be:

$$I = \log_2 n, \tag{4.25}$$

where n is the number of *equally likely* messages we might receive. Each message contains this same amount of information. In the general case, some messages will be more likely than others, and in this case, the greater the likelihood, the less information contained. This makes sense, given our claim that the information in a message represents the "surprise" we experience at receiving it. In this section, we come on to the topic of *information theory* proper, which will enable us to both generalize and make more rigorous our previous considerations.

We'll take a simple example first. Suppose our message is built from an alphabet of symbols. There could be any number of these, such as the four bases of DNA, or whatever: we certainly do not want to restrict ourselves to the letters of English. Let the number of symbols be i, and label them:

$$a_1, a_2, \ldots\ldots a_i.$$

Messages in this language are long strings of these symbols, say of length N. Now before we go any further, we have to make some assumptions about the way these symbols are distributed throughout messages. We assume firstly, that we can assign a probability, p_i, to each symbol, which is the probability that any given symbol in the message is the symbol a_i. The quantity p_i tells us the frequency of occurrence of a_i. We also assume that each symbol in the message is independent of every other; that is, which symbol appears at a given position does not depend on symbols at other positions, such as the one before. This is actually quite an unrealistic assumption for most languages. We will consider cases for which it is not true shortly.

How much information is carried by a given message? A simple way in which we can work this out is as follows. Suppose the message we have is length N. Then we would expect to find symbol a_1 turn up Np_1 times on average, a_2 Np_2 times, ... a_i Np_i times. The bigger N is, the better these guesses are. How many different messages do we have? Combinatorics comes to our rescue, through a standard formula. If we have N objects, m of one type, n of another, p of another, ..., and $m+n+p+... = N$, then the number of possible arrangements of the m, n, p... is given by:

$$\frac{N!}{m!n!p!...}. \tag{4.26}$$

On average, then, we can say that the number of different messages in N symbols is

$$\frac{N!}{(Np_1)!(Np_2)!...}. \tag{4.27}$$

We earlier defined information to be the base two logarithm of the number of possible messages in a string. That definition was based on the case where all messages were equally likely, but you can see that it is a good definition in the unequal probability case too. We therefore find the expected information in a message, which we write $<I>$, by taking the $\log_2$ of (4.27). Assuming N to be very large, and using Stirling's approximation, with which you should be familiar by now, we find:

$$<I> = N \sum_{i=1}^{M} (-p_i \log_2 p_i). \tag{4.28}$$

We can therefore obtain the *average information per symbol*:

$$<I>/N = \sum_{i=1}^{M} (-p_i \log_2 p_i). \tag{4.29}$$

This derivation appeals to intuition but it is possible to make it more rigorous. Shannon defined the information in a message to be the base two logarithm of the probability of that message appearing. Note how this ties in with our notion of information as "surprise": the less likely the message to appear, the greater the information it carries. Clearly, the information contained in one particular symbol a_n is:

$$-\log_2 p_n, \tag{4.30}$$

and if a message contains n_1 a_1's, n_2 a_2's, and so on, its information will be:

$$I = -\log_2[(p_1^{n_1})(p_2^{n_2}) \ldots (p_M^{n_M})] \tag{4.31}$$

which is:

$$-(n_1 \log_2 p_1 + n_2 \log_2 p_2 + \ldots n_M \log_2 p_M). \tag{4.32}$$

Incidentally, this shows that if we place two messages end to end, the total information they convey is twice that in the messages individually, which is satisfying. Check this for yourselves. Now the average information in a message is calculated in standard probabilistic fashion; it is just:

$$\begin{aligned} \textit{Average information} &= \Sigma\ \textit{information in symbol } a_i \\ &\quad \cdot\ (\textit{expected number of appearances of } a_i) \\ &= -\Sigma\ (\log_2 p_i) \times (Np_i) \end{aligned} \tag{4.33}$$

which is our previous result. Incidentally, Shannon called this average information the "entropy", which some think was a big mistake, as it led many to overemphasize the link between information theory and thermodynamics[6].

Here is a nice, and slightly different, illustration of these ideas. Suppose we work for a telegraph company, and we send, with unequal probabilities, a range of messages — such as "Happy birthday", "Isn't Christmas wonderful", and so on. Each one of these messages has a probability P_m of being requested by a customer (m=1 to M, say). We can define two types of information here. There is that calculated by the person who receives the telegram — since it's my birthday it is not very surprising I get a "happy birthday" message, so there is not much information there. There is also the information as seen by the telegraphist who gets requested to send the message. It's interesting to look at the second type. To work this out, we would have to look at the operation of the telegraphy business for some time, and calculate the proportions of each type of message that are sent. This gives the probability P_m of message m being sent and we can treat each message as the symbol of some alphabet, similar to labeling each one, rather like the parent-student card we looked at earlier. We can hence calculate the information from the viewpoint of the telegraphist.

4.7: Further Coding Techniques

Let me now return to the topic of coding and describe a couple of popular techniques for coding messages to show you some of the wonderful and ingenious ways in which this can be done. These codes are unlike those we've considered so far in that they are designed for messages in which the symbol probabilities vary.

[6]Legend has it that Shannon adopted this term on the advice of the mathematician John Von Neumann, who declared that it would give him ". . . a great edge in debates because nobody really knows what entropy is anyway." [RPF]

4.7.1: Huffman Coding

Consider the following system of eight symbols, where by each I have written the probability of its occurrence (I have arranged the probabilities in descending order of magnitude):

E	0.50
THE	0.15
AN	0.12
O	0.10
IN	0.04
R	0.04
S	0.03
PS	0.02

The probabilities add to one, as they should. A sample message in this system might be:

ANOTHER

which has probability 0.12 x 0.10 x 0.15 x 0.04. Now we notice that the symbol E appears much more often than the others: it turns up 25 times as often as the symbol PS, which takes twice as much effort to write. This symbol system doesn't look very efficient. Can we write a new code that improves it? Naively, we might think: "Well, we have eight symbols, so let's just use a three-bit binary code." But since the E occurs so often, would it not be better to describe it, if we can, by just *one* bit instead of three? We might have to use more bits to describe the other symbols, but as they're pretty rare maybe we might still gain something. In fact, it is possible to invent a *non-uniform* code that is much more efficient, as regards the space taken up by a message, than the one we have. This will be an example of compression of a code. Morse had this idea in mind when he assigned a single "dot" to the common E but "dash dash dot dash" to the much rarer Q.

The idea is that the symbols will vary in their lengths, roughly inversely according to their probability of appearance, with the most common being represented by a single symbol, and with the upshot that the typical overall message length is shortened. We will actually replace our symbols by binary strings. The technique I will outline for you — I will let you figure out for yourselves why it works — is due to Huffman. It is quite a popular method, although I believe frequently costly to implement. It is a two-stage process, and

is best understood by considering the following tree diagram, where initially the symbols are arranged in ascending order of probabilities (Fig. 4.3):

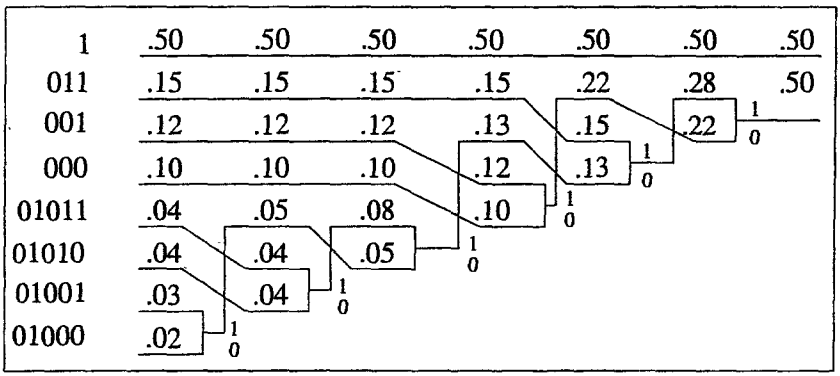


Fig. 4.3 Huffman Coding Tree

Begin by coalescing the two lowest probability symbols into one, adding the probabilities. We can now pretend that we have a source alphabet consisting of the original symbols, less the lower two, plus a new "joint" symbol with probability of occurrence (in this case) .05. Redraw the column, placing the joint symbol at its appropriate point in the probability hierarchy, as shown in Figure 4.3. Now iterate. Coalesce the next two to shrink the list further. Continue in this vein until we reach the right hand of the tree, where we have an "alphabet" of two symbols, the original maximally probable one, plus a summed "joint" symbol, built from all the others.

To figure out the actual assignment of coding symbols, we now retrace a path back through the tree. The rule is straightforward: at each branch in the path required to get back to the original symbol, you add a digit to its code. If you follow the upper path at the branch, you add a one; a lower branch gives you a zero (this is purely a matter of convention). You move from right to left across the tree, but the code you write out from left to right as you go. What is happening is shown in Figure 4.4:

1	E					.50
011	THE		.15	.22	.28	.50
001	AN	.....	.13	.13		
000	O					
01011	IN					
01010	R	.....				
01001	S					
01000	PS					

Fig. 4.4 "Trellis" for Huffman Coding

Let us look at the code for "THE". To get to it, we have to start with a 0. We follow the upper path from the first branch, giving us 01 so far. Then, again, we have to follow the upper path from the next branch. We end up with 011, which is the code for THE. The other codes are as shown above. It is worth pointing out that other Huffman codes can be developed by exploiting the ambiguity that occasionally arises when a joint probability at some point equals one of the original probabilities. Do we put the joint above its equal in the table, or beneath? You might like to think about this.

We can easily calculate the length saving of this code in comparison with a straight three-bit code. With three bits, the average length of a symbol is obviously three! With this Huffman code, the average symbol length is:

$$(1 \times 0.5) + 3 \times (0.15 + 0.12 + 0.10) + 5 \times (.09+.04+.03+.02) = 2.06.$$

which is a saving of nearly a third!

There is a nice subtlety in constructing non-uniform codes that the Huffman method takes care of nicely. It has the property that *no code word is the prefix of the beginning of any other code word.* A little thought shows that a code for which this is not true is potentially disastrous. Suppose we had the following set of symbols:

1, 01, 10, 101, 010, 011.

Try and decode this message: 011010110. You can't do it! At least, not uniquely. You do not know whether it is 01-1-01-01-10 or 011-01-01-10 or 01-101-01-10 or another possibility. There is an ambiguity due to the fact that the symbols can run into each other. A good, uniquely decodable symbol choice is necessary to avoid this, and Huffman coding is one way forward. You can check that the code choice we have found for our symbols leads to unique decoding.

Problem 4.4: Huffman coding differs from our previous coding methods in that it was developed for compression, not error correction. The technique gives us nicely-packed codes, but they are quite sensitive to errors. If we have the following message:

00100001101010 (= ANOTHER)

then a single glitch can easily result in swapped symbols. For example, an error in position 2 would give us THEOTHER[7]. This throws up an interesting question that you might like to address. For general non-uniform coding, what is the greatest number of symbol errors that can be generated by a one-bit mistake? You are used to thinking of one error — one bit, but with non-uniform coding that might not be true. For example, might it not be possible that a single error might change one symbol to one of a different length, and that this will affect the next symbol, and the next, and so on, so that the error propagates throughout the entire message string? It turns out not. Can you figure out why?

4.7.2: Predictive Coding

Thus far, I have only considered situations in which the probabilities of symbols occurring in a message are independent: symbols exert no influence across the message. However, as I have stressed by the example of English, such dependence is extremely common. The full mathematical treatment of source alphabets comprising varying probabilities of appearance and intersymbol influence is quite complex and I will not address it here. However, I would like to give you some flavor of the issues such influence raises, and I will do this by considering *predictive coding*. This is another way of compressing codes, rather than correcting them.

[7]It could be said that too many glitches would drive us 01011010010011. [RPF]

Let us suppose that we have a source alphabet which is such that, if we know the contents of a message up to a point, we can predict what the next symbol will be. This prediction will not typically be certain — it will be probabilistic — and the details of how we make it do not matter. The method we use might require the knowledge of only the previous symbol, or the previous four, or even the whole message. It does not matter. We just have some rule that enables us to predict the next symbol.

Imagine now that we do our predicting with a predictor, a black box that we stick next to the message to be sent, which contains some formula for making predictions. Here is how things work. The predictor is fully aware of the message that has been sent so far (which we illustrate by feeding a message line into it), and on the basis of this it makes a prediction of what symbol is to be sent next. This prediction is then compared with the actual source symbol that comes up. If the prediction is right, then we send a zero along the transmission channel. If the prediction is wrong, we send a one. The easiest way to implement this is to bitwise add the source symbol and the prediction and ignore any carry. Schematically, we have, at the transmission end (Fig. 4.5):

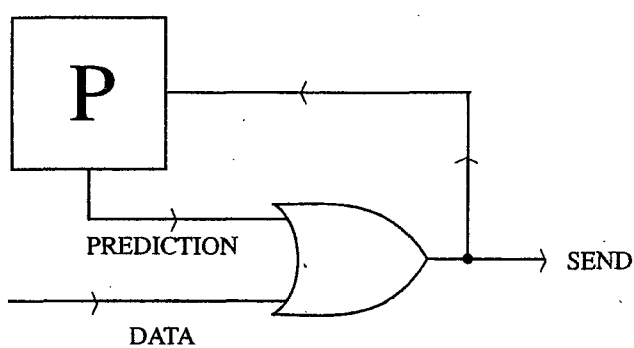


Fig. 4.5 A Predictive Encoder

Note that we have incorporated a feedback loop to let the predictor know whether its prediction was correct or not. A good predictor will produce a long string of zeroes, with the occasional one interspersed where it made a mistake: a random predictor, one that just guesses, will have ones and zeroes split fifty-fifty, if that is the base-rate in the source. It's not difficult to see how, if we send this string, we can reconstruct the original message at the other end by using an identical predictor as a decoder. It simply works backwards. This is all very nice, of course, but what is the point of this rather exotic procedure? Well,

if the first predictor is good, making pretty accurate predictions, then it will generate far more zeroes than ones. Interspersed between the ones will be long runs of zeroes. The key is this — when sending the message, we do not send these runs: *instead we send a number telling us how many zeroes it contained.* We do this in binary, of course. If there is a run of twenty two zeroes before the next one digit, we don't send out:

0000000000000000000000

but rather its binary equivalent:

10110.

That's some saving of transmission space! All we have to do is get the guy at the receiving end to break the binary numbers down into strings of zeroes, and use his predictor to figure out what we were saying. Predictive coding enables us to compress messages to quite a remarkable degree.

Problem 4.5: An interesting problem with which you can entertain yourself is how to compress things still further. The average length of the runs of zeroes is dependent on how good the predictor is. Once we know how good it is, we can work out the probability that a run will have a given length. We can then use a Huffman technique to get an even tighter code! Work out the details if we are sending an equally likely binary code, and the probability of the predictor being wrong in its prediction is q. You can get pretty close to Shannon's limit using compression of this sort.

4.8: Analogue Signal Transmission

I would like to discuss one more coding problem before leaving the subject. This is the question of how we can send information that is not naturally in the form of bits; that is, an analogue signal. Ordinarily, information like the oil pressure in a car, the torque in a drive shaft, the temperature variation on the Venusian surface, is *continuous*: the quantities can take any value. If we only have the capacity to transmit bits, how do we send information of this kind? This is not a matter of fundamental principle; it is actually a practical matter. I will say a few words about it despite the fact it is somewhat peripheral. You could say that the whole course is somewhat peripheral. You just wait!

Let us suppose for starters that our continuous quantity — S, say — is

restricted to lie between 0 and 1:

$$0 \leq S \leq 1 \tag{4.34}$$

The secret of sending the value of S is to approximate it. The most important question to ask is with what accuracy we want to send data. Suppose we want S to within 1%. Then, all we need do is split the interval [0,1] up into one hundred slices (usually referred to as "bins"), and transmit information about which slice the value of S is in; in other words, a number between 0 and 100. This is easy. However, as we prefer to use binary, it is better to split the range of S into 128 slices ($=2^7$), and send the S value as a 7-bit number. Similarly, if we want to send S to an accuracy of one part in a thousand, we would send a 10-bit number, having split [0,1] into 1024 bins.

What happens if the variable S is unbounded? This is not uncommon. Usually, such a variable will have values that are not evenly distributed. In other words, it will be more likely to have some values rather than others (very, very few physical quantities have flat probability distributions). We might have a variable with a probability distribution such as that shown in Figure 4.6:

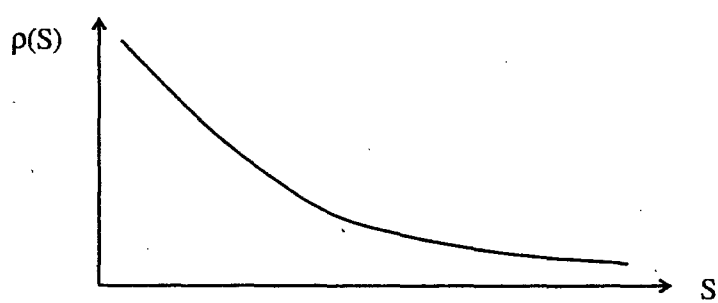


Fig. 4.6 A Sample Probability Distribution for a Physical Variable

The probability density $\rho(S)$ has the usual definition: if we make a measurement of S, the probability of finding its value to lie between S_1 and S_2 is:

$$\int_{S_1}^{S_2} \rho(S)\, dS \tag{4.35}$$

or, if S_1 and S_2 lie infinitesimally close to one another, $S_2 = S_1 + \delta s$:

$$\rho(S_1).\delta s \tag{4.36}$$

The basic idea for transmitting S in this general case is the same. We divide the effective range of S into a number of bins with the important difference that we size these bins so that they are all of equal probability (Fig. 4.7):

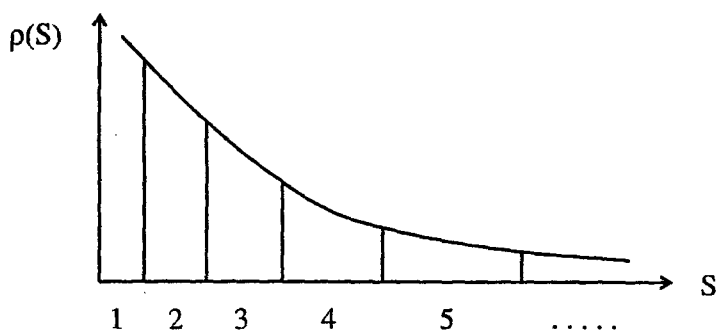


Fig. 4.7 Division of ρ(S) into Equal Volume Bins

Clearly the bins are of different width, but they are chosen to have the same area when viewed probabilistically. They are defined by the formula:

$$(1/128) \int_{S_i}^{S_{i+1}} \rho(S)\, dS \tag{4.37}$$

where i runs from 0 to 127, and the ith bin corresponds to the S-values S_i to S_{i+1}.

Alternatively, we can make a change of variables. For each value s of S we can define the function $P(s)$ by:

$$P(s) = \int_0^s \rho(S)\, dS \tag{4.38}$$

$P(s)$ is just the cumulative probability function of S, the probability that $S \leq s$. It clearly satisfies the inequality ($0 \leq P \leq 1$). One well-known statistical property of this function (as you can check) is that its own distribution is *flat*: that is, if we were to plot the probability distribution of $P(s)$ as a function of s in Figure 4.6, we would see just a horizontal line. A consequence of this is that if we make equal volume bins in P, they will automatically be of equal width. That takes us back to the first case.

A different, but related, problem is that of transmitting a function of time (Fig. 4.8):

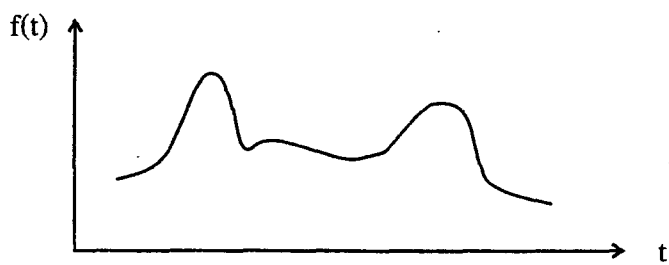


Fig. 4.8 A Typical Function of Time

Consideration of such a problem will bring us on to consider the famous *Sampling Theorem*, another baby of Claude Shannon. The basic idea here would be to sample the function at certain regular time intervals, say τ, and send the value of the function at each time digitally. The receiver would then attempt to reconstruct the original signal from this discrete set of numbers. Of course, for a general function, the receiver will have to smooth out the set, to make up for the "gaps". However, for certain types of continuous function, it is actually possible to sample in such a way as to encode *completely* the information about the function: that is, to enable the receiver to reconstruct the source function exactly! To understand how it is possible to describe a continuous function with a finite number of numbers, we have to take a quick look at the mathematical subject of Fourier analysis. I will cover this because I think it is interesting;

those without the mathematical background might wish to skip it!

It turns out that if the "Fourier transform" of the function $g(\omega) = 0$ for all $|\omega| \geq \upsilon$, and we sample at intervals of $\tau = \pi/\upsilon$, then these samples will completely describe the function. What does this mean? Recall that, according to Fourier theory, any periodic function $f(t)$ can be written as a sum of trigonometric terms. For a general function of time, $f(t)$, we have:

$$f(t) = (1/2\pi)\int_{-\infty}^{\infty} g(\omega)e^{-2\pi i\omega t}\, d\omega \tag{4.39}$$

where $g(\omega)$ is the *Fourier Transform* of $f(t)$. What we have effectively done here is split $f(t)$ up into component frequencies, suitably weighted. Now the typical function (signal) that is encountered in communication theory has a limited *bandwidth*; that is, there is an upper limit to the frequencies that may be found in it (for example, the channel through which the signal is sent might not be able to carry frequencies above a certain value). In such a case, the limits of integration in (4.39) become finite:

$$f(t) = (1/2\pi)\int_{-W(\upsilon)}^{W(\upsilon)} g(\omega)e^{-2\pi i\omega t}\, d\omega, \tag{4.40}$$

where W is the bandwidth, and υ is now the highest frequency in the Fourier expansion of $f(t)$[8].

It is possible (the math is a bit tough) to show that this expression reduces to the infinite sum over the integers:

$$f(t) = \sum_{n=-\infty}^{\infty} f(n\pi/\upsilon).[\sin\pi(\upsilon t - n\pi)]/(\upsilon t - n\pi) \tag{4.41}$$

This is the Sampling Theorem. If you look at this expression, you will see that as long as we know the values of the function $f(t)$ at the times:

[8]Conventionally, the bandwidth W is given by $W=\upsilon/2\pi$. [RPF]

$$t = n\pi/\upsilon, \tag{4.42}$$

where n is an integer, then we can work it out at all other times, as a superposition of terms weighted by the signal samples. This is a subtle consequence of using the well-known relation:

$$(\sin x)/x \rightarrow 1 \ as \ x \rightarrow 0 \tag{4.43}$$

in (4.41): setting $t = n\pi/\upsilon$ in the summand, we find that all terms except the nth vanish: the nth is just unity multiplied by the value of f at $t = n\pi/\upsilon$. In other words, if we sampled the function at times spaced (π/υ) time units apart, we could reconstruct the entire thing from the sample! This finding is of most interest in the physically meaningful case when the function $f(t)$ is defined only over a finite interval (0,T). Then, the sum (4.41) is no longer infinite and we only need to take a finite number of sample points to enable us to reconstruct $f(t)$. This number is $(T\upsilon/\pi)$.

Although I have skated over the mathematical proof of the Sampling Theorem, it is worth pausing to give you at least some feel for where it comes from. We are sampling a function $f(t)$ at regular intervals, τ. The graph for the sampled function arises from multiplying that of the continuous $f(t)$ by that of a spikey "comb" function, $C(t)$, which is unity at the sample points and zero elsewhere (Fig. 4.9):

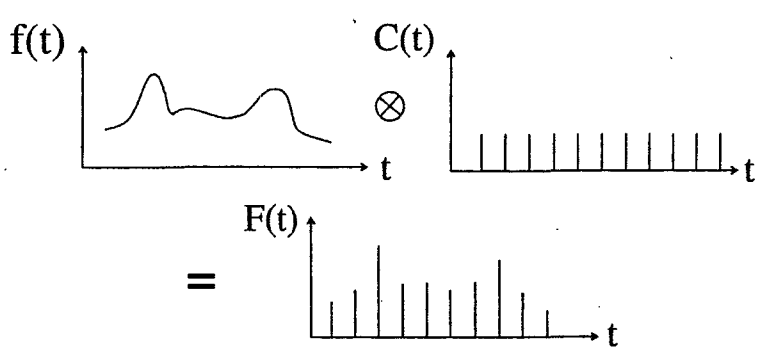


Fig. 4.9 The Sampled Function

Now, corresponding to $f(t)$ is a Fourier Transform $\phi(\omega)$. $C(t)$ also has an associated transform, $\chi(\omega)$ another comb function (Fig. 4.10):

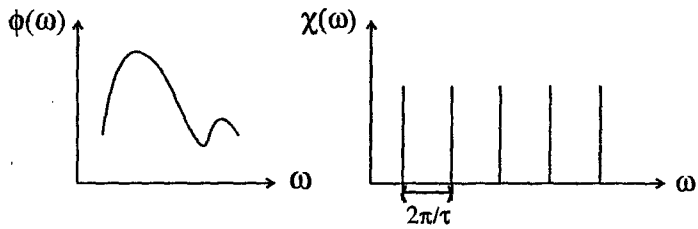


Fig. 4.10 Fourier Transforms of f(t) and C(t)

The transform χ is actually a set of equally-spaced delta functions ($2\pi/\tau$ apart). The Fourier transform of the sampled function, $F(t)$, is obtained by the process of "convolution", which in crude graphical terms involves superposing the graph of ϕ with that of χ. We find that the transform of $F(t)$ comprises copies of the transform of $f(t)$, equally-spaced along the horizontal axis, but scaled in height according to the trigonometric ratio in (4.41) as in Figure 4.11:

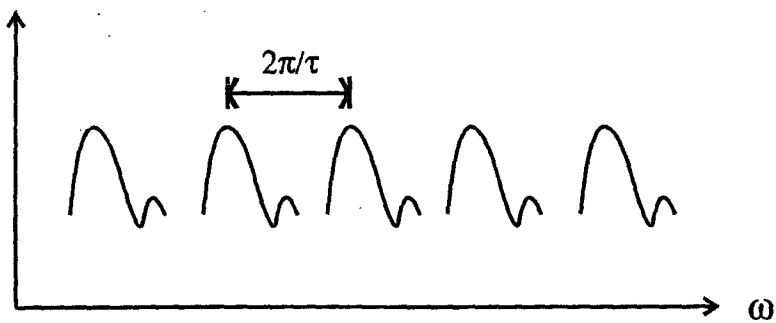


Fig. 4.11 The Fourier Transform of the Sampled Function

Look closely at this graph. What it is telling us is that information about the

whole of $f(t)$ could, in principle, be extracted from $F(t)$ alone. There is as much information in one of the Fourier-transformed bumps in Figure 4.11 as there is in the whole of Figure 4.9! As the former transform comes solely from the sampled function $F(t)$, we can see the basic idea of the Sampling Theorem emerging.

An interesting subtlety occasionally arises in the sampling process. The Sampling Theorem tells us that, if a signal has an upper frequency limit of ν (i.e. a bandwidth of $\nu/2\pi$) then we need at least $(T\nu/\pi)$ sample points to enable us to reconstruct the signal. If we take more points than this, all well and good. However, if we take fewer (and this can arise by accident if the function $f(t)$ has "tails" that lie outside the interval (0,T)), our sampling will be insufficient. Under such circumstances we get what is known as *aliasing*. The sampling interval will be too coarse to resolve high frequency components in $f(t)$, instead mapping them into low frequency components — their "aliases". A familiar example of this phenomenon can be found in movies. Movies, of course, are samples — 24 times a second, we take a snapshot of the world, creating the illusion of seamless movement to our eyes and brains. However, evidence that sampling has occurred often shows up. Maybe the best known is the behavior of wagon wheels in old westerns. As kids we all noticed that, when a wagon started moving, at first the spokes in the wheels seemed to go around the right way. Then, as things sped up, they appeared to stop rotating altogether. Finally, as things sped up still further, the wheels appeared to be going the wrong way around! The explanation for this phenomenon lies in inadequate sampling. Another example of aliasing occurs when we inadequately sample audio data, and end up with frequencies that we cannot ordinarily hear being aliased into ones we can. To avoid aliasing, we would need to filter out of the signal any unduly high frequencies before we sampled. In the case of the movies, this would mean taking pictures with a wider shutter, so that the picture is just a blur, or smoothed out.

It is now possible to send sound *digitally* — sixteen bits, 44.1 kHz reproduces perfectly, and is pretty resistant to noise; and such a method is far superior to any analog technique. Such developments will transform the future. Movies will be cleaned up, too — optical fibers, for example, are now giving us *over*capacity. The soap ad will appear with absolute clarity. It seems that the technological world progresses, but real humanistic culture slides in the mud!

FIVE

REVERSIBLE COMPUTATION AND THE THERMODYNAMICS OF COMPUTING

I would now like to take a look at a subject which is extremely interesting, but almost entirely academic in nature. This is the subject of the energetics of computing. We want to address the question: *how much energy must be used in carrying out a computation?* This doesn't sound all that academic. After all, a feature of most modern machines is that their energy consumption when they run very fast is quite considerable, and one of the limitations of the fastest machines is the speed at which we can drain off the heat generated in their components, such as transistors, during operation. The reason I have described our subject as "academic" is because we are actually going to ask another of our fundamental questions: what is the *minimum* energy required to carry out a computation?

To introduce these more physical aspects of our subject I will return to the field covered in the last chapter, namely the theory of information. It is possible to treat this subject from a strictly physical viewpoint, and it is this that will make the link with the energy of computation.

5.1: The Physics of Information

To begin with, I would like to try to give you an understanding of the physical definition of the information content of a message. That physics should get involved in this area is hardly surprising. Remember, Shannon was initially interested in sending messages down real wires, and we cannot send messages of any kind without some interference from the physical world. I am going to illustrate things by concentrating on a particular, very basic physical model of a message being sent.

I want you to visualize the message coming in as a sequence of boxes, each of which contains a single atom. In each box the atom can be in one of two places, on the left or the right side. If it's on the left, that counts as a 0 bit, if it's on the right, it's a 1. So the stream of boxes comes past me, and by looking to see where each atom is I can work out the corresponding bit (Fig. 5.1):

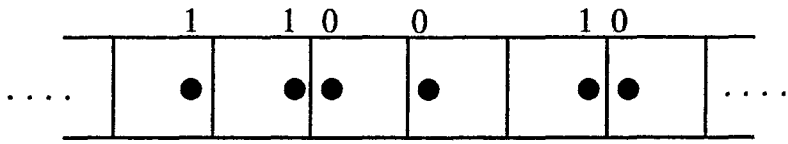


Fig. 5.1 A Basic Atomic Message

To see how this model can help us understand information, we have to look at the physics of jiggling atoms around. This requires us to consider the physics of gases, so I will begin by taking a few things I need from that. Let us begin by supposing we have a gas, containing N atoms (or molecules), occupying a volume V_1. We will take this gas to be an exceptionally simple one; each atom, or molecule, within it (we take the terms to be interchangeable here) is essentially free — there are no forces of attraction or repulsion between each constituent (this is actually a good approximation at moderately low pressures). I am now going to shrink the gas, pushing against its volume with a piston, compressing it to volume V_2. I do all this isothermally: that is, I immerse the whole system in a thermal "bath" at a fixed temperature T, so that the temperature of my apparatus remains constant. Isn't it wonderful that this has anything to do with what we're talking about? I'm going to show you how. First we want to know how much work, W, it takes to compress the gas (see Fig. 5.2):

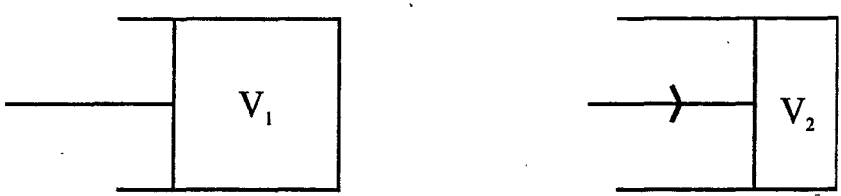


Fig. 5.2 Gas Compression

Now a standard result in mechanics has it that if a force F moves through a

small distance δx, the work[1] done δW is:

$$\delta W = F\ \delta x \tag{5.1}$$

If the pressure of the gas is p, and the cross-sectional area of the piston is A, we can rewrite this using $F = pA$ and letting the volume change of the gas $\delta V = A\delta x$ so that:

$$\delta W = p\ \delta V. \tag{5.2}$$

Now we draw on a standard result from gas theory. For an ideal gas at pressure p, volume V and temperature T, we have the relation:

$$pV = NkT \tag{5.3}$$

where N is the number of molecules in the gas and k is Boltzmann's constant (approximately 1.381×10^{-23} J K^{-1}). As T is constant — our isothermal assumption — we can perform a simple integration to find W:

$$W = \int_{V_1}^{V_2} \frac{NkT}{V}\, dV = NkT \log \frac{V_2}{V_1}. \tag{5.4}$$

(Here, $\log x = \log_e x$.) Since V_2 is smaller than V_1, this quantity is negative, and this is just a result of the convention that work done on a gas, rather than by it, has a minus sign. Now, ordinarily when we compress a gas, we heat it up. This is a result of its constituent atoms speeding up and gaining kinetic energy. However, in our case, if we examine the molecules of the gas before and after compression, we find no difference. There are the same number, and they are jiggling about no more or less energetically than they were before. There is no difference between the two at the molecular level. So where did the work go? We put some in to compress the gas, and conservation of energy says it had to go somewhere. In fact, it *was* converted into internal gas heat, but was promptly

[1]Another one of those awkward words, like "information". Note that, with this definition, a force must move through a distance to perform work; so it does not take any of this kind of "work" to hold up a suitcase — only to lift it! [RPF]

drained off into the thermal bath, keeping the gas at the same temperature. This is actually what we mean by isothermal compression: we do the compression slowly, ensuring that at all times the gas and the surrounding bath are in thermal equilibrium.

From the viewpoint of thermodynamics, what we have effected is a "change of state", from a gas occupying volume V_1 to one occupying volume V_2. In the process, the total energy of the gas, U, which is the sum of the energies of its constituent parts, remains unchanged. The natural thermodynamical quantities with which such changes of state are discussed are the *free energy* F and the *entropy* S, which are related by:

$$F = U - TS. \tag{5.5}$$

The concept of free energy was invented to enable us to discuss the differences between two states even though there might be no actual mechanical differences between them. To get a better feel for its meaning, look at how expression (5.5) relates small variations at constant temperature:

$$\delta F = \delta U - T\delta S. \tag{5.6}$$

For the change under consideration, the total gas energy remains constant, so $\delta U=0$ and $\delta F = -T\,\delta S$. δF is just the "missing" heat energy siphoned off into the heat bath, $NkT \log(V_1/V_2)$, and we use this to write (5.6) as an entropy change:

$$\Delta S = Nk \log \frac{V_2}{V_1}. \tag{5.7}$$

Note that as we are dealing with a finite change here, we have replaced the infinitesimal δ with a finite Δ.

Entropy is a rather bizarre and counter-intuitive quantity, and I am never sure whether to focus on it or on the free energy! For those who know a little thermodynamics, the general equation $\delta S = -\delta F/T$ is a variant of the standard formula $\delta S = \delta Q/T$ for the infinitesimal change in entropy resulting from a thermodynamically reversible change of state where, at each stage, an amount

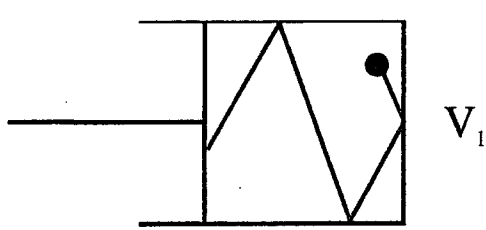
of heat δQ enters or leaves the system at absolute temperature T. For an irreversible process, the equality is replaced by an inequality, ensuring that the entropy of an isolated system can only remain constant or increase – this is the Second Law of Thermodynamics. I'll say a little more about entropy in a moment.

Now we take a bit of a leap, and it is not obvious that we can do this, but we can. We consider the case where our gas contains only one molecule. That is, we put N=1 into our formulae. Now it's difficult to get a feeling for concepts like temperature, pressure and volume, never mind free energy and entropy, when you only have one molecule! However, these concepts make sense as long as we consider them to be time averaged, smoothing out the irregularities of this one particle as it bounces back and forth. Indeed, our formulae actually work with N=1, as long as there is this hidden smoothing. The situation is more fun, too!

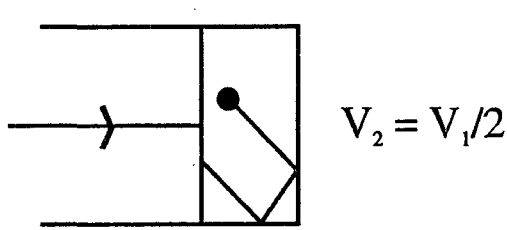
Let us suppose that we are halving the volume occupied by the molecule: $V_2 = V_1/2$. We then find that the free energy and the entropy of the particle change by:

$$+ kT \log 2 \textit{ and } -k \log 2 \tag{5.8}$$

respectively. What does this mean? Pictorially, the situation has changed from:



to:



The physical state of the molecule before and after the compression appears to

be the same — its actual (kinetic) energy has not changed, for example — yet for some reason we have a change in these quantities F and S. What has happened, and this is very subtle, is that my *knowledge of the possible locations of the molecule has changed.* In the initial state, it could be hiding anywhere in volume V_1: after the compression, it must be somewhere within V_2. In other words, there are fewer places it can be in.

This concept of "knowledge" is extremely important, and central to the concept of entropy, so I will dwell on it awhile. It arises from the deeply statistical nature of thermodynamics. When doing the mathematics of vast numbers of particles that make up gases, we cannot practically follow the paths and momenta of every molecule in the gas, so we are forced to turn to probability theory. Concepts such as temperature and pressure of a gas are essentially defined to be statistical averages. We assign certain physical properties to each molecule, assume particular distributions for these molecules, and calculate the average by a weighting process: so many molecules will move this fast, corresponding to one temperature; so many will move that fast, giving another temperature; and we just average over everything. The entropy of a gas is defined statistically, indeed this is its core definition, but in a different way to quantities such as temperature and energy. Unlike these, it is not a macroscopic property that arises from a sum of microscopic properties. Rather, it is directly related to the *probability that the gas be in the configuration in which it is found.* By "configuration" I mean a particular arrangement, or cluster of arrangements, of positions and momenta for each of the N constituent molecules (or, if you want to be fancy, a particular point or region in "phase space"). The existence of such a probability should not come as too much of a surprise: if you look at any given gas it is far less likely at the outset that you will find all the molecules moving in the same direction or paired up and dancing than you will find them shooting all over the place at random. Entropy quantifies this notion. Loosely speaking, if the probability of a particular gas configuration is W, we have:

$$S \approx k \log W. \tag{5.9}$$

The bigger W, the bigger the entropy, and, like all probabilities, the W's add, so we can straightforwardly calculate the chances of being in some range of configurations. The gas with molecules going all one way has a W much less than that of the one with a more random — or *more disordered* — structure, and hence has a lower entropy. What has all this got to do with our knowledge of

a system? Simply, the less we know about the configuration of a gas, the more states it could be in, and the greater the overall W — and the greater the entropy. This gives us a nice intuitive feel of what is happening when we compress a gas into a smaller volume. Working isothermally, the momenta of the molecules within the container remains the same (δU=0), but each molecule has access to fewer possible spatial positions. The gas has therefore adopted a configuration with smaller W, and its entropy has decreased. As an aside, the Second Law of Thermodynamics tells us that in any isolated system:

$$\delta S \approx k\ \delta W/W \geq 0, \tag{5.10}$$

i.e. the entropy never decreases. The fact that the entropy of our compressed gas has dropped is a reminder that the system is not isolated — we have been draining heat into a heat bath. The heat flow into the bath increases its entropy, preserving the Second Law. Generally speaking, *the less information we have about a state, the higher the entropy.*

As the definition of entropy is essentially statistical, it is perfectly all right to define it for a gas with a single molecule, such as the one we have been considering, although there are a few subtleties (which we will avoid). You can see that if we compress the volume by a factor of 2, then we halve the number of spatial positions, and hence the number of configurations that the molecule can occupy. Before, it could be in either half of the box: now, it can only be in one half. You should be able to see in our probabilistic picture how this leads to a decrease in entropy by an amount:

$$\delta S = k \log 2 \tag{5.11}$$

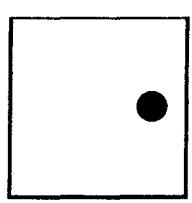
This is the same as we obtained with our work and free energy considerations.

We can now return to the topic of information and see where all this weird physics fits in. Recall the atomic tape with which we opened this section, in which the position of atoms in boxes tells us the binary bits in the message. Now if the message is a typical one, for some of these bits we will have no prior knowledge, whereas for others we will — either because we know them in advance, or because we can work them out from correlations with earlier bits that we have examined. We will *define* the information in the message to be *proportional to the amount of free energy required to reset the entire tape to zero*. By "reset to zero", we mean compress each cell of the tape to ensure that

its constituent atom is in the "zero" position.

Straightaway, we note what seems to be an obvious problem with this definition, namely, that it introduces an unnatural asymmetry between 0 and 1. If an atom is already in the zero part of the compartment, then surely the reset operation amounts to doing nothing, which costs no free energy. Yet if it is in the one position in the compartment, we have to do work to move it into the zero position! This doesn't seem to make sense. One would expect to be able to introduce an alternative definition of information for which the tape is reset to one — but then we would only seem to get the same answer if the message contained an equal number of ones and zeroes! But there is a subtlety here. Only if we *do not know* which side of the compartment the atom is in do we expend free energy. It is only in this circumstance that the phase space for the atom is halved, and the entropy increases. If we know the atom's position, then we expend *no* energy in resetting, irrespective of where the atom starts out. In other words, as one would hope, the information in the message is contained in the surprise bits. Understanding why this is so is worth dwelling on, as it involves a style of argument often seen in the reversible computing world. It seems a bit counter-intuitive to claim that the energy required to reset a one to a zero is no more than leaving a zero alone — in other words, nothing.

To clear this point up, I first have to stress the idealized nature of the set-up we are considering. Although I have talked freely about atoms in "boxes", these boxes are not real boxes made of cardboard and strung together, with mass and kinetic and potential energy. Moreover, when I talk about "energy", I certainly don't mean that of the tape! We are only interested in the content of the message, which is specified by the positions of the atoms. Let us suppose we have a message bit that we know is a one — the atom is on the right hand side — so we have the following picture:



We can show that to reset this to zero costs no energy in several ways. One pretty abstract way is to first slip in a little partition to keep the atom in place. All I have to do now is *turn the box over*. The end result is that we now have a zero on the right hand side (Fig. 5.3):

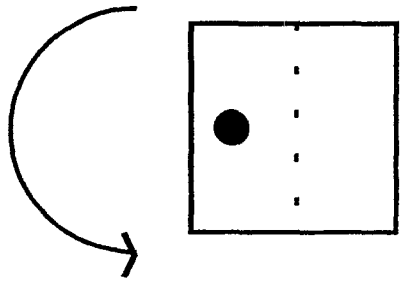


Fig. 5.3 A Simple Reset Procedure

This is abstract because it might seem odd to be able to insert pistons and turn boxes without expending energy. In the real world, of course, you can't — but we are dealing with abstractions here and, as I have said, we are not interested in the kinetic energy or weight of the "boxes". Given our assumptions, it is possible to do so, although the downside is that we would have to take an eternity to do it! (We will return to this sort of argument in §5.2.) Another way, perhaps a little less abstract, would be to introduce two pistons, one on each side of the box, and push the atom over with one, while drawing the other out (Fig. 5.4):

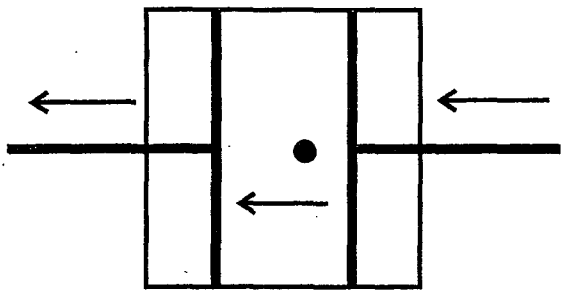


Fig. 5.4 A More "Realistic" Reset

Now the bombardment on the left is equal to that on the right, and any work put in at one end will be taken out at the other, and so is recovered. One could even join the pistons by an up-and-over rod, and you should be able to see that the tiniest touch on one piston will send the whole thing coasting over to its final position. So, if you do it slowly enough — "infinitesimal in the limit" — no work is done in resetting. Clearing, or resetting, the tape is what occurs when we don't know what compartment the atom is in. Then we must perform a compression, and this will take free energy, as we discussed earlier, as we are

lessening our ignorance of the atom's position.

Another way of looking at these ideas is due to Bennett, who suggests using a message tape as *fuel*, and relates the information in the tape to its fuel value – that is, to the amount of energy we can get from it. His idea, which is quite subtle, goes as follows. We suppose we have a machine, in contact with some kind of heat bath, which takes in tapes at one end, and spits them out at the other. We assume to begin with, that the tape the machine eats is blank, i.e. all of its atoms are in the zero state. We will show how such a tape can be used to provide us with useful work, which we can use to power our machine.

What we do is incorporate a piston into the system. As each cell comes in, we bring the piston into it, up to the halfway position in each box (Fig. 5.5):

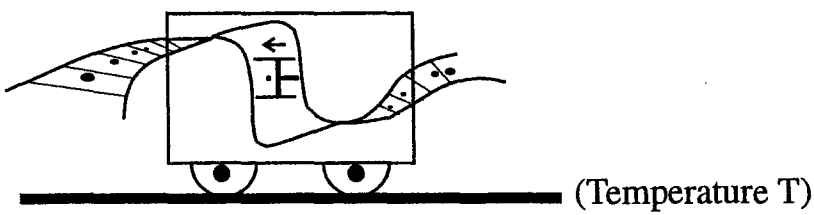


Fig. 5.5 An Information-driven Engine

We now let the heat bath warm the cell up. This will cause the atom in the cell to jiggle against the piston, isothermally pushing it outwards as in Figure 5.6:

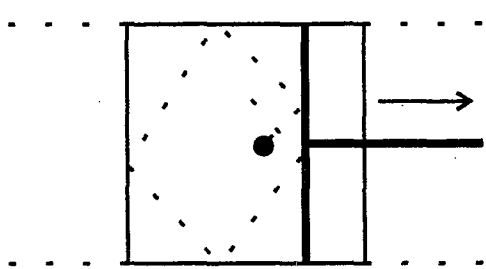


Fig. 5.6 Work Generation Mechanism in the Engine

This is just the opposite process to the compression of a gas we considered at the beginning of this section. The net result is that work is done on the piston which we can subsequently extract: in other words, we can get our tape to do work for us. You should be able to see that for a tape of n bits this work is equal to $nkT\log 2$, the free energy, where T is the temperature of the heat bath. An important consequence of our procedure is that the tape that the machine spits out has been *randomized*: after the piston has been pushed out, the atom that did the pushing can be anywhere in that cell, and we have no way of knowing where, short of performing a measurement.

We now generalize the argument by assuming that our piston is maneuverable. This allows us to extract work from tapes which have a 1 in them. If we get a 1, we switch the piston to the other side of the cell, bring it up to the edge of the 1 half, and proceed as before. Again we get $kT\log 2$ of useful work given out, and again the tape that emerges from the machine is randomized. What is crucial here is that we *know* what bit is about to enter the machine. Only then can we ready the piston to ensure that it does work for us. Obviously, if we left the piston in the 0 position, and we got a 1 in, we would actually have to do work to shift the atom into the 0 cell, and when the atom expands back into the full cell we would get that work back: that is, no useful work would be done. Clearly, *a random tape has zero fuel value*. If we do not know what bit is coming in next, we do not know how to set our piston. So we would leave it in one position, and just push it in and hope, push it in and hope, boom, boom, boom. Sometimes we would get lucky, and find an atom pushing our piston out again, giving us work; but equally likely, for a truly random message, we have to do work on the atom. The net result is zero work to power our machine.

Clearly, Bennett's tape machine seems to do the opposite to our reset process. He uses a message tape to extract work, ending up with a random tape: we took a random tape and did work on it, to end up with a tape of standard zeroes. This inverse relationship is reflected in the definition of information within Bennett's framework. Suppose we have a tape with N bits. We *define* the information, I, in the tape by the formula:

$$\textit{Fuel value of tape} = (N-I).kT \log 2. \tag{5.12}$$

From this we see that a tape giving us a full fuel-load — that is, $kT\log 2$ per bit — carries zero information. This is what we would expect since such a tape must have completely predictable contents. There is a nice physical symmetry

between these two approaches. If we run a message tape through the machine, we will be able to extract a certain energy E from it: this energy E will be precisely what we need to reset the newly randomized tape to its original form. It is, of course, up to you which picture you prefer to adopt when thinking about these things. I opt for the erasure picture partly because I do not like having to subtract from N all the time to get my information!

You might like to contemplate some problems on Dr. Bennett's machine.

Problem 5.1: Suppose we have two tapes: an N-bit random tape, and an exact copy. It can be shown that the fuel value of the two tapes combined is $NkT\log 2$. See if you can design a machine that will be able to extract this amount of energy from the two tapes. (Hint: you have to expand one tape "relative" to the other.)

Problem 5.2: We have a tape in which three bits are repeated in succession, say 110110110110... For a $3N$-bit tape, what is the fuel value? How do you get it out?

5.1.1: Maxwell's Demon and the Thermodynamics of Measurement

Those of you who wish to take your study of the physics of information further could do no better than check out many of the references to a nineteenth century paradox discovered by the great Scottish physicist James Clerk Maxwell. *Maxwell's Demon*, as it is known, resulted in a controversy that raged among physicists for a century, and the matter has only recently been resolved. In fact, it was contemplation of Maxwell's demon that partly led workers such as Charles Bennett and Rolf Landauer to their conclusions about reversible computing, the energy of computation, and clarified the link between information and entropy. Importantly, such research has also shed light on the role of *measurement* in all this. I will not go into the matter in great detail here, but supply you with enough tidbits to at least arouse your interest. A full discussion of the demon and of the attempts to understand it can be found in *Maxwell's Demon: Entropy, Information, Computing*, by H.S. Leff and A.F. Rex (Adam Hilger, 1990).

With Maxwell, we will imagine that we have a small demon sitting on a partitioned box, each half of which is filled by a gas of molecules with a random distribution of positions and velocities (Fig. 5.7):

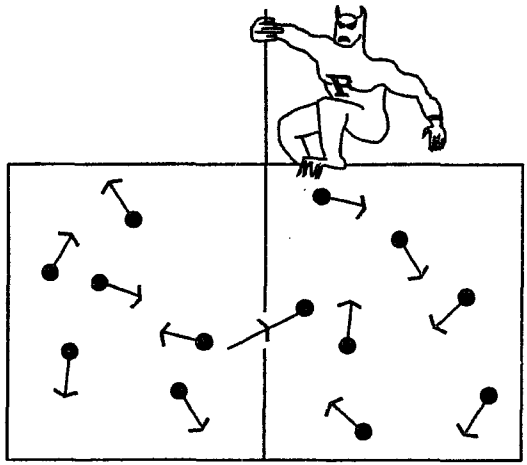


Fig. 5.7 Maxwell's Demon at Work

The demon has a very simple task. Set into the partition is a flap, which he can open and shut at will. He looks in one half of the box (say, the left) and waits until he sees a fast-moving molecule approaching the flap. When he does, he opens the flap momentarily, letting the molecule through into the right side, and then shuts the flap again. Similarly, if the demon sees a slow-moving molecule approaching from the right side of the flap, he lets that through into the side the fast one came from. After a period of such activity, our little friend will have separated the fast- and slow-moving molecules into the two compartments. In other words, he will have separated the hot from the cold, and hence created a temperature difference between the two sides of the box. This means that the entropy of the system has decreased, in clear violation of the Second Law!

This seeming paradox, as I have said, caused tremendous controversy among physicists. The Second Law of Thermodynamics is a well-established principle in physics, and if Maxwell's demon appears to be able to violate it, there is probably something fishy about him. Since Maxwell came up with his idea in 1867, many people have tried to spot the flaw in his argument. Somehow, somewhere, in the process of looking for molecules of a given type and letting them through the flap, there had to be some entropy generated.

Until recently, it was generally accepted that this entropy arose as a result of the demon's *measurement* of the position of the molecules. This did not seem unreasonable. For example, one way in which the demon could detect fast-moving molecules would be to shine a demonic torch at them; but such a

process would involve dispersing at least one photon, which would cost energy. More generally, before looking at a particular molecule, the demon could not know whether it was moving left or right. Upon observing it, however this was done, his uncertainty, and hence entropy, would have reduced by half, surely accompanied by the corresponding generation of entropy in the environment.

In fact, and surprisingly, Bennett has shown that Maxwell's demon can actually make its measurements with zero energy expenditure, providing it follows certain rules for recording and erasing whatever information it obtains. The demon must be in a standard state of some kind before measurement, which we will call S: this is the state of uncertainty. After it measures the direction of motion of a molecule, it enters one of two other states – say L for "left-moving", or R for "right-moving". It overwrites the S with whichever is appropriate. Bennett has demonstrated that this procedure can be performed for no energy cost. The cost comes in the next step, which is the *erasure* of the L or R to reset the demon in the S state in preparation for the next measurement. This realization, that it is the erasure of information, and not measurement, that is the source of entropy generation in the computational process, was a major breakthrough in the study of reversible computation.

5.1.2: Energy and Shannon's Theorem

Before leaving physics and information, I would like to return to something we studied in the previous chapter, namely, the limits on sending information down a channel. It will come as no surprise to you that we can revisit Shannon's Theorem with our physical tools too! Let us combine our study of the physics of information with our earlier work on errors. An interesting question is: How does the occurrence of an error in a message affect its information content? Let's start off with a message with all its M bits perfectly known, containing information N, and suppose that we want to send it somewhere. We're going to send it through a noisy channel: the effect of this is that, in transit, each bit of the message has a probability q of coming through wrong. Let us ask a familiar question: what is the minimum number of bits we have to send to get the information in the M bits across? We will have to code up the message, and in keeping with our earlier look at this question, we'll say the coded message has length M_C. This is the number of bits we actually send. Now we have said that to clear the tape, assuming we know nothing about its contents, we need to expend the following amount of free energy:

$$M_C kT \log 2. \tag{5.13}$$

However, some of this energy is taken up in clearing errors. On average, using our earlier derivations, this amount will be:

$$M_C kT \log 2[-q\log_2 q-(1-q)\log_2(1-q)]=[1-f(q)]M_C kT \log 2. \tag{5.14}$$

This energy we consider to be wasted. This leaves us with the free energy:

$$M_C kT \log 2 - [1-f(q)]M_C kT \log 2 = f(q)M_C kT \log 2 \tag{5.15}$$

to expend in clearing the message. By conservation of energy, then, and using our relationship between free energy and information, the greatest amount of information I can send through this channel will be:

$$M_C\ [q\log_2(1/q) + (1-q)\log_2(1/1-q)] \tag{5.16}$$

You can see how this kind of physical argument now leads us on to Shannon's result.

5.2: Reversible Computation and the Thermodynamics of Computing

It has always been assumed that any computational step required energy[2]. The first guess, and one that was a common belief for years, was that there was a minimum amount of energy required for each logical step taken by a machine. From what we have looked at so far, you should be able to appreciate the argument. The idea is that every logical state of a device must correspond to some physical state of the device, and whenever the device had to choose between 0 and 1 for its output — such as a transistor in an AND gate — there would be a compression of the available phase-space of the object from two

[2]Detailed accounts of the history of this subject can be found in the papers "Zig-zag Path to Understanding", R. Landauer, Proceedings of the Workshop on Physics and Computation Physcomp '94, and "Notes on the History of Reversible Computation", C.H. Bennett, IBM J. Res. Dev. 32(1), pp. 16-23 (1988). [Editors]

options to one, halving the phase-space volume. Therefore, the argument went, a minimum free energy of $kT\log 2$ would be required per logical step[3]. There have been other suggestions. One focused on the reliability of the computational step. The probability of an error, say q, was involved and the minimum energy was supposed to be $kT\log q$. However, recently this question has been straightened out. The energy required per step is less than $kT\log q$, less than $kT\log 2$, in fact less than any other number you might want to set — provided you carry out the computation carefully and slowly enough. Ideally, the computation can actually be done with *no* minimal loss of energy. Perhaps a good analogy is with friction. In practice, there is always friction, and if you take a look at a typical real-world engine you will see heat energy dissipated all over the place as various moving parts rub against one another. This loss of energy is ordinarily large. However, physicists are very fond of studying certain types of idealized engines, so-called Carnot heat engines in which heat energy is converted into work and back again, for which it is possible to calculate a certain maximum efficiency of operation. Such engines operate over a *reversible* closed cycle: that is, they start off in a particular state and, after one cycle of operation, return to it. The Second Law ensures that this cannot be done for zero energy cost but it is theoretically possible to operate such machines in such a way as to achieve the maximum efficiency, making the losses due to friction, for example, as small as possible. Unfortunately, they have to be run infinitesimally slowly to do this! You might, for example, want to drain heat from the engine into a surrounding reservoir to keep everything at thermal equilibrium, but if you operate the machine too quickly you will not be able to do this smoothly and will lose heat to parts of the engine that will simply dissipate it. But the point is that, in principle, such engines could be made, and physicists have learned much about thermodynamics from studying them. The crucial requirement is reversibility. Now it turns out that a similar idea works in computers. If your computer is reversible, and I'll say what I mean by that in a moment, then the energy loss could be made as small as you want, provided you work with care and slowly — as a rule, infinitesimally slowly. Just as with Carnot's engines, if you work too fast, you will dissipate energy. Now you can see why I think of this as an academic subject. You might even think the question is a bit dopey — after all, as I've said, modern transistors dissipate something like $10^8\ kT$ per switch — but as with our discussion of the limits of what is computable, such questions are of interest. When we come to design the Ultimate Computers of the far future, which might have "transistors" that are

[3] This is actually a lower limit far beneath anything practically realizable at present. Conventional transistors dissipate on the order of $10^8\ kT$ per step! [RPF]

atom-sized, we will want to know how the fundamental physical laws will limit us. When you get down to that sort of scale, you really have to ask about the energies involved in computation, and the answer is that there is no reason why you shouldn't operate below kT. We shall look later at problems of more immediacy, such as how to reduce the energy dissipation of modern computers, involving present-day transistors.

5.2.1: Reversible Computers

Let me return to the matter of "reversible computing". Consider the following special kind of computation, which we draw as a black box with a set of input and output lines (Fig. 5.8):

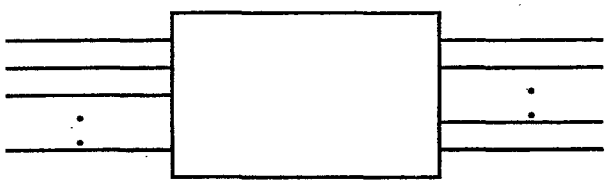


Fig. 5.8 A Reversible Computation

Suppose that for every input line there is one, and only one, output, and that this is determined by the input. (In the most trivial case, the signals simply propagate through the box unchanged.) In such circumstances, the output carries no more information than the input — if we know the input, we can calculate the output and, moreover, the computation is "reversible". This is in sharp contrast to a conventional logic gate, such as an AND (Fig. 5.9):

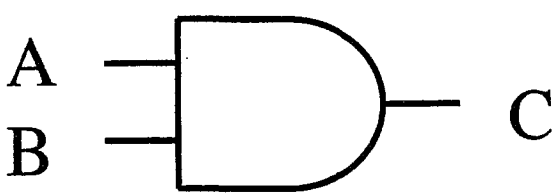


Fig. 5.9 The AND Gate

In this case we have two lines going in but only one coming out. If the output is found to be zero, then any one of three possible states could have led to it. I have irretrievably lost information about the input so the AND gate is irreversible. So too, is the OR gate (but not the NOT!). In other words, the phase space of the inputs has shrunk to that of the output, with an unavoidable

decrease in entropy. This must be compensated by heat generation somewhere. The mistake everyone was making about energy dissipation in computers was based on the assumption that logical steps were necessarily like AND and OR – irreversible. What Bennett and others showed was that this is not necessarily the case. The fact that there is no gain in information in our abstract "computation" above is the first clue that maybe there's no loss of entropy involved in a reversible computation. This is actually correct: reversible computers are rather like Carnot engines, where the reversible ones are the most efficient. It will turn out that the only entropy loss resulting from operating our abstract machine comes in resetting it for its next operation.

We can consider a "higher" kind of computer which is reversible in a more direct sense: it gives as its output the actual result of a computation plus the original input. That is, it appends the input data to the output data printed on its tape (say). This is the most direct way of making a computation reversible. We will later show that, in principle, such a calculation can be performed for zero energy cost. The only cost is incurred in resetting the machine to restart, and the nice thing is that this does not depend on the complexity of the computation itself but only on the number of bits in the answer. You might have billions of components whirring away in the machine, but if the answer you get out is just one bit, then $kT\log 2$ is all the energy you need to run things.

We actually studied some reversible gates earlier in the course. NOT is one, as I've said. A more complicated example we looked at was Fredkin's CONTROLLED CONTROLLED NOT gate (Fig. 5.10):

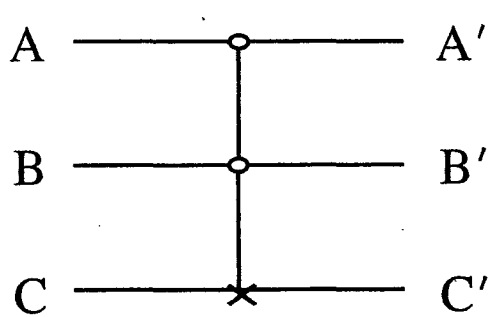


Fig. 5.10 The CCN Gate

in which the lines A and B act as control lines, leaving C as it is unless both are one, in which case C becomes NOT C. This is reversible in the sense that we

can regain our input data by running the output through another CCN gate (see section 2.3).

I would now like to take a look in more detail at some reversible computations and demonstrate the absence of a minimum energy requirement. I'll start with a computation that you might not ordinarily think of as a computation: the act of copying (recall our discussion of Turing copying machines, §3.5). This seems like a dumb sort of computation, as you're not getting anywhere, but it is a useful introduction to some of the ideas underlying issues of energy dissipation. It's not at all obvious that you can copy information down from one place to another without expending at least some energy, even in principle. Having said this, it is easy to suggest why it might not cost any energy. We can consider a set of data and its copy as two messages on tape, both identical. Either we know what the original message is, or we don't. In the first case, no free energy is expended in clearing the tape, and none need be for the copy tape: we just turn it over when necessary, as we discussed previously. In the second case, clearing the tape will cost free energy, but not for the copy: knowing what the first tape says, we can use this information to clear the copy by turning bits over again. Simply, there is no more information in the (data plus copy) set than is in just the single data set. Clearing the system should not, therefore, require more free energy in the first case than the second. This is a common type of argument in the reversible computing world.

5.2.2: The Copy Computation

Let us make these ideas a little more concrete. In a moment, I will examine a copying machine found in Nature, namely the RNA molecule found in living cells. But first, I will take a look at two rather artificial examples of copying machines. Our discussion follows Bennett.

We start with a very general copy process. We will have an original object, which we'll call the model, which can somehow hold a zero or one. It's some kind of bistable physical device. We want another object, which we'll call the copier, which can also hold a zero or one. An example of a bistable device would be one which could be modeled by the following potential well (Fig. 5.11):

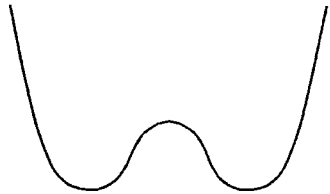


Fig. 5.11 A Potential Well

I will give one possible physical realization of this shortly. What this rather abstract diagram means is that some part of the device, which we will represent by a dot, can be in either of two stable states — here, in the left or the right trough, meaning one or zero, say. The curve displays the potential energy of the dot according to its position in the device. The troughs are the minima of this energy, and are favored by the dot: they are of equal depth, and are hence equally likely to be occupied at the outset. A useful way to think of this operation is to have the dot as a ball, and the curve an actual shape constraining it. Putting energy into the ball makes it move up and down the sides of its trough; enough energy and the ball will go over the hill and into the next trough — equivalent to our model changing its bit-state. The height of the hill, the amount of energy needed for the transition to occur, is called the barrier potential. In actual operation, we would want the typical thermal fluctuations of whatever it is the dot represents to be much less than this, to keep the device stable. Another way of visualizing this is to imagine the dot to be in a box separated into two halves by a partition. The barrier potential will be the energy required to get the dot from one half into the other.

We suppose both model and copier to be modeled by such a potential, and the model to be in some state. This can be random — we need not know what it is, but for sake of illustration let's say it is as shown in Figure 5.12 (where we have used an X for the model's dot):

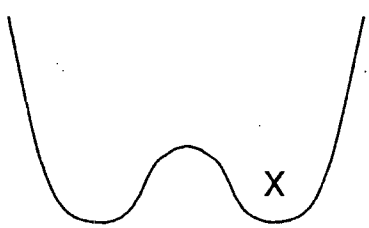


Fig. 5.12 Initial State of the Model

How does the copier start out? It must be in some standard state. It cannot be in a random state, because copying will involve getting it into a definite state, and to do this we must do work (compressing, if we use the box and partition analogy). Alternatively, you can use phase space considerations, comparing the number of possible model-copier options before copying (four, if the model is randomly set) and after (just two): this would be a logically irreversible step. Let's say the copier starts out in the state opposite to the model (Fig. 5.13):

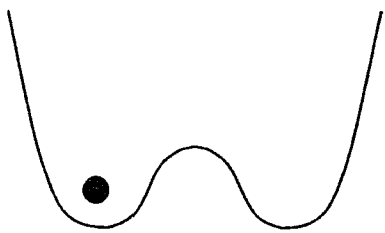


Fig. 5.13 Initial State of the Copier

Clearly, copying is going to involve somehow getting the dot from one trough to the other. To do this, we need to be able to manipulate the potential curve; we have to make the other trough energetically more favorable to the dot. We shall assume that there are two parameters associated with the copier that we can adjust: the barrier height, and the relative depths of the troughs. Furthermore, we assume that the depths of the troughs can be altered by some force of interaction between the copier and the model. (Don't worry if this is all horribly confusing and abstract! All will become clear.) We'll call this a "tilt" force, since it tilts the graph. We will combine these two operations to move the copier dot, but we will combine them in such a way — and this is important — that there will always be a unique minimum accessible to the dot at all times.

What we do is this. We start with the model some way away from the copier. Even at a distance it will exert a slight tilt force on the copier. We take this force to have the consequence of increasing the depth of whichever trough of the copier corresponds to that occupied in the model. The copier potential will hence be slightly distorted at the outset, as shown overleaf:

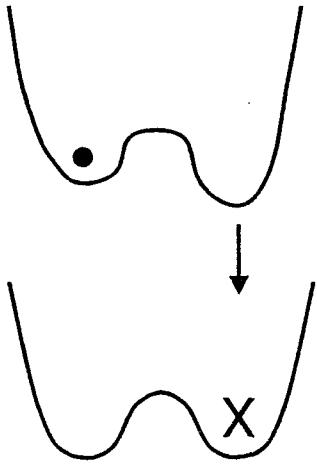


Fig. 5.14 Initial Copier Distortion

The first step in the copy process involves gently lowering the copier's potential barrier. This removes the obstacle to the dot switching positions: it can now wander over to the other bit state. What will make it do this? This is where the "tilt" from the model comes in. In step two, we slowly bring the model up closer to the copier, and in the process its tilt force increases. This gradually distorts the copier's potential even more, lowering the energy of the appropriate trough as shown in Figure 5.15:

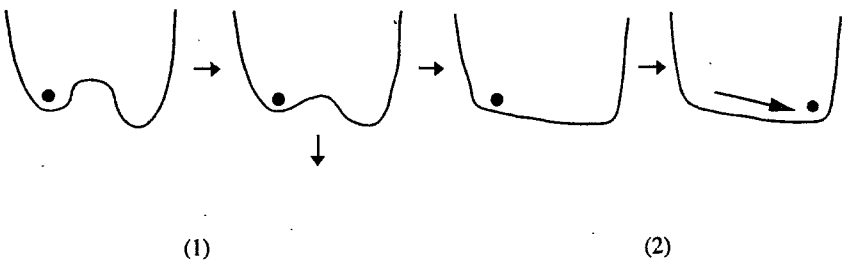


Fig. 5.15 Lowering the Potential Barrier and Tilting

The dot now slides smoothly down the potential curve, occupying the new, energetically more favorable trough. In step three, we replace the potential barrier to secure the dot in its new position, and finally, step four, we take the

model away, restoring the copier's potential to its normal state (Fig. 5.16):

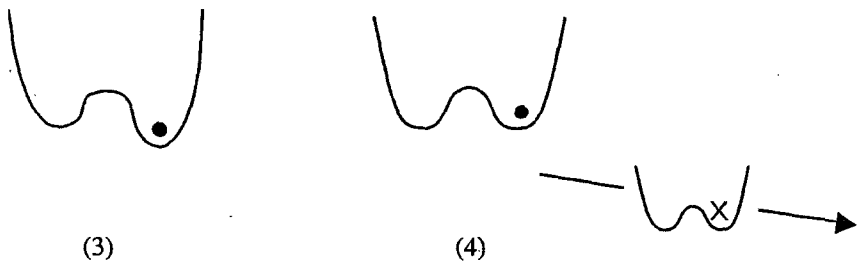


Fig. 5.16 Final State of the System

That is the basic idea of this copy machine. It's possible to play around with it further. For example, for appropriate physical systems, we can envisage bringing the model up to the copier in step one in such a way as that the tilt force lowers the state the dot is already in so that the dot is held steady while we lower the potential barrier, if this is a concern. The model is then moved over to the other side to provide the new tilt. This is one variation, but it does not significantly alter the basic idea.

The crucial thing about this process is that it needs to be carried out slowly and carefully. There are no jumps or sudden changes. The easiest way to get the dot from one trough to the other would be to bring the model up rapidly to bias the troughs in the desired way, then to rip away the potential barrier. The dot would then slosh over into its new trough, but the whole process, while nice and quick, would invariably involve dissipation in a real system. However, if the procedure is graceful enough, the lowering of the barrier, the tilting of the trough and the copying can be done for nothing. This is basically because the physical quantities that contribute to the energy dissipation — such as the kinetic energy of the dot moving to its new state, the work done in raising and lowering the barrier — are negligible under such circumstances. You should be able to see, incidentally, that this procedure will

work even if we don't know what state the model is in.

When Bennett discovered all this, no one knew it could be done, although much of the preliminary groundwork had been carried out by his IBM colleague, Landauer, as far back as 1961. There was a lot of prejudice around that had to be argued against. I see nothing wrong with his arguments. I was asked by Carver Mead at CalTech to look into the energy consumption of computers, so I looked at all this stuff and gradually concluded that there was no minimum energy. This was something of a surprise to me! Bennett's result was four years old by then but there were still people fighting over it. Also it's nice to work this sort of thing out for yourself: as I said in Chapter One — OK, you're not the first, but at least you understand it!

5.2.3: A Physical Implementation

Let me return to the preceding example and give you something that is essentially a physical realization of it. It is also fun to think about! We need some kind of bistable physical device, and here it is: two compass needles — just two magnetic dipoles on pivots. One end is North and the other South, and as we all know North attracts South and vice versa; otherwise we have repulsion. Now suppose that both the model and the copier are made up of such a pair. To make the analysis easier, we insist that the each member of a pair is linked to the other, in such a way that both members must point in the same direction. This means that we can analyze each system in terms of just one variable, the angle ϕ the needles make with the horizontal. So we have the allowed and disallowed situations shown below:

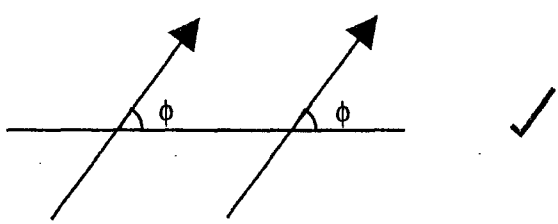


Fig. 5.17(a) Allowed Angular Configuration

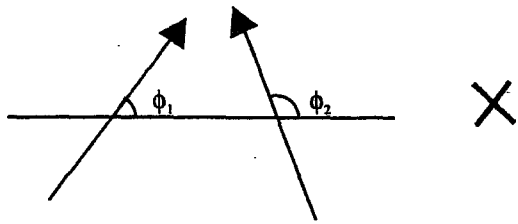


Fig. 5.17(b) Disallowed Angular Configuration

The disallowed case would, in any case, clearly be unstable. Now, not all alignments of the needles within a pair have the same potential energy. This is obvious by comparing the states shown in Figure 5.18:

(horizontal) S ⟶ N S ⟶ N with (vertical) N N ↑ ↑ S S

Fig. 5.18 Stable and Unstable States

The first is evidently quite stable, with the tip of one needle attracting the base of the other. The second, with both arrows vertical, is quite unstable: the North poles will repel, and the needles will seek to occupy the first state or its mirror image. We can actually calculate the potential energy for a state with angle ϕ. It is approximately (close enough for us) given by:

$$\textit{Potential energy} \approx \sin^2 \phi \tag{5.17}$$

This potential energy function looks like the graph of Figure 5.19:

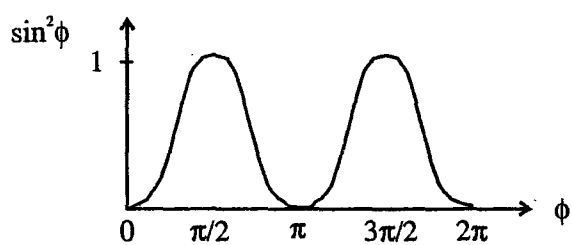


Fig. 5.19 Potential Energy as a function of ϕ

Note how similar this is to our abstract potential well. The minima are at $\phi = 0$ and $\phi = \pi$, corresponding to the stable "horizontal" states, whilst the maxima correspond to the vertical states at $\pi/2$ and $3\pi/2$. (Remember that the graph wraps around at $\phi = 0$ and 2π.) The system is clearly bistable and we can see that once the needles are in one of the two minima, energy would have to be expended to push them to the other.

To manipulate the barrier in this case, we introduce a vertical magnetic field B. It can be shown that this adds a term:

$$-B \sin \phi \tag{5.18}$$

to the potential energy. As we increase B, the effect is to lower the barrier between the 0 and π states as shown in Figure 5.20:

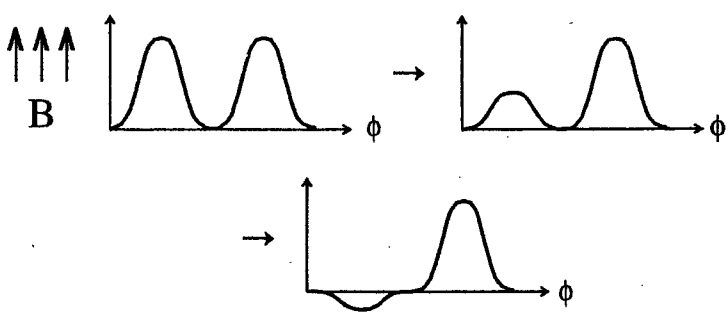


Fig. 5.20 Barrier Manipulation in the Dipole Copier

(You can play with numbers to gauge the exact effect of this.) The tilt force, as before, results from bringing the model closer to the copier; this time, we can see what it is about the model that causes this force — it is the magnetic field from the data bit. The force is perpendicular to B, and in the direction of the needles in the model. If we call it b, then it contributes:

$$-b \cos \phi \tag{5.19}$$

to the potential energy. This clearly removes the symmetry about $\pi/2$ and $3\pi/2$ and represents a tilting. We can now see how the copying process works. We start with the copier in a standard state, which we take to be the $\phi = 0$ state (→→). We gently turn up the field B — or alternatively slowly move the copier from a region of weak B to one of high B — until the barrier is removed. At this stage, the dipole is vertical (Fig. 5.21):

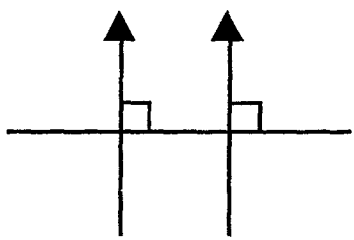


Fig. 5.21 Initial (Unstable) Copier State

Now we bring in the model. This has already been slightly perturbing the copier pair, but not enough to have a noticeable effect so far. Now, as it gets closer, its field biases the copier needles to flip over — but not suddenly! — into a new state. (This is if a new state is appropriate: if the standard state and the model state coincide, the needles will simply return to their original position.) The model is removed, the copier taken out of the field B to restore the barrier, and the copying is finished.

Once again, you can check that this copying method will work if we do not know what the model state is. It is not difficult to see that, if performed slowly, it will cost no energy — no current, no nothing. My previous discussion was to show you the principles; this specific example is probably easier to understand.

5.2.4: A Living Computer

The foregoing example of two dipoles has a certain physical basis, but is undeniably artificial. However, here's a copying process that really is found in

Nature and is one that involves thermodynamical, rather than mechanical, forces[4]. It occurs as one of the steps in the synthesis of proteins in a living cell. Now you probably know what proteins are — long, twisted molecular chains of amino acids (such as tryptophan or alanine) — and you may know how central they are both to the structure and functioning of living things. However, a proper understanding of the complex business that is their manufacture would require an understanding of biochemistry lying way beyond this course! I can't make up for that here, so I'll just try to give you enough background to let you see how the copying "machine" I have in mind behaves.

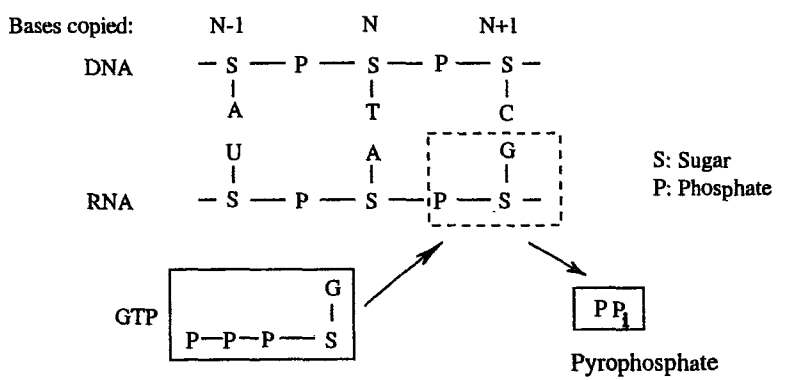
A living creature typically contains a huge number of different types of protein, each uniquely defined by some combination of specific amino acids. If the cell is to manufacture these molecules, then clearly a set of "design rules" for each protein-type must be available somewhere. This information actually resides in the DNA (Deoxyribonucleic Acid) molecule, the famous "double-helix" structure which resides in the cell nucleus. DNA comprises a double chain, each strand of which is made up of alternating phosphate and pentose sugar groups. To each sugar group is attached one of four bases, A (adenine), T (thymine), C (cytosine) and G (guanine) (a base-sugar-phosphate group is called a nucleotide). It is a certain sequence of bases that provides the code for protein synthesis.

We can break down the synthesis of proteins into two stages. The first stage, and it is only this which interests us, requires the formation of another, linear, strand of sugar phosphates with bases attached, called messenger RNA (or m-RNA). The code on the DNA is copied onto the RNA strand base by base (according to a strict matching rule, which I shall come on to), and the m-RNA, once completed, leaves the nucleus and travels elsewhere to assist in the making of the protein. The machine that does the copying is an enzyme called RNA polymerase. What happens is this. The DNA and enzyme are floating around in a crazy biological soup which contains, among other things, lots of triphosphates — such as ATP (adenosine triphosphate), CTP, GTP and UTP (U is another base, Uracil). These are essentially nucleotides with two extra phosphates attached.The polymerase attaches itself to whichever part of one of the DNA strands it is to duplicate and then moves along it, building its RNA copy base by base by reacting the growing RNA strand with one of the four nucleotides present in the soup. (A crucial proviso here is that RNA is built out of the four bases A, G,

[4] For a discussion of this topic in the literature, see C. Bennett, Int. J. Theor. Phys 21, pp. 905-940 [1982].[RPF]

C and U (not T), and that the RNA strand must be complementary to that on the DNA; the complementarity relationships are that As on the DNA must match with Us on the RNA, Ts with As, Cs with Gs and Gs with Cs). The nucleotides are provided in the triphosphate form, and during the addition process two of the phosphates are released back into the soup, still bound together (as a pyrophosphate). The nucleotide chosen must be the correct one, that is, complementary to the base on the DNA strand that is being copied. For example: suppose the enzyme, traveling along the DNA strand, hits a C base. Now at this particular stage of its operation a bonding between the polymerase and a GTP molecule from the soup happens to be more energetically favorable than one between it and UTP or ATP: complementarity is actually enforced by energy considerations. Preferentially, then, it will pick up a GTP molecule. It releases a pyrophosphate back out into the soup, moves along the DNA and looks for the next complementary nucleotide.

Schematically, we have the following picture (Fig. 5.22):



RNA (N bases) + (G-S-P-P-P) → RNA (N+1 bases) + P_i-P (pyrophosphate)

Fig. 5.22 Formation of m-RNA

Now the role of enzymes in biochemical processes is as catalysts: they influence the rate at which reactions occur, but not the direction in which they proceed. Chemical reactions are reversible, and it would be just as possible for

the polymerase reaction to go the other way — that is, for the enzyme to undo the m-RNA chain it is working on. In such an event, it would extract a pyrophosphate from the surrounding soup, attach it to a base on the m-RNA, and then release the whole lot back into the environment as one of our triphosphates. The enzyme could just move along the wrong way, eat a G, move along, eat a C, move along,.., undoing everything[5]. Which way the reaction goes depends on the relative concentrations of pyrophosphates and triphosphates in the soup. If there is a lot of ATP, GTP, and so on, but not much free pyrophosphate, then the rate at which the enzyme can run the reaction backwards is lowered, because it can't find much pyrophosphate with which to pull off the m-RNA nucleotides. On the other hand, if there is an excess of free pyrophosphates over triphosphates, the reaction will tend to run the wrong way, and we'll be uncopying and ruining our copy.

We can actually interpret these relative concentrations in terms of the number of possible states available to our system at any given computational point. If there are plenty of triphosphates around, then there are plenty of forward-moving, and comparatively few backward-moving, states available: the RNA polymerase will tend to enter the former state, in the process lowering its entropy. The difference in free energies, measured by the differing concentrations, determines the way it goes. If we get the concentrations just right, the copier will oscillate forever, and we will never get around to making copies. In an actual cell, the pyrophosphate concentration is kept low by hydrolysis, ensuring that only the copying process occurs, not its inverse. The whole RNA polymerase system is not particularly efficient as far as energy use goes: it dissipates about $100kT$ per bit. Less could be wasted if the enzyme moved a little more slowly (and of course, the reaction rate does vary with concentration gradient), but there has to be a certain speed for the sake of life! Still, $100kT$ per bit is considerably more efficient than the 10^8kT thrown away by a typical transistor!

To reiterate: The lesson of this section is that there is no absolute minimum amount of energy required to copy. There *is* a limit, however, if you want to copy at a certain speed.

---

[5]Bennett has nicely christened machines like this "Brownian computers" to capture the manner in which their behavior is essentially random but in which they nevertheless progress due to some weak direction of drift imposed on their operation. [RPF]

5.3: Computation: Energy Cost versus Speed

The question of speed is important and I would like to write down a formula for the amount of free energy it takes to run a computation in a finite time. This at least makes our discussion a bit more practical. There is little room for reversible computing in the computer world at the moment, although one can foresee applications that are a little more immediately useful than getting from $10^8 kT$ to under kT. (You can actually get to 2 or 3 kT irreversibly, but you can't get under this.) For example, we can look at the problem of errors arising in parallel processing architectures where we might have thousands of processors working together. The question of error correction through coding in such a situation has arisen and is unsolved. It occurs to me that maybe the devices in the machine could all be made reversible, and then we could notice the errors as we go. What would be the cost of such reversible devices? Maybe these things will find application soon. That would make this discussion more practical to you and since computing is engineering you might value this! In any case, I shouldn't make any more apologies for my wild academic interest in the far future.

An example we gave of reversible computing was that of the chemical process of copying DNA. This involved a machine (if you like) that progressed in fits and starts, going forward a bit, then backwards, but more one than the other because of some driving force, and so ended up doing some computation (in this case, copying). We can take this as a model for more general considerations and will use this "Brownian" concept to derive a formula for the energy dissipation in such processes. This will not be a general formula for energy dissipation during computation but it should show you how we go about calculating these things. However, we will precede this discussion by first giving the general formula[6], and then what follows can be viewed as illustration.

Let us suppose we have a reversible computer. Ordinarily, the free energy expended in running it reversibly will be zero, when the process is infinitesimally slow, but let us suppose that we are actually driving it forward at a rate r. In other words, at any given stage, it is r times more likely to make a forward calculational step than a backwards one. Then, the general result is that minimum energy that must be expended per computational step in the process is:

[6]This rule is pretty general, but there will be exceptions, requiring slight corrections. We will [illegible] "ballistic" computer, in §5.5. [RPF]

$$kT \log r. \tag{5.20}$$

Note that the smaller r is, the lower the energy.

Let us illustrate this rule by looking at a Brownian-type computer. Imagine we have a system, or device, in a particular state, which has a particular energy associated with it. It can go forwards or backwards into a new state, each transition corresponding either to doing a computation (forward) or undoing it (backward). We can model this situation using the energy level diagram of Figure 5.23:

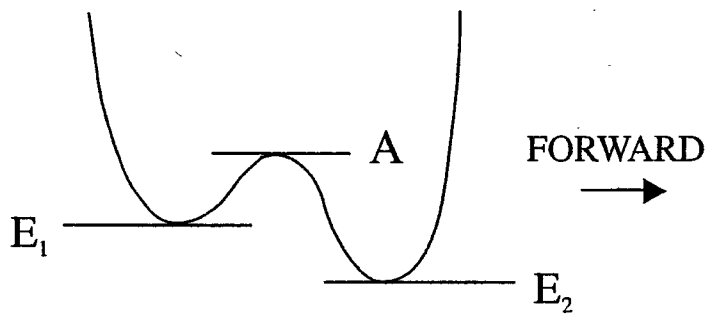


Fig. 5.23 The General Transition

We assume our computer to be sitting in one of the two states, with energy E_1 or E_2. These energies will not generally be equal. Now our device can go from E_1 to E_2, a forward step – the idea is that the energies are lower in the direction of computation – or from E_2 to E_1, a backward step. The energies of the two states might be equal, but one of them could be effectively lowered by the imposition of an external driving force. We have introduced into this diagram the "activation energy" A, which is the energy that must be supplied to the system to cause a transition of any kind. We will focus on the effects of thermal fluctuations which will, quite randomly, cause the computer to move between states, whenever the energy of these fluctuations exceeds A. Such fluctuations can make the device go either way, and we can calculate the rate at which it goes in either direction. These will not be equal. Roughly, the chance of the system going into the state with energy E_i is the chance that by accident it acquires enough energy to get past the barrier (that is, A) and into E_i. Clearly, the energy needed to get from E_1 to E_2, a forward step, is $(A\text{-}E_1)$, while to get from E_2 to E_1 it is $(A\text{-}E_2)$. It is a standard result in statistical mechanics that the

probability of a transition from one state to another differing in (positive) energy δE is:

$$C \exp(-\delta E/kT), \tag{5.21}$$

where C is a factor that carries information about the thermal fluctuations in the environment. This can be calculated through a phase-space (entropy-type) analysis, examining the probabilities of ensemble transitions between states. However, we are interested in the transition rates between states and this is describable by a similar formula. We simply have to insert another factor, say X, giving us:

$$\textit{forward rate} = CX \exp[-(A-E_1)/kT] \tag{5.22}$$

and

$$\textit{backward rate} = CX \exp[-(A-E_2)/kT]. \tag{5.23}$$

The factor X depends on a variety of molecular properties of the particular substance (the mean free path, the speed, and so on), but the property that interests us is that it does not depend on E (consider the transition rates for the case $E_1 = E_2$). We can therefore write for the ratio of the forward to backward rates:

$$\exp[(E_1-E_2)/kT]. \tag{5.24}$$

This depends only on the energy difference between successive states. This gives us some insight into the rate at which our computation (= reaction) proceeds, and the energy difference between each step required to drive it. The bigger the energy difference E_1 - E_2, the quicker the machine hops from E_1 to E_2, and the faster the computation.

We can tie this result into our earlier general formula by setting the above rate equal to r. We then have, for the energy expended per step:

$$kT \log r = E_1-E_2 \tag{5.25}$$

which makes sense.

Let me give you one more illustration of driving a computer in a particular direction. This time we will look at computational states that do not differ in their energy, but in their availability. That is, our computer selects which state of a certain kind to go into next, not on the basis of the energy of the state, but on the number of equivalent states of that kind available for it to go into. We have an example of such a situation in our DNA copier. A calculational step there involved the RNA enzyme attaching bases to the RNA chain and liberating pyrophosphates into the surroundings. The inverse step involved taking up phosphates from the solution and breaking off bases. Each step is energetically equivalent but one can be favored over the other, depending on the relative concentrations of chemicals in the soup. Suppose there is a dearth of phosphates but a wealth of bases available. Then, the number of (forward) states of the system in which a base is attached to the RNA strand and a phosphate is released — and we consider all such states equivalent — exceeds the number of states in which a phosphate is grabbed and a base released (again, all such states we take to be the same). So we can envisage a computer designed so that it proceeds by diffusion, in the sense that it is more likely to move into a state with greater, rather than lower, availability. Schematically, we have the situation shown in Figure 5.24:

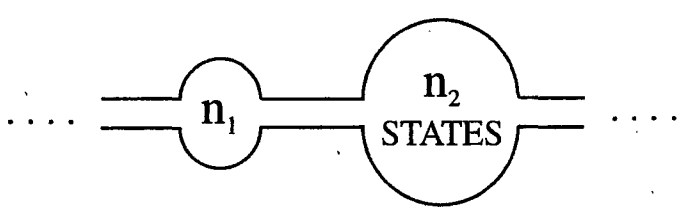


Fig. 5.24 The Availability of States

where n_i is the number of available states. It is possible to show in this situation (although it takes a litle thought) that the ratio of the forward rate to the backward rate is:

$$r = n_2/n_1. \tag{5.26}$$

If you recall, we defined the entropy of a configuration of a system to be:

$$S \approx k \log W \tag{5.27}$$

where W is the probability of finding the system in that configuration. Hence we may write:

$$kT \log r \approx kT (\log n_2 - \log n_1) = (S_2 - S_1)T \tag{5.28}$$

(with various constant factors canceling to leave the equality). In other words, for this process the energy loss per step is equal to the entropy generated in that step, up to the usual temperature factor. Again, this makes sense.

So we can see that our general formula reduces to the specific formulae we have obtained in these instances. An interesting question that arises is: in a real world situation, can we minimize the energy taken per computational step? We know that if we have an effectively reversible computer, the chances of forward and backward movement are equal, and we have no energy loss. The price we pay for this is that a computation will take an infinite time. We will never know when we're finished. So as we've said, to get it going we want to give things a tug, lower the energies of successive steps, make them more available, or whatever. Let us suppose that we have the forward rate, f, just a little bigger than the backward rate, b, so the computation just goes. We write:

$$f = b + \Theta \tag{5.29}$$

where Θ is small. Our general formula now gives:

$$\textit{energy per step} = kT \log [1+(\Theta/b)] \approx kT\Theta/b = kT(f-b)/b \tag{5.30}$$

for small Θ. We can provide a nice physical interpretation of this expression, although at the cost of mathematical inaccuracy. We replace the formula above by one that is nearly equal to it:

$$\textit{energy per step} = kT \frac{(f-b)}{(f+b)/2}. \tag{5.31}$$

This differs from the original formula by terms of order Θ^2. Now the numerator of this fraction is the speed at which we go forward and do the calculation. It is a bit like a velocity, in that it represents the rate at which the computer drifts through its calculation, measured in steps per second. The denominator is the average rate of transition — it is a measure of the degree to which our computer

is oscillating back and forth. We can interpret this roughly as the fastest speed at which you could possibly go, backwards or forwards, which would be the speed found if the computer underwent a series of steps in one direction with no reverses: it is the greatest possible drift. So we can write, approximately:

$$\textit{energy loss per step} = kT\,\frac{v_{drift}}{v_{max}}. \tag{5.32}$$

Alternatively, we can emphasize time as our variable and write:

$$\textit{energy loss/step} = kT\frac{\textit{minimum time taken/step}}{\textit{time/step actually taken}} \tag{5.33}$$

Let us now take a look at more general issues in reversible computing.

5.4: The General Reversible Computer

We have repeatedly stated that, if a computation is to be reversible, then we have to store a lot of information that we would ordinarily lose or throw away in order that we can subsequently use it to undo something. The logic gates of such a machine give us not just the answer to the logical calculation we want, but a whole lot of extra bits. A simple illustration of this for a realistic gate is a simple adder built out of reversible gates. In §2.3 I set as a problem for you, the construction of a full three-bit adder from reversible gates — specifically, using CN and CCN gates (or alternatively, just CCN gates, out of which all others can be built). An easier example, the simple two-bit adder, is built as follows:

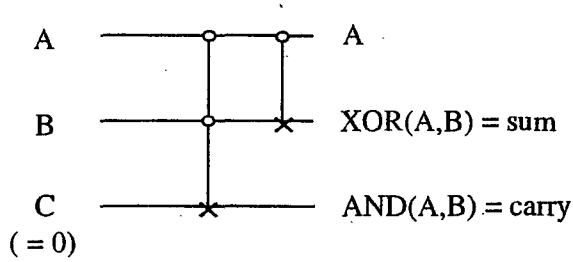


Fig. 5.25 A Two-bit Adder From Reversible Gates

The C-input is kept set at zero (the full three-bit adder requires the addition of a fourth input line, kept set at zero). As well as the sum and carry of $A+B$, we

find this gate feeds the A-line through. We can see that this bit is necessary if we are to be able to reconstruct the input (A,B) from the output. If you look at the three-bit adder, you will find two spare bits at the output. Generally, then, we will always need a certain amount of junk to remind us of the history of the logical operation. We can summarize the main constraint on reversible gates as follows: it is obvious that, when running a computer forward, there must be no ambiguity in the forward step — if you have a "goto", you have to know where to go to. With a reversible machine, there cannot be any ambiguity in *backward* steps either. You should never have a situation where you do not know where to go back to. It is this latter feature that makes reversible computing radically different from ordinary, irreversible computing.

We can, following Bennett, consider the most general computational process, and also answer a criticism leveled at advocates of reversible computing. Let us suppose we have a system of (reversible) logic units tied together, and we put into it some input data. We also have to feed in a set of "standard" zeroes, the bits that are kept set at particular values to control the reversible gates. (If we want a "standard" one instead of a zero, we can just NOT one of the zeroes: this is reversible, of course!) The logic unit will do its business — dup, dup, dup — and at the end we will find an output — the answer we want plus a pile of garbage bits, forming the history tape. This is shown in Figure 5.26 below:

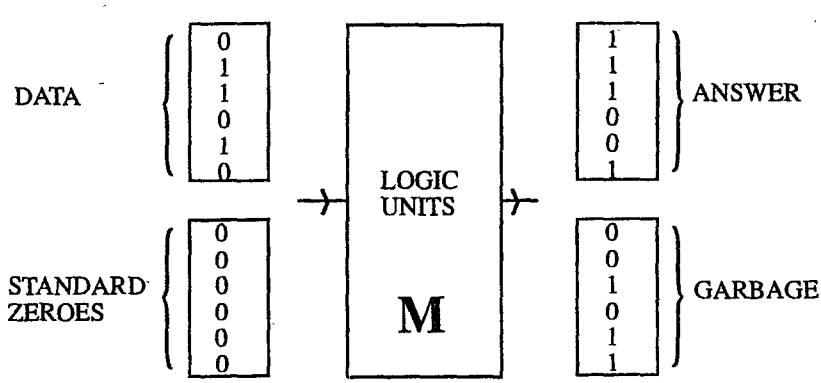


Fig. 5.26 The General Reversible Computation

Now this picture makes it look like you start up with a blank tape (or a preset one) and end up with a lot of chaos. Not surprisingly, everyone said that was where the entropy was going: "This randomization of zeroes is (in Bennett's picture) fueling the running of your machine. How can keeping this data make your computation practically reversible? It's rather like claiming that you can make an irreversible heat engine reversible by keeping the water that all the heat has gone into, rather than throwing it away. If you don't throw the water away, sure you have all the information you need about the history of the system, but that hardly means the engine is going to be able to run backwards, reversing the motions of water molecules!" In the thermodynamic case, that would indeed be silly. But it isn't so for computing. By adding one more tape to the system, and feeding the results through another machine, we can bypass this difficulty (Fig. 5.27):

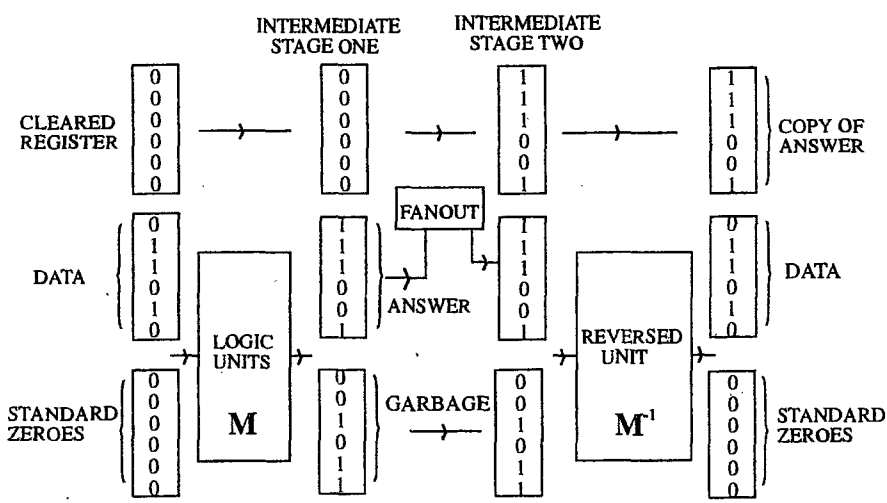


Fig. 5.27 A Zero Entropy Loss Reversible Computer

Let us try to make sense of this! The new logic unit that we have added is the reverse of the original (hence we have labeled it M^{-1}) and is also reversible. M^{-1}

is such that if we feed the output of **M** through this, it undoes all the work on it and feeds us back the original inputs to **M**. The new tape is a cleared register which we will use to copy the answer to our computation. We begin as before, feeding into **M** the input data and the standard bits for control. **M** gives us an output and a history tape (marked garbage in the diagram). The history tape we feed directly into $\mathbf{M}^{-1}$. We also feed the data output tape in. However, before we do this we make a copy of it onto the cleared register. We have shown this schematically as a fanout, but this actually represents a copy process (which is, of course, a reversible operation).

The reverse machine $\mathbf{M}^{-1}$ now undoes all the work done by **M**, producing as its output the standard bits and the input data. At the end of the whole process we are left with the answer to the computation, plus an exact copy of the inputs we started with. So our grand machine has done a calculation for no entropy loss (ideally — in practice we would have to drive the system a little as discussed) and reversible computing really can save us work. Of course, there will be an energy loss when we wipe our tapes clean to do another calculation.

Reversible computing is quite a strange concept for those used to thinking in classical Boolean terms, so let me suggest a few problems for you to work on to help you become more comfortable with the ideas.

Problem 5.3: Suppose a reversible computer is carrying out a calculation and it needs to execute a subroutine. So it gets sent off to some other place to execute a compact set of instructions. Now these instructions must be reversible, as are the basic computing elements, and so there is a chance that once we are into our subroutine we might find ourself running backward. It might even happen that we get back to the start of the routine — and then have to re-enter the main body of the program where we left it! The question is: Given that this same subroutine might be used several times throughout the computation, how does the machine know where to return to when this reverse happens? You might like to think about this. Somehow you have to have a number of memory stacks to keep track of where you have to go to find the subroutine, but also where to go back to should you reverse. This is your first problem in reversible computing — how to handle subroutines.

Problem 5.4: A related problem concerns how to get "if" clauses to work. What if, after having followed an "if... then..." command, the machine starts to reverse? How can the machine get back to the original condition that dictated which way the "if" branched? Of course, a set of initial conditions can result in a single "if" output ("if $x = 2, 3, 4$ or 6.159 let $F = d$"), so this condition may not

be uniquely specified. Here is a nice way to analyze things. Simply bring in a new variable at each branch, and assign a unique value to this variable for each choice at a branch point. You might like to work this through in detail.

Problem 5.5: A simple question to ask about a general reversible computer is: How big a history tape do we need? The gates we have considered so far have had the number of outputs equal to the number of inputs. Is this always necessary for reversibility? As far as I know, this question hasn't even been asked by theorists. See if you can work it out. Certainly the minimum has something to do with the number of possible inputs that the output could represent, and we'll apparently need a number of bits to keep track of that (on top of the actual outputted answer). So the questions are: firstly, what is the minimum number of bits needed to keep a gate reversible in principle, and secondly, could we actually accomplish it?

5.5: The Billiard Ball Computer

To give you a demonstration of a reversible computer that can actually do calculations, I am now going to discuss an ingeniously simple machine invented by Fredkin, Toffoli and others. In this device, the movement of billiard balls on a plane is used to simulate the movement of electronic signals (bits) through logic gates. We fire balls into the machine to represent the input, and the distribution of balls coming out gives us our output. The balls all move diagonally across a planar grid and obey the laws of idealized classical mechanics (i.e zero friction and perfectly elastic collisions). To introduce you to the basic idea, examine the following diagram (Fig. 5.28):

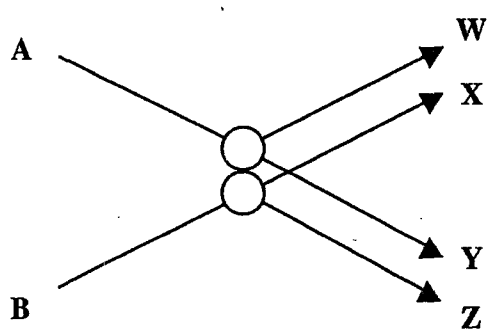


Fig. 5.28 The Basic Two-ball Collision Computation

This illustrates how a two-ball collision realizes a two-input four-output logic function. The data to this gate is represented by the presence of a ball at a particular position (1) or its absence (0). For example, the gate has two input channels, *A* and *B*. If we fire a ball in at *A*, then the input at *A* is binary 1. If there is no ball, it is zero. Similarly with *B*. If we find a ball coming out at point *X*, this means output *X* is 1, and so on. There are four possible input states, and for each we use basic mechanics to calculate the configuration of balls coming out of the device. There are four possible outputs, two corresponding to one input ball being absent and the other going straight through, and two corresponding to a collision.

Let us suppose there is no ball at *A*. If there is a ball at *B*, it will continue on through the "machine", coming out at *X*. We can see that we will only get a ball at *X* if there is no ball at *A and* one ball at *B*. In logic terms, *X* is 1 if and only if *B* is 1 and *A* is 0, so we have:

$$X = B \text{ AND NOT } A \tag{5.34}$$

Similarly, we find that:

$$Y = A \text{ AND NOT } B \tag{5.35}$$

Output *W* is a little trickier. We will find a ball there only if there is a ball at both *A* and *B*. Likewise for output *Z*. Hence, both *W* and *Z* realize the same AND function:

$$W, Z = A \text{ AND } B \tag{5.36}$$

Let us summarize this with some fancy notation (Fig. 5.29):

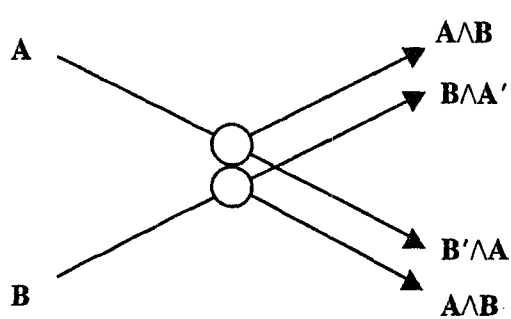


Fig. 5.29 Logical Structure of the Basic Collision Computation

This is the fundamental collision of this billiard ball computer and you can see how neatly the logic element drops out of it. We can build other logic functions besides AND with this gate. For example, we can use it to make a FANOUT. If we set $A = 1$ (the billiard equivalent of a control line set to "on") and take our output from W and Z, then clearly this has the effect of branching our B input: a ball at B produces one at each of W and Z; no ball at B leaves both outputs blank. You can also make a CN gate with this unit (try it). However, by itself, the basic collision gate will not make enough elements to build a whole computer — we'd be stuck with pairs of balls going along two lines, and we could never change anything! How do we reroute balls? We have to introduce two fundamental mechanical devices. The first, which you would never invent if you were a logician, as it seems a damn silly thing to do, I'll call a collision gate; in this device, two balls go in, but four come out (Fig. 5.30):

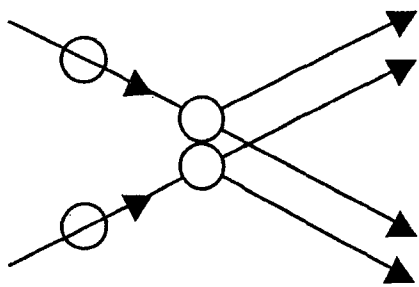


Fig. 5.30 The Collision Gate

This is a sort of all-in "double-FANOUT" process, which we achieve by letting the two incoming balls collide with two stationary ones. (You might find it an interesting exercise to consider the energy and momentum properties of this gate.) The second and more important device is a redirection gate. This is just a mirror to reflect a ball. It can be oriented any way you wish, although we restrict ourselves to four possibilities (Fig. 5.31):

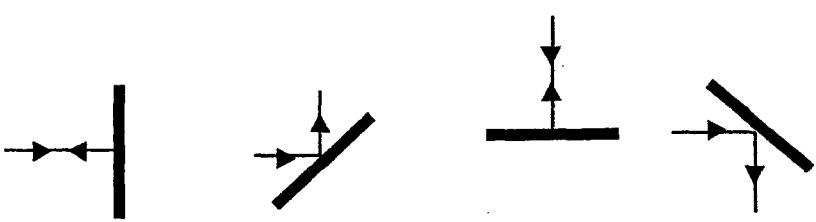


Fig. 5.31 Four Redirection Gates

Mirrors enable us to do a lot of things. For example, we can use mirrors to construct a "crossover" device (Fig. 5.32):

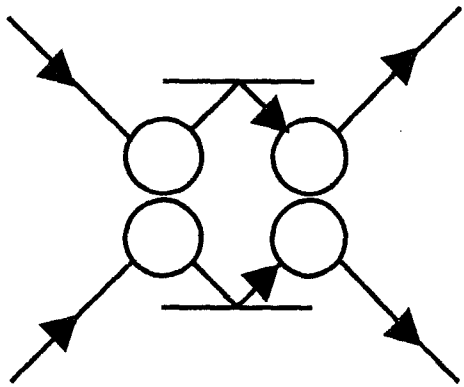


Fig. 5.32 A Crossover Device

Incidentally, this device tells us something important about the balls, namely, that they are indistinguishable. We do not tell them apart, and are interested only in their presence or absence. The above crossover device actually switches the incoming balls, but as we can't tell them apart, it looks as if they just pass each other by. Note that if one ball is missing, the other just sails right through.

To show you the sorts of thing you can build with these basic structures, I will first give you a unit that acts as a switch (Fig. 5.33):

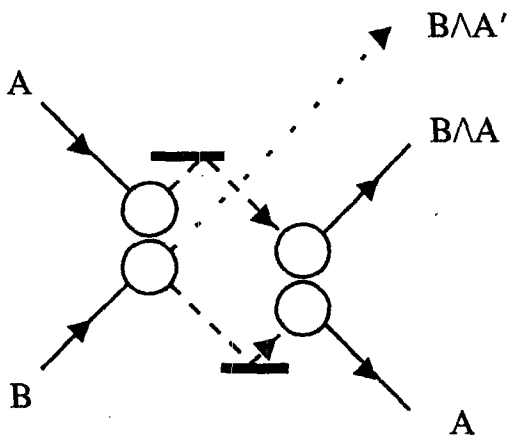


Fig. 5.33 A Switching Device

This is a sort of offset crossover. Note that, irrespective of whether or not there is a *B* input, the lower right output is always the same as *A*. This is a "debris" bit, corresponding to the control line fed through the gate. Of course, we are used to such outputs by now.

A question that arises in the context of this chapter is obviously: "OK, show me how to make reversible gates with all these mirrors and balls." Specifically, can we build, say, a CN gate? The answer is that we can, and a CCN gate too if we like. However, it is more enlightening to build a Fredkin, or controlled exchange gate. This is because it is possible to build everything we could want, just out of Fredkin gates! I'll remind you of what such a gate is (Fig. 5.34):

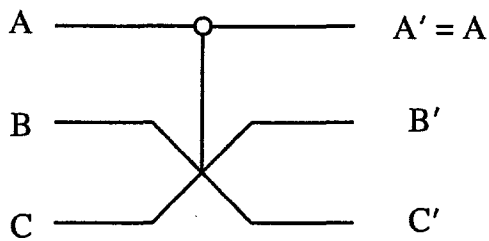


Fig. 5.34 The Fredkin Gate

Line *A* goes through unchanged. This is true of *B* and *C* also, if *A*=0; but if *A*=1, *B* and *C* switch. I won't leave building a Fredkin gate as an exercise. It is constructed from four switching devices of the kind depicted in Figure 5.33, put together with considerable ingenuity as shown in Figure 5.35:

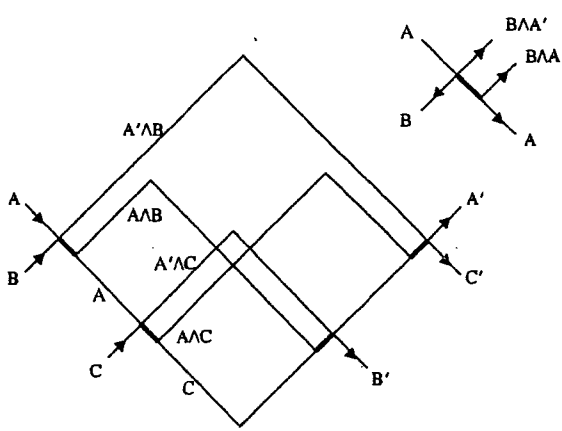


Fig. 5.35 The Fredkin Gate Realized by Billiard Ball Gates

Obviously, there is no point in making a computer like this except for fun. However, it does show how profoundly simple the basic structure of a machine can be.

Now anybody who is familiar with bouncing balls knows that if there's a slight error, it is rapidly magnified. Suppose you have a ball on a table and you drop another onto it from above, right in the middle. You might think: "Oh, it'll go straight down, then straight up, and so on." Everybody has an intuition about this but if you played with balls as a baby you know that you can't bounce one ball on another. It doesn't work! What happens is that as soon as the ball bounces ever so slightly wrongly, the next bounce is further out and the ball comes down slightly more cock-eyed. When it comes down next time, it is further out still and hits the lower ball in an even more glancing fashion. Next time, the balls will probably miss altogether.

The reasons for this are not hard to fathom. Although at the macroscopic level, balls seem stable and solid, at the microscopic level, they are a seething mass of jiggling molecules. Thermal oscillations, statistical mechanical fluctuations and whatnot, all contribute corrections to the naive collision of ideal balls. In fact, even the tiniest effects of quantum mechanics get in the way. According to the Uncertainty Principle, we cannot know both the precise location and momentum of a ball, so we cannot drop one perfectly straight. Suppose we have two ideal 1cm balls, and we drop one onto the other from a height of 10cm. How many bounces can we get away with before, according to quantum mechanics, things *have* to go awry? We can actually calculate this and the answer is about seventeen bounces. Of course, in reality the disturbances from more classical phenomena are far more significant and we would never get anywhere near this quantum limit. Don't forget, even your hand will be shaking from Brownian motion!

So surely the billiard ball machine idea is nonsense? We may not be dropping balls from a height, but we are colliding them and we would therefore expect errors to accrue just as inevitably. So how can we claim to have a physically implementable reversible computer? After all, all you have to do is give me an error per collision, and I will tell you how long you have before the machine falls apart. 10^{-3}? Five minutes. 10^{-6}? OK, ten minutes. It looks completely hopeless. In order to get this system to work, we have to find some way to keep straightening out the balls. Perhaps we could put them in troughs, guiding them in some way. But if you put a ball in a trough it'll slosh back and forth, getting worse and worse, unless there are losses — absorption, resistance, dissipation. Even if we design our troughs to cope with these difficulties,

inevitably energy will be lost because of friction in the trough. We would have to pull the balls through to drive the machine. Now if you drive it just a little, you can find that the energy required to drive it is a minimum of the ratio:

$$kT \frac{\textit{time to make collision}}{\textit{speed at which it happens}}. \tag{5.37}$$

This expression has not been analyzed in any great detail for the billiard ball machine.

5.6: Quantum Computation

The billiard ball computer operates chiefly according to the laws of classical mechanics. However, inspired by the questions it brings up, people have asked me (and others have thought about this too[7]): "What would the situation be if our computer is operating according to the laws of *quantum* mechanics?" Suppose we wanted to make extremely small computers, say the size of a few atoms. Then we would have to use the laws of quantum mechanics, not classical mechanics. Wouldn't the Uncertainty Principle screw things up? Not necessarily. I will wind up this chapter by briefly considering what may become the computers of the future — quantum computers.

We are asking yet another question about absolute limitations! This time, it is: "How *small* can you make a computer?" This is one area where, I think, I've made a contribution. Unlike an airplane, it turns out that we can make it pretty much as small as we want. There will be engineering details about wires[8], and we will have to find a way of magnifying outputs and whatnot, but we are here discussing questions of principle, not practicality. We cannot get any smaller than atoms[9] because we will always need something to write on,

[7]Notably the physicist Paul Benioff (see, for example, "Quantum Mechanical Models of Turing Machines that Dissipate No Energy", Phys. Rev. Lett. 48, pp. 1581-1585 [1982]). [Editors]

[8] It is interesting to note that most computer theorists treat wire as idealized thin string that doesn't take up any room. However, real computer engineers frequently discover that they just can't get enough wires in! (We'll return to this in Chapter Seven.) [RPF]

[9]I am not allowing for the possibility that some smart soul will build a computer out of more fundamental particles! [RPF]

but all we actually need are bits which communicate. An atom, or a nucleus will do since they are natural "spin systems", i.e, they have measurable physical attributes that we can put numbers to and we can consider each different number to represent a state. We can make magnets the size of atoms. (It'll put some chemists out of a job, but that's progress). But the point is that there are no further limitations on size imposed by quantum mechanics, over and above those due to statistical and classical mechanics.

I won't go into too much detail here: I will return to the subject, and all its lovely math — in the next chapter. For now, I'll just give you the gist of the ideas. Let us begin with some idealized quantum mechanical system (anything very small) and suppose that it can be in one of two states — say "up", which might correspond to an excited state, and "down", corresponding to a de-excited state. Alternatively, the two states might refer to the spin of the quantum system (spin is a crude classical analogy). We can actually allow it to be in other states as well, but for our purposes it just needs at least two states to represent a binary number: up is one, down is zero. I'll call this quantum mechanical system an atom, so that you can get a grip on its basic nature, but bear in mind that it could be something more complex, or even something simpler, like an electron (which has two spin states). Now the idea is that we build our computing device out of such atoms by stringing them together in a particular way. We start with part or all of the system — a string of atoms in one or other of their two states — representing a number, our input. We then let the whole system evolve over time according to the laws of quantum theory, interacting with itself — the atoms change states, the ones and zeroes move around — until at some point we have a bunch of atoms somewhere which will be in certain states, and these will represent our answer.

We could set the machine running with a single input bit — say firing an atom into the system — and design things such that the machine itself tells us when the calculation is complete, say by firing an atom out of the system. Nothing would be trustable until the output bit was one. You would measure this bit, then change it to zero and freeze the answer for examination. Putting the information in and out is not, incidentally, a particularly quantum mechanical process — it is a matter of amplification. Interestingly, as a rule one cannot predict the time the computer will take to complete its calculation. It turns out to be ballistic, like Fredkin's, but at the end you only get a wave packet for the arrival of the answer. We test to see whether or not the answer is in the machine or not. For the simple machine I have designed (see next chapter), there exist several quantum mechanical "amplitudes" (certain physical properties of the system) which, upon measurement, tell us how far through the calculation we

have gone, but ultimately we have to wait for the machine to let us know it's finished.

So, in 2050, or before, we may have computers that we can't even see! I will return to these strange beasts in the next chapter.

SIX

QUANTUM MECHANICAL COMPUTERS

6.1: Introduction

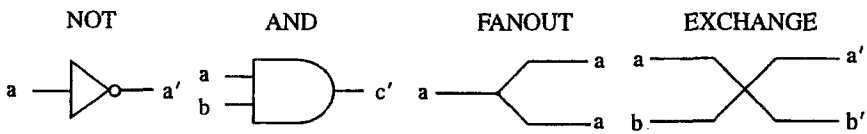
In this chapter[1], we discuss our part in an effort to analyze the physical limitations of computers due to the laws of physics[2]. For example, Bennett[1] has made a careful study of the free energy dissipation that must accompany computation. He found it to be virtually zero. He suggested to me the question of the limitations due to quantum mechanics and the Uncertainty Principle. I have found that, aside from the obvious limitation to size if the working parts are to be made of atoms, there is no fundamental limit from these sources either. We are here considering ideal machines; the effects of small imperfections will be considered later. This study is one of principle; our aim is to exhibit some Hamiltonian for a system which could serve as a computer. We are not concerned with whether we have the most efficient system, nor how we could best implement it.

Since the laws of quantum physics are reversible in time, we shall have to consider computing engines which obey such reversible laws. This problem already occurred to Bennett[1], and to Fredkin and Toffoli[2], and a great deal of thought has been given to it. Since it may not be familiar to you here, I shall review this and, in doing so, take the opportunity to review, very briefly, the conclusions of Bennett[3], for we shall confirm them all when we analyze our quantum system.

---

[1] Reprinted with permission from Optics News, February 1985, pp. 11-20. © Optical Society of America. Readers will require some understanding of elementary quantum mechanics to fully comprehend this chapter. [Editors]

[2] Although some of the notational and stylistic conventions of this chapter differ from those adopted elsewhere in the book, we have decided to retain them to preserve the flavor of Feynman's original published discourse. There is also some repetition of material discussed in previous chapters. [Editors]



a	a'
0	1
1	0

a	b	c'
0	0	0
0	1	0
1	0	0
1	1	1

a	b	a'	b'
0	0	0	0
0	1	1	0
1	0	0	1
1	1	1	1

Fig. 6.1 Primitive Elements

It is a result of computer science that a universal computer can be made by a suitably complex network of interconnected primitive elements. Following the usual classical analysis, we can imagine the interconnections to be ideal wires carrying one or two standard voltages representing the local 1 and 0. We can take the primitive elements to be just two, NOT and AND. (Actually just the one element NAND = NOT AND suffices, for if one input is set at 1 the output is the NOT of the other input.) They are symbolized in Figure 6.1, with the logical values resulting from different combinations of input wires. From a logical point of view, we must consider the wire in detail, for in other systems, and our quantum system in particular, we may not have wires as such. We see we really have two more logical primitives, FANOUT when two wires are connected to one, and EXCHANGE, when wires are crossed. In the usual computer the NOT and NAND primitives are implemented by transistors, possibly as in Figure 6.2:

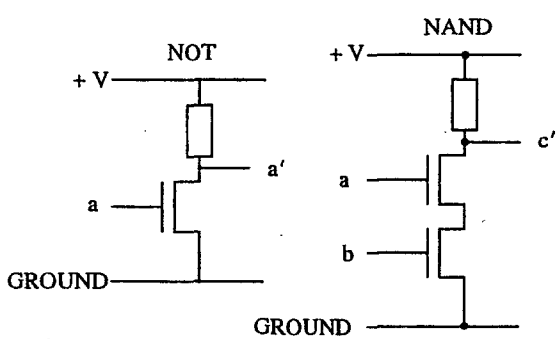


Fig. 6.2 Transistor Circuits for NOT and NAND

What is the minimum free energy that must be expended to operate an ideal computer made of such primitives? Since, for example, when the AND operates the output line, c' is being determined by one of two values no matter what was before, the entropy change is log2 units. This represents a heat generation of kT log 2 at temperature T. For many years it was thought that this represented an absolute minimum to the quantity of heat per primitive step that had to be dissipated in making a calculation.

The question is academic at this time. In actual machines we are quite concerned with the heat dissipation question, but the transistor system used actually dissipates about $10^{10}kT$! As Bennett[3] has pointed out, this arises because to change a wire's voltage we dump it to ground through a resistance; and to build it up again we feed charge, again through a resistance, to the wire. It could be greatly reduced if energy could be stored in an inductance, or other reactive element. However, it is apparently very difficult to make inductive elements on silicon wafers with present techniques. Even nature, in her DNA copying machine, dissipates about 100 kT per bit copied. Being, at present, so very far from this kT log 2 figure, it seems ridiculous to argue that even this is too high and the minimum is really essentially zero. But, we are going to be even more ridiculous later and consider bits written on one atom instead of the present 10^{11} atoms. Such nonsense is very entertaining to professors like me. I hope you will find it interesting and entertaining also.

What Bennett pointed out was that this former limit was wrong because it is not necessary to use irreversible primitives. Calculations can be done with reversible machines containing only reversible primitives. If this is done, the minimum free energy required is independent of the complexity or number of logical steps in the calculation. If anything, it is kT per bit of the output answer. But even this, which might be considered the free energy needed to clear the computer for further use, might also be considered as part of what you are going to do with the answer — the information in the result if you transmit it to another point. This is a limit only achieved ideally if you compute with a reversible computer at infinitesimal speed.

6.2: Computation With a Reversible Machine

We will now describe three reversible primitives that could be used to make a universal machine (Toffoli[4]). The first is the NOT which evidently loses no information, and is reversible, being reversed by acting again with NOT. Because the conventional symbol is not symmetrical we shall use an X on the

wire instead (see Figure 6.3(a)):

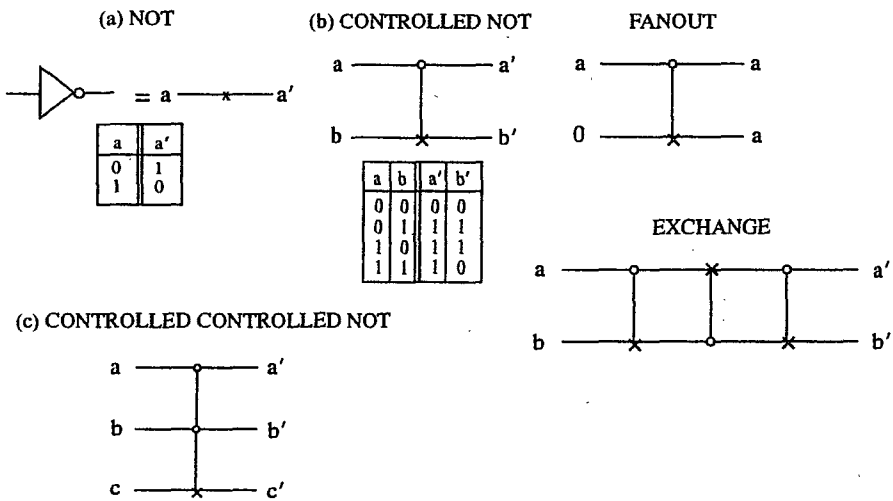


Fig. 6.3 Reversible Primitives

Next is what we shall call the CONTROLLED NOT (see Figure 6.3(b)). There are two entering lines, a and b and two exiting lines a' and b'. The a' is always the same as a, which is the control line. If the control is activated $a = 1$ then the output b' is the NOT of b. Otherwise b is unchanged, $b = b'$. The table of values for input and output is given in Figure 6.3. The action is reversed by simply repeating it. The quantity b' is really a symmetric function of a and b called XOR, the exclusive or; a or b but not both. It is likewise the sum modulo two of a and b, and can be used to compare a and b, giving a 1 as a signal that they are different. Please notice that this function XOR is itself not reversible. For example, if the value is zero we cannot tell whether it came from $(a,b) = (0,0)$ or from $(1,1)$ but we keep the other line $a = a'$ to resolve the ambiguity. We will represent the CONTROLLED NOT by putting a 0 on the control wire, connected with a vertical line to an X on the wire which is controlled. This element can also supply us with FANOUT, for if $b = 0$ we see that a is copied onto line b'. This COPY function will be important later on. It also supplies us with EXCHANGE, for three of them used successively on a pair of lines, but with alternate choice for control line, accomplishes an exchange of the information on the lines (Fig. 6.3(b)).

It turns out that combinations of just these two elements alone are insufficient to accomplish arbitrary logical functions. Some element involving three lines is necessary. We have chosen what we can call the CONTROLLED CONTROLLED NOT. Here (see Figure 6.3(c)) we have two control lines a,b which appear unchanged in the output and which change the third line c to NOT c only if both lines are activated ($a = 1$ and $b = 1$). Otherwise $c' = c$. If the third line input c is set to 0, then evidently it becomes 1 ($c' = 1$) only if both a and b are 1 and therefore supplies us with the AND function (see Table 6.1 below). Three combinations for (a,b), namely (0,0), (0,1) and (1,0) all give the same value, 0, to the AND (a,b) function so the ambiguity requires two bits to resolve it. These are kept in the lines a,b in the output so the function can be reversed (by itself, in fact). The AND function is the carry bit for the sum of a and b.

From these elements it is known that any logical circuit can be put together by using them in combination, and in fact, computer science shows that a universal computer can be made. We will illustrate this by a little example. First, of course, as you see in Figure 6.4, we can make an adder by first using the CONTROLLED CONTROLLED NOT and then the CONTROLLED NOT in succession to produce from a and b and 0, as input lines, the original a on one line, the sum on the second line and the carry on the third:

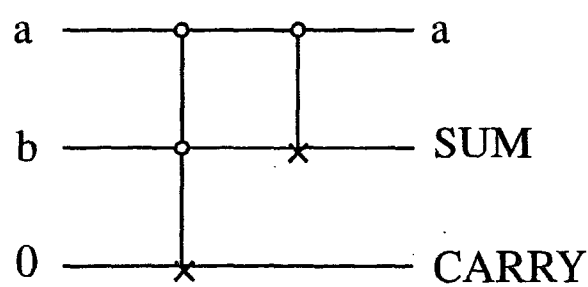


Fig. 6.4 An Adder

A more elaborate circuit is a full adder (see Figure 6.5) which takes a carry, c (from some previous addition), and adds it to the two lines a and b and has an additional line, d, with a 0 input. It requires four primitive elements to be put together. Besides this total sum, the total of the three, a, b, and c and the carry, we obtain on the other two lines two pieces of information. One is the a that we started with, and the other some intermediary quantity that we calculated en route:

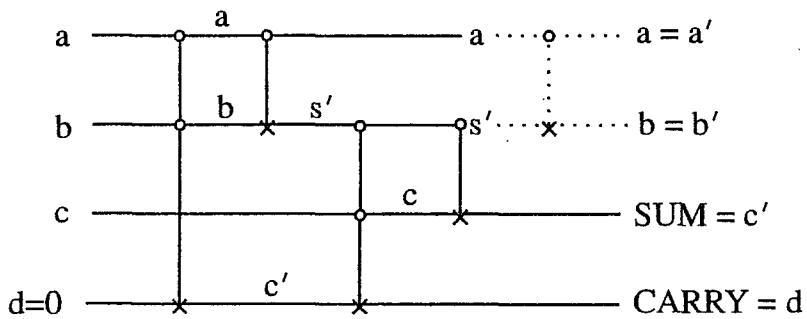


Fig. 6.5 A Full Adder

This is typical of these reversible systems; they produce not only what you want in output, but also a certain amount of garbage. In this particular case, and as it turns out in all cases, the garbage can be arranged to be, in fact, just the input. If we would just add the extra CONTROLLED NOT on the first two lines, as indicated by the dotted lines in Figure 6.5, we see that the garbage would be *a* and *b*, which were the inputs of at least two of the lines. (We know this circuit can be simplified but we do it this way for illustrative purposes.)

In this way, we can by various combinations, produce a general logic unit that transforms *n* bits to *n* bits in a reversible manner. If the problem you are trying to do is reversible, then there might be no extra garbage, but in general, there are some extra lines needed to store up the information which you would need to reverse the operation. In other words, we can make any function that the conventional system can, plus garbage. The garbage contains the information you need to reverse the process. And how much garbage? It turns out, in general, that if the output data that you're looking for has *k* bits, then starting with an input and *k* bits containing 0, we can produce, as a result, just the input and the output and no further garbage. This is reversible because knowing the output and the input permits you, of course, to undo everything. This proposition is always reversible. The argument for this is illustrated in Figure 6.6.

Suppose we begin with a machine **M**, which, starting with an input and some large number of 0's, produces the desired output plus a certain amount of extra data which we call garbage. Now we've seen that the copy operation can

be done by a sequence of CONTROLLED NOTs, so if we have originally an empty register, with the k bits ready for the output, we can, after the processor **M** has operated, copy the output from the **M** onto this new register (Fig. 6.6):

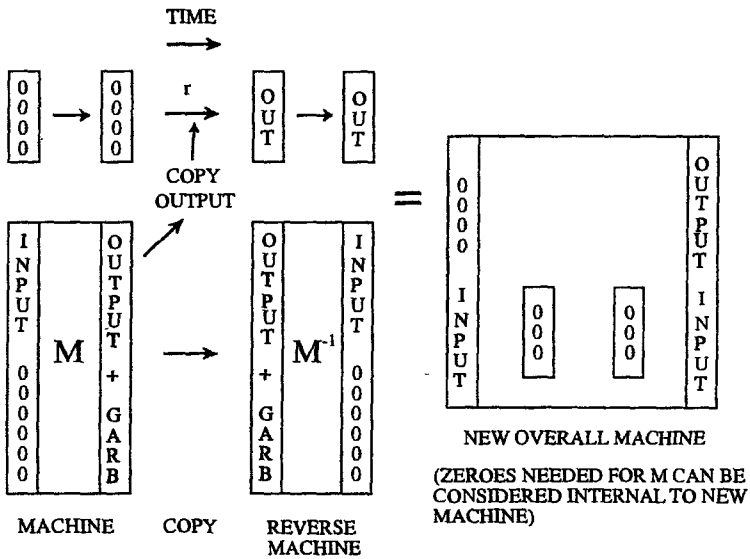


Fig. 6.6 Clearing Garbage

After that, we can build the opposite machine, the **M** in reverse, the reverse machine, which would take this output of **M** and garbage and turn it into the input and 0's. Thus, seen as an overall machine, we would have started with the k 0's of the register for the output, and the input, and ended up with those k 0's occupied by the output data, and repeat the input data as a final product. The number of 0's that was originally needed in the **M** machine in order to hold the garbage is restored again to 0, and can be considered as internal wires inside the new complete machine (**M**, $\mathbf{M}^{-1}$ and copy). Overall, then, we have accomplished what we set out to do, and therefore garbage need never be any greater than a repetition of the input data.

6.3: A Quantum Mechanical Computer

We now go on to consider how such a computer can also be built using the laws of quantum mechanics. We are going to write a Hamiltonian, for a system of interacting parts, which will behave in the same way as a large system in serving as a universal computer. Of course the large system obeys quantum

mechanics, but it is in interaction with the heat baths and other things that could make it effectively irreversible. What we would like to do is make the computer as small and as simple as possible. Our Hamiltonian will describe in detail all the internal computing actions but not, of course, those interactions with the exterior involved in entering the input (preparing the initial state) and reading the output.

How small can such a computer be? How small, for instance, can a number be? Of course a number can be represented by bits of 1's and 0's. What we're going to do is imagine that we have two-state systems, which we will call "atoms". An n-bit number is then represented by a state of a "register", a set of n two-state systems. Depending upon whether or not each atom is in one or another of its two states, which we call $|1\rangle$ and $|0\rangle$, we can of course represent any number. And the number can be read out of such a register by determining, or measuring, in which state each of the atoms is at a given moment. Therefore one bit will be represented by a single atom being in one of two states, the states we will call $|1\rangle$ and $|0\rangle$.

What we will have to do then can be understood by considering an example: the example of a CONTROLLED CONTROLLED NOT. Let G be some sort of an operation on three atoms a, b and c, which converts the original state of a,b, and c into a new appropriate state, a',b',c', so that the connection between a', b' and c' and a,b,c, are just what we would have expected if a,b, and c represented wires, and the a',b' and c' were the output wires of a CONTROLLED CONTROLLED NOT. It must be appreciated here that, at the moment, we are not trying to move the data from one position to another, we are just going to change it. Unlike the situation in the actual wired computer in which the voltages on one wire then go over to voltages on another, what we're specifically making is something simpler, that the three atoms are in some particular state, and that an operation is performed which changes the state to new values a',b',c'. What we would have then is that the state, in the mathematical form $|a',b',c'\rangle$ is simply some operation G operating on $|a,b,c\rangle$. In quantum mechanics, state changing operators are linear operators, and so we'll suppose that G is linear. Therefore, G is a matrix, and the matrix elements of $G, G^{a',b',c',a,b,c}$ are all 0 except those in the following table which are, of course, 1 (Table 6.1):

A	B	C	A′	B′	C′
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

Table 6.1 The Non-zero Matrix Elements of G

This table is the same table that represents the truth value table for the CONTROLLED CONTROLLED NOT. It is apparent that the operation is reversible, and that can be represented by saying that $G^*G = 1$, where the * means Hermitian adjoint. That is to say G is a unitary matrix. (In fact G is also a real matrix $G^* = G$, but that's only a special case.) To be more specific, we're going to write $A_{ab,c'}$ for this special G. We shall use the same matrix A with different numbers of subscripts to represent the other primitive elements.

To take a simple example, the NOT, which would be represented by A_a is the simple matrix:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{6.1}$$

This is a 2 x 2 matrix and can be represented in many ways, in different notations, but the particular one we will use to define this is by the method of creation and annihilation operators. Consider operating in this case, on a single line a. In order to save alphabets, let us call the matrix:

$$\underline{a} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \tag{6.2}$$

which annihilates the 1 on atom a and converts it to 0; $\underline{a}$ is an operator which

converts the state of |1> to |0>. But, if the state of the atom were originally |0>, the operator $\underline{a}$ produces the number 0. That is, it doesn't change the state, it simply produces the numerical value zero when operating on that state. The conjugate of this thing, of course, is:

$$\underline{a}^* = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \tag{6.3}$$

which creates, in the sense that operating on the 0 state, it turns it to the 1 state. In other words, it moves from |0> to |1>. When operating on the |1> state, there is no further state above that you can create, and therefore it gives it the number zero. Every other operator 2 x 2 matrix can be represented in terms of these $\underline{a}$ and $\underline{a}^*$. For example, the product of $\underline{a}^*\underline{a}$ is equal to the matrix:

$$\underline{a}^*\underline{a} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \tag{6.4}$$

which you might call N_a. It is 1 when the state is |1> and 0 when the state is |0>. It gives the number that the state of the atom represents. Likewise the product:

$$\underline{a}\underline{a}^* = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \tag{6.5}$$

is 1 - N_a, and gives 0 for the up-state and 1 for the down-state. We'll use 1 to represent the diagonal matrix:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{6.6}$$

As a consequence of all this, $\underline{a}\underline{a}^* + \underline{a}^*\underline{a} = 1$.

It is evident then that our matrix for NOT, the operator that produces NOT, is $A_a = \underline{a} + \underline{a}^*$. And further, of course, it is reversible, $A_a^*A_a = 1$, and A_a is unitary. In the same way, the matrix $A_{a,b}$ for the CONTROLLED NOT can be worked out. If you look at the table of values for CONTROLLED NOT (Fig. 6.3), you see that it can be written this way:

$$\underline{a}^*\underline{a}\ (\underline{b} + \underline{b}^*) + \underline{a}\underline{a}^* \tag{6.7}$$

In the first term, the $\underline{a}^*\underline{a}$ selects the condition that the line $a = 1$ in which case we want $\underline{b} + \underline{b}^*$, the NOT, to apply to b. The second term selects the condition that the line a is 0, in which case we want nothing to happen to b and the unit matrix on the operators of b is implied. This can also be written as $1 + \underline{a}^*\underline{a}(\underline{b} + \underline{b}^* - 1)$, the 1 representing all the lines coming through directly, but, in the case that a is 1, we would like to correct that by putting in a NOT instead of leaving the line b unchanged. The matrix for the CONTROLLED CONTROLLED NOT is:

$$A_{ab,c} = 1 + \underline{a}^*\underline{a}\underline{b}^*\underline{b}\ (\underline{c} + \underline{c}^* - 1) \tag{6.8}$$

as perhaps you may be able to see.

The next question is what the matrix is for a general logic unit which consists of a sequence of these. As an example, we'll study the case of the full adder which we described before (see Figure 6.5). Now we'll have, in the general case, four wires represented by a,b,c and d; we don't necessarily have d as 0 in all cases, and we would like to describe how the object operates in general (if d is changed to 1, d' is changed to its NOT). It produces new numbers a', b', c' and d', and we could imagine with our system that there are four atoms labeled a,b,c,d in a state labeled $|a',b',c',d'\rangle$ and that a matrix M operates which changes these same four atoms so that they appear to be in the state $|a',b',c',d'\rangle$ which is appropriate for this logic unit. That is, if $|\psi_{IN}\rangle$ represents the incoming state of the four bits, M is a matrix which generates an outgoing state $|\psi_{OUT}\rangle = M|\psi_{IN}\rangle$ for the four bits. For example, if the input state were the state $|1,0,1,0\rangle$ then, as we know, the output state should be $|1,0,0,1\rangle$; the first two a',b' should be 1,0 for those two first lines come straight through, and the last two c',d' should be 0,1 because that represents the sum and carry of the first three, a,b,c bits in the first input, as $d = 0$. Now the matrix M for the adder can easily be seen as the result of five successive primitive operations, and therefore becomes the matrix product of the five successive matrices representing these primitive objects:

$$M = A_{a,b}\ A_{b,c}\ A_{bc,d}\ A_{a,b}\ A_{ab,d} \tag{6.9}$$

The first, which is the one written farthest to the right, is $A_{ab,d}$ for that represents the CONTROLLED CONTROLLED NOT in which a and b are the CONTROL lines, and the NOT appears on line d. By looking at the diagram in Figure 6.5 we can immediately see what the remaining factors in the sequence represent. The last factor, for example, $A_{a,b}$ means that there's a CONTROLLED NOT with a CONTROL on line a and NOT on line b. This matrix will have the unitary property $M^*M = 1$ since all of the A's out of which it is a product are unitary. That is to say M is a reversal operation, and M^* is its inverse.

Our general problem, then, is this. Let $A_1, A_2, A_3, \ldots A_k$ be the succession of operations wanted, in some logical unit, to operate on n lines. The $2^n \times 2^n$ matrix M needed to accomplish the same goal is a product $A_k \ldots A_3 A_2 A_1$, where each A is a simple matrix. How can we generate this M in a physical way if we know how to make the simpler elements?

In general, in quantum mechanics, the outgoing state at time t is $e^{iHt}\psi_{IN}$ where ψ_{IN} is the input state, for a system with Hamiltonian H. To try to find, for a given special time t, the Hamiltonian which will produce $M = e^{iHt}$ when M is such a product of non-commuting matrices, from some simple property of the matrices themselves appears to be very difficult.

We realize, however, that at any particular time, if we expand the e^{iHt} out (as $1 + iHt - H^2t^2/2..$) we'll find the operator H operating an innumerable arbitrary number of times, once, twice, three times and so forth, and the total state is generated by a superposition of these possibilities. This suggests that we can solve this problem of the composition of these A's in the following way. We add to the n atoms, which are in our register, an entirely new set of $k + 1$ atoms, which we'll call "program counter sites". Let us call q_i and q_i^* the annihilation and creation operators for the program site i for $i = 0$ to k. A good thing to think of, as an example, is an electron moving from one empty site to another. If the site is occupied by the electron, its state is $|1\rangle$, while if the site is empty, its state is $|0\rangle$.

We write, as our Hamiltonian:

$$H = \sum_{i=0}^{k-1} q_{i+1}^* \, q_i \, A_{i+1} + \textit{complex conjugate}$$

$$= q_1^* \, q_0 \, A_1 + q_2^* \, q_1 \, A_2 + q_3^* \, q_2 \, A_3 + \ldots$$
$$+ q_0^* \, q_1 \, A_1^* + q_1^* \, q_2 \, A_2^* + q_2^* \, q_3 \, A_3^* + \ldots \tag{6.10}$$

The first thing to notice is that, if all the program sites are unoccupied so that all the program atoms are initially in the state 0, nothing happens because every term in the Hamiltonian starts with an annihilation operator and it gives 0 therefore.

The second thing we notice is that, if only one or another of the program sites is occupied (in state |1>), and the rest are not (state |0>), then this is always true. In fact the number of program sites that are in state |1> is a conserved quantity. We will suppose that, in the operation of this computer, either no sites are occupied (in which case nothing happens) or just one site is occupied. Two or more program sites are never both occupied during normal operation.

Let us start with an initial state where site 0 is occupied, is in the |1> state, and all the others are empty, in the |0> state. If later, at some time, the final site k is found to be in the |1> state (and therefore all the others in |0>) then, we claim, the n register has been multiplied by the matrix M, which is $A_k \cdots A_2 A_1$ as desired.

Let me explain how this works. Suppose that the register starts in any initial state, ψ_{in}, and that the site, 0, of the program counter is occupied. Then the only term in the entire Hamiltonian that can first operate, as the Hamiltonian operates in successive times, is the first term, $q_1{}^* q_0 A_1$. The q_0 will change site number 0 to an unoccupied site, while $q_1{}^*$ will change the site number 0 to an occupied site. Thus the term $q_1{}^* q_0$ is a term which simply moves the occupied site from the location 0 to the location 1. But this is multiplied by the matrix A_1 which operates only on the n register atoms, and therefore multiplies the initial state of the n register atoms by A_1. Now, if the Hamiltonian happens to operate a second time, this first term will produce nothing because q_0 produces 0 on the number 0 site because it is now unoccupied. The term which can operate now is the second term, $q_2{}^* q_1 A_2$ for that can move the occupied point, which I shall call a "cursor". The cursor can move from site 1 to site 2 but the matrix A_2 now operates on the register, therefore the register has now got the matrix $A_2 A_1$

operating on it. So, looking at the first line of the Hamiltonian, if that is all there was to it, as the Hamiltonian operates in successive orders, the cursor would move successively from 0 to k, and you would acquire, one after the other, operating on the n register atoms, the matrices, A, in the order that we would like to construct the total M.

However, a Hamiltonian must be Hermitian, and therefore the complex conjugate of all these operators must be present. Suppose that, at a given stage, we have gotten the cursor on site number 2, and we have the matrix A_2A_1 operating on the register. Now the q_2 which intends to move that occupation to a new position needn't come from the first line, but may have come from the second line. It may have come, in fact, from $q_1{}^*q_2A_2{}^*$ which would move the cursor back from the position 2 to the position 1. But note that, when this happens, the operator $A_2{}^*$ operates on the register, and therefore the total operator on the register is $A_2{}^*A_2A_1$ in this case. But $A_2{}^*A_2$ is 1 and therefore the operator is just A_1. Thus we see that, when the cursor is returned to the position 1, the net result is that only the operator A_1 has really operated on the register. Thus it is that, as the various terms of the Hamiltonian move the cursor forwards and backwards, the A's accumulate, or are reduced out again. At any stage, for example, if the cursor were up to the j site, the matrices from A_1 to A_j have operated in succession on the n register. It does not matter whether or not the cursor on the j site has arrived there by going directly from 0 to j, or going further and returning, or going back and forth in any pattern whatsoever, as long as it finally arrived at the state j. Therefore it is true that, if the cursor is found at the site k, we have the net result for the n register atoms that the matrix M has operated on their initial state as we desired.

How then could we operate this computer? We begin by putting the input bits onto the register, and by putting the cursor to occupy the site 0. We then check at the site k, say, by scattering electrons, that the site k is empty, or that the site k has a cursor. The moment we find the cursor at site k, we remove the cursor so that it cannot return down the program line, and then we know that the register contains the output data. We can then measure it at our leisure. Of course, there are external things involved in making the measurements, and determining all of this, which are not part of our computer. Surely a computer has eventually to be in interaction with the external world, both for putting data in and for taking it out.

Mathematically it turns out that the propagation of the cursor up and down this program line is exactly the same as it would be if the operators A were not

in the Hamiltonian. In other words, it represents just the waves which are familiar from the propagation of the tight binding electrons or spin waves in one dimension, and are very well known. There are waves that travel up and down the line, and you can have packets of waves and so forth. We could improve the action of this computer and make it into a ballistic action in the following way: by making a line of sites in addition to the ones inside, that we are actually using for computing, a line of, say, many sites both before and after. It's just as though we had values of the index i for q_i, which are less than 0 and greater than k, each of which has no matrix A, just a 1 multiplying there. Then we'd have a longer spin chain, and we could have started, instead of putting a cursor exactly at the beginning site 0, by putting the cursor with different amplitudes on different sites representing an initial incoming spin wave, a wide packet of nearly definite momentum. This spin wave would then go through the entire computer in a ballistic fashion and out the other end into the outside tail that we have added to the line of program sites, and there it would be easier to determine if it is present and to steer it away to some other place, and to capture the cursor. Thus the logical unit can act in a ballistic way.

This is the essential point and indicates, at least to a computer scientist, that we could make a universal computer, because he knows if we can make any logical unit we can make a universal computer. That this could represent a universal computer for which composition of elements and branching can be done, is not entirely obvious unless you have some experience, but I will discuss that to some further extent later.

6.4: Imperfections and Irreversible Free Energy Loss

There are, however, a number of questions that we would like to discuss in more detail such as the question of imperfections. There are many sources of imperfections in this machine, but the first one we would like to consider is the possibility that the coefficients in the couplings, along the program line, are not exactly equal. The line is so long that in a real calculation little irregularities would produce a small probability of scattering, and the waves would not travel exactly ballistically but would go back and forth. If the system, for example, is built so that these sites are built on a substrate of ordinary physical atoms, then the thermal vibrations of these atoms would change the couplings a little bit and generate imperfections. (We should even need such noise for with small imperfections there are shallow trapping regions where the cursor may get caught.) Suppose then, that there is a certain probability, say p per step of calculation (that is, per step of cursor motion $i \to i + 1$) for scattering the cursor

momentum until it is randomized ($1/p$ is the transport mean free path). We will suppose that the p is fairly small. Then in a very long calculation, it might take a very long time for the wave to make its way out the other end, once started at the beginning — because it has to go back and forth so many times due to the scattering. What one then could do, would be to pull the cursor along the program line with an external force. If the cursor is, for example, an electron moving from one vacant site to another, this would be just like an electric field trying to pull the electron along a wire, the resistance of which is generated by the imperfection or the probability of scattering. Under these circumstances we can calculate how much energy will be expended by this external force.

This analysis can be made very simply; it is an almost classical analysis of an electron with a mean free path. Every time the cursor is scattered, I'm going to suppose it is randomly scattered forward and backward. In order for the machine to operate, of course, it must be moving forward at a higher probability than it is moving backward. When a scattering occurs, therefore, the loss in entropy is the logarithm of the probability that the cursor is moving forward, divided by the probability that the cursor was moving backward. This can be approximated by (the probability forward – the probability backward)/(the probability forward + the probability backward). That was the entropy lost per scattering. More interesting is the entropy lost per net calculational step which is, of course, simply p times that number. We can rewrite the entropy cost per calculational step as:

$$p\ \upsilon_D/\upsilon_R \tag{6.11}$$

where v_D is the drift velocity of the cursor, and v_R is its random velocity. Or, if you like, it is p times the minimum time that the calculation could be done in, (that is, if all the steps were always in the forward direction), divided by the actual time allowed. The free energy loss per step, then, is kT x p x the minimum time that the calculation could be done, divided by the actual time that you allow yourself to do it. This is a formula that was first derived by Bennett. The factor p is a coasting factor, to represent situations in which not every site scatters the cursor randomly, but it has only a small probability to be thus scattered. It will be appreciated that the energy loss per step is not kT but is that divided by two factors. One ($1/p$) measures how perfectly you can build the machine, and the other is proportional to the length of time that you take to do the calculation. It is very much like a Carnot engine in which, in order to obtain reversibility, one must operate very slowly. For the ideal machine where p is 0,

or where you allow an infinite time, the mean energy loss can be 0.

The Uncertainty Principle, which usually relates some energy and time uncertainty, is not directly a limitation. What we have in our computer is a device for making a computation, but the time of arrival of the cursor and the measurement of the output register at the other end (in other words, the time it takes in which to complete the calculation), is not a definite time. It's a question of probabilities, and so there is a considerable uncertainty in the time at which a calculation will be done. There is no loss associated with the uncertainty of cursor energy; at least no loss depending on the number of calculational steps. Of course, if you want to do a ballistic calculation on a perfect machine, some energy would have to be put into the original waves, but that energy can, of course, be removed from the final waves when it comes out of the tail of the program line. All questions associated with the uncertainty of operators and the irreversibility of measurements are associated with the input and output functions. No further limitations are generated by the quantum nature of the computer per se; nothing that is proportional to the number of computational steps.

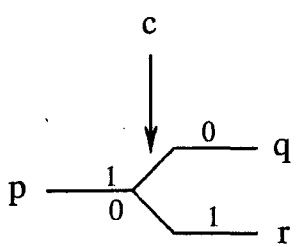
In a machine such as this there are very many other problems due to imperfections. For example, in the registers for holding the data, there will be problems of cross-talk, interactions between one atom and another in that register, or interaction of the atoms in that register directly with things that are happening along the program line that we didn't exactly bargain for. In other words, there may be small terms in the Hamiltonian besides the ones we've written. Until we propose a complete implementation of this, it is very difficult to analyze. At least some of these problems can be remedied in the usual way by techniques such as error correcting codes and so forth, that have been studied in normal computers. But until we find a specific implementation for this computer, I do not know how to proceed to analyze these effects. However, it appears that they would be very important in practice. This computer seems to be very delicate and these imperfections may produce considerable havoc.

The time needed to make a step of calculation depends on the strength or the energy of the interactions in the terms of the Hamiltonian. If each of the terms in the Hamiltonian is supposed to be of the order of 0.1 electron volts, then it appears that the time for the cursor to make each step, if done in a ballistic fashion, is of the order 6×10^{-15} sec. This does not represent an enormous improvement, perhaps only about four orders of magnitude over the present values of the time delays in transistors, and is not much shorter than the very short times possible to achieve in many optical systems.

6.5: Simplifying the Implementation

We have completed the job we set out to do — to find some quantum mechanical Hamiltonian of a system that could compute, and that is all that we need to say. But it is of some interest to deal with some questions about simplifying the implementation. The Hamiltonian that we've written involves terms which can involve a special kind of interaction between five atoms. For example, three of them in the register for a CONTROLLED CONTROLLED NOT and two of them as the two adjacent sites in the program counter. This may be rather complicated to arrange. The question is, can we do it with simpler parts? It turns out, we can indeed. We can do it so that in each interaction there are only three atoms. We're going to start with new primitive elements instead of the ones we began with. We'll have the NOT all right, but we have in addition to that simply a "switch" (see also Priese[5]).

Supposing that we have a term, $q^*cp + r^*c^*p +$ its complex conjugate in the Hamiltonian (in all cases we'll use letters in the earlier part of the alphabet for register atoms, and in the latter part of the alphabet for program sites (see Figure 6.7):



IF c=0 GO p TO q AND PUT c=0

IF c=0 GO p TO r AND PUT c=1

IF c=1 GO r TO p AND PUT c=0

IF c=0 GO q TO p AND PUT c=1

Fig. 6.7 Switch

This is a switch in the sense that, if c is originally in the |1> state, a cursor at p will move to q, whereas if c is in the |0> state, the cursor at p will move to r. During this operation the controlling atom c changes its state. (It is possible also to write an expression in which the control atom does not change its state, such as $q^*c^*cp + r^*cc^*p$ and its complex conjugate, but there is no particular advantage or disadvantage to this, and we will take the simpler form.) The

complex conjugate reverses this. If, however, the cursor is at q and c is in the state $|1\rangle$ (or cursor at r, c in $|0\rangle$) the H gives 0, and the cursor gets reflected back. We shall build all our circuits and choose initial states so that this circumstance will not arise in normal operation, and the ideal ballistic mode will work.

With this switch we can do a number of things. For example, we could produce a CONTROLLED NOT as in Figure 6.8:

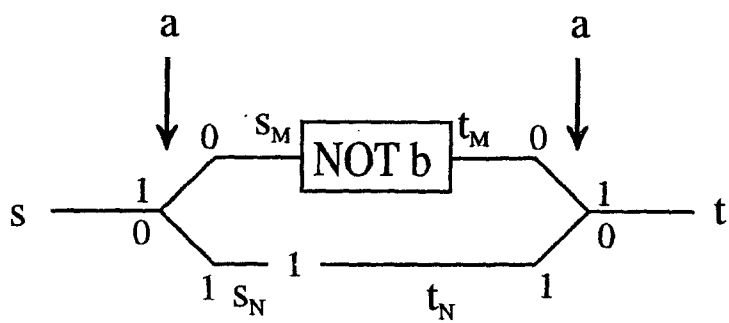


Fig. 6.8 CONTROLLED NOT realized by Switches

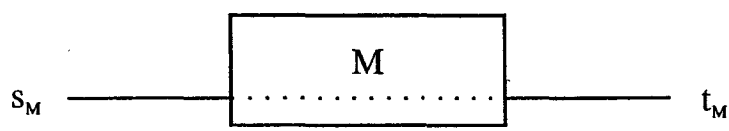
The switch a controls the operation. Assume the cursor starts at s. If $a = 1$ the program cursor is carried along the top line, whereas if $a = 0$ it is carried along the bottom line, in either case terminating finally in the program site t. In these diagrams, horizontal or vertical lines will represent program atoms. The switches are represented by diagonal lines, and in boxes we'll put the other matrices that operate on registers such as the NOT b. To be specific, the Hamiltonian for this little section of a CONTROLLED NOT, thinking of it as starting at s and ending at t, is given below:

$$\begin{aligned} H_c(s,t) &= s_M^* a s + t^* a^* t_M \\ &+ t_M^*(b + b^*)s_M + s_N^* a^* s \\ &+ t^* a t_N + t_N^* s_N + c.c. \end{aligned} \tag{6.12}$$

(The c.c. means to add the complex conjugate of all the previous terms.) Although there seem to be two routes here which would possibly produce all kinds of complications characteristic of quantum mechanics, this is not so. If the

entire computer system is started in a definite state for a, by the time the cursor reaches s the atom a is still in some definite state (although possibly different from its initial state due to previous computer operations on it). Thus only one of the two routes is taken. The expression may be simplified by omitting the s_N* t_N term and putting $t_N = s_N$. One need not be concerned in that case that one route is longer (two cursor sites) than the other (one cursor site), for again there is no interference. No scattering is produced in any case by the insertion into a chain of coupled sites, an extra piece of chain of any number of sites with the same mutual coupling between sites (analogous to matching impedances in transmission lines).

To study these things further, we think of putting pieces together. A piece (see Figure 6.9) M might be represented as a logical unit of interacting parts in which we only represent the first input cursor site as s_M and the final one at the other end as t_M. All the rest of the program sites that are between s_M and t_M are considered internal parts of M, and M contains its registers. Only s_M and t_M are sites that may be coupled externally (Fig. 6.9):



s_M = Starting program site for piece

t_M = Terminal program site for piece

Fig. 6.9 One "piece"

The Hamiltonian for this sub-section we'll call H_M, and we'll identify s_M and t_M as the names of the input and output program sites by writing $H_M(s_M,t_M)$. So therefore H_M is the part of the Hamiltonian representing all the atoms in the box and their external start and terminator sites.

An especially important and interesting case to consider is when the input data (in the regular atoms) comes from one logical unit, and we would like to transfer it to another (see Figure 6.10):

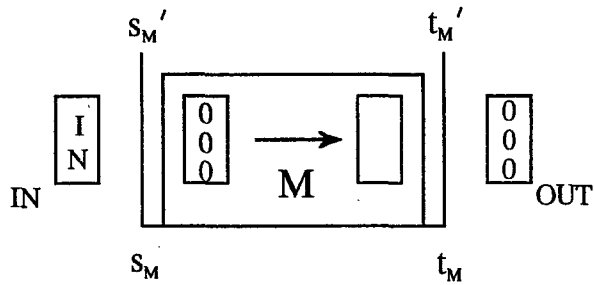


Fig. 6.10 Piece with External Input and Output

Suppose that we imagine that the box M starts with its input register with 0 and its output (which may be the same register) also with 0. We could now use it in the following way. We could make a program line, let's say starting with s_M' whose first job is to exchange the data in an external register which contains the input, with M's input register which at the present time contains 0's. Then the first step in our calculation starting, say, at s_M', would be to make an exchange with the register inside of M. That puts 0's into the original input register and puts the input where it belongs inside the box M. The cursor is now at s_M. (We have already explained how exchange can be made of CONTROLLED NOTs.) Then, as the program goes from s_M to t_M, we find the output now in the box M. The output register of M is now cleared as we write the results into some new external register provided for that purpose, originally containing 0's. This we do from t_M to t_M' by exchanging data in the empty external register with the M's output register.

We can now consider connecting such units in different ways. For example, the most obvious way is succession. If we want to do first M and then N we can connect the terminal side of one to the starting side of the other as in Figure 6.11 to produce a new effective operator K:

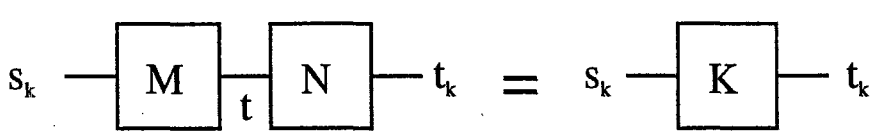


Fig. 6.11 Operations Performed in Succession

The Hamiltonian for H_K is then:

$$H_K(s_K,t_K) = H_M(s_K,t) + H_N(t,t_K) \tag{6.13}$$

The general conditional, if $a = 1$ do M, but if $a = 0$ do N, can be made, as in Figure 6.12:

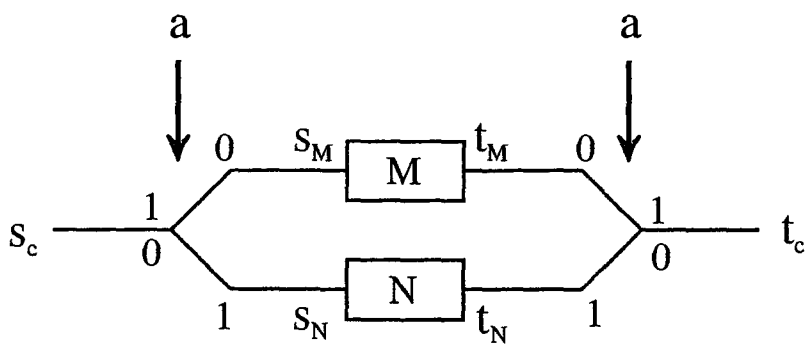


Fig. 6.12 Conditional: if a = 1 then M, else N

For this:

$$\begin{aligned} H_{cond}(s_c,t_c) = (&s_M^* a s_c + t_c^* a^* t_M + s_N^* a^* s_c \\ &+ t_c^* a t_N + c.c.) + H_M(s_M,t_M) + H_N(s_N,t_N) \end{aligned} \tag{6.14}$$

The CONTROLLED NOT is the special case of this with M = NOT b for which H is:

$$H_{NOT\ b}(s,t) = s^*(b + b^*)t + c.c. \tag{6.15}$$

and N is no operation: s^*t.

As another example, we can deal with a garbage clearer (previously described in Figure 6.6) not by making two machines, a machine and its inverse, but by using the same machine and then sending the data back to the machine in the opposite direction, using our switch (Fig. 6.13):

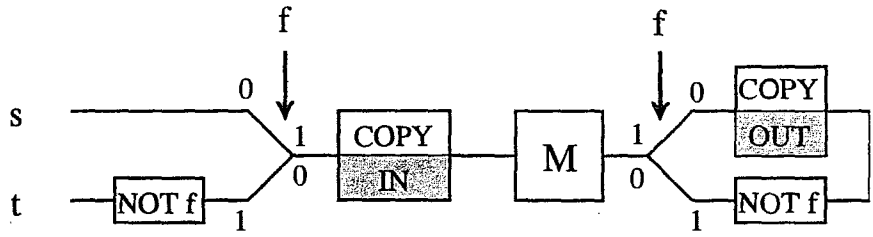


Fig. 6.13 Garbage Clearer

Suppose in this system we have a special flag *f* which is originally always set to 0. We also suppose we have the input data in an external register, an empty external register available to hold the output, and the machine registers all empty (containing 0's). We come on the starting line *s*. The first thing we do is to copy (using CONTROLLED NOTs) our external input into *M*. Then *M* operates, and the cursor goes on the top line in our drawing. It copies the output out of *M* into the external output register. *M* now contains garbage. Next it changes *f* to NOT *f*, comes down on the other line of the switch, backs out through *M* clearing the garbage and uncopies the input again. When you copy data and do it again, you reduce one of the registers to 0, the register into which you copied the first time. After the copying, it goes out (since *f* is now changed) on the other line where we restore *f* to 0 and come out at *t*. So between *s* and *t* we have a new piece of equipment which has the following properties. When it starts we have, in a register called IN, the input data. In an external register, which we call OUT, we have 0's. There is an internal flag set at 0, and the box, *M*, is empty of all data. At the termination of this, at *t*, the input register still contains the input data, the output register contains the output of the effort of the operator *M*. *M*, however, is still empty, and the flag *f* is reset to 0.

Also important in computer programs is the ability to use the same subroutine several times. Of course, from a logical point of view, that can be done by writing that bit of program over and over again each time it is to be used, but in a practical computer it is much better if we could build that section of the computer which does a particular operation just once, and use that section again and again. To show the possibilities here, first just suppose we have an operation we simply wish to repeat twice in succession (Fig. 6.14):

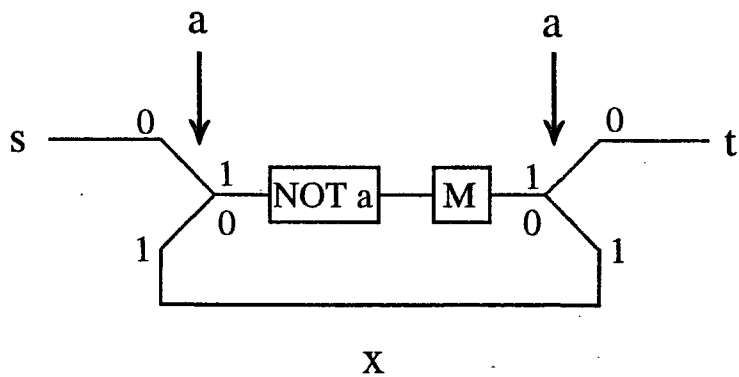


Fig. 6.14 The Operation "Do M Twice"

We start at s with the flag a in the condition 0, and thus we come along the line and the first thing that happens is we change the value of a. Next we do the operation M. Now, because we changed a, instead of coming out at the top line where we went in, we come out at the bottom line which recirculates the program back into changing a again, and it restores it. This time as we go through M, we come out and we have the a to follow on the upper line and thus come out at the terminal t. The Hamiltonian for this is:

$$\begin{aligned} H_{MM}(s,t) &= (s_N^* a^* s + s_M^*(a^* + a)s_N \\ &+ x^* a^* t_M + s_N^* a x + t^* a t_M + c.c) \\ &+ H_M(s_M, t_M) \end{aligned} \tag{6.16}$$

Using this switching circuit a number of times, of course, we can repeat an operation several times. For example, using the same idea three times in succession, a nested succession, we can do an operation eight times by the apparatus indicated in Figure 6.15:

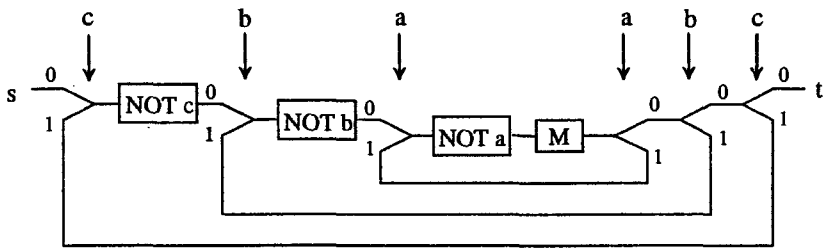


Fig. 6.15 The Operation "Do M Eight Times"

In order to do so, we have three flags, *a*, *b* and *c*. It is necessary to have flags when operations are done again for the reason that we must keep track of how many times it's done and where we are in the program or we'll never be able to reverse things. A subroutine in a normal computer can be used and emptied and used again without any record being kept of what happened. But here we have to keep a record — and we do that with flags — of exactly where we are in the cycle of the use of the subroutine. If the subroutine is called from a certain place and has to go back to some other place, and is called another time, its origin and final destination are different. We have to know and keep track of where it came from and where it's supposed to go individually in each case, so more data has to be kept. Using a subroutine over and over in a reversible machine is only slightly harder than in a general machine. All these considerations appear in papers by Fredkin, Toffoli and Bennett.

It is clear by the use of this switch, and successive uses of such switches in trees, that we would be able to steer data to any point in a memory. A memory would simply be a place where there are registers into which you could copy data and then return to the program. The cursor will have to follow the data along and I suppose there must be another set of tree switches set the opposite direction to carry the cursor out again after copying the data so that the system remains reversible.

In Figure 6.16 below we show an incremental binary counter (of three bits *a,b,c* with *c* the most significant bit) which keeps track of how many net times the cursor has passed from *s* to *t*. These few examples should be enough to show that indeed we can construct all computer functions with our SWITCH and NOT. We need not follow this in more detail.

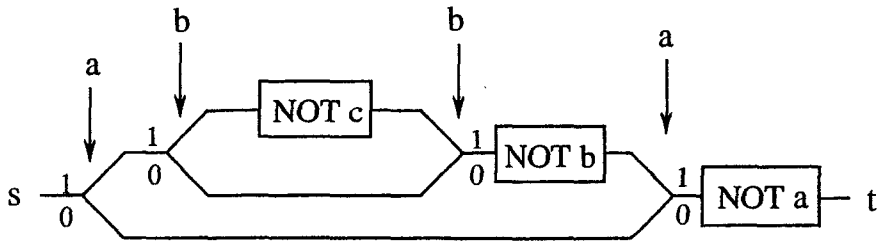


Fig. 6.16 Increment Counter (3-bit)

6.6: Conclusions

It is clear from these examples that this quantum machine has not really used many of the specific qualities of the differential equations of quantum mechanics. What we have done is only to try to imitate as closely as possible the digital machine of conventional sequential architecture. It is analogous to the use of transistors in conventional machines where we don't properly use all the analog continuum of the behaviour of transistors, but just try to run them as saturated on or off digital devices so the logical analysis of the system behavior is easier. Furthermore, the system is absolutely sequential — for example, even in the comparison (exclusive OR) of two k bit numbers, we must do each bit successively. What can be done, in these reversible quantum systems, to gain the speed available by concurrent operation has not been studied here.

Although, for theoretical and academic reasons, I have studied complete and reversible systems, if such tiny machines could become practical there is no reason why irreversible and entropy creating interactions cannot be made frequently during the course of operations of the machine. For example, it might prove wise, in a long calculation, to ensure that the cursor has surely reached some point and cannot be allowed to reverse again from there. Or, it may be found practical to connect irreversible memory storage (for items less frequently used) to reversible logic or short term reversible storage registers, etc. Again, there is no reason we need to stick to chains of coupled sites for more distant communication where wires or light may be easier and faster. At any rate, it seems that the laws of physics present no barrier to reducing the size of

computers until bits are the size of atoms, and quantum behavior holds dominant sway.

6.7: References[3]

[1] **C.H. Bennett**, "Logical Reversibility of Computation," *IBM Journal of Research and Development*, 6 (1979), pp. 525-532.

[2] **E. Fredkin and T. Toffoli**, "Conservative Logic," *Int. J. Theor. Phys.*, 21 (1982), pp. 219-253.

[3] **C.H. Bennett**, "Thermodynamics of Computation — a Review," *Int. J. Theor. Phys.* 21 (1982), pp. 905-940.

[4] **T. Toffoli**, "Bicontinuous Extensions of Invertible Combinatorial Functions," *Mathematical Systems Theory*, 14 (1981), pp. 13-23.

[5] **L. Priese**, "On a Simple Combinatorial Structure Sufficient for Sublying Non-Trivial Self Reproduction," *Journal of Cybernetics*, 6 (1976), pp. 101-137.

[3] I would like to thank T. Toffoli for his help with the references. [RPF]

SEVEN

PHYSICAL ASPECTS OF COMPUTATION

A Caveat from the Editors

This chapter covers the most time-dependent of all the topics in these lectures — the advances in silicon technology over the past decade have been truly startling. Nonetheless, we believe it worthwhile to include Feynman's overview of the state of the subject in the early 1980's — despite the fact that some of the technological goalposts have moved considerably since Feynman looked at the subject. In particular, the mid 1980's saw the widespread adoption of CMOS technology and Feynman's discussion of devices in terms of nMOS technology now looks somewhat dated: we have therefore edited out a few of his more complex nMOS examples. Nonetheless, his brief discussion of CMOS devices does concentrate on their favorable energetics and savings in power compared to nMOS. Feynman's discussion of design rules is restricted to single metal layer nMOS — as specified by Mead and Conway in their classic 1980 book on VLSI systems. Rather than attempt to update the material to a CMOS context, we have decided to remain faithful to Feynman's original presentation, apart from some minor editorial updates. In this way we hope that Feynman's unique ability to offer valuable physical insight into complex physical processes still comes through. Moreover, it should be remembered that, in actuality, Feynman's lectures were supplemented by lectures from experts from many fields. It is intended to capture this element of Feynman's course in a forthcoming accompanying volume containing state-of-the-art lectures and papers by some of the same experts who contributed to his original courses in the early 1980's. Now read on.

The unifying theme of this course has been what we can and cannot do with computers, and why. We have considered restrictions arising from the organization of the basic elements within machines, the limitations imposed by fundamental mathematics, and even those resulting from the laws of Nature themselves. In this final chapter, we come to address perhaps the most practical of obstacles: the constraints that arise from the technology we employ to

actually *build* our machines — both from the materials we use and from the way in which we arrange the elementary component parts.

Presently, the majority of computers are based on *semiconductor* technology, which is used to fashion the basic building blocks of machines — devices such as transistors and diodes. VLSI (Very Large Scale Integration), the field of microelectronics dealing with the construction and utilization of silicon chips — and hence of central importance to computing — is a vast subject in itself and we can only scratch the surface here. The reader will certainly find what follows easier to understand if he or she has some knowledge of electronics. However, we hope that our presentation will be intelligible to those with only a passing acquaintance with electricity and magnetism, and we provide several references in the section on suggested reading for the curious to take their interest further.

To begin with, we shall take a look at one simple kind of device, the *diode*. This is a cunning device which allows current to flow in one direction only. We shall consider the physical phenomena involved in its operation, and how it works in the engineering context of a Field Effect Transistor.

7.1: The Physics of Semiconductor Devices

Our current understanding of the electrical properties of metals and other materials is based on the so-called "Band Theory" of solids. Loosely speaking, this theory predicts that the possible physical states that can be occupied by electrons within a material are arranged into a series of (effectively continuous) strata called "bands", each characterized by a specific range of energies for the allowed electron energy levels within it. These bands arise from the complex interplay of electrons with their parent atoms located within the atomic lattice of the material and are an intrinsically quantum mechanical effect. Electrons in different atomic states occupy different bands. In a general substance, we can identify two essentially distinct types of band relevant to the conduction of electric current: these are the "filled" or "valence" band, and the "conduction" band. States in the filled band correspond to electrons which are bound to their parent atoms, and are effectively confined to a certain region within the material — they are not free to roam around. Electrical conduction occurs when electrons leave their parent atoms and are able to move freely through the conductor. Mobile electrons of this type are said to occupy states within the "conduction band". Typically, there will be a *discrete* energy gap between the filled and conduction bands. The size of this gap largely determines whether our material

is to be classified as a conductor or an insulator, as we'll see. Let us examine the energy band structure shown in Figure 7.1:

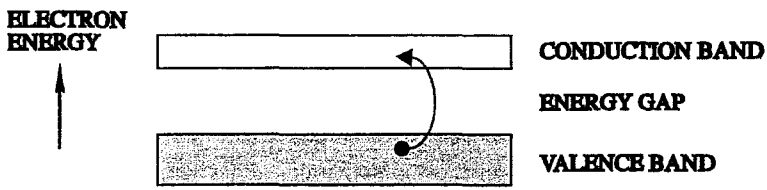


Fig. 7.1 Band Structure

As you can see, we have valence and conduction bands separated by an energy gap — in the diagram, the energy associated with the bands increases as we move up vertically. When the lower band is full, the material acts like an insulator: there are no available energy states for electrons to gain energy from the applied electric field and form a current. To support an electrical current, we need electrons in the conduction band where there are plenty of empty states available. To produce such electrons, enough energy must be supplied to occupants of the valence band to help them leap above the gap and make the transition into the conduction zone. This minimum energy is called the "band gap energy" and its value largely determines the electrical properties of a substance, as I've said. Good conductors have a plentiful supply of free electrons under normal conditions, the band gap energy being tiny or non-existent (filled and conduction bands can even overlap). Hence it will not be difficult to excite a current in such a material. Insulators, however, have prohibitively wide gaps (several eV) and only conduct under pretty extreme conditions. There is, however, a third class of material needing consideration, and that is a sort of hybrid of conductors and insulators — the *semiconductor* — for which the energy gap is relatively small (1 eV or so).

The primary mechanism responsible for getting electrons out of the filled band and into the conduction band is *thermal excitation* (neglecting the application of external electric fields). This is simply the process whereby the energy changes of random thermal fluctuations are themselves enough to supply the energy required to enable electrons to make a transition. A typical thermal energy might be of the order of 25 meV and if this exceeds the band gap energy, it will be sufficient to cause transitions. This is the case for metals but

not for insulators — with their large band gap energies of several eV. For any given material, we can calculate how likely it is for a thermal fluctuation to result in a conduction electron. If the temperature of the substance is T, and E is the band gap energy, then the rate at which electrons spontaneously pop up to the higher band is determined by the Boltzmann distribution and is proportional to $\exp(-E/kT)$, where k is Boltzmann's constant. At room temperature ($T \approx 300$K), we have $kT \approx 1/40$ of an electron-volt or 25 meV. Note that, due to the exponential in the formula, this transition rate rises rapidly with temperature. Nonetheless, for most insulators, this rate remains negligible right up to near the melting point.

Let us take a look at a semiconductor. At zero degrees (and low temperatures generally), the semiconductor silicon (henceforth Si) is effectively an insulator. Its band gap is of the order of 1.1 eV and thermal transitions are rare. However, we can certainly excite a current by supplying energy to the valence electrons and when we do, we find something interesting happening, something which is of central importance in our study of semiconductors. When we excite an electron to the conduction band, not only does it become free to run around and give rise to some conductivity, but it leaves behind, in the lower band, a *hole*. This hole has an effective positive charge and, like the electron in the conduction band, is also able to move about and carry electric current: if a nearby electron fills the space vacated by the thermally excited particle, it will leave a positive charge in its own original location, as if the hole had moved sites. Holes are not "real" free particles — they are just empty spaces in the valence band that behave as if they are particles with positive charge. Holes also appear in insulators but rarely in metals.

There is a special trick that we can perform with Si which modifies its properties so that it is ideal for use in computers. This is the process of *doping*. Doping involves adding atoms of another substance (an "impurity") into the Si lattice[1]. A common dopant is the element phosphorus (P), which sits next to Si in the Periodic Table. P has a valency of five rather than the four of Si: this means it has five electrons in its outer shell compared to silicon's four. In an ordinary silicon crystal lattice, all four of these valence electrons play a role in holding the atom in place in the lattice and they are not free to move through the crystal — the valence band is fully occupied. When some impurity P atoms are introduced, each impurity atom bonds to four silicon atoms using up four of

[1] An undoped semiconductor is usually referred to as an *intrinsic* semiconductor. If it is doped, it is *extrinsic*. [Editors]

the five valence shell electrons of the phosphorus. This leaves an extra electron per P atom free to roam through the metal and carry a current (Fig. 7.2):

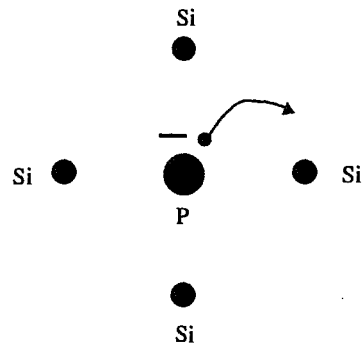


Fig. 7.2 Liberation of a Phosphorus Electron During Doping

The resulting material is called an "n-type" semiconductor, as there is an excess of *negative* charge carriers. At modest levels of doping, substances of this sort conduct quite weakly compared to metals; the latter may have one or two free electrons per metal atom whereas an n-type semiconductor has but one electron for each phosphorus atom.

There are very few holes in n-type Si, even when the temperature is high enough to dislodge electrons thermally, because holes in the lower band are filled in by the P electrons preferentially before they fill levels in the conduction band. The venerable "Law of Mass Action", as used for chemical reactions, gives an important relation between electron and hole densities, n_e and n_h respectively, and one which, interestingly enough, is actually independent of the fraction of dopant in the material:

$$n_e \, n_h = n_i^2 \tag{7.1}$$

where n_i is the density of electrons and holes at that temperature for pure, undoped Si. (This relationship is pretty obvious for undoped Si, since we must have $n_e = n_h$.) Ideally, we would like to be able to design components which still work when material specifications such as n_i or the temperature are slightly but unpredictably changed.

Another type of doping involves replacing selected Si atoms with atoms from group 3 of the Periodic Table. Thus we could add an impurity atom such

as Boron (B) which has one less electron than Si in its outer shell[2]. If we do this, then clearly we will find ourselves with an excess of *holes* rather than electrons, and another type of semiconductor. Due to the wonders of the laws of electromagnetism, holes can be viewed rather like bubbles of positive charge in an electric field — just as air bubbles in a liquid go up in a gravitational field (having an effective "negative weight"), so do holes go the "wrong way" in an electric field. Since they act like positive charges, B-doped Si is called "p-type" Si to indicate this. Note that, once again, relation (7.1) still holds.

7.1.1: The np Junction Diode and npn Transistor

We will now look at what it is about semiconductors that makes them useful in the manufacture of parts for computers. We start by examining the particularly interesting situation that occurs when slabs of p-type and n-type silicon are brought into *contact* with each other. This forms the basis of a device called a *diode*. We will give an idealized, qualitative discussion, and not allow ourselves to get bogged down in the murky details. We envisage a situation like that shown in Figure 7.3:

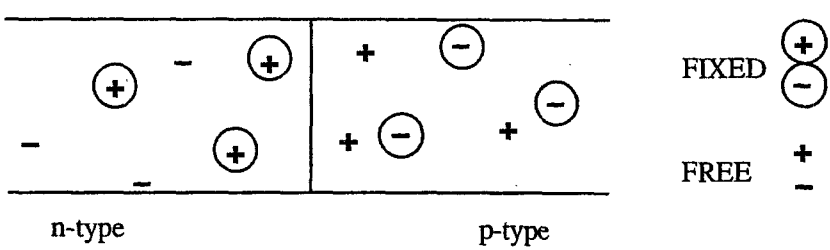


Fig. 7.3 The np Junction

On the left hand side we have the n-type material, which we can view as comprising a bunch of fixed positive charges and free negative charges. On the right hand side, we have the opposite situation. We know how many free electrons and free holes there are since they match up with the extra fixed B and P atoms — one electron per P, and one hole per B atom. At room temperature there will also be extra carriers due to thermal fluctuations.

[2] Feynman actually used Aluminum instead of Boron in his lectures. Aluminum immediately precedes Si in the Periodic Table but has rarely, if ever, been used as a valence-3 dopant. [Editors]

In a moment we'll stick this device into a circuit and put a voltage across it. First, let's see what's going on in the absence of any such field. The charge carriers will not only move about within their respective halves but will billow out, like steam escaping, into the adjacent material. However, this process of diffusion — of electrons into the p-type material and holes into the n-type — does not go unopposed. The fixed positive charges in the left hand block will create an electric field that tends to pull the escaping electrons back; this field is increased by the holes percolating into the n-type material. These holes also experience a "tug", from the fixed negative charges in their original half, and the electrons that have migrated over the barrier. We can actually list four separate phenomena operating at the join of our two slabs:

(1) Creation and annihilation of electron-hole pairs by thermal fluctuations,

(2) Conductivity (carrier drift prompted by electric forces),

(3) Diffusion (carriers trying to smooth out the charge density), and

(4) Electrostatic processes (due to the fixed charges).

After a while, this complicated physical system will settle down into an equilibrium state in which there is a concentration of fixed charges either side of the junction (Fig. 7.4):

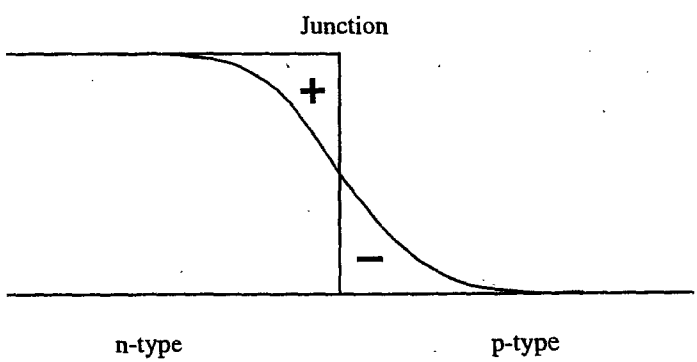


Fig. 7.4 The Equilibrium Charge Distribution for an np Junction

The central region is actually depleted of charge carriers and is referred to as the *depletion* region. The density of the fixed charges in this region is not quite [illegible] different effective mass, but we will treat

them as the same. We can add the signed densities of electrons, holes, and the fixed charges to obtain the net charge density in the device (Fig. 7.5):

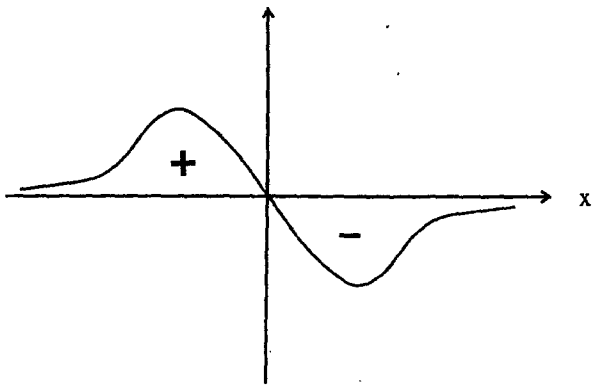


Fig. 7.5 The Net Charge Density

The physical situation in equilibrium, it must be stressed, is not static. There are currents flowing due to diffusion. However, in equilibrium, the current flowing to the right cancels that to the left, resulting in no *net* current flow.

Let us see what happens if we apply a voltage across this system. We have two choices as to how we hook up our battery: we can connect the positive terminal to either the n-type or the p-type material. Let us consider connecting it to the p-type material first. If you think about it, you should be able to see that the effect will be to reduce the opposition to current flow caused by the fixed charges in the depletion region — the positive potential on the right will attract the electrons from the n-type material into the p-type. As the voltage increases, more and more electrons are able to diffuse across the boundary, and more and more holes, of course, can go in the opposite direction. Put bluntly if we wire it right our "device" conducts madly! (An important rider to this bald statement is that it is *essential* for the maintenance of a current that "external" free electrons be drawn into the n-type part of the material from the point of contact with the battery, to continually replenish the flow. This is necessary because many of the "indigenous" carriers in the semiconductor will recombine with their opposite charges once over the boundary.)

What happens if we apply the voltage in the opposite direction? Now it gets interesting — we find that *the material does not conduct!* Why so? Well any free electrons in the n-type material can happily go left, away from the junction region, and free holes in the p-type can go right, flowing out of the

semiconductor and into the circuit. However, the application of the voltage has increased the height of the potential barrier across the depeletion zone, in fact, to the point where *electrons in the p-type material cannot traverse it.* (Needless to say, neither can any electrons that might be sourced from the point of contact with the battery.) Similarly, holes cannot maintain a current to the right, so after an initial blip, the current just drops off. There are too few free carriers available in the right places to sustain it. We say that the voltage *reverse-biases* the junction: in the current flow condition, the junction is said to be forward-biased. We call this device a *junction diode* and it has the fundamental property that it conducts in one direction but not the other.

Is there absolutely no current when the junction is reverse-biased? Well, not quite — there will be some flow due to thermal electron-hole pair creation at the junction. We wait for it to happen, it happens, and the electron scoots off in one direction, the hole in the other. The magnitude of the current created will clearly be temperature dependent — it actually increases exponentially with temperature — and largely independent of the applied reverse voltage. If we wish, we can aid the thermal process in a reverse-biased diode by creating electron-hole pairs ourselves. The thermal current is typically so small that, if we do create any pairs, we can easily detect them over the thermal background. How we go about producing pairs depends on the magnitude of the semiconductor's band gap. In gallium arsenide (GaAs), for example, we can create pairs utilizing photons (in fact, this process is quite efficient even in Si). Naturally, the reverse is true: if we flip the electric field so that the diode becomes "forward-biased" and current can flow freely, electrons and holes move towards each other and annihilate, producing photons (in GaAs) or phonons (in Si). In this way, we can make semiconductor LASERs and LEDs.

Recall that electrons and holes annihilate at a rate n_i^2. If this were zero, then in the reverse-biased case there would be no current since, by Equation 7.1, n_e and n_h would have to vanish also! However, in the forward-biased case, current could flow by filling up the p-type material farther and farther from the junction with electrons. If the applied voltage were to flip again, these electrons would have to all be laboriously brought back, and the diode would no longer prevent current flowing the wrong way — the device would be acting like a large capacitor. Annihilation solves this problem by letting the electrons in the p-type material fill in holes rather than be stored at increasingly great distances from the junction. So in fact, the diode would not really work without this annihilation process.

Let us return to the case when the field allows current to flow fully. It is

possible to calculate the current I that flows through the diode as a function of the applied voltage V. The math isn't exceptionally difficult, but we will not go into it here (see suggested reading). The relationship turns out to be non-linear:

$$I(V) = I_0 \, [\exp(qV/kT) - 1] \tag{7.2}$$

where V is the effective potential difference across the device — the voltage after allowing for a voltage in the "wrong" direction caused by the fixed charges at the boundary — and q is the magnitude of the charge on the carriers (in Coulombs) and I_0 is a constant. The graph of I against V can now be drawn (Fig. 7.6):

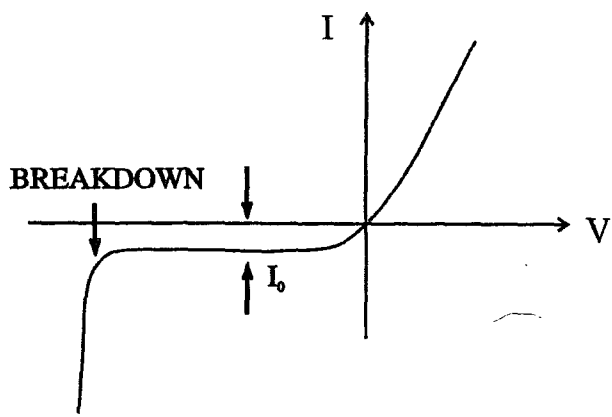


Fig. 7.6 Variation of Diode Current With Voltage

In the real world, $I(V)$ cannot just keep on growing exponentially with V; other phenomena will come into play, and the potential difference across the junction will differ from that applied. Note also that the current trickle that exists in the reverse-biased case, catastrophically increases (negatively) at a certain voltage, the so-called *breakdown* voltage. This can vary from as few as five volts to hundreds, and can actually be exploited in some situations to limit the voltages in a circuit.

The next step in this chapter is to look at the operation of another semiconductor device, the famous *transistor*. Now transistors come in all shapes and sizes, and those that are deployed extensively in VLSI chips are quite different from those that are used by hobbyists. As an example of this latter type we mention very briefly the venerable *npn bipolar junction transistor*, one of

several historical antecedents of more modern transistor devices. This transistor is formed by sandwiching an extremely thin slice of p-type material between two of n-type material (hence "npn"). The various slabs are denoted the base, the collector, and the emitter, as shown in Figure 7.7:

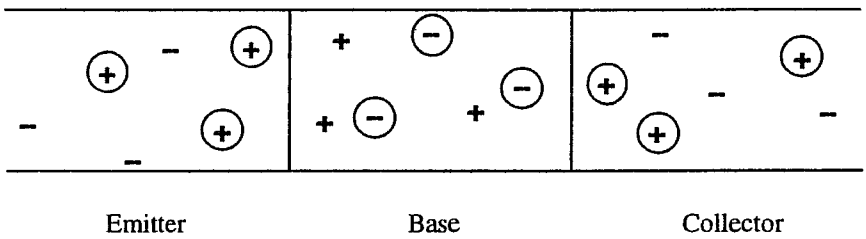


Fig. 7.7 Structure of the npn Bipolar Junction Transistor

The "base" got its name from the fact that the transistor was built of this material: the terms "emitter" and "collector" both derive from prehistoric vacuum technology. Note the relative thinness of the base to the slabs on either side of it — this feature is essential to the transistor's correct operation. What this device is, if operated properly, is an *amplifier*: small changes in the current to the base are amplified at the emitter. It can also act as a switch and can be used in all of the transistor circuits discussed thus far in this book. However, bipolar transistors are not the most commonly used transistors in modern VLSI chips and I will therefore not discuss the (complicated) mode of operation of this transistor at the electron and hole level here. Good discussions can be found in many standard texts (see suggested reading). Instead, we will take a look at the type of transistor that *is* most commonly employed in VLSI systems — the sort of transistor that is usually built onto silicon chips. This is the MOSFET, an acronym for Metal Oxide Semiconductor Field Effect Transistor.

7.1.2: The MOSFET

We begin with a sketch of the structure of the MOSFET (and will worry about how to actually build such a device on a chip later):

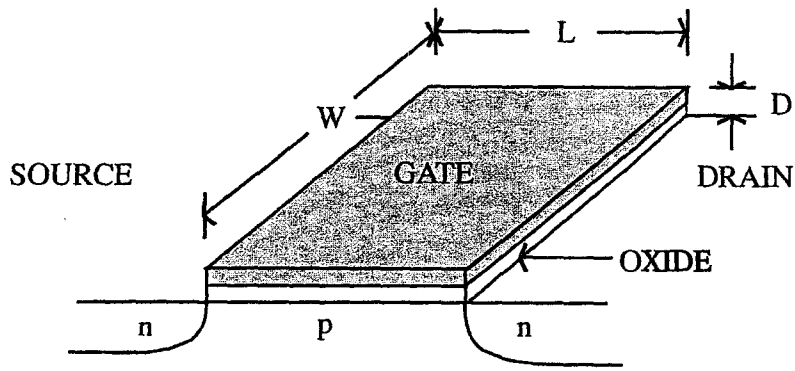


Fig. 7.8 The MOSFET

The bulk of a silicon chip consists of a slab of lightly doped silicon, the substrate onto which the transistors and whatnot are laid down. If the doping is of the *p*-type, we are dealing with so-called *nMOS* technology; if the substrate is *n*-type, we have *pMOS*. We will focus on nMOS, as shown in Figure 7.8. We can identify three ports for our MOSFET: the *gate* is a conducting layer of "polysilicon" (a substance rather like a metal), which is separated from the silicon by a thin layer of non-conducting oxide. To either side of the gate, also separated from it by the oxide layer, are two n-type *diffusion layers*, materials so-called because they have doping diffused into them (as illustrated): they are referred to as the *source* and *drain* — as opposed to the emitter and collector of the npn device. These diffusion layers also conduct.

The transistor works as easy as pie. From our discussion of the np junction diode, we can see roughly what's happening inside the device before we apply any external voltages. The substrate material is lightly p-type, and is usually kept grounded. The substrate forms a diode-like structure with the n-type layers — in essence, the MOSFET is built from two back-to-back diodes. As before, an equilibrium state will arise in which there are depletion regions at the n-p junctions with very few electrons and holes diffusing across them. So, the source and drain are effectively cut off from one another and if a voltage is put across them no current will flow. However, if we now put a positive voltage on the gate, things are different. The effect of such a voltage is to attract electrons to the underside of the oxide (they won't conduct into the gate because the oxide layer is an insulator). These electrons chiefly come from the source and drain — the positive potential having lowered the barriers at the depletion layers

which prevented their free movement. The electrons under the oxide form what is called an *inversion layer*. If we *now* put a voltage across the source and drain, we *will* get a current — the inversion layer is essentially a channel that allows electrons to flow freely between the two contacts. So we have a fantastic device — a switch! The voltage that controls it is the gate-source voltage V_{gs} — the bigger this is, the more charge carriers there are under the gate, and the more current that can flow. Note, however, that if $V_{gs} < 0$, the MOSFET will not conduct. In such a case, electrons are repelled from beneath the gate, and an inversion layer cannot be formed.

Actually, we have been a little simplistic here. Simply having $V_{gs} > 0$ does not automatically allow a current to flow. It is necessary for V_{gs} to exceed a certain minimum voltage, the *threshold* voltage V_{th}, before this happens (typically, V_{th} is of the order of $0.2V_{DD}$, where V_{DD} is the supply voltage, say five volts). Because of this, it turns out to be convenient to define a shifted gate-source voltage:

$$V'_{gs} = V_{gs} - V_{th} \tag{7.3}$$

in terms of which the condition for current flow is $V'_{gs} > 0$. We can actually design our MOSFET to make this threshold voltage either positive or negative. A negative threshold voltage is obtained by doping the silicon slab so that there is a thin conducting layer of n-type semiconductor under the gate, connecting the source and the drain. Transistors with $V_{th} > 0$ such as we discussed above are called "enhancement mode" transistors: if $V_{th} < 0$, they are called "depletion mode" transistors. (Depletion mode transistors turn out to be useful for fabricating resistors in nMOS VLSI, as we shall see later.)

Let's make all this a bit more quantitative. Suppose we want to find the drain-source current I_{ds} for given gate-source and source-drain voltages. Those readers not interested in details can skip this as we will not need it subsequently — it's just a nice bit of physics! We can consider the gate/oxide/silicon sandwich to be a capacitor, modeled by two conducting plates of area $A = WL$ (where W is the width of, and L the distance between source and drain), separated by a material of depth D and permittivity ε. Let us denote the charge on this capacitor (i.e. the charge under the gate) at any time by q_g. We can calculate the capacitance C_g for this system, using the well-known general formula for a parallel plate capacitor $C = \varepsilon A/D$. We have:

$$C_g = \epsilon WL/D. \tag{7.4}$$

Using the standard relationship between the voltage across a capacitor and the charge stored in it, we can write:

$$q_g = C_g V'_{gs} \tag{7.5}$$

Let us suppose first that the drain-source voltage V_{ds} is small. We know that the current I_{ds} is just the charge under the gate divided by the time it takes for the electrons to drift from the source to the drain. This is a standard result in electricity. How long is this drift time? Drawing on engineering practice, we can write the drift velocity v_{drift} in terms of the "mobility" μ of the charge carriers as $v_{drift} = \mu E$, where E is the electric field across the drain/source. E is easily seen to be given by V_{ds}/L. We can now straightforwardly find the drift time, which we denote by τ:

$$\tau = L/\mu E = L^2/\mu V_{ds} \tag{7.6}$$

Combining this expression with that for the charge q_g in Equation 7.5, we find that the current (charge divided by time) is given by:

$$I_{ds} = (\mu\epsilon W/LD)V'_{gs}V_{ds} \tag{7.7}$$

Our calculation has been a little simplistic: this result strictly only holds for small values of the source-drain voltage V_{ds}. However, we can see that as long as V_{ds} is fairly small, the transistor has the interesting property that *the current through it is proportional to the applied voltage*. In other words, it effectively functions as a *resistor* (remember $V = IR$!), with the resistance proportional to $(1/V'_{gs})$.

As the drain-source voltage increases, matters become more complicated. The drift velocity of the charge carriers depends upon the electric field E, and this in turn is determined by both V_{ds} and V'_{gs}. However, it so happens that, if V_{ds} gets too big, the current across the transistor actually becomes *independent* of V_{ds}, a phenomenon known as *saturation* — at this point, the current becomes proportional to $(V'_{gs})^2$. We can understand this strange phenomenon better by use of a fluid model analogy (described in more detail in the book by Mead and Conway). If you like water and gravity better than electricity, you should like

this!

Let us envisage two reservoirs of water, separated by a partition. We will actually take this water to be without inertia — a little like honey — so it will flow slowly, and not splash about all over the place. To begin, consider the state of affairs shown in Figure 7.9 where the "water" is on either side of a movable central partition:

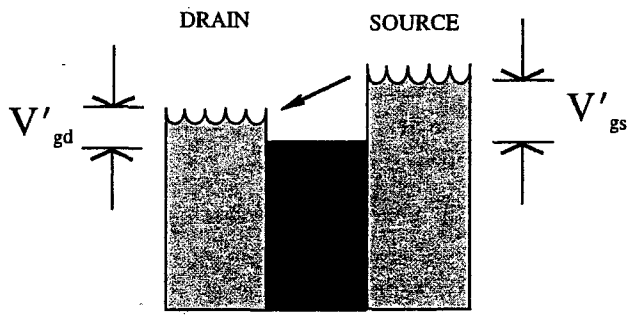


Fig. 7.9 Fluid Analogy for the Forward-Biased MOSFET

The diagram has been annotated with MOSFET-related words to force the analogy on the reader. In this case, there will clearly be a flow of water from the "source" to the "drain": in the transistor, this equates to a current, with the partition playing the role of the potential barrier the charge carriers have to overcome. In this situation the transistor is said to be "forward-biased". The height of the water column above the partition on the right represents V_{gs}, of the left column above the partition, V_{gd}, and so on. Note that the precise nature of the flow across the partition will depend on the level of the "water" in the drain reservoir. Consider now a second state shown in Figure 7.10:

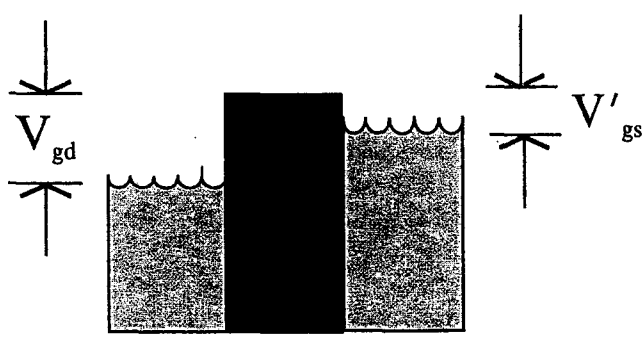


Fig. 7.10 Fluid Analogy for the Back-Biased MOSFET

In this case there will be no flow at all: for the MOSFET, this represents the "back-biased" case, where $V_{gs} < 0$. Thirdly, consider the case of saturation (Fig. 7.11):

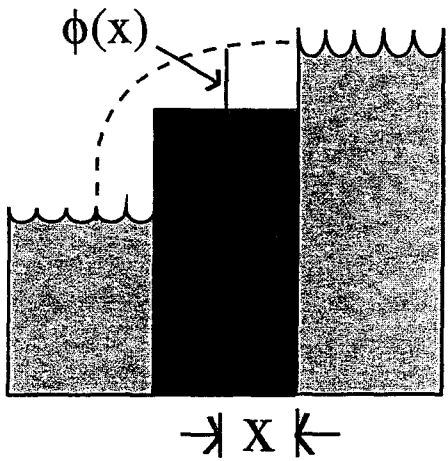


Fig. 7.11 Flnid Analogy for the Case of Saturation of the MOSFET

In this instance, where the level of the drain reservoir is below the level of the partition, the water from the source will simply "waterfall" into the drain, at a rate *independent of the actual drain-level.* In the MOSFET, this behavior would begin to occur when the "drain reservoir" and the "partition" were at the same height, e.g. when $V_{ds} = V'_{gs}$. At such a point, the current flow will become constant, irrespective of V_{ds}.

We can implement this analogy physically to gain some insight into the process of saturation. This is an alternative derivation of the magnitude of the current flowing through the MOSFET. Let us take Figure 7.11 to define a voltage $\phi(x)$ beneath the gate which, due to the energy level structure of Si, happens to be proportional to the number of free electrons under the gate. Now the electric field under the gate is proportional to the derivative of this voltage ($d\phi/dx$). Hence, since the current must depend on the density of electrons multiplied by the electric field (which controls the electron drift velocity), we must have:

$$I = K\ \phi(x)\ d\phi/dx = (K/2)\ d/dx\ (\phi^2) \qquad (7.8)$$

where K is a constant. Now if you think about it, you should be able to see that the current I has to be a constant, independent of x. This means that the function ϕ^2 must be linear in x and thus $\phi \propto \sqrt{x}$. We therefore obtain the general

expression:

$$I = (K/2L)\ (\phi^2[0] - \phi^2[L]) \tag{7.9}$$

However, at saturation we have $\phi(0) = V'_{gs}$ and $\phi(L) = 0$ so that we recover the quadratic dependence of I on V_{gs} that we flagged earlier. What happens if we are not at saturation? Will we rederive Equation 7.7? Actually, no. We get something better. In the unsaturated case, $\phi(L) = V'_{gd}$, the gate-drain voltage, and we find that:

$$I \propto (1/2)\ (V'_{gs} + V'_{gd})V_{ds} \tag{7.10}$$

where $V_{ds} = V'_{gs} - V'_{gd}$. Why is this an improvement? It removes an anomaly in our earlier expression (Equation 7.7). In that expression, the current I was strangely asymmetric between source and drain. Intuitively, one might prefer to replace it with an average over the gate-source voltage and the gate-drain voltage — that is just what Equation 7.10 amounts to. For the record, we give here a plot of the variation of the current with the various voltages we have considered in this section (Fig. 7.12):

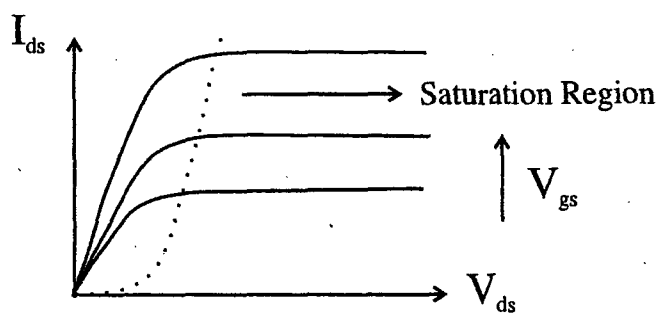
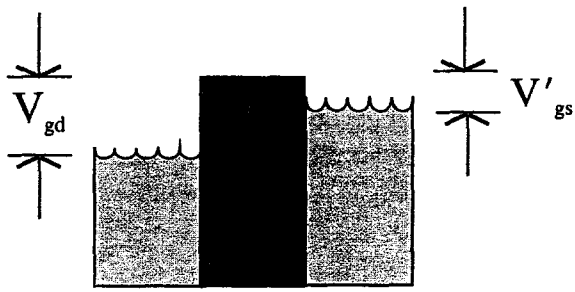


Fig. 7.12 Current-Voltage Variations for MOSFET

Another question we can ask about the MOSFET is: in the state for which no current flows, how good an insulator is the device? You will remember that the junction diode permitted a small thermally-induced current to flow even when reverse-biased. Let us briefly discuss this effect in the transistor. Recall Figure 7.10, the "water" diagram for the back-biased case:



The probability of an electron jumping from the source to the drain must be proportional to the probability for it to have enough thermal energy for it to overcome the potential barrier V'_{gs}, namely, $\exp(-qV'_{gs}/kT)$. Thus the forward current will also be proportional to this factor. There will also be a backward current — we can similarly see that this will be proportional to $\exp(-q[V'_{gs} + V_{ds}]/kT)$. There will therefore be an overall current given by

$$I = (constant).\exp(-qV'_{gs}/kT)[1-\exp(-qV_{ds}/kT)] \qquad (7.11)$$

when $V'_{gs} < 0$. So, if we switch from a current-flow to a back-biased case, the current will not switch off instantly (note that the temperature of the device is likely to be higher than room temperature). However, if we have $kT \sim 1/40$ electron-volt, then the current turns off quite quickly for circuitry run at about 5 volts. We can amend Figure 7.12 to include the reverse current:

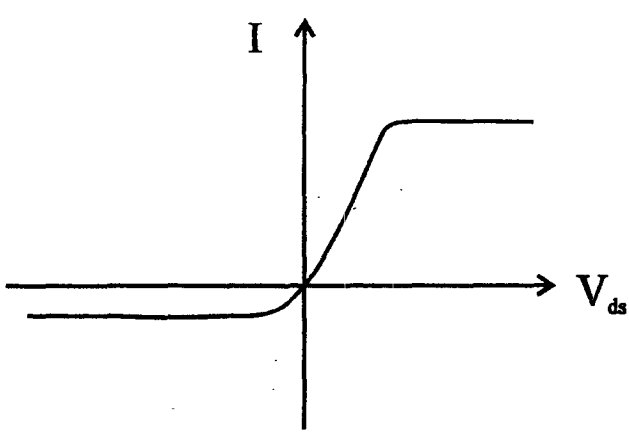


Fig. 7.13 Transistor Current as a Function of Source-Drain Voltage

Let us take stock and summarize what we have learned about the properties of MOSFETs. Firstly, rather than redraw the physical picture of Figure 7.8 every time we want to discuss the device and draw circuits involving it, we'll need a diagrammatic symbol. There are many such symbols for the differing varieties of transistors encountered in VLSI technology. Figure 7.14 illustrates the fairly common convention for the MOSFET that we shall adopt:

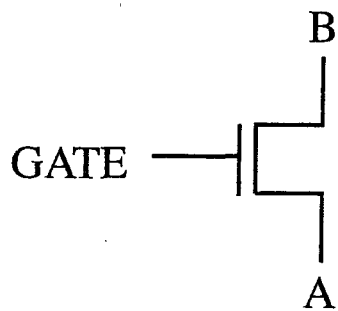


Fig. 7.14 Conventional MOSFET Symbol

This can represent either an n- or p-type transistor. (We will show later how to amend this figure to indicate which — this is necessary in CMOS technology which uses both types.) The rule for both is that they act like switches — when the gate voltage V_g is positive enough they conduct. For the n-type MOSFET, we have the following rules: the most negative of A and B is called the source, the other called the drain; and if the gate voltage is more positive than a certain threshold voltage, V_{th}, above the source voltage the device conducts — the switch is "closed" and current flows. For a p-type device, the most positive of A and B is the source and the transistor conducts if the voltage on the gate is more *negative* than a certain threshold *below* the source voltage. We also defined two modes of operation for a MOSFET — the *enhancement* and *depletion* modes. The former is the case when the threshold voltage, V_{th}, > 0 for n-type and < 0 for p-type. In depletion mode, it is the other way around. Now, a nice feature of depletion mode MOSFETs is that, if $V_g = V_s$ (the source voltage), the device always conducts. Thus, if we directly connect the source to the gate so that each is automatically at the same voltage, we find our transistor acting no longer as a switch but as a *resistance* (Fig. 7.15):

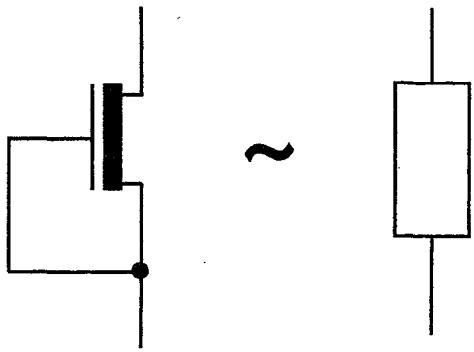


Fig. 7.15 Depletion Mode MOSFET operating as a Resistor

Why should we want to do this? It's a matter of simple economics and design. Implementing a standard resistance on a chip is both expensive and takes up a lot of space, neither of which are obstacles to the use of depletion mode MOSFETS[3]. (Note, incidentally, that this trick would not work with an enhancement mode device.)

Problem 7.1: I will now set you a problem — I will actually give you the answer shortly, but you might like to work through the math to get some practice on the theory of the internal guts of these things. The question has to do with capacitance. In the diode, not only was the current a non-linear function of the voltage but so was its capacitance. This is also the case for the MOSFET. Now although most of the capacitance is in the oxide, the overall set up turns out to be actually highly non-linear — and quite interesting! The problem I am about to give you is designed to illustrate the nature of this non-linearity.

We model the electrode contact at the gate in the MOSFET as shown below in Figure 7.16. Suppose we have a large mass of lightly doped p-type material — in principle, this material should be of infinite depth (measured vertically in the diagram). On top of this we place a metal plate, also of very large extent (but now in the horizontal plane):

[3] With the advent of CMOS technology (see later) nMOS depletion mode transistors are now rarely used; they are replaced by n-channel enhancement mode transistors. [Editors]

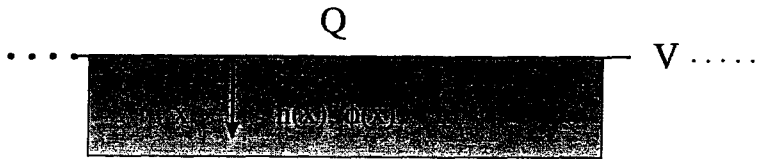


Fig. 7.16 Schematic Model of the Gate-Semiconductor Electrode Contact

We put a positive voltage V on the plate and, as a result, negative charge carriers are attracted to its underside. Deep into the material, where the electrostatic forces due to the plate are negligible, the number of negative carriers will just equal the number of doping ions. However, because of the voltage on the plate, the density of carriers near to it will be greater, falling off to a constant as we go further away. If we label the depth into the material by x, we can define a position-dependent carrier density $n(x)$. We can also define a resulting electrical potential within the material, $\phi(x)$. Finally, let us denote by n_0 the initial positive doping concentration. Now the question I want you to answer is: how much charge is there on the electrode? Put differently, what is the capacitance of this physical set up?

To help you practice, I'll give you some hints. Firstly, you have to take $\phi(0)$ (i.e., at the electrode) to be the plate voltage V, and take the idealized value $\phi(\infty)$ to be zero. I will hand you on a plate a relationship between $n(x)$ and $\phi(x)$ resulting from thermodynamical considerations:

$$n(x) = n_0 \exp[q\phi(x)/kT] \tag{7.12}$$

where q is the charge on the negative carriers, and T is the temperature, as usual. n_0 is a constant. Another essential relationship is that between the rate of change of $\phi(x)$ as we go deeper into the material (in other words, the electric field within the semiconductor) and the charge density on the plate, Q. Note that defining a charge *density* here is important — it would be meaningless to discuss the *total* charge for an *infinite* plate. We find (cf. Equation 7.4 or by Gauss' theorem) the result that at $x = 0$:

$$\partial\phi/\partial x = Q/\epsilon. \tag{7.13}$$

where ε is the *permittivity* of the doped material and determines how rapidly the

electric field drops off with distance from the plate. Using the standard Poisson equation $\partial^2\phi/\partial x^2 = -\rho(x)$, where $\rho(x)$ is the charge density within the material (which you can find in terms of $n(x)$ and N) and integrating using the boundary conditions, you should find the result of the form of Equation 7.14:

$$Q = V[2(e^V - V - 1)/V^2]^{1/2} \tag{7.14}$$

in some set of units ($kT/q = 1$ and $n_0 q\varepsilon = 1$). The rather odd appearance of a V^2 within the square root, which you might think ought to cancel with the V outside it, is necessary to get the correct sign for Q. Now you can see by comparison with the standard formula defining capacitance, $Q = CV$, that the capacitance of this system displays an extremely non-linear relationship with the plate voltage, V. To my knowledge, this property is not much exploited in VLSI — although there have been recent applications in "hot clocking" (which we discuss later).

Thus far, we have considered an isolated MOSFET device on a silicon substrate. The next stage in our journey into the heart of VLSI is to take a look at how these transistors might actually be put together on chips to make logic circuits. We now come to real machines!

7.1.3: MOSFET Logic Gates and Circuit Elements

To build logic circuits we need to be able to build logic gates and we have already seen, in Chapter Two, how to do this using generic "transistors". We use the same approach with MOSFETs. Consider what happens when we hook up a transistor to a supply voltage, V_{DD} across a resistance as shown in Figure 7.17:

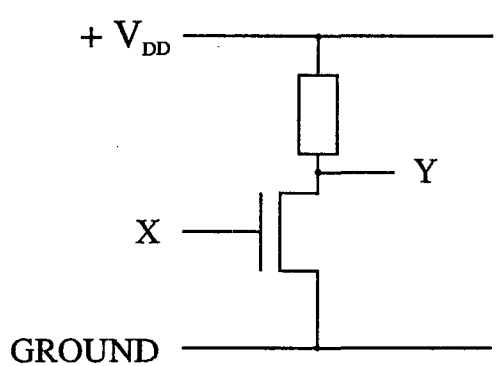


Fig. 7.17 Inverter Circuit

We will take our transistor to be of the nMOS variety, operating in enhancement mode. (There are many types of VLSI design, and we cannot consider all of them — it makes sense to focus on one in particular.) If terminal X (the gate) is near zero, then the transistor is an insulator, and the output voltage at Y is near the supply voltage, V_{DD}: we interpret this state of affairs as meaning that the output Y is at logical 1. However, if X is near V_{DD}, then the transistor conducts. If we suppose it conducts much better than the resistance, then Y is near zero: this state we equate with logical 0. As a rule, we do not operate between these extremes, except perhaps temporarily. This single MOSFET device therefore operates as a NOT gate (an inverter), as we saw in Chapter Two, since it just flips the input signal.

We can follow Chapter Two's lead for the other canonical gates. For example, the NAND (NOT AND) gate is built as follows (Fig. 7.18):

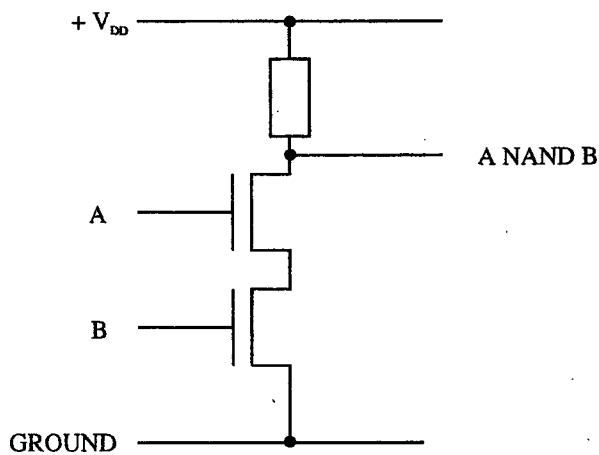


Fig. 7.18 The NAND Gate Realized by MOSFETS

In this system, both inputs A and B must be logical 1 for the output Y to be 0. To get the AND gate, we obviously just tag an inverter onto the output. To remind yourself of how to get a NOR gate, check out Chapter Two!

One can build other useful elements onto chips using MOSFETs apart from logic gates. Consider the matter of resistors — as I stated earlier, it turns out to be expensive and area-consuming to put standard forms of resistor onto silicon chips so it is normal practice to employ depletion-mode transistors in this role. Thus, in nMOS technology, the MOSFET structure of our inverter would actually be that shown in Figure 7.19:

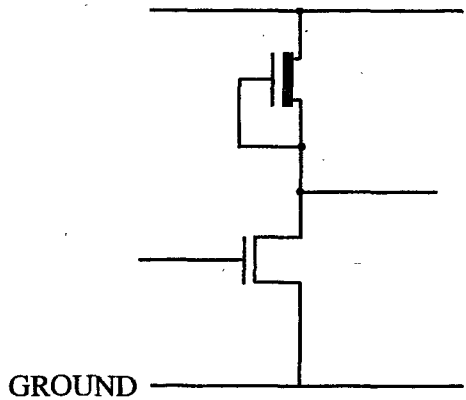


Fig. 7.19 nMOS Inverter with MOSFET Resistance

Now there is another essential property of MOSFETs that is not evident from strictly logical considerations. This is their behavior as *amplifiers*. Consider what happens if we place two inverters in sequence (Fig. 7.20):

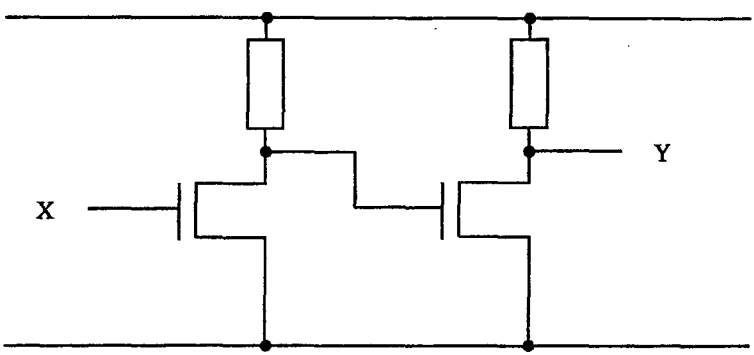


Fig. 7.20 "Follower" Circuit

From a logical viewpoint, this is a pretty trivial operation — we have just produced the identity. We're not doing any computing. However, from the viewpoint of machinery, we have to be careful; transistors dissipate energy, and one might naively think that the output of a chain of devices such as this would ultimately dwindle away to nothing as the power dropped at each successive stage. This would indeed be disastrous! However, this clearly doesn't happen: the input current to the second transistor may drop slightly, but it will not be enough to alter the mode of operation of this transistor (i.e. conducting or not), and the output Y will still be pulled up to the supply voltage (or down to

ground, whichever is appropriate). In other words, the output will always represent a definite logical decision, being relatively insensitive to minor power fluctuations along the chain. This circuit is an extremely effective so-called "follower", which jacks up the power or impedance behind the line (if you like, it is a double amplifier). In a sense, we can control the whole dog just by controlling its tail. Needless to say, this amplifying property is crucial to the successful operation of circuits containing thousands or millions of transistors, where we are constantly needing to restore the signals through them. The presence of amplification is essential for any computing technology.

With VLSI, as with other areas of computing, we are often concerned with matters of timing. In this regard, it is interesting to ask how fast an inverter can go. That is, if we switch the input at the far left of a chain of connected inverters, what happens at the output on the right? The switching certainly won't be instantaneous: the output of each transistor must feed the input of another and charge up its gate, and this will take time. Each gate voltage must be changed by some value V with the gate having some effective capacitance C_g, say. If we can find how long the process takes, and maybe think up ways of speeding it up, we might be able to get better machines. We can shed some light on this process by examining the circuit depicted in Figure 7.21, in which we have explicitly inserted a capacitor to represent the gate capacitance:

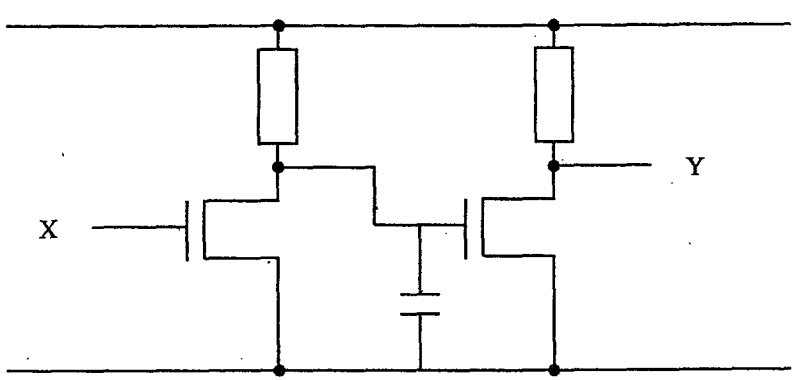


Fig. 7.21 Effective Electrical Analogue of a Follower Circuit

Suppose the accumulated charge needed for a decision (i.e. for the gate voltage to be adequate for the transistor to switch) is Q. Then, $Q = C_g V$. How fast can we deliver this charge, or take it away? Firstly, note that the state $X = 1$ does not correspond to the first transistor's output being exactly at ground; the transistor will have a certain minimum resistance (which we call R_{min}) resulting

in a slight voltage drop across the device. Now it is a standard result in electronics that the discharge time is determined by the product $R_{min}C_g$, assuming an analogy with the standard RC circuit shown in Figure 7.22:

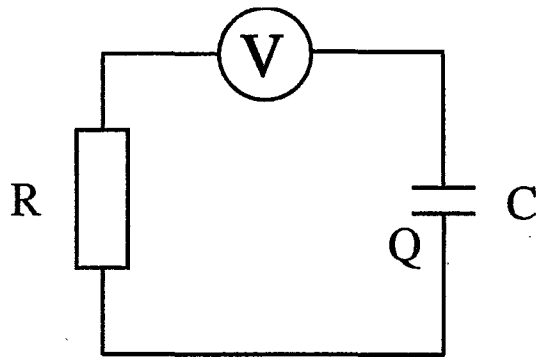


Fig. 7.22 Equivalent RC Circuit

Again from standard circuit theory, the charge Q on the capacitor at time t, $Q(t)$, obeys

$$Q(t) \propto \exp(-t/R_{\min}C_g) \equiv \exp(-t/\tau), \tag{7.15}$$

with $\tau = R_{min}C_g$.

Clearly, if we were interested solely in getting the inverter to go faster, then we could achieve this by decreasing both R and C, something we could do by making the circuit smaller. However, there is a limit to this: recall that, even in an inactive state, electrons from the source and drain nonetheless seep a small distance into the silicon substrate of the MOSFET. As we shrink the device down, these carriers drift closer and closer to the opposite pole, until there comes a point where they actually *short-circuit* the region under the gate, and we will get a current flowing without having to manipulate the gate voltage. When this happens, it is back to the drawing board: a redesign is now needed, as the transistor will no longer work the old way. This is a nice example of how Nature places limitations on our technology!

So what do we do if we want to build smaller machines? Well, when the rules change, redesign, as I have said. Consider, for example, the case of aeronautical engineering with incompressible air and low-speed aircraft. A detailed analysis concluded that propeller-based machines would not work for

speeds in excess of that of sound: there was a "sound barrier". To get a faster-than-sound plane, it was necessary to go back pretty much to square one. At this moment in time, we have yet to find a fundamental limit on sizes for Si computers — there is no analogue of the sound barrier. This problem is an instance of how thinking differently from everyone else might pay dividends — you might blunder into something new! Currently, state-of-the-art devices have $RC \approx 10$ picoseconds[4]. By the time you have managed to reduce this significantly, you'll probably find that others will have undercut you using some other technology! This actually happened with superconducting computing devices: as researchers were working in this area, its advantages were continually disappearing as advances were made in conventional VLSI technology. This sort of thing is quite a common occurrence.

Thus far in this chapter, we have reviewed the structure of various semiconductor devices used in computing but have so far had little to say on the practical limitations in this area. We address some examples of this now, beginning with a discussion of the important topics of heat generation and power loss in computers.

7.2: Energy Use and Heat Loss in Computers

In Chapter Five, we pointed out that a typical transistor dissipates some $10^8 kT$ in heat per switch. This is a phenomenal amount — if we could get it down by a factor of ten or a hundred, we could simplify our machines considerably just by getting rid of all the fans! One particularly annoying problem with the nMOS technology we have discussed up to now is that even in the steady state of a MOSFET's operation — when $X=1$ ($Y=0$), say, and the transistor is merely *holding* this value, not changing it — *current flows continuously*. So even if our transistors aren't doing anything, they're throwing away power! Obviously, any technology that offers the hope of more economical behavior is worth exploring; and the *Complementary Metal Oxide Semiconductor* (CMOS) technology that we will look at in this section is just such a technology.

[4] When Feynman delivered his course, the value of RC was actually of the order of 4 *nano*seconds. This 400-fold improvement in timing is an illustration of the extraordinary rate at which VLSI technology has advanced. [Editors]

7.2.1: The CMOS Inverter

In the CMOS approach, we employ a mixture of n-type and p-type MOSFETs in our circuitry. The way in which we combine these to make a standard CMOS inverter is shown in Figure 7.33. As with the nMOS inverter, logical 1 is held to be near $+V$, for some voltage V, but logical 0 is not at ground but can be chosen to be at $-V$:

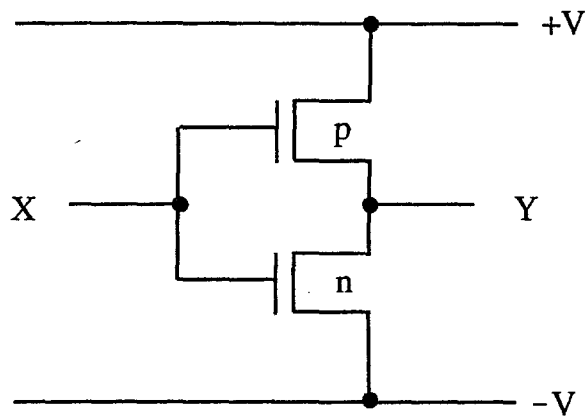


Fig. 7.33 The CMOS Inverter

To indicate the doping type of each MOSFET (n or p) we have followed the convention of writing the appropriate letter adjacent to its symbol. Note that the nMOS depletion mode transistor has effectively been replaced by a conventional p-channel transistor. Is this circuit worth building? Yes, for the following reason. Suppose the input X is positive. Then the n-type MOSFET has its gate voltage above that of the source and it conducts: the p-type device, on the other hand, is reverse biased and therefore *doesn't* conduct. The output Y is pulled down to $-V$. Now switch X to zero. As you can see from Figure 7.33, the upper transistor now conducts and the lower doesn't; the voltage Y rises to the supply. So far, nothing new — this is just the standard operation of an inverter. However, this circuit has a novel feature: specifically, after the transition occurs, *no current flows through the circuit!* The route to $-V$ is cut off by the insulating n-type MOSFET. (I'll leave it to the reader to see what happens when the input is switched back again.)

This is a remarkable property. In a CMOS inverter, no energy is required

to *hold* a state, just to *change* it[5]. The CMOS inverter can also serve as a useful simple 'laboratory' for investigating some of the energetics of logic gates. The matter of how much energy is required for a logical process was considered in the abstract in Chapter Five, but it is obviously important to get a handle on the practicalities of the matter. We would, unsurprisingly, like our devices to use the very minimum of energy to function — and to this end we will have to take into account the amount of energy required to make a decision, the time taken in switching, the reliability of our components, their size, and so forth. Let's start by considering in more detail the electrical behavior of a CMOS inverter as part of a chain. This will enable us to examine also the amplification properties of CMOS devices. To proceed we will employ a simplified (and none too accurate) model due to Mead and Conway.

In this model, we treat the two transistors simply as controlled resistors. We thus have the following equivalence (Fig. 7.34):

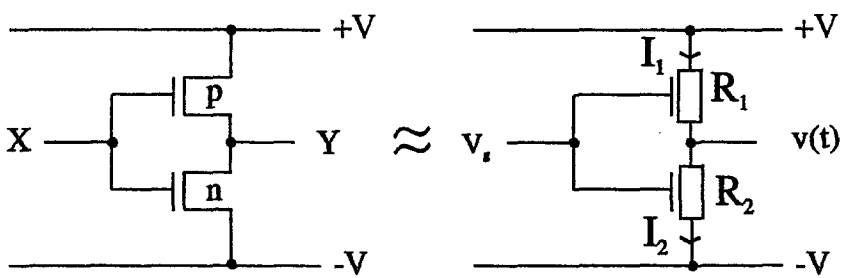


Fig. 7.34 Simplified Model of a CMOS Inverter

This CMOS device is to be visualized as one in a linear sequence. The input at X is fed in from the previous gate: the output at Y is to be considered the input to the next gate, which has an effective capacitance to ground of C, say (which we take to be a constant, although this isn't strictly true). Ultimately, we want to examine the behavior of the output voltage as we vary the gate voltage, $V_g(t)$, at X — i.e., as we perform a switch. Let us first consider the simple case where we keep the voltage to the I input gate constant. This will prompt a flow of current. What will the final, equilibrium voltage at the output be? Denote the

[5] Strictly speaking, there will be a small current flowing through the reverse-biased transistor, but we largely neglect this in our considerations. [RPF]

currents through the transistors by I_1 and I_2, and define the difference between them (that is, the current that transfers charge to any subsequent component connected to Y) to be $I = I_1 - I_2$. The voltage at Y is a function of time, say v(t). Let us also take the charge accumulating at Y to be Q(t). From standard circuit theory, we know that:

$$dQ/dt = I_1 - I_2 = Cdv(t)/dt \tag{7.16}$$

and from Equation 7.11, we can see that for small drain-source voltages the currents I_n are given by:

$$I_n \approx V_{ds\,n}/R_n \tag{7.17}$$

where the interpretation of $V_{ds\,n}$ is obvious, and the effective resistances R_n are given by

$$R_1 = R_0\exp(qV_g/kT),\ R_2 = R_0\exp(-qV_g/kT). \tag{7.18}$$

Note that we have sneakily removed all sign of the threshold voltage in V_g — we are considering our devices to be somewhat ideal.

If we now combine the basic equations for I_1 and I_2 given below:

$$I_1 = (V - v)/R_1,\ I_2 = (v + V)/R_2 \tag{7.19}$$

with the Equations 7.16 and 7.18, we can straightforwardly derive a differential equation relating C, $v(t)$ and R:

$$Cdv/dt = -(2V/R_0)\sinh(V_g/V_T) - (2v/R_0)\cosh(V_g/V_T). \tag{7.20}$$

So, if we keep the voltage on the gate fixed, what is the equilibrium value at the output, i.e. the value it has when everything has settled down? Well, when everything has stopped sloshing about, $dv/dt = 0$, and we see directly that the equilibrium value, v_e say, is given by:

$$v_e = -V \tanh(V_g/V_T) \tag{7.21}$$

where v_e is a constant. Since V_g/V_T is typically a large positive number or a large negative number, the equilibrium voltage asymptotically approaches $+V$ or $-V$.

We can use this result to analyze the amplification properties of a chain of CMOS inverters. Suppose we vary the gate input slightly, say let $V_g \to V_g + \delta V_g$. In response to this, the output will vary by some amount which we denote $\delta v_e = A.V_g$. In response to this, the output of the gate fed by v_e, v' say, will itself vary, by $\delta v' = Av_e = A^2V_g$; and so on down the chain. Clearly, if this CMOS device is to work, it must be the case that the magnitude of this factor $|A|$ is *greater than one*: if it were not, then any change of input at the left hand of the chain would not propagate all the way through and eventually peter out. The amplification factor A is the slope of the graph of v_e against V_g at the origin, $V_g = 0$ (Fig.7.35):

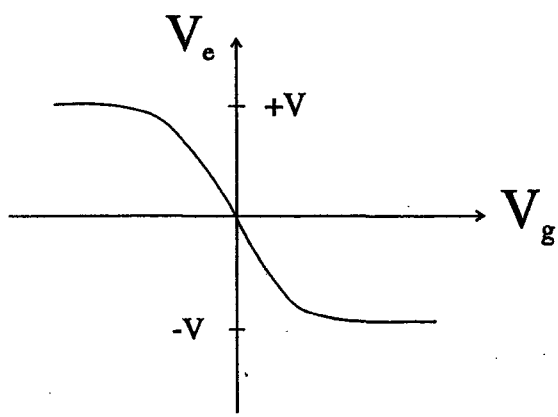


Fig. 7.35 Gate versus Output Voltages for the CMOS Inverter

The slope at the origin is $-V/V_T$ (as you can show). Hence, we only need our DC supply voltage to exceed of the order of 1/40th of a volt for the chain to work. In practice, of course, the supply voltage is much higher (say five or six volts) so we see that the amplification is quite significant. The output voltage is an extremely sensitive function of the input since small input changes are magnified many times at the output.

Problem 7.2: Here are some problems, not easy, for you to try. So far, we have considered the equilibrium behavior of a CMOS circuit. What I'd like you to do now is analyze its behavior in time, by solving the equation (7.20) to find how

long it takes the output to switch if we switch the input. The general solution, for which V_g is an arbitrary function of time, is obviously too difficult, so assume in your calculation that the input voltage switches *infinitely rapidly*. Next, consider the dissipation of energy in the inverter. I stated earlier that, while it is a useful qualitative idealization to think of no current flowing through the circuit in equilibrium, this is not actually the case (indeed, our previous calculation presumes otherwise!). The reverse-biased transistor just has a very high resistance. This results in a small perpetual power loss, which you can find using the standard electrical formula for the power dissipated by a voltage drop V across a resistance R: V^2/R, where R is the "non-conducting" resistance (alternatively, you could use I^2R, where I is the leakage current). There will also be power loss in the switching process — this occurs when we dump the charge on the gate through the (now conducting) resistance. You should find the energy lost during switching to be $2Cv_e^2$. Also, what is the time constant τ of the effective gate capacitance?

Although we are interested in CMOS technology chiefly for what it can tell us about the energetics of VLSI, for completeness I will briefly digress to illustrate how CMOS inverters can be used to construct general logic gates. Consider the implementation of a NAND gate — remember, if we can build one of these, we can build everything. A NAND gate then results from the arrangement shown in Figure 7.36:

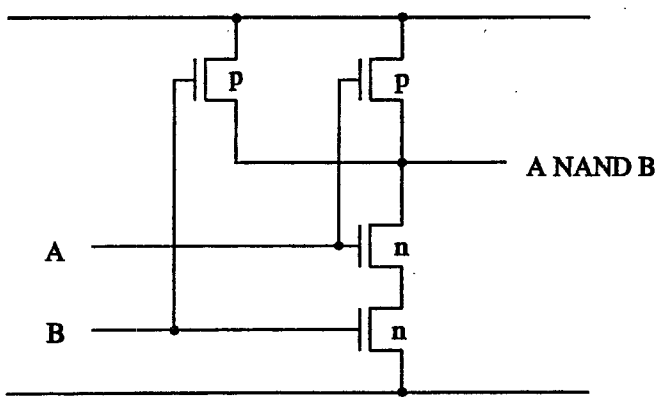


Fig. 7.36 NAND Gate Realised in CMOS

Let us see how this works. Recall that, for a NAND gate, the output is zero if both inputs are one, and one for all other inputs. That is clearly what will occur here: the output voltage in Figure 7.36 can only be pulled down to $-V$, i.e., logical zero, *if both of the lower transistors conduct.* This can only occur when

both inputs are positive. If either input is negative, the respective transistor will fail to conduct, and the output voltage will stay at $+V$.

Let us return to the matter of energy dissipation in CMOS devices. In practice, the energy dissipated per switch is of the order of $10^8 kT$. This is very big so here is an opportunity for people to make a splash in the engineering world: *there is no reason why it should be so high.* Obviously, the voltage must be a certain size depending on the technology implemented in our devices, but this is not a fundamental limitation, and it should be possible to decrease the energy dissipated. (Remember our analysis in Chapter Five where we saw that $kT\log 2$ was theoretically attainable.) Let us discuss what can be done in this area.

Consider what actually happens in the switching process. Before we make the switch, there is a voltage on the input capacitance and a certain energy stored there. After we switch, the voltage is reversed, but the energy in the capacitance is the *same* energy. So we have done the stupid thing of getting from one energy condition to the same energy condition by dumping all the juice out of the circuit into the sewer, and then recharging from the power supply! This is rather analogous to driving along the highway at great speed, slamming on the brakes — screeech! — until we come to a halt; and then pushing the car back up to speed again in the opposite direction! We start off at sixty miles an hour, and we end up there, but we dissipate an awful lot of energy in the process. Now, in principle it should be possible to put the energy of the car into (say) a flywheel and store the energy. Then, having stopped, we can get started again by drawing power from the flywheel rather than from a fresh source. We shouldn't have to throw the energy away. Is there some parallel in VLSI to this flywheel?

One suggestion is to store the energy in an inductance, the electrical analogue of inertia. We build the circuit so that the energy is not thrown away, but stored "in a box" so that we can get it out again subsequently. Is this possible? Let's see. To explain the concept of inductance, I'll turn to another useful analogy using water. Those of you who are electrically-minded are used to analogies between water and electricity: those more comfortable with mechanics than electricity will also find water is easy! Imagine we have the arrangement shown in Figure 7.37, consisting of a large water-holding vessel with a couple of pipes leading into it:

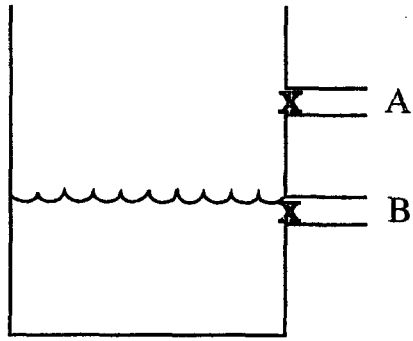


Fig. 7.37 Water Analogy for the CMOS Switch

Each pipe is connected to an essentially bottomless reservoir (not shown), into or from which water can flow — this flow is regulated by a valve on each pipe. The analogy here is that the pipes plus valves represent the transistors, and the water in the reservoirs is charge from the power supply just waiting to be dumped through them. The upper reservoir corresponds to the voltage $+V$, the lower to voltage $-V$, and the height of the water in the tank can be interpreted as the voltage through which the charge will be dumped. To keep the analogy meaningful, the valves are rigged so that if one is open (conducting), the other is closed (insulating). To model the switching process in an inverter, we open and close the valves in this system and see what happens.

The initial condition is that shown in Figure 7.37, with the upper valve closed and the lower valve open. The water sits at some equilibrium level. Suppose we now switch the system by closing the lower valve and opening the upper (corresponding to a negative gate voltage). The water from the upper reservoir rushes in — sploosh! — filling up the tank up until a new equilibrium depth is reached. In the process there is noise, friction, turbulence and whatnot, and energy is dissipated. There is a power loss. Eventually, everything settles down to a fresh equilibrium point. We now want to go back to our initial situation, so we switch again, opening the lower valve and closing the upper. Down comes the water level, dissipating energy in a variety of ways, until the water in the tank reaches its original height. We are back where we started, but we have used up a heck of a lot of energy in getting there!

We would like to alter this set-up so that we don't lose so much energy every time we switch. One way we could do this is as follows. We put another tank next to the first, and join the two by a tube containing a valve (Fig. 7.38):

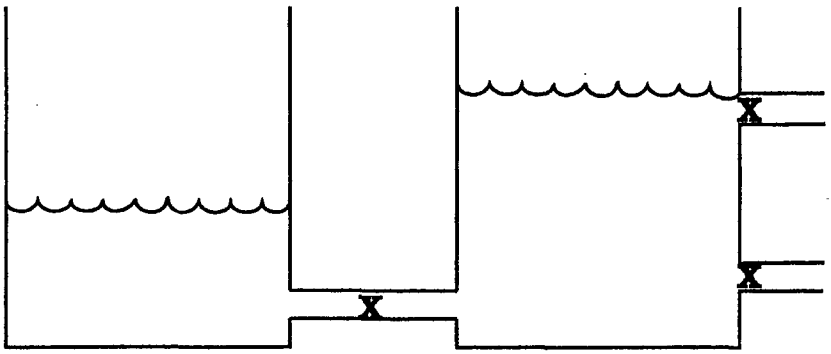


Fig. 7.38 Energy-Saving Analogy for the CMOS Switch

Suppose we have the upper valve open so the water level of our original tank is as shown in Figure 7.38. If we now close the upper valve and open up the valve into the adjoining tank, the water goes splashing through the connecting tube into the new tank. When the water level reaches its maximum height in this tank, we close the valve. If we were to just leave the adjoining valve open, the water would slosh back and forth, back and forth between the two tanks and eventually settle down into a state where the height in both tanks was the same. In this case the pressure would be equalized but this finite time to stability results from the fact that water has inertia. When the valve is first opened, the water level reaches a height in the new tank that is higher than what would be the equilibrium value if we let the system continue sloshing about. Likewise, the initial level in our original tank will be lower than its equilibrium value. By closing the valve just after this high point is reached, we have actually managed to catch most of the water, and hence its potential energy, in the new tank. Not all of it, of course – there will be losses due to friction, etc., and we might have to top the new tank up a bit. But now if we want the energy of the water back, we just have to open up the adjoining valve to the adjoining tank when the right-hand tank is at a low ebb.

To implement this in silicon we need the electrical analogue of this and that means we need the analogue of inertia. As I've said, for electricity this is inductance. One way to implement the above idea can be seen by considering the following circuit (Fig. 7.39):

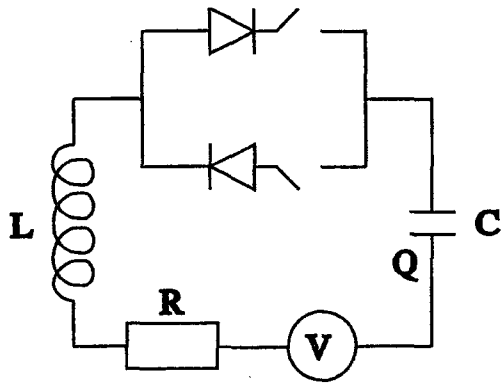


Fig. 7.39 An Inductive Circuit

This circuit contains a capacitor, an inductance, a resistance and two "check valves", based on diodes. When one of the switches is closed, the diode ensures that the current can only flow one way, mimicking the one-way flow of water through the two pipes in the water model. You should be familiar with the basic equation defining the behavior of the circuit:

$$Ld^2Q/dt^2 + RdQ/dt + (Q/C) = V \tag{7.22}$$

where V is the voltage across the circuit. I will leave it to you to see if you can implement this sort of idea using CMOS as the basis. Unfortunately, it turns out that it is extremely difficult to make appreciable inductances with silicon technology. You need long wires and coils and there's no room! So it turns out that this is not a practical way of getting the energy losses down. However, that need not mean we have to abandon the basic idea — a very clever thing we can try is to have just one inductance, off the chip, instead of many small ones, as in one per switch.

7.2.2: Hot-Clocking

Here is a completely different, and very clever, way to get the energy dissipation down. It is a technique known as *hot-clocking*. In this approach, we try to save the energy by *varying the power supply voltages*. How and why might something like this work? Let's return to our water analogy. Earlier, we saw that if we opened the upper valve while the level of water in the tank was low, then

we would lose energy as water flooded in from above and cascaded down. Where we are going wrong is in *opening the switch while there was a difference in water level.* If we do that, we will unavoidably lose energy. In principle, however, there are other ways of filling tanks which aren't nearly so wasteful. For example, suppose we have a tank to which is attached a single switched pipe, at the end of which is a water reservoir. If we fill the tank by the gradual process shown in Figure 7.40, opening the switch and moving the pipe up the tank as it pours so that it is always at the height of the water, then we will dissipate no energy:

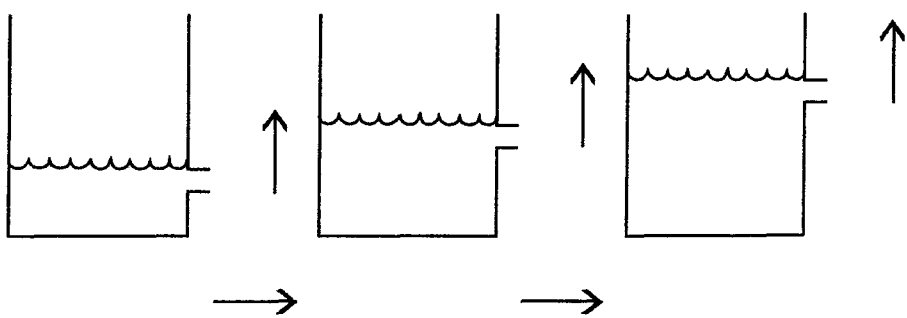


Fig. 7.40 Non-dissipative Filling of a Tank

Of course, we would have to perform the operation infinitestimally slowly to *completely* avoid a dissipative waterfall (this type of argument was used frequently in Chapter Five). However, it is clear that if we could move things so slowly, then we could really get the energy loss down as long as we never opened the switch when there was a difference in level between the pipe and the head of water in the tank. There is an analogous principle in electricity: *Never open or close a switch when there's a voltage across it.* But that's exactly what we've been doing!

Here's the basic principle of hot-clocking. Consider the amended inverter circuit in Figure 7.41:

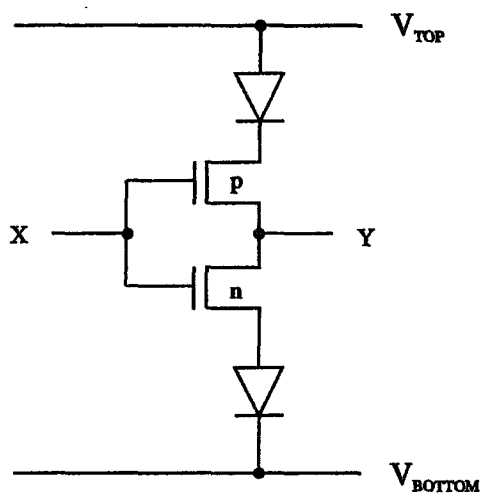


Fig. 7.41 Sample "Hot-Clocking" Circuit

In Figure 7.41, the upper and lower voltages V_{TOP} (= V, say) and V_{BOTTOM} (= $-V_{TOP}$) are *not* to be considered constant: they can, and will, vary, so watch out! We will define the two main states in which they can be as the *quiescent* state, which corresponds to the upper voltage being negative and the lower positive, and the *hot* state, the inverse of this, with the upper voltage positive and the lower negative. (These designations are arbitrary — we could just as well have them the other way around.) The principle of operation of this device is this. Suppose we start in the quiescent state, so the upper voltage in Figure 7.41 is negative, and have X positive (= $+V$). Then, the p-MOSFET is open, the n-MOSFET is closed, and no current flows (there is no voltage across the n device). In fact, even if X is negative, no current will flow due to the rectification property of the diode. So we can switch the input willy-nilly while in the quiescent state — the circuit is quite insensitive to the input voltage. This clearly leaves us free to choose our initial state for Y: we will take this to be positive.

Now, we let the voltages go hot — we gradually turn them around. Now a positive voltage gradually grows across the bottom diode, which conducts. This draws the output Y down to that of the lower voltage (which is now negative). The energy dissipation as this occurs is small as the resistance of the diode is low. When this lower voltage bottoms out and things have settled down, we switch back to the quiescent state again: the output Y would like to revert to its previous value but cannot, as the diode prevents any current from flowing.

We can change X, that is, make a switch, as we please once in this stable state. It is necessary to run the first part of the cycle, when Y changes, rather slowly; the second stage, the return to quiescence, can be performed rapidly.

Now the output of Y must feed another gate. Clearly, we cannot use it while it is changing so the voltage cycle of the next gate must take place somewhat out of phase, with a different power supply (rather like a two-phase clock). It is possible, as is common with flip-flops, to have the second signal simply the inverse of the first, and hence use just the one supply — but this is dangerous, and slightly confusing, as going back to quiescence allows Y to vary a little. It is safer to design conservatively, with two separate power supplies. We can exhibit this diagrammatically by plotting the voltage changes of the two supplies (Fig. 7.42):

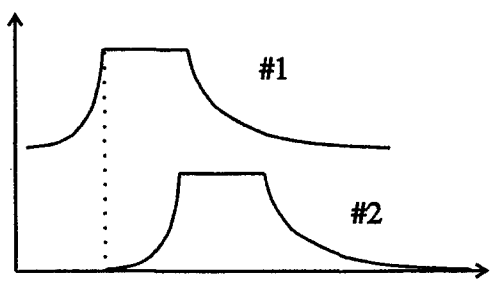


Fig. 7.42 The Supply Voltages

Note that the leading edge of each pulse is more leisurely than the trailing edge, reflecting the differing times of switching in the two stages. Let us also point out that these power supplies are universal to the entire chip or chips: otherwise, we could see that the amount of energy required to vary the supply voltage would effectively offset any savings we might make. We store outflowing energy in the power supply machinery.

Let's go back to the diode arrangement and calculate the energy lost during the switch. Let's suppose that the "rise time" we are allowing for the supply voltage to shift is t. The charge that we have to move during the change is $Q = CV$ and hence the current that will flow is (on average) just $Q/t = CV/t$. If we further suppose that the resistance we encounter in the diodes when we close them is R, a small quantity similar to that of the transistors, then the rate of energy loss, i.e. the power loss, is just $P = I^2R = Q^2R/t^2$. Hence, the total energy loss in switching is:

$$\Delta E = Pt = C^2V^2R/t = (CV^2)(CR)/t = (CV^2)\tau/t, \tag{7.23}$$

where τ is the time constant of the original, naive CMOS inverter circuit. Also, recall that CV^2 was the energy loss during switching in that circuit. Therefore, we see that the energy loss multiplied by the time in which this loss takes place is the same for both the old and the new circuits. This is suggestive of a general relationship of the form:

$$(\textit{Energy loss})(\textit{Time of loss}) = \textit{Constant} \tag{7.24}$$

for each switching step or simple logical operation. This expression certainly appears to be in sympathy with the findings of Chapter Five: the slower we go, the less energy we lose. In actual circuits, the clocks are much slower than the transistors (e.g., a factor of fifty to one), and so clocking enables us to save a great deal of energy in our computations. Unfortunately, such is the current obsession with speed that full advantage is not being taken of the opportunities that power savings might offer. Yes, the machines would be slower, and bigger because of the extra components, but this might be offset by the fact that they would be cheaper to run, and there would be considerably less need for all the pumps and the fans and so forth needed to keep the things cool!

Now although I used diodes in my example of Figure 7.41, I ought to point out that a more realistic set-up, if we don't want to use too many different types of component, is that shown in Figure 7.43 below, in which the diodes are replaced by transistors:

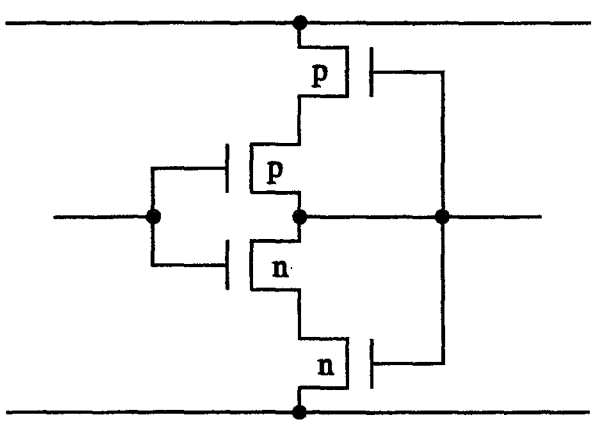


Fig. 7.43 "Diode-less" Hot-Clocking Circuit

We have looked at just one of the so-called "hot-clocking" methods for reducing energy dissipation. These techniques (developed largely at CalTech[6]) allow clock lines to deliver power but were not originally intended for trading time for energy. Let me finish this section by pointing out that hot-clocking is a fairly recent development, and there are still many unanswered questions about it — so you have a chance to actually do something here, to make a contribution! The circuit I drew was my own, different from others that have been designed and built, and I'm not sure if it has any advantages over them. But you can check out all manner of ideas. For example: what if the supply voltage was AC, i.e., sinusoidal? Could we perhaps use two power supplies, both AC and out of phase? Why not let the voltage across the logic elements be AC? Perhaps we could define two states, in phase with the power supply (logical one) and out of phase (logical zero). There are many opportunities, and perhaps if you delved further and kept at it, you might uncover something interesting.

7.2.3: Some General Considerations and an Interesting Relationship

One of the central discoveries of the previous section, which might be general, is that the energy needed to do the switching, multiplied by the time used for this switching, is a constant — at least for resistive systems. We will call this constant the "dissipated action" (a new phrase I just made up). Now the typical time constant τ of an inverter is of the order of 0.3ns, which is pretty small. Does it have to be so tiny? Well, yes, if we want to go as fast as possible. But we can approach the matter from a different angle. Because of delays on the lines, and because each element might have to feed others, and so forth, the actual clock cycles used are a hundred times greater in length — you can't have everything changing too quickly, or you'll get a jam. Now it is not obvious that we cannot slow the inverter down a bit — if we do so, it is not necessarily true that we will lose time overall in our computation in proportion to this reduction. Since this is unclear, it is interesting to find out exactly what is the value of our constant, which we shall write as $(Et)_{sw}$.

One way to do this is to work out the value of the constant for a specific switch for which it is directly calculable. We will therefore focus on the fastest possible switch and evaluate it for this — this is as good as any other choice.

[6] A 1985 paper on 'Hot-Clock nMOS' by Chuck Seitz and colleagues at CalTech ends with the following acknowledgement: "We have enjoyed and benefitted from many interesting discussions about 'hot-clocking' with our CalTech colleagues Alain J. Martin and Richard P. Feynman." [Editors]

Let's first recap our basic equations. Our switch, a single transistor, will have a certain capacitance C_g, and we put a voltage V_g on it, and hence a charge $Q = C_g V_g$. This gives us a switching energy $E_{sw} = C_g V_g^2$. Now $\tau = C_g R$, so $(Et)_{sw} = C_g^2 V_g^2 R = Q^2 R$, that is, the square of the charge needed to make the switch work multiplied by the minimum resistance we get when the switch is turned on. (You can also understand this in terms of power losses, working with currents.) Now we're naturally interested in asking what this dissipated action constant is for our ordinary transistors. We want to know this to see if, by redesigning such devices, we can get it down a bit, and perhaps use less energy or less time.

In order to proceed with the calculation, which is rather easy, we will need some physical parameters. Firstly, the electron charge $e = 1.6 \times 10^{-19}$ C. Also, at room temperature we have $kT/e = 1/40$ Volt. Using the kinetic energy relationship $(1/2)mv^2 = (3/2)kT$ (where m is the effective mass of the electron), we can define a "thermal velocity" v_{th} — this turns out to be roughly 1.2×10^7 cm/s. We also need some of the properties of weakly-doped silicon material: the electron carriers have surface channel mobility $\mu = 800$ cm^{2}V^{-1}s^{-1}, and a mean free path $l_{col} = 5 \times 10^{-6}$ cm. As with our earlier analysis of the MOSFET, we take the silicon under the gate to be L cm in length, W in width: in 1978, a typical value for L was 6 microns, falling to 3 by 1985[7].

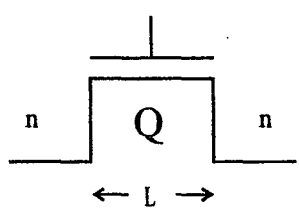


Fig. 7.44 The Simple MOSFET

Suppose we have electrons sloshing about under the gate, and we impose a force F on them, for a time τ_{col}. This latter quantity we take to be the average time between electron collisions, which is a natural choice given the physics of the situation. There is an intuitively satisfying relationship between the mean free path and the collision time: $l_{col} = v_{th}\tau_{col}$. Now, from standard mechanics, at the end of this time an electron will have gained a momentum $mv_D = F\tau_{col}$ where

[7] Standard technology in 1996 is now 0.5 micron with 0.35 micron available to major manufacturers like Intel. [Editors.]

We have looked at just one of the so-called "hot-clocking" methods for reducing energy dissipation. These techniques (developed largely at CalTech[6]) allow clock lines to deliver power but were not originally intended for trading time for energy. Let me finish this section by pointing out that hot-clocking is a fairly recent development, and there are still many unanswered questions about it — so you have a chance to actually do something here, to make a contribution! The circuit I drew was my own, different from others that have been designed and built, and I'm not sure if it has any advantages over them. But you can check out all manner of ideas. For example: what if the supply voltage was AC, i.e., sinusoidal? Could we perhaps use two power supplies, both AC and out of phase? Why not let the voltage across the logic elements be AC? Perhaps we could define two states, in phase with the power supply (logical one) and out of phase (logical zero). There are many opportunities, and perhaps if you delved further and kept at it, you might uncover something interesting.

7.2.3: Some General Considerations and an Interesting Relationship

One of the central discoveries of the previous section, which might be general, is that the energy needed to do the switching, multiplied by the time used for this switching, is a constant — at least for resistive systems. We will call this constant the "dissipated action" (a new phrase I just made up). Now the typical time constant τ of an inverter is of the order of 0.3ns, which is pretty small. Does it have to be so tiny? Well, yes, if we want to go as fast as possible. But we can approach the matter from a different angle. Because of delays on the lines, and because each element might have to feed others, and so forth, the actual clock cycles used are a hundred times greater in length — you can't have everything changing too quickly, or you'll get a jam. Now it is not obvious that we cannot slow the inverter down a bit — if we do so, it is not necessarily true that we will lose time overall in our computation in proportion to this reduction. Since this is unclear, it is interesting to find out exactly what is the value of our constant, which we shall write as $(Et)_{sw}$.

One way to do this is to work out the value of the constant for a specific switch for which it is directly calculable. We will therefore focus on the fastest possible switch and evaluate it for this — this is as good as any other choice.

[6] A 1985 paper on 'Hot-Clock nMOS' by Chuck Seitz and colleagues at CalTech ends with the following acknowledgement: "We have enjoyed and benefitted from many interesting discussions about 'hot-clocking' with our CalTech colleagues Alain J. Martin and Richard P. Feynman." [Editors]

Let's first recap our basic equations. Our switch, a single transistor, will have a certain capacitance C_g, and we put a voltage V_g on it, and hence a charge $Q = C_gV_g$. This gives us a switching energy $E_{sw} = C_gV_g^2$. Now $\tau = C_gR$, so $(Et)_{sw} = C_g^2V_g^2R = Q^2R$, that is, the square of the charge needed to make the switch work multiplied by the minimum resistance we get when the switch is turned on. (You can also understand this in terms of power losses, working with currents.) Now we're naturally interested in asking what this dissipated action constant is for our ordinary transistors. We want to know this to see if, by redesigning such devices, we can get it down a bit, and perhaps use less energy or less time.

In order to proceed with the calculation, which is rather easy, we will need some physical parameters. Firstly, the electron charge $e = 1.6 \times 10^{-19}$ C. Also, at room temperature we have $kT/e = 1/40$ Volt. Using the kinetic energy relationship $(1/2)mv^2 = (3/2)kT$ (where m is the effective mass of the electron), we can define a "thermal velocity" v_{th} — this turns out to be roughly 1.2×10^7 cm/s. We also need some of the properties of weakly-doped silicon material: the electron carriers have surface channel mobility $\mu = 800$ cm^{2}V^{-1}s^{-1}, and a mean free path $l_{col} = 5 \times 10^{-6}$ cm. As with our earlier analysis of the MOSFET, we take the silicon under the gate to be L cm in length, W in width: in 1978, a typical value for L was 6 microns, falling to 3 by 1985[7].

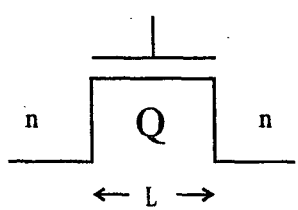


Fig. 7.44 The Simple MOSFET

Suppose we have electrons sloshing about under the gate, and we impose a force F on them, for a time τ_{col}. This latter quantity we take to be the average time between electron collisions, which is a natural choice given the physics of the situation. There is an intuitively satisfying relationship between the mean free path and the collision time: $l_{col} = v_{th}\tau_{col}$. Now, from standard mechanics, at the end of this time an electron will have gained a momentum $mv_D = F\tau_{col}$ where

[7] Standard technology in 1996 is now 0.5 micron with 0.35 micron available to major manufacturers like Intel. [Editors.]

the velocity v_D is the "drift velocity" in the direction of the force, and is quite independent of (and much smaller than) v_{th}. Since mobility is defined by the relation, $v_D = \mu F$, we have $\mu = \tau_{col}/m$. Now, take the current flowing under the gate to be I. We have $I = Q/$(time of passage across the gate) $= Q/(L/v_D) = (Q/L).(\mu e).(V_{ds}/L)$. However, the source-drain voltage $V_{ds} = IR$, so we have, for the resistance, $R = L^2/(Q\mu e) = mL^2/(Qe\tau_{col})$. (Incidentally, the effective mass of an electron moving through Si is within 10% of its free mass, so we can take m to be the latter.) Now, using our expression for the dissipated action in terms of Q and R, and using the relationships we have derived, we find:

$$(Et)_{sw} = N(L/l_{col})^2(3kT)\tau_{col} \tag{7.25}$$

where we have introduced the number N of (free) electrons under the gate, $N = Q/e$. Now focus on the last two factors on the right hand side of Equation 7.25 — $3kT$ is an energy, of the order of the kinetic energy of an electron, and τ_{col} is a time, the time between collisions. Maybe it will help us to understand what is going on here if we define the product of these terms to *itself* be a dissipated action — just that dissipated during a single collision. This isn't forced on us: we'll just see what happens. Let us call such an action $(Et)_{col}$. Then we have:

$$(Et)_{sw}=N(L/l_{col})^2(Et)_{col}. \tag{7.26}$$

So we find that the (Et) that we need for the whole switch is larger than the Et for a single collision by two factors. One is the number of electrons under the gate, and the other is the ratio of the width of the gate to the mean free path. Taking L to be 6 microns (hence L/l_{col} to be about 100), and the number of electrons N under the gate to be about 10^6, we find:

$$(Et)_{sw} = 10^{10}(Et)_{col} \approx 10^8 kT\tau_{col}. \tag{7.27}$$

This ties in with what we have quoted before. Now this is an *awful* amount, and we would certainly hope that we can improve things somehow! Why is this number so large? We know from the considerations of the Chapter Five that it in no way reflects a fundamental energetic limit. What can we do to get it down a bit?

Of course, all of our calculations thus far have been rooted in the conventional silicon VLSI approach — so perhaps what we ought to do is step

outside that technology and look at another. Let us take a more general, and somewhat abstract, look at this question. Suppose that you design for someone a beautiful switch, the fundamental part of a computational device, which has a certain switching energy E_{part} and corresponding switching time t_{part}. Now you give this guy a pile of these parts, and he proceeds to build a circuit with them. But he does this in a most absurdly inefficient manner. He does this as follows (this might all seem a bit abstract at first, but bear with me). Firstly, he connects up, say, p switches in parallel, and hooks them all up to the same input:

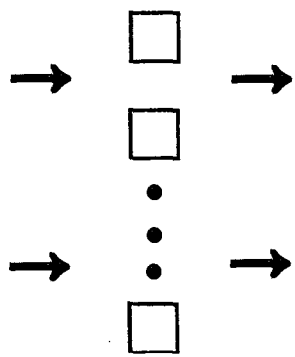


Fig. 7.45 A Possible Parallel Connection of Fundamental Parts

These switches all operate simultaneously, the signal propagating from left to right. Clearly, the energy dissipated in switching all of these parts is $E_{sys} = pE_{part}$, and the time for it to occur is just $t_{sys} = t_{part}$. In other words, $(Et)_{sys} = p(Et)_{part}$. This is ridiculous, given that they all give out the same answer.

Next, the guy does something even dumber and connects up some parts in series as well, in chains s parts long:

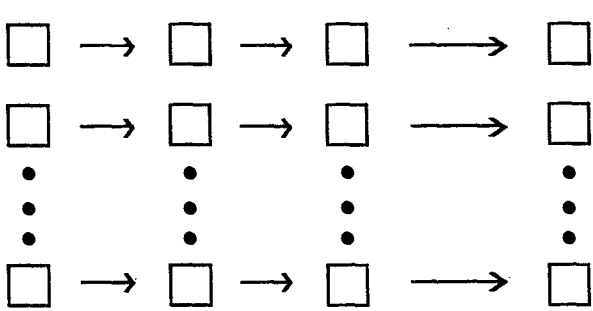


Fig. 7.46 A Serial Connection of Parts

This really is dumb! Each switch in the chain just inverts the previous one, so all he has overall is a simple switch, effectively no better than the parallel arrangement he started out with! Yet compared to that, now $(Et)_{sys} = ps^2(Et)_{part}$, as you should be able to see. So of what relevance is all this? Well, an electron colliding is rather like a *1-electron switch*, with which we can associate a quantity $(Et)_{part} = 3kT\tau_{col}$. We can consider such a collision to be the fundamental operation. Now all the electrons jiggling beneath the gate are doing the same thing, bumping into one another, drifting and so forth, and so we can consider them to be operating effectively in *parallel*; with the number of parallel parts $p = N$, the number of electrons beneath the gate. Of course, one collision is not sufficient to account for the whole of an electron's activity between poles — the actual number of hits, on average, is $(L/l_{col}) \sim s$, using the serial analogy here. So we can actually interpret our result for $(Et)_{sw}$ in terms of the crazy handiwork of our engineer: $10^{10} = ps^2$! Surely room for improvement?

Okay — so how *can* we improve on this? Firstly, is it completely silly to put things in parallel? Not at all: it's good for accuracy. It might be the case that we are working with parts that are extremely sensitive and which can easily be flipped the wrong way by thermal fluctuations and what-not. Putting such parts in parallel and deciding the output on the basis of averaging, or by a majority vote, improves system reliability. If we have a part whose probability of malfunctioning is 1/4, then with just 400 of these in parallel, we can guarantee that the chance of the system spitting out a wrong answer is about 1 in 10^{18} — wow! And what about putting parts in *series*? Well, I've thought a lot about this, but have yet to come up with *any* resulting advantage. It wouldn't help with reliability — all it does is increase the lag. In fact, I can see no reason for having anything other than $s = 1$.

Problem 7.3: In our electron model, $s = 1$ would correspond to getting the fundamental ratio (L/l_{col}) down to unity. An interesting question arises if we actually take this notion seriously. In fact, I would like you to consider the most extreme case, that where the mean free path of the electrons below the gate is *infinite*: in other words, they suffer no collisions. Analyze the characteristics and behavior of such a device. Sure, on first impression, such a device could never function as a switch — it would always conduct. But we have forgotten about *inertia*: in order to conduct, the electrons have to speed up and change their speed, and can only start at zero; so there is a certain density of charge beneath the gate anyhow. In fact, this whole analysis, with zero mean path, was originally made for vacuum tubes, and these certainly worked. So a switch of this kind can be devised, and analyzed — it's just that we can't do it with silicon

(in which the electrons can be thought of as moving through some sort of "honey").

Generally, however we do it, we should make every effort to increase the mean free path and to decrease L. There is a factor of 100 to be found in $(Et)_{sw}$ (not 10^4, because if we change the mean free path we change τ_{col} as well). Current hardware design stinks! The energy loss is huge and there is no physical reason why we shouldn't be able to get that down at the same time as speeding things up. So go for it – you're only up against your imagination not Nature.

An obvious suggestion is to simply reduce the size of our machines. We can make good gains this way. Let us scale L by a factor of $\alpha < 1$: $L \rightarrow \alpha L$. We then find $(L/l_{col}) \rightarrow \alpha^2(L/l_{col})$ since τ_{col} scales as $1/\alpha$. The number of electrons under the gate scales with area: $N \rightarrow \alpha^2 N$. Hence, we arrive at the result:

$$(Et)_{sw} \rightarrow \alpha^4 (Et)_{sw} \tag{7.28}$$

This is excellent scaling behavior, and though we cannot trust it down to too small values of α, it shows that simply shrinking our components will be advantageous.

The (Et) ideas I've put forward here are my own way of looking at these things and might be wrong. The idea that (Et) might be a constant is very reminiscent of the Uncertainty Principle in quantum mechanics, and I would love to have a fundamental explanation for it, if it turns out to be so. There is certainly room for you to look into such questions to see if you can notice something. Anything you can do to criticize or discuss these ideas could be valuable. If nothing else, because the simple relationship:

$$Power = E/t = (Et)/t^2, \tag{7.29}$$

shows that reducing the dissipative action (Et) should reduce the power loss from faster machines.

7.3: VLSI Circuit Construction

We now come, at last, to discuss the actual physical technology underlying VLSI. How are transistors actually made? How do we, being so big, get all this stuff onto such tiny chips? The answer is: very, very cleverly – although the

basic idea is conceptually quite simple. The whole VLSI approach is a triumph of engineering and industrial manufacture, and it's a pity that ordinary people in the street don't appreciate how marvelous and beautiful it all is! The accuracy and skill needed to make chips is quite fantastic. People talk about being able to write on the head of a pin as if it is still in the future, but they have no idea of what is possible today! We can now easily get a whole book, such as an encyclopedia or the Bible, onto a pinhead – rather than angels! In this section we will examine, at a fairly simplistic level of analysis, the basic processes used to make VLSI components. We shall once again focus solely on nMOS technology.

7.3.1: Planar Process Fabrication Technology

The process all begins with a very pure crystal of silicon. This material was known and studied for many years before an application in electronics was found, and at first, it tended to be both rare and, when unearthed, riddled with impurities – nowadays, in the laboratory, we are able to make it extremely pure. We start with a block of the stuff, about four inches square[1], and deep, and we slice this into thin wafers. Building integrated circuits on this substrate involves a successive layering of a wafer, laying down the oxide, polysilicon and metals that we need according to our design. Remember from our earlier discussion that the source and drain of a MOSFET were n-type regions seeded into, rather than grafted onto, lightly doped p-type Si material – it is important to keep in mind that the silicon wafer we are using is actually this p-type stuff. To see the sort of thing that goes on, we'll explain in some detail the first step, which is to create and manipulate the non-conducting oxide layer on the silicon that will ultimately play a role in constructing the insulation layer under the gate of a transistor. We start by passing oxygen over the surface of the wafer, at high temperature, which results in the growth of a layer of silicon oxide (SiO_2). This oxide layer is shown in Figure 7.47. We now want to get rid of this oxide in a selective fashion. We do this very cunningly. On top of the oxide we spread a layer of "resist", an organic material which we bake to make sure it stays put. A property of this resist is that it breaks down under ultraviolet light, and we use this property to etch an actual outline of our circuitry on the wafer. We take a template – a "mask" – and lay this over the material. The mask comprises a transparent material overlaid with an ultraviolet opaque substance, occupying regions beneath which, on the chips, we will want SiO_2 to remain. (Usually, the mask will repeat this pattern over its area many times, enabling us to produce

[1] In 1996, cylinders of silicon 12 inches in diameter are common. [Editors]

many chips on one wafer, which may be cut into separate chips later.) We next bombard the wafer with UV light (or X-rays). The affected resist, that not shielded by the opaque regions of the mask, breaks down, and can be sluiced off. This exposes channels of SiO_2 which we can now remove by application of a strong acid, such as hydrofluoric acid. The beauty of the resist is that it is not removed by the acid so that it protects the layer of SiO_2 beneath it — unlike the stuff we've just sluiced off — we want to keep in place. After this stage, we have an upper grid of resist, under which lies SiO_2, and beneath this a bared grid of the original silicon. We now apply an organic solvent to the wafer which removes the resist and leaves the underlying oxide intact. The result is, if you like, a layer of oxide with "silicon holes" in it (Fig. 7.47):

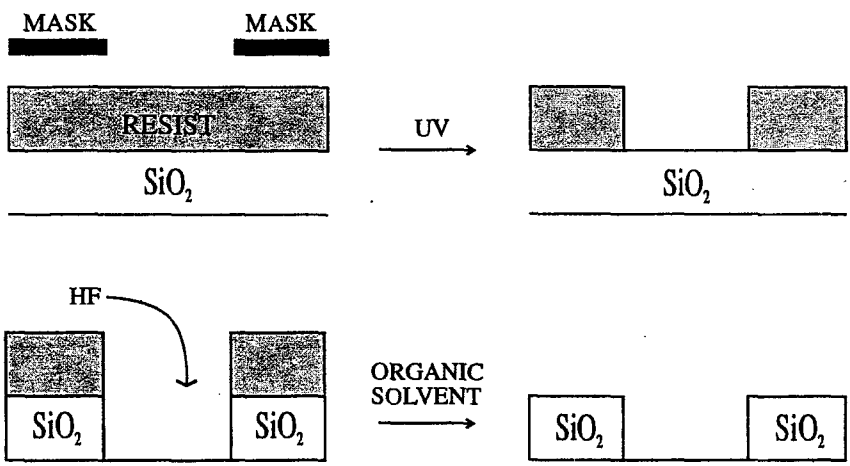


Fig. 7.47 The First Stages of Chip Fabrication

That is the first of several steps. Step two involves laying down the basic material for any *depletion mode* transistors that may be required in the circuit (for use as resistors, for example). Such transistors differ in their construction from enhancement mode devices by having a shallow layer of n-type Si strung beneath the gate between the source and drain:

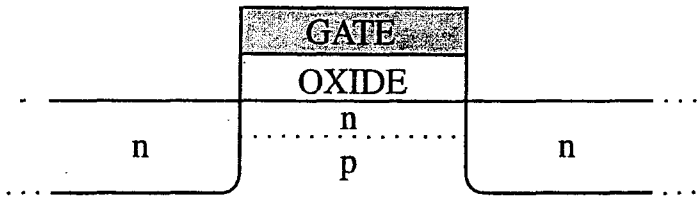


Fig. 7.48 The Depletion Mode Transistor

Such a transistor is perpetually *closed* and current can always flow unless we place a negative charge on the gate to stem this current flow and open the switch (hence, $V_{th} < 0$, as stated earlier). To put such transistors on the chip, it is necessary to lay down their foundations before we go any further: this entails first delineating their gate regions and then creating a very thin region of n-type doped Si over these areas. To do this, we cover the wafer with resist again, and place on it a mask whose transparent regions represent the depletion areas. Once again, we blast the wafer with UV or X-radiation, and this time we are left with a wafer comprising a covering of resist, dotted among which are spots of exposed silicon substrate. These open areas we dope with phosphorus, arsenic or antimony, to create the required depletion region. The resist prevents these ions from penetrating into the rest of the silicon. This done, we wash off the remaining resist.

The next layer to be taken care of is the *polysilicon* (polycrystalline silicon) layer. Recall that highly-doped polysilicon conducts well, although not as well as a metal, and will be used to construct, among other things, the gates of transistors. As these gates are separated from the substrate by a thin layer of insulating oxide (see Figure 7.8), it should come as no surprise to you that before we do anything with our polysilicon, we have to coat the wafer with another thin layer of oxide as we did initially. As before, we do this by heating the wafer in oxygen (note that this will leave the depth of oxide across the wafer uneven). The wafer is then coated in polysilicon and another mask overlaid — this time designed to enable us to remove unwanted polysilicon. Having done this, we have to build the drains and sources (and, generally, the diffusion layer) of our transistors — and we do this by doping all of the remaining silicon appropriately (i.e. with phosphorus). We achieve this by removing any oxide that is not lying under the polysilicon and mass-doping the exposed Si regions. The depletion layers beneath the polysilicon are protected will not be additionally contaminated.

We can now see how an enhancement mode transistor will arise from this process. To draw a diagram, we will adopt the conventions for the various layers shown in Figure 7.49 (the conventions in most common usage are actually color-coded):

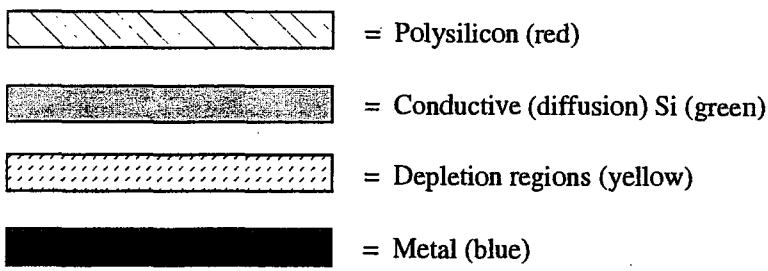


Fig. 7.49 Conventions for Chip Paths

We have added one more layer here — that of metal[2]. This layer comprises the "flat wires" we use to carry current a sizeable distance, in preference to polysilicon or the diffusion layer. (The power supply is usually drawn from metal paths.) It will also be necessary to add contact points to enable the current to flow freely between layers, as required. With this convention, we can draw an enhancement mode transistor as:

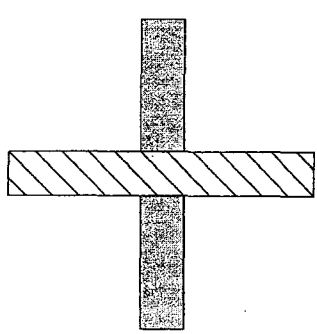


Fig. 7.50 Schematic Diagram for Enhancement Mode Transistor

[2] Three metal layers are now typical in 1996, with five available for specialists. Typically one of these 'metal' layers would be polysilicon. [Editors]

The transistor is just the crossing point of a polysilicon path and a diffusion path! Of course, the two paths do not cross in the sense of making physical contact — there is a layer of insulating oxide between them.

A full inverter requires a resistance in series. As we discussed earlier, we use a depletion mode transistor for this task. The inverter circuit is shown in Figure 7.51 below:

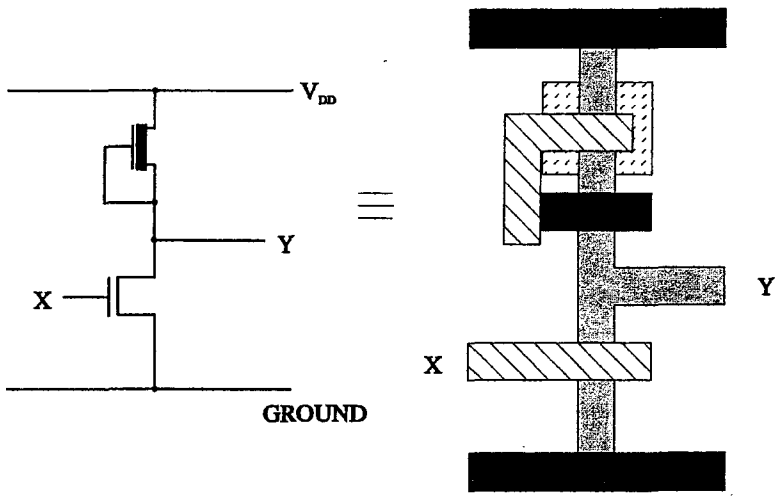


Fig. 7.51 The Full Inverter

You will note that I have included here the power supply and ground lines, both of which are metal paths. It is necessary in the fabrication process to leave patches of the diffusion paths exposed at the point where the metal crosses, so as to ensure an electrical contact. These features you cannot see from a vertical picture. (The actual circuit is not laid out wholly flat as in Figure 7.51; it's all built on top of itself, in a clever, tight little box. See Mead and Conway for more details.) A similar procedure is necessary if we want to, say, use the source or drain of a transistor as the input to another gate — we then have to connect a diffusion path to a polysilicon path. Obviously, some kind of direct contact is needed; otherwise, we would find a capacitor or transistor where the lines cross. We can use a so-called "butting contact" where we overlay a direct diffusion/poly contact with metal, as shown in Figure 7.52:

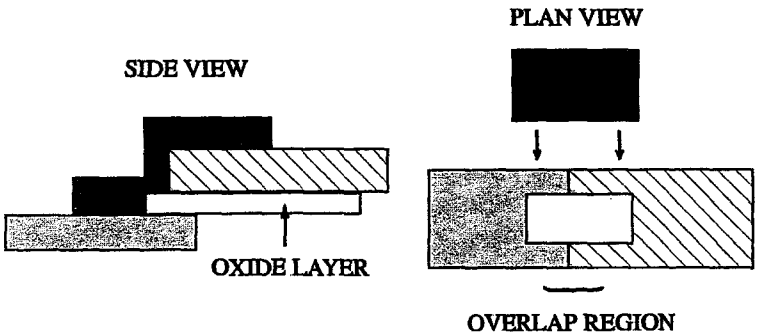


Fig. 7.52 Polysilicon-Diffusion Layer Contact

To give an illustration of a more involved logic unit, we will look at the NAND gate. To make this, all we need to do is take our previous circuit, and cross the diffusion path with another polysilicon path to make another transistor (Fig. 7.53):

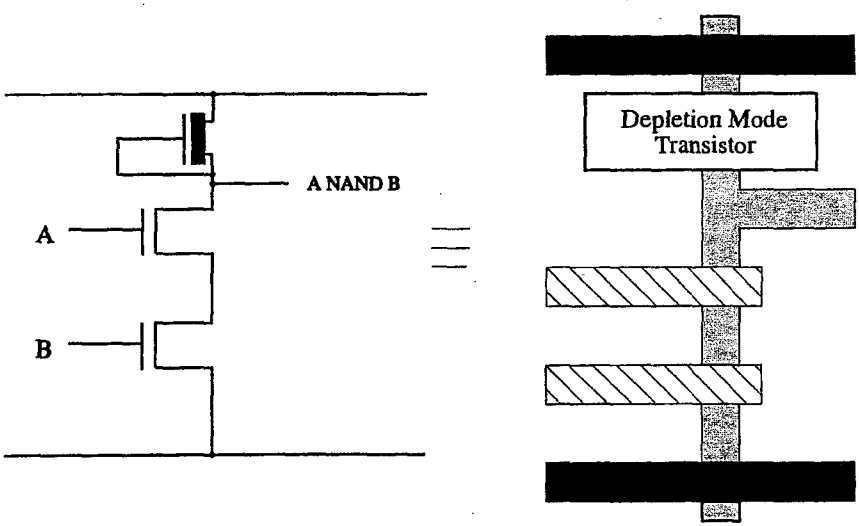


Fig. 7.53 The NAND Gate

Note that in this circuit the polysilicon paths extend a litle way beyond the diffusion path at the each of the two transistors. Why? Well, there are many design rules governing precisely how we should arrange the various paths on a

chip with regard to each other, how big the paths must be, and so on, and I'll briefly list some here. (For a fuller exposition of these 'lambda-based' design rules, see Mead & Conway). Let us begin by defining a certain unit of length, λ, and express all lengths on the chip in terms of this variable. In 1978, λ was about 3 microns; by 1985, it had fallen to 1 micron, and it falls further as time progresses. The minimum width for the diffusion and polysilicon paths is 2λ. The metal wire, however, must be at least 3λ across, to counter the possibility of what is known as "electromigration", a phenomenon whereby atoms of the metal tend to drift in the direction of the current. This can be a seriously destructive effect if the wire is especially thin (Fig. 7.54):

Fig. 7.54 Silicon Chip Path Widths

Again, these are minima: the paths can be wider if we desire. Another set of rules pertains to how closely we can string wires together. Conducting paths cannot be placed too near each other because of the danger of voltage breakdown, which would allow current to criss-cross the circuit (Fig. 7.55):

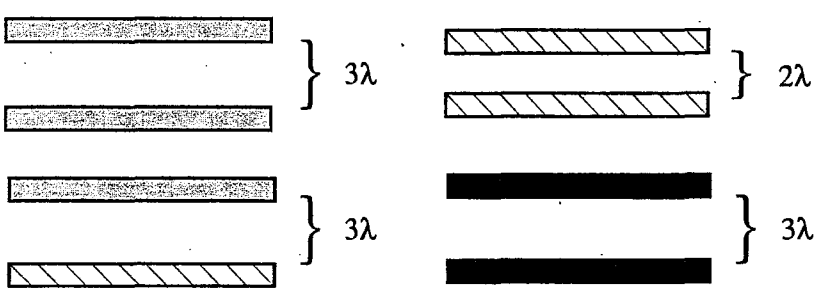


Fig. 7.55 Silicon Chip Path Separations

Metal paths (blue) can go on top of poly (red) and diffusion ones (green)

without making contact. Where red crosses green, as we've said, there is a transistor. It is important with such devices that the poly line forming the gate extends over the edge of the diffusion region, to prevent a conducting path forming around it resulting from shorting the drain to the source. We usually require an overlap of at least 2λ, to allow for manufacturing errors (Fig. 7.56):

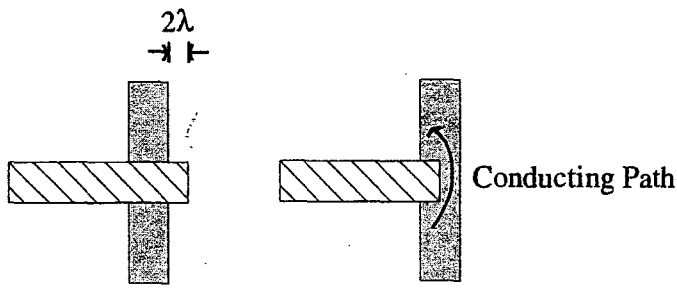


Fig. 7.56 Rules for a Transistor

We must also consider the connections between levels. If we are hooking a metal line to another path, we must be sure the contact is good (the contact is typically made square). To ensure this, we do not just place the metal in contact with the path, area for area, but must have at least a distance λ of the path substance surrounding the contact to prevent leakage through the metal and into the surroundings. This is true whether we are connecting to poly, diffusion or metal lines (Fig. 7.57):

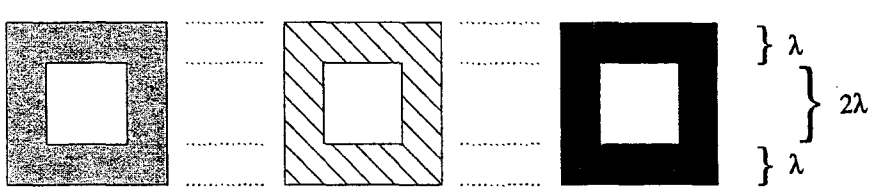


Fig. 7.57 Rules for Contacts

7.3.2: Circuit Design and Pass Transistors

To actually make a specific circuit, we would design all of the necessary masks (typically enormously complex) and send them to a manufacturer. This

manufacturer would then implement them in the construction process we have described to provide us with our product. There is a standard heuristic technique for drawing out circuits, one which tells us the topology of the layout, but not its geometry — that is, it tells us what which paths are made of, and what is connected to where; but it does not inform us as to scale, i.e. the relevant lengths of paths and so on. For example, the drawing (the so-called "stick figure") for the NAND gate is shown in Figure 7.58 (in which we have also indicated the new linear conventions we adopt for each type of path):

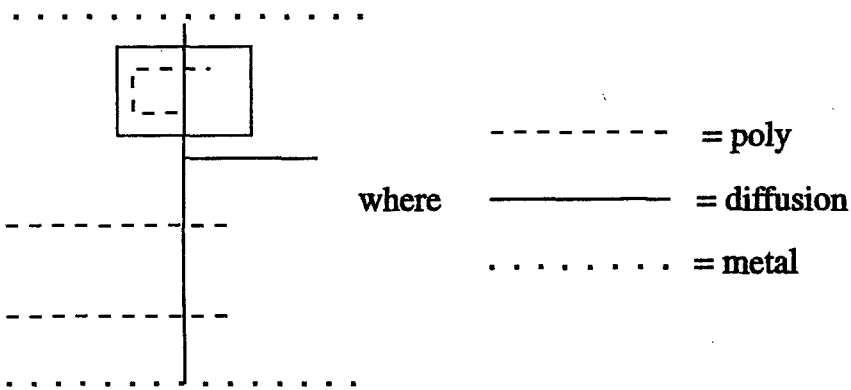


Fig. 7.58 "Stick Figure" for the NAND Gate

This tells us all the important interconnections in the circuit but if we were to actually trace the final physical product, the actual scaling of the respective parts might be radically different. This latter need not concern us here and we will adopt the stick figure approach in what follows, when we want to take a look at some specific circuits. To make things simpler still, we can sometimes deploy a kind of "half and half" shorthand, in which we represent sub-circuits on the chip by black boxes (a common enough procedure). So, for example, if we had a simple chain of inverters, it would be easier, rather than drawing the entire transistor stick figures over and over, to use the scheme of Figure 7.59:

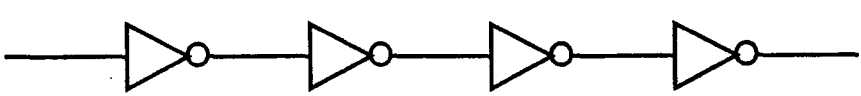


Fig. 7.59 Simplified Circuit Diagram for Chain of Inverters

where the triangles are just the conventional symbols for inverters, and the line convention is as explained in Figure 7.58.

A common type of circuit is the shift register. We represent this in Figure 7.60 as a doubly-clocked inverter chain, crossed by polysilicon paths:

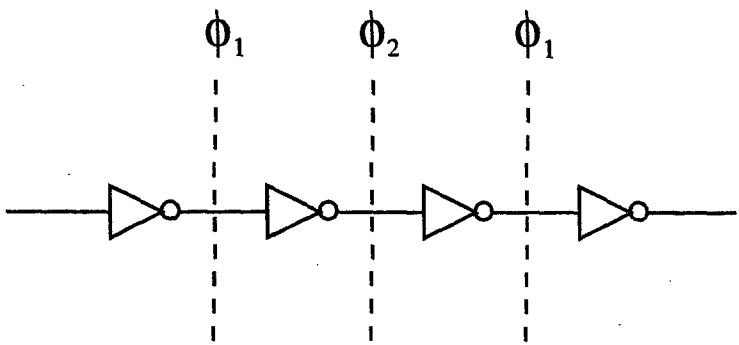


Fig. 7.60 A Shift Register

The two (complementary) clock pulses are sent down polysilicon lines, and where these cross a diffusion line, they form what is known as a *pass transistor*, so-called because it only allows a current to flow from source to drain (i.e. from left to right in the above picture) if the gate is foward-biased. This occurs whenever the clock pulse to the polysilicon line is on. At the next pulse the next inverter in the chain switches and will hold its new value until the next clock pulse. The reader should be able to make contact with our discussion of clocked registers in Chapter Two to figure out how Figure 7.60 works. It is a simpler, more accessible arrangement than a bunch of flip-flops and logic gates. Note, incidentally, that we can close such an arrangement (i.e., make it go "in a circle") if we want to use it as a memory store.

7.3.3: Programmable Logic Arrays

With Programmable Logic Arrays (PLAs), we come on to examine the issue of *if-then control* in machines — that is, the matter of how, given a certain set of input data, the machine should determine what it does next. For example, "if such-and such is zero, then stop" or "if both bits are 1, then carry 1". Abstractly, there is information coming out of some part of the machine which will tell us what we're to do next. This information hits some "sensors" (my own word, not the technical one), which tell us our present state. Once we know this, we can

act on it, for example, by telling an adder to add or subtract. This instruction, or more generally, set of instructions, will take the form of data coming out on a set of lines (Fig. 7.61):

Fig. 7.61 A Generic Control Device

The first stage in designing a device to do this is, obviously, to know what set of instructions are associated with a given sensory set. This is pretty straightforward. For example, we might represent the instructions as in Table 7.1:

SENSE LINES	INSTRUCTIONS
1 2 3 4 5	a b c d e f
1 1 0 1 0	1 0 1 1 0 1
1 0 0 1 1	1 0 1 1 0 1
0 1 1 0 1	0 1 1 0 1 0
⋮	⋮

Table 7.1: Example Instruction Set for a Control Device

What this means is as follows. Each row in the left hand column represents some configuration of bits on the sensor lines (of which there are five in this example). The corresponding rows on the right represent the bits sent out along the instruction lines (six, in this case), given the sensor set on the left. In this column, a 1 might mean "do something if the input from this line is 1" — such as "add" or "switch on light" — while a 0 might mean do nothing, or do something else — "leave state X as it is" or "switch off light". A very direct, and very inefficient, way of making a control system would be to simply store this

table in memory, with the sensing lines as memory addresses, and the control lines as the contents of these addresses. Thus we would separately store the actions to be performed for all possible combinations of sense lines. Since the contents of this memory are to be fixed, we might as well store everything on a Read Only Memory (ROM) device. The only potential hitch in this otherwise straightforward procedure arises from timing: it is conceivable that some instructions could leave the ROM device before the rest, changing the state of the machine and confusing the sensing. The effect of this might be fed back into the ROM before it has completely dealt with its previous sense set. This would be pretty bad if it happened but is usually avoided (you should be way ahead of me here) by deploying clocked registers at each end of the memory to ensure that the retrieval and use of an instruction occur at different times (Fig. 7.62):

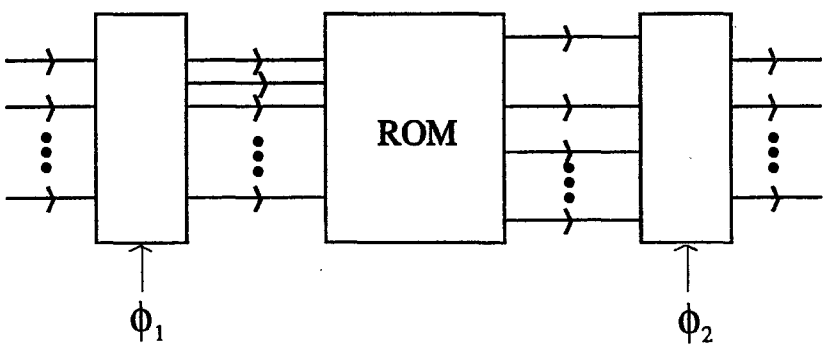


Fig. 7.62 Clocked ROM Control System

When ϕ_1 is on, the sense lines feed through to the memory, which looks up the corresponding control signals. These latter signals cannot get out because ϕ_2 is off. Only when we can be sure that everything has settled down — that all the sensing information is in and that the instruction set has been chosen — do we switch ϕ_2 on. ϕ_1 has meanwhile gone off to freeze the memory input. With the external clock on, the instruction set can now get out and reach the rest of the machine without affecting the memory input. And so it goes on.

Thus, we see that control can be very, very simple. However, we are dissatisfied with this kind of approach because we would also like to be efficient! As a rule, stuffing our memory with 2^n entries is somewhat extravagant. Often, for example, two or more given input states will result in the same output state, or we might always filter a few sense lines through a

multiple-OR gate before letting them into the ROM. This would leave us with a high degree of redundant information in our table, and naturally enough we find ourselves tempted to eliminate the ROM completely and go back to basics, developing a circuit involving masses of logic gates. This was how things were done in the early days, carefully building immensely complicated logic circuits, deploying theorems to find the minimum number of gates needed, and so forth, without a ROM in sight. However, these days the circuits are so complex that it is frequently necessary — given the limitations of human brain power! — to use a ROM approach. But there are intermediate cases for which a ROM is not necessary because the number of possible outputs is small enough to enable a much more compact implementation using just a logic circuit — the set up is not *too* complicated for us to design. To illustrate one such instance, we shall examine a so-called "Programmable Logic Array" (PLA), something we first encountered in Chapter Two. This is an ordered arrangement of logic gates into which we feed the sense input, and which then spits out the required instruction set. Ideally, such an array would exhibit no redundancy. In a "black box" scheme, a generic PLA would have the form shown in Figure 7.63:

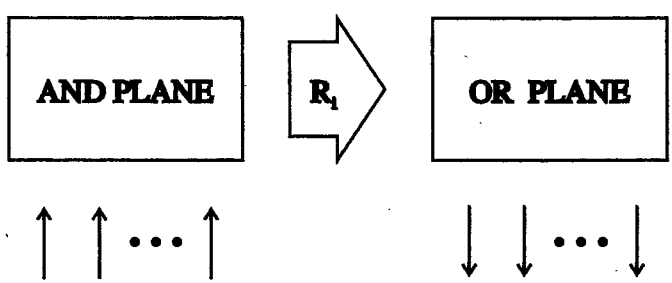


Fig. 7.63 The Generic PLA

As can be seen, the PLA comprises two main sections: the "AND-plane" — formed exclusively from AND gates — and the "OR-plane" — formed exclusively from ORs. The planes are connected by a bridge of wires, which we label *R*. The inputs are fed into the AND-plane, processed and fed into the OR-plane by the *R*-wires. A further level of processing then takes place and a signal emerges as output from the OR-plane. This output is the set of "what next" instructions corresponding to the particular input.

Let us consider a case where we have three input lines, *A*, *B* and *C*, and four output lines, $Z_1 ... Z_4$. Each input, before being fed into the AND-plane, is

split into two pieces — itself and its complement, for example, A and NOT A. We now have a device that can manipulate each signal with NOT, AND and OR — in other words, it can represent any logical function whatsoever. Let us pick a specific PLA to show the actual transistor structure of such an array. We have three inputs telling us the state of certain parts of the machine and four possible outputs — pulses that will shoot off and tell the machine what to do next. Now suppose that the output Z's are to be given in terms of the inputs according to the following Boolean functions ($\vee$ = OR; $\wedge$= AND; $'$ = NOT):

$$\begin{aligned} Z_1 &= A \\ Z_2 &= A \vee (A' \wedge B' \wedge C) \\ Z_3 &= B' \wedge C' \\ Z_4 &= (A' \wedge B' \wedge C) \vee (A' \wedge B \wedge C') \end{aligned} \tag{7.30}$$

It is not immediately obvious that the particular Boolean functions of A, B and C that we need to calculate the Z's can be written as the product of a series of ANDs followed by ORs. However, it is in fact the case, as we demonstrated for the general logical function in Chapter 2. In this instance, an acceptable output R_i from the AND-plane must only involve the ANDs and NOTs of A, B, and C. Thus we can define the R_i as:

$$R_1 = A,\ R_2 = B' \wedge C',\ R_3 = A' \wedge B' \wedge C,\ R_4 = A' \wedge B \wedge C', \tag{7.31}$$

and it is now straightforward to see that the Z-outputs can be written purely in terms of OR operations (or identities) on these R's:

$$\begin{aligned} Z_1 &= R_1 \\ Z_2 &= R_1 \vee R_3 \\ Z_3 &= R_2 \\ Z_4 &= R_3 \vee R_4 \end{aligned} \tag{7.32}$$

It is a general result that any Boolean function can be factorized in this way. The PLA for this function is shown in Figure 7.64:

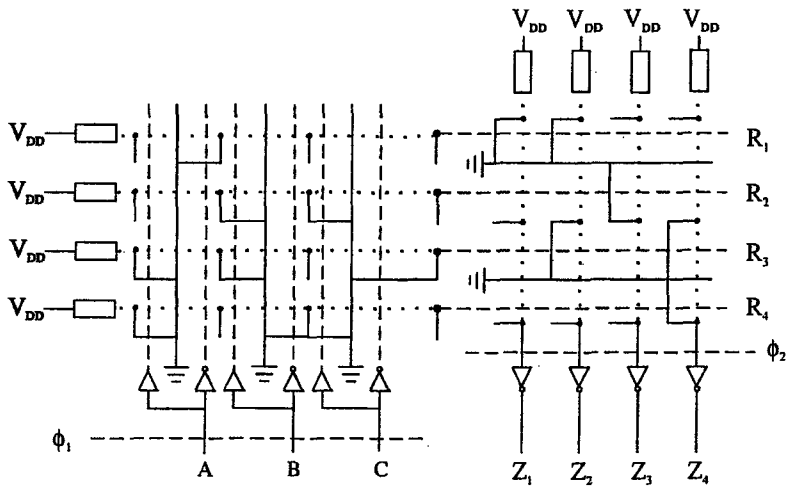
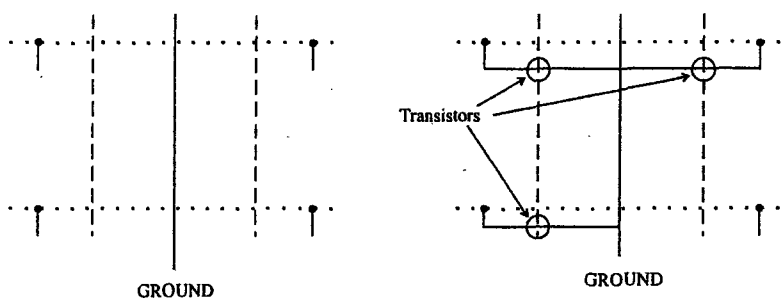


Fig. 7.64 Circuit Diagram for a PLA

I will leave it to you to work out at the electronic level how this circuit gives us the advertised transformation!

As a rule, some 90% of the structure of a PLA is independent of its actual function. In consequence, PLAs are usually constructed by overlaying a standard design with select additions. For example, the above circuit results from taking the generic AND-plane and changing it into the circuit we want by the judicious addition of several diffusion paths in the right places (Fig. 7.65):



This is a very practical approach. Of course, if you wanted more lines you would have to look up in a manufacturer's catalog which core arrays were available. Incidentally, note, from Figure 7.64 that the generic OR-plane is essentially the AND-plane rotated through a right angle.

Problem 7.4: Let me now give you an interesting problem to solve. This actually arose during the design of a real device. The problem is this: we would like to switch, that is, exchange, a pair of lines A and B by means of a control line, C. We are given C and its complement C' — they come shooting in from somewhere and we don't care exactly where — and if the control C is hot, then A and B change places: if C is cold, they don't. This is a variant of our old friend the Controlled Exchange. The circuit diagram we will use is that of Figure 7.66:

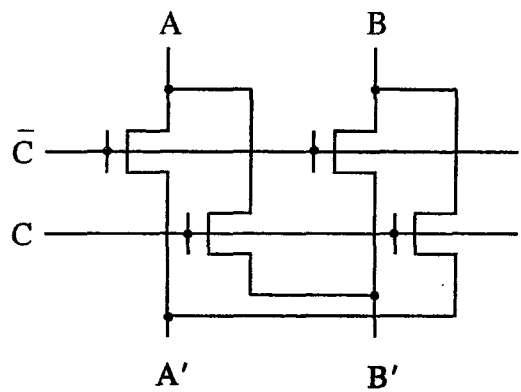


Fig. 7.66 An Exchange Circuit

To reiterate the rules: $C = 0 \Rightarrow A' = A$, $B' = B$; $C = 1 \Rightarrow A' = B$, $B' = A$. You should be able to see how it all works. Here is what I want you to do:

(a) Draw a stick figure with the correct conventions for diffusion, poly and metal (Hint: the inputs A and B are fed in on metal lines),

(b) Draw a legitimate layout on graph paper, obeying the λ design rules.

(c) This circuit can easily be amended to allow for more A, B... inputs simply by iterating its structure (and extending the C, NOT C lines). Suppose now that we have eight input pairs coming in from the top. There are only fourteen λ's available horizontally for each pair, and sixteen or twenty extra λ's on the

want the A's and B's to go out of the circuit in metal too. Can you design it? You may assume more C's from the left if you want.

7.4: Further Limitations on Machine Design

It doesn't take much thought to realize that one of the most important components of any computer is *wire*. We're so used to treating wires — more generally, transmission paths, including polysilicon lines — in an idealized way that we forget they are real physical objects, with real physical properties that can affect the way our machine needs to be designed. In this final section, I'd like to look at two ways in which wires play an important role in machine design. The first relates to how wire lengths can screw up our clocking, the so-called "clock skew" problem; the second to an even simpler issue, the fact that wires take up space, and that when we build a computer, we'd better make sure we leave enough room to get the stuff in!

7.4.1: Clock Skew

Let's return to our discussion of clocking the general PLA. Remember, we employed two clock pulses, ϕ_1 and ϕ_2, taking the general form:

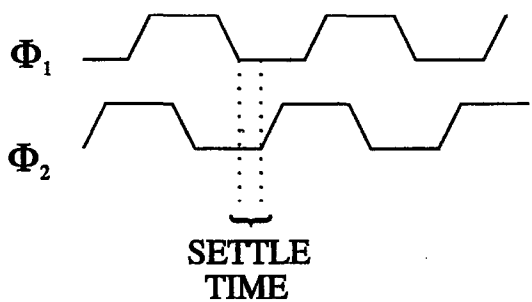


Fig.7.67 The PLA Clock Pulses

The idea is that we feed data into the PLA while ϕ_1 is on, and then let things settle down for a while — let the logic gates go to work and ready their outputs, and so on. This is the reason for introducing a delay time, and not simply having the two clocks complementary. Then, we switch on ϕ_2, and during this time we allow the data to spew out. This sounds all very straightforward and simple.

However, in a real machine, there can be problems. For a start, charging up the gates of circuit elements takes a non-zero time, and this will introduce delays and time-lags. Also, of course, the clock signals are current pulses sent along *wires* — metal, polysilicon, whatever — and *these pulses will take a finite time to travel.* A clock pulse sent along a short wire will reach the end before a pulse sent along a long wire. We can actually model a simple wire in an interesting way as an infinite sequence of components as shown in Figure 7.68 (which in the finite case could be taken as modeling a chain of pass transistors):

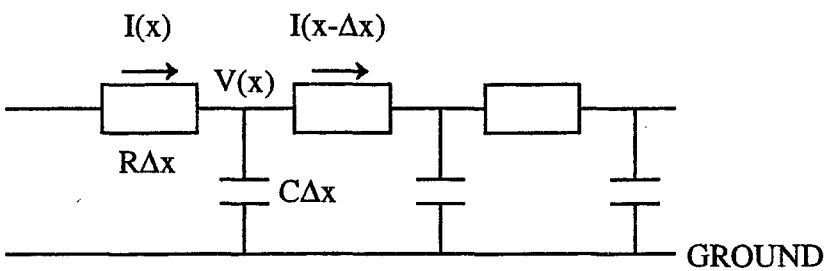


Fig. 7.68 An Infinite-Limit Model of a Simple Wire

We have a line of resistors interspersed with capacitors. If we assume we have infinitely many small capacitors and resistors, bunched up infinitely closely, then we effectively have a wire, with a resistance per unit length of R, and a capacitance per unit length of C. Now what we want to do is to load up one end of the line (which needn't be metal — it could be polysilicon), and wait for the signal to propagate along to the other end. Let the distance along the wire from the origin be x. At each junction we can define a potential $V(x)$, and a current flowing into it, $I(x)$. Taking the limit as $\Delta x \rightarrow 0$, elementary math and electricity gives us the set of equations:

$$\partial I/\partial x = C\partial V/\partial t \tag{7.33}$$

$$\partial V/\partial x = IR \tag{7.34}$$

$$\partial^2 V/\partial x^2 = RC\partial V/\partial t = \tau\partial V/\partial t \tag{7.35}$$

flows in at one end and diffuses through the system. The general form of the solution in terms of Green's functions is well-known. With our boundary conditions the solution is:

$$V(x,t) \propto \exp(-x^2/4t\tau) \tag{7.36}$$

It is easy to see from this that if the overall length of the wire is X, then the time to load the wire scales as X^2. For 1mm of polysilicon, this time comes to 100ns. For 2mm, it is 400ns. This is a pretty lousy line, especially if you're more used to transmission lines for which the loading time is proportional to the distance. Metal, however, has such a low resistance that the load time is relatively much shorter — so if you want to send a signal any great distance, you should put it on metal.

The issue of clocking is of such importance to computing (indeed, much more important than you'd think given how little I've talked about it) that we are naturally encouraged to explore other avenues, other ways of controlling our information flow. The problem with standard way, so-called synchronous clocking — the only type we've considered so far — is that in designing our machine we have, at each part of the system, to allow for the "worst case scenario". For example, suppose we have to take an output from a complex adder that could take anywhere from, say t units of time up to $5t$ to show. Now even if the output zips through after t units, we still have to put our machine on hold for at least $5t$ just on the off-chance that we get a slow decision. This can lead to severe time inefficiencies. Now, another way to design machines — although one which is not yet used commercially — is an "asynchronous" method: we *let the adder control the timing*. Let it tell us when it's ready! It carries out its computation, and then sends a signal saying it's ready to send the data. In this way, the timing is controlled by the computing elements themselves, and not a set of external clocks.

Interestingly enough, a little thought will show you that even synchronous systems have asynchronous problems of their own to solve. For example, consider what happens if such a machine has to accept data from a keyboard, or another machine hooked up to it? Keyboards don't know anything about the "right time" to send in the data! We have to have a *buffer*, a little box which lets data into the machine only if the machine clocks are in the right state. It has to make a decision: whether to accept the data right now, or to wait until the next cycle, as the data came in too late. The fact that a decision has to be made introduces the theoretical possibility of a hang-up caused by the data coming in at *just* such a time that the buffer is not quick enough to make a decision — it

can't make its mind up. It's a fascinating problem, and one well worth thinking about.

7.4.2: Wire Packing: Rent's Rule

Up until now we've been discussing transistors, VLSI, and this and that — and we think that's the hard part of machine design. But whenever you get to the end of a big design, and you set out to build it, you'll discover that all the algorithms and so forth that you've worked out are not enough — something always ends up getting in the way. That something is *wires*. We look at that now.

I would like to emphasize that wires represent a real problem in system design. We've discussed one difficulty they cause: timing problems resulting from the finite time it takes to load them. But another problem is that the space needed for the wiring, connecting this chip to that and the other[3] is greater than that needed for the functioning components, like transistors! Now there is no guarantee that wires will forever reign supreme: with optical fibers, for example, we can send multiple messages down single wires by using light of differing frequencies. People occasionally break down and begin to dream, having brilliant ideas such as that of building a machine, by analogy with our broadcasting system, in which each component radiates light of a particular color (say via a LED), which is broadcast throughout the machine to be picked up and acted on by frequency-sensitive components. However, at this moment in time the predominant method of current transmission is via wires, and I'd like to spend some time discussing them. Specifically, I want to address the question of how *much* wire we might need for a generic design.

Now there's very little I can say here about wire-packing — they're just wires, after all — but it turns out that there is an empirical rule, *Rent's Rule*, which purports to shed some light on this question. It's a curious rule, and I can't really vouch for how accurate it is in general, but it appears to be the case based on the *experience* of IBM. Here's how it goes. Let us suppose we have a unit, like a circuit board, and suppose further that we can segregate elements on the unit into "cells" — not too big, not too small. These cells could be

[3] I am not now concerned so much with the "wires" on the chips, but those connecting chips together — real bunches of wires that interfere with how closely chips can be stacked, and so on. [RPF]

individual chips, for example. Now suppose that:

(1) Each cell has t pins, or terminals,
(2) N cells make up a unit, and
(3) The number of terminals, or output pins, on our unit is T.

Needless to say, these numbers have to be interpreted with a certain latitude. Let us suppose we try to connect everything up so that the components talk a lot, that is, we try to minimize the wire length by packing. Then Rent's Rule states that:

$$T = tN^r \tag{7.37}$$

where $0.65 \leq r \leq 0.70$. (Since this inequality is only approximate we will take $r = 2/3$.) In other words, it claims to relate the number of wires leading to and from the unit ($\sim T$) to the density of cell packing on the unit ($= N$). A naive first question to ask might be: why not just $T = tN$? Well, for the obvious reason that many of the wires will be *internal* to the unit (Fig. 7.69):

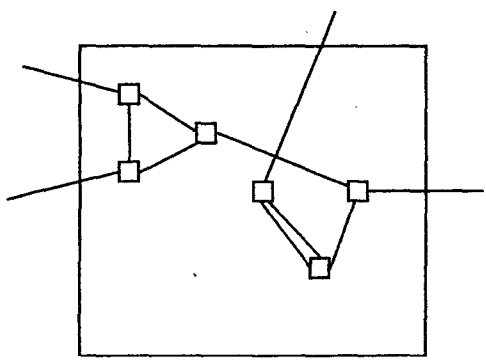


Fig. 7.69 Schematic Depiction of Fundameutal Cells on a Board

We can see how an expression such as that in Equation 7.37 might arise by moving "up" in our hierarchy of units and cells. We have considered units on which cells were joined together. We now consider *units* joined together. So let us imagine that we have a bigger unit, a "superunit", the cells of which are the units bearing the original cells. Suppose this superunit contains M units. Now, because we have set no fundamental level of analysis, there must be some consistency of scaling between these two situations. Let the number of terminals on the superunit be T_S. Clearly, each of the M units will have T terminals. Then,

$$T_S = TM^r \tag{7.38}$$

However, returning to our initial level of analysis, we can treat the superunit as comprising NM of the original cells, each of which has t pins. Using Rent's Rule again, we get:

$$T_S = t(NM)^r \tag{7.39}$$

Clearly, using (7.37), we see that (7.38) and (7.39) agree so that Rent's Rule has the correct scaling properties. This is very important.

Note that this treatment tells us nothing about the value of r (although it should be obvious from the form of the rule and the discussion following (7.37) that r would have to be less than 1). Where does this exponent come from? Well, you should remember that the value that was chosen was derived from experience, and this experience must have been influenced by problems of geometry in designing and connecting up logic circuits. That is, while it might be enticing to think that there is some neat logical reason for the value of r, that it might drop out of a pretty mathematical treatment, it's possible that it's an artifact of conventional design approaches. But for the moment, with this caveat in mind, let's assume it *is* true in the general case and see what it might teach us about wire packing.

Let's go back to the two-dimensional case. Suppose we have a square board, of side length L cm, say. Let this be the unit. We pack it with cells, each of length l cm; so we can write the number of cells on the board as $N = (L/l)^2$ (Fig. 7.70):

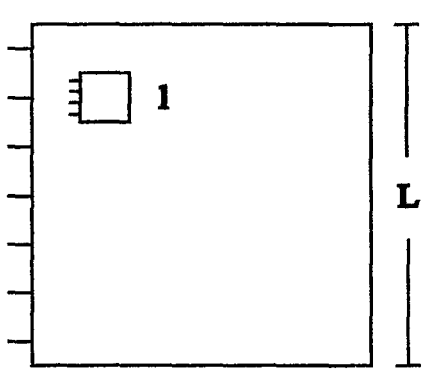


Fig. 7.70 A General Two-dimensional Unit

Now suppose that there is a restriction on how many terminals we can fit on each of the cells — that we can only place them so closely together. Let the maximum number of pins per cm on a cell perimeter be s_C. Suppose that there is also a minimum pin separation for the board terminals, with the maximum number per cm of perimeter being s_B. Rent's rule then becomes:

$$T = (4s_B L) = tN^r = (4s_C l)(L/l)^{2r} \tag{7.40}$$

and we have:

$$s_B = s_C(L/l)^{2r-1}. \tag{7.41}$$

It is clear from this that if $r > 1/2$ then, as we increase L, we need more and more pins per inch on the perimeter to take care of all the junk inside it. Therefore, we'll eventually get a jam. So as we build the machine bigger, the wiring problem becomes more serious. At the heart of this is the fact that the length of the perimeter varies as the square root of the area but the number of terminals (according to Rent) goes as the $(2/3)^{rd}$ power, a much faster scaling factor. A big incompressible mess of wires is unavoidable, and we have to increase the spacing between cells, leading to more boards, and increasing the spacing between boards, and so on, to make room. Now interestingly enough, if we were to rework this argument in three dimensions, rather than two, we get a different result: in 3-D we replace the perimeter by the surface area (length^2) and the area by the volume (length^3). Clearly, the former scales as the latter to the $(2/3)^{rd}$ power, the same as the number of terminals! So in 3-D we could just make it — we could always use the same density of pins over the surface, and we wouldn't get into a wire hassle. The problem with this sort of 3-D design, of course, is that for anyone to look at it — to see what's going on — they have to be able to get inside it, to get a hand or some tools in. At least with two dimensions we can look at our circuits from above!

Still assuming the validity of Rent's Rule, we can ask another interesting question. What is the distribution of wire lengths in a computer? Suppose we have a big, two-dimensional computer, a board covered in cells and wires. Some of the wires are short, maybe going between adjacent cells, but others may have to stretch right across the board. A natural question to ask is: if we pick a wire at random, what is the chance that is of a certain length? With Rent's Rule we can actually have a guess at this, after a fashion. Return to the two-dimensional case shown in Figure 7.70, and now take L to be the side length of some

arbitrary unit on the board. We can consider any wires connecting cells within this unit to other cells within it to be less than L in length. This is not strictly true, of course, as we might have diagonals. However, if we just deal with orders of magnitude, we shall assume we can neglect this subtlety. There will also be wires going out of this unit and hooking up to other units on the board. We take these to be longer than L. From Rent's Rule, we can calculate the number of wires of length greater than L — this will be T, the number of terminals on the unit, given in this case by:

$$T = t(L/l)^{2r}. \tag{7.42}$$

We can now calculate the probability that a random wire will have a length greater than L. It is just the right hand side of Equation 7.42 divided by the total number of wires on the unit. This is easily seen to be $\propto t(L/l)^2$. So, if the probability of a wire having length greater than L is $P(L)$, we clearly have:

$$P(L) \propto (L/l)^{2r-2}, \text{ i.e., } P(L) \propto 1/L^{2/3}, \tag{7.43}$$

using $r = 2/3$ in Rent's expression.

We can take these statistics further. Introduce the probability density $\rho(L)$, which is defined in the standard way — the probability of finding the wire length to lie between L and $L + \delta L$ is $\rho(L)\delta L$. Then we have:

$$P(L) = \int_L^\infty \rho(L')dL' \tag{7.44}$$

with

$$\rho(L) = dP/dL \propto 1/L^{5/3}. \tag{7.45}$$

Let us compute a quantity of particular interest, the mean wire length. By conventional statistical reasoning, this is:

$$[\int_l^{L_{max}} L\rho(L)dL] \,/\, [\int_l^{L_{max}} \rho(L)dL] \tag{7.46}$$

Note that we have tinkered with the limits of integration in (7.46); if we let the length L range from zero to infinity, then the numerator gives us trouble at its upper limit (infinity), as the integral is of positive dimension in L, and the denominator gives us trouble at its lower limit (zero), as its integral is of negative dimension in L. We hence set an upper limit for L, L_{max}, and also set a natural lower limit, the cell-size l. The reader can perform the integrals in (7.14) to obtain the mean wire length. The answer is:

$$2l(L_{max}/l)^{1/3}. \tag{7.47}$$

Note that this quantity is divergent: the bigger our machine (its size being given roughly by L_{max}), the bigger the mean wire length. No surprise there. However, note how it is the *cell-size*, l, that is calling the shots in (7.47); the mean length scales half as quickly with machine size as it does with cell-size (which is equivalent to cell spacing in our model). If we space our cells a little further apart, the size of the machine must balloon out of proportion.

It used to be said in the early eighties that a good designer, with a bit of ingenuity and hard work, could pack a circuit in such a way as to beat Rent's Rule. But when it came to the finished product, something always came up — extra circuits were needed, a register had to be put here, an inductance there — and, when the machine was finally built, it would be found to obey the Rule. When it comes to the finished product, Rent's Rule holds sway, even though it can be beaten for specific circuits. Nowadays, we have "machine packing programs", semi-intelligent software which attempts to take the contents of a machine and arrange things so as to minimize the space it takes up.

A Final Comment from the Editors

What remains to be said? Well, there are a few scattered lectures on the Feynman tapes that we have not attempted to put into publishable form. These lectures cover interesting topics such as the physics of optical fibers and the possibilities for optical computers. In these cases, however, technical developments have been so substantial that we have thought it best to leave topics such as these for an expert up-to-date overview in the accompanying

constitute an accurate representation of Feynman's overview of the field of computation. Moreover, these lectures, by his choice of topics, also demonstrate the subject areas that he felt were important for the future.

Afterword: Memories of Richard Feynman

I well remember my arrival at CalTech on a sunny October morning in 1970. Fresh from Oxford where even graduate students – at that time – wore ties and shirts, I was unsure what to wear for my first meeting with Murray Gell-Mann. I gambled, wrongly, on a suit and arrived in the office of the theory group secretary, Julie Curcio, feeling more and more overdressed and as if I had a large label dangling from my collar saying 'New Ph.D. from Oxford'. I had seen Gell-Mann once before in England but was unsure if the bearded individual dressed in an open-necked shirt and sitting in Julie's office was indeed the eminent professor. A moment after I had introduced myself my doubts were dispelled by Gell-Mann putting out his hand and saying "Hi, I'm Murray." This episode illustrates only a small part of the (healthy) culture shock I experienced in California. Six years in Oxford had left me used to calling my professor "Professor Dalitz, sir". At that time, I would certainly not have dared to address Dalitz by his first name!

One of my first tasks on arrival in Pasadena was to buy a car. This was not as easy as it sounds. The used car lots in Pasadena are sprinkled down Colorado Boulevard for several miles in typical US fashion and getting to them in the days when public transport in Los Angeles was probably at its lowest ebb was not straightforward. It was only after my wife and I were stopped by the police and asked why we were walking on the streets of Pasadena that I understood the paradox that, in California, you had to have a car to buy a car. Another 'chicken and egg' problem arose in connection with 'ID' – a term we had not encountered before. As a matter of routine, the police demanded to see our ID and of course the only acceptable ID in deepest Pasadena at that time was a California driver's license. A British driving license without a photograph of the bearer was clearly inadequate and even our passports were looked on with suspicion. An introduction to America via used car salesmen is not the introduction I would recommend to my worst enemy and it is not surprising that I sought advice from the CalTech grad students. I was pointed in the direction of Steve Ellis whose advice was valued because he came from Detroit and was believed to be worldly wise. I tracked Steve down to the seminar room where I saw he was engaged in a debate with a character who looked mildly reminiscent of the used car salesmen I had recently encountered. This was, of course, my first introduction to Dick Feynman – I did not at first recognize him from the much earlier photograph I knew from the three red books of the 'Feynman Lectures'. Curiously enough, even after ten years or more, I always

Compared to my previous life as a graduate student in Oxford, adjusting to life at CalTech was like changing to the fast lane on a freeway. Firstly, instead of Oxford being the center of the universe, it was evident that, to a first approximation, Europe and the UK did not exist. Secondly, I rapidly discovered that the ethos of the theory group of Feynman and Gell-Mann was that physics was all about attacking the outstanding fundamental problems of the day: it was not about getting the phase conventions right in a difficult but ultimately well-understood area. I remember asking George Zweig — a co-inventor of the whole quark picture of matter — for his comments on a paper of mine. This was the not very famous 'SLAC-PUB 1000', a paper that I had written with an experimenter friend at the Stanford Linear Accelerator Center (SLAC) about the analysis of three-body final states. George's uncharacteristically gentle comment to me was: "We do, after all, understand rotational invariance." In fact, the paper was both useful and correct but, on the CalTech scale of things, amounted to doodling in the margins of science. In those days I aspired to be as good a physicist as Zweig: this ambition strikes me now as similar to wanting to emulate the achievements of Jordan in the early days of quantum mechanics, rather than those of his collaborators, Heisenberg and Born.

One of the nicest things about CalTech was the sheer excitement of being around Feynman and Gell-Mann. As a post-doc from England, where we gain a rapid but narrow exposure to research, I was contemporary in age with the final year grad students and a lot of our social life was spent with them. Feynman was actively working with two of them — Finn Ravndal and Mark Kislinger, who had just been awarded his Ph.D. — on his own version of the quark model. Perhaps because of his work with Ravndal and Kislinger, Feynman was very involved with the final year graduate students and we all had lunch with him most days at the 'Greasy' — as the CalTech self-service cafeteria was universally known. Needless to say, our table was always the center of attraction. One frequent topic for discussion was Feynman's explanation of some new experimental results obtained at SLAC on electron proton scattering. Feynman's 'parton model' — an intuitively appealing picture of the proton made up of point-like constituents — was sweeping all before it, much to Murray's annoyance. It was not surprising that I had left Oxford full of enthusiasm for working on the parton model and looking forward to hearing Feynman on the subject he had invented. Curiously, Feynman's only publication on partons was applied to proton-proton scattering. It was when he was visiting SLAC and the experimenters told him of their surprising results with electrons and protons that Feynman realized that this was a much simpler application of his parton model. There and then, Feynman gave a seminar in which he explained their results

left to Bjorken, who had been away from SLAC at the time of Feynman's visit, and Paschos, a post-doc at SLAC, to write up the analysis of the experimental results in terms of 'Feynman's Parton Model'.

My first encounter with Feynman on a technical level was intimidating. Two CalTech experimenters, Barry Barish and Frank Sciulli, had just had a proposal for a neutrino-proton experiment accepted. Since I liked to work with experimenters, they asked me to give an informal lunch-time seminar to their group explaining the application of the 'parton model' to their experiment. Imagine my surprise when I turned up to talk to the experimental group, on finding Feynman sitting in the audience. Still, I started out and even managed to score a point off Feynman. At an early stage in the lecture, he asked how I derived a particular relation. I replied, with what now seems like foolhardy temerity: "I used Conserved Vector Current theory — you should know, you invented it!" In fact all went well until I had nearly reached the end of the seminar. I was just outlining what further predictions could be made when Feynman said: "Stop. Draw a line. Everything above the line is the parton model — below the line are just some guesses of Bjorken and Paschos." As I rapidly became aware, the reason for Feynman's sensitivity on this point was that Murray was going around the fourth floor of Lauritsen at CalTech, growling that "Partons are stupid" and that "Anyone who wants to know what the parton model predicts needs to consult Feynman's entrails!" In fact, all the results above Feynman's line in my seminar were identical to predictions that Murray had been able to derive using much more sophisticated algebraic techniques. Feynman wanted to dissociate himself from some of the wilder 'parton model' predictions of others and to stress that his simple intuitive parton approach gave identical predictions to Gell-Mann's much more fancy methods. Unfortunately for me, my lecture just happened to be a handy vehicle for him to make this point!

There were, of course, drawbacks to being in the same group as Feynman and Gell-Mann. I came to CalTech with the firm intention of pursuing research on Feynman's parton model. What I had not realized was that CalTech was the one place that one could not publish research on partons! Why was this? There was the obvious distaste of Gell-Mann for the whole approach but that would not have mattered if it had not been for the awkward fact of 'Feynman's notebooks'. I used to go to Feynman with some idea and proudly display my analysis on his blackboard. Each time Feynman listened, commented and corrected — and then proceeded to derive my 'new' results several different ways, pulling in thermodynamics, rotational invariance or what have you, and

derive the same result by a number of different physical approaches he felt more confidence in its correctness. Although this was very educational and stimulating, it was also somewhat dispiriting and frustrating. After all, one could hardly publish a result that Feynman already knew about and had written down in his famous working 'notebooks' but had not bothered to publish. So it was somewhat in desperation that I turned to Gell-Mann's algebraic approach for a more formal framework within which to work. With Jeff Mandula, an assistant professor, I looked at electron-proton scattering when both the electron and proton were 'polarized' — with their spins all lined up in the same direction. We found a new prediction whose parton equivalent was obscure. Roughly speaking, at high energies the spin direction of the parton is unchanged by collision with an electron. Our result concerned the probability of the parton spin changing its direction in the collision: this was related to 'spin-flip' amplitudes normally neglected in the parton model. Armed with this new result, I went to Feynman and challenged him to produce it with his parton approach. In the lectures he gave at CalTech the next term, later published as the book *Photon-Hadron Interactions*, you will find how Feynman rose to this challenge.

Life at CalTech with Feynman and Gell-Mann was never boring. Stories of their exploits abounded — many of Feynman's now preserved for posterity by his friend Ralph Leighton in *Surely You're Joking, Mr. Feynman!* There were many other stories. A friend told me of the time he was about to enter a lecture class and Gell-Mann arrived at the door to give the class. My friend was about to open the door but was stopped by Murray saying: "Wait!" There was a storm raging outside the building and at the appearance of a particularly violent flash of lightning, Gell-Mann said "Now!", and entered the class accompanied by a duly impressive peal of thunder. Another story that circulated was of Feynman giving a talk about the discovery, with Gell-Mann, of the V-A model of weak interactions. After the talk, one of the audience came up to him and said: "Excuse me, Professor Feynman, but isn't it usual in giving a talk about joint research to mention the name of your collaborator?" Feynman reportedly came back with: "Yes — but it's usual for your collaborator to have done something!" Obviously these stories get inflated in the telling but I did ask Feynman about this one since it seemed so out of character to the Feynman I knew. He smiled and said "Surely you don't believe I would do a thing like that!" I only knew Feynman after he had received the Nobel Prize and found happiness in his marriage to Gweneth. A somewhat more abrasive and aggressive picture of him before this time emerges from the Feynman biographies, so I am still not sure! Certainly he enjoyed making a quick and amusing response. This feature of Feynman's was often in evidence in seminars given by visiting speakers. On one memorable occasion, the speaker started out by writing the title of his talk on

the board: "Pomeron Bootstrap". Feynman shouted out: "Two absurdities" and the room dissolved into laughter. Alas for the speaker, he was deriving theoretical results supposedly valid in one energy regime but going on to apply them in another. This was just the kind of academic dishonesty that Feynman hated and on that particular occasion the speaker had a very uncomfortable time fielding brickbats thrown from the entire audience. Feynman could be restrained: on the occasion of another seminar he leaned over to me and whispered "If this guy wasn't a regular visitor, I would destroy him!"

It was during this time at CalTech that Feynman gave his celebrated lecture in the Beckman Auditorium on 'Deciphering Mayan Hieroglyphics'. Feynman's account of his honeymoon in Mexico with his second wife Mary Lou, and his efforts to decipher the Dresden Codex is contained in *Surely You're Joking, Mr. Feynman!* The lecture itself was a typical Feynman tour de force. The story illustrates perfectly Feynman's approach to tackling a new subject. Rather than look at a translation of the Codex, Feynman made believe he was the first to get hold of it. Struggling with the Mayan bars and dots in the tables, he figured out that a bar equalled five dots and found the symbol for zero. The bars and dots carried at twenty the first time but at eighteen the second time, giving a cycle of 360. The number 584 was prominent in one place and was made up of periods of 236, 90, 250 and 8. Another prominent number was 2920 or 584 x 5 and close by there were tables of multiples of 2920 up to 13 x 2920. Here Feynman says he did the equivalent of looking in the back of the book. He scoured the astronomy library to find something associated with the number 584 and found out that 583.92 days is the period of Venus as it appears from the Earth. The numbers 236, 90, 250 and 8 were then connected with the different phases of Venus. There was also another table that had periods of 11,959 in the Codex which Feynman figured out were to be used for predicting lunar eclipses. With a typical down-to-earth analogy, Feynman likened the Mayans' fascination with such 'magic' numbers to our childish delight in watching the odometer of a car pass 10,000, 20,000, 30,000 miles and so on. As Feynman says, "Murray Gell-Mann countered in the following weeks by giving a beautiful set of six lectures concerning the linguistic relations of all the languages of the world". For these lectures, Murray used to arrive clutching armfuls of books and proceed to tell his audience about the classification of languages into 'Superfamilies' with a common origin. He was always fond of drawing attention to the similarities between English and German and, for example, delighted in calling George Zweig, George Twig. I still have some notes of his lectures – with examples from the Northern, the Afro-Asiatic, the Indo-Pacific, the Niger-Kardofanian, the Nilo-Saharian Superfamilies amongst others. Even though it seemed a bit strange for professional particle physicists

to be attending lectures on comparative linguistics, life at CalTech was always interesting! I have always suspected that Feynman's account of his time with his father in the Catskills described in *What Do You Care What Other People Think?*, the second volume of anecdotes produced with Ralph Leighton, was partly directed at Gell-Mann's passion for languages and names. In the story, Feynman's father says "You can know the name of that bird in all the languages of the world, but when you're finished, you'll know absolutely nothing whatever about the bird". Feynman credits his "knowing very early on the difference between knowing the name of something and knowing something" to these experiences with his father.

Other recollections of Feynman are still fresh in my memory. One time I went to get the coffee at lunch in the Greasy and returned to find that Feynman had invited my wife down to their house in Mexico for the weekend – with his family, I hasten to add. As an afterthought he invited me too and we found ourselves strolling along the beach in Mexico, talking physics with Feynman late into the night. Feynman's advice to me on that occasion was: "You read too many novels." He had started out very narrow and focused and only later in life had his interests broadened out. Good advice perhaps, but during the years I knew Feynman I also learnt how impossible he was for anyone to emulate – in his disregard for the 'unimportant' things of life, like committees and administration, and in his unique ability to attack physics problems from many different angles. On another visit to CalTech many years later, sitting with him in the garden of his house in Altadena, Feynman proceeded to take off his belt and demonstrate his new understanding of the spin-statistics rule. He later wrote this up in a memorial lecture to his hero in physics, Paul Dirac, discoverer of anti-matter. This was some twenty years after the publication of *The Feynman Lectures on Physics* in which he had apologized for not being able to give an elementary explanation of this rule. As he said then: "This probably means we do not have a complete understanding of the fundamental principle involved."

What made Feynman's lectures unique? The well-known Cornell physicist David Mermin, himself noted for his thoughtful and penetrating analyses of supposedly well-understood problems in physics, was moved to say: "I would drop everything to hear him lecture on the municipal drainage system." In 1967 the Los Angeles Times Science editor wrote: "A lecture by Dr. Feynman is a rare treat indeed. For humor and drama, suspense and interest it often rivals Broadway stage plays. And above all, it crackles with clarity. If physics is the underlying 'melody' of science, then Dr. Feynman is its most lucid troubador." In the same article, the author, Irving Bengelsdorf, sums up the essence of

Feynman's approach: "No matter how difficult the subject — from gravity through quantum mechanics to relativity — the words are sharp and clear. No stuffed shirt phrases, no 'snow jobs', no obfuscation." A New York Times article in the same year said that Feynman "uses hand gestures and intonations the way Billy Rose used beautiful women on the stage, spectacularly but with grace."

For me, it was Feynman's choice of words that made a Feynman lecture such a unique experience. The same New York Times article went on to say that "his lectures are couched in pithy often rough-cut phrases." There are innumerable examples to choose from. In the middle of pages of complicated mathematics Feynman deliberately lightens up the text by introducing phrases like "you can cook up two new states . . ." or by personalizing the account by introducing imagined conversations of physicists as in "Now — said Gell-Mann and Pais — here is an interesting situation." In his invited lecture in 1971, on the occasion of the award of the Oersted medal for his services to the teaching of physics, Feynman began disarmingly by saying "I don't know anything about teaching" and then proceeded to give a fascinating account of the research problem he was working on — "What is the proton made out of? Nobody knows but that's what we're going to find out." In the talk he likened smashing two protons together to smashing two watches together: one could look at the gearwheels and all the other bits and pieces that resulted and try to figure out what was happening. In this way he was able to explain that smashing a simple point particle like an electron into a proton was much simpler because there was only one watch to look at. At a summer school in Erice in Italy one summer he was asked a question about conservation laws. Feynman replied: "If a cat were to disappear in Pasadena and at the same time appear in Erice, that would be an example of global conservation of cats. This is not the way cats are conserved. Cats or charge or baryons are conserved in a much more continuous way."

Feynman's Nobel Prize lecture should be required reading for all aspiring scientists. In it, Feynman forgoes the customary habit of removing the scaffolding that was used to construct the new theory. Instead, he tells us of all the blind alleys and wrong ideas that he had on the way to his great discoveries. The article also reveals more of Feynman's lecture technique when he says: "I shall include details of anecdotes which are of no value scientifically nor for understanding the development of the ideas. They are included only to make the lecture more entertaining." In the article we find out how Feynman first started on his attempt to answer the challenge of Dirac concerning the troublesome infinities that plagued relativistic quantum mechanics. In the last sentence of his famous book Dirac said: "It seems that some essentially new physical ideas are

here needed." Of his youthful idea to solve the problem Feynman says: "That was the beginning and the idea seemed so obvious to me and so elegant that I fell deeply in love with it. And, like falling in love with a woman, it is only possible if you do not know too much about her, so you cannot see her faults. The faults will become apparent later, but after the love is strong enough to hold you to her. So, I was held to this theory, in spite of all difficulties, by my youthful enthusiasm." Later in the lecture Feynman writes: "I suddenly realized what a stupid fellow I am; for what I had described and calculated was just ordinary reflected light, not radiation reaction." This refreshing honesty from one of the greatest physicists of the twentieth century reminds me of another of my heroes, Johannes Kepler — who was first to write down laws of physics as precise, verifiable statements expressed in mathematical terms. Unlike Copernicus and Newton, Kepler wrote down all the twists and turns in his thought processes as he was forced to the shocking conclusion that the orbit of Mars was not a circle but an ellipse. Kepler summed up his struggle with the words: "Ah, what a foolish old bird I have been!"

One of the best anecdotes in the lecture concerns a physicist called Slotnick and his encounter with 'Case's theorem'. This described the moment when Feynman realized that his 'diagrams' really were something new. In its full form the story runs as follows. At a meeting of the American Physical Society in New York, Slotnick presented a paper comparing two different forms for the electron-neutron coupling. After a long and complicated calculation, Slotnick concluded that the two forms gave different results. At this point, Robert Oppenheimer rose from the audience and remarked that Slotnick's calculation must be wrong since it violated Case's theorem. Poor Slotnick had to admit he had never heard of this theorem, so Oppenheimer kindly told him he could remedy his ignorance by listening to Professor Case presenting his result the next day. That evening, in his hotel, Feynman could not sleep so he decided to use his new methods to repeat Slotnick's calculations. Feynman then goes on to say: "The next day at the meeting, I saw Slotnick and said, 'Slotnick, I worked it out last night; I wanted to see if I got the same answers you do. I got a different answer for each coupling — but, I would like to check with you because I want to make sure of my methods.' And he said, 'What do you mean you worked it out last night, it took me six months!' And, when we compared the answers he looked at mine, and he asked, 'What is that Q in there, that variable Q?' I said, 'That's the momentum transferred by the electron, the electron deflected by different angles.' 'Oh,' he said, 'no, I only have the limiting value as Q approaches zero, the forward scattering.' Well it was easy enough to just substitute Q equals zero in my form and I then got the same answers as he did. But it took him six months to do the case of zero momentum

transfer, whereas during one evening I had done the finite and arbitrary momentum transfer. That was a thrilling moment for me, like receiving the Nobel Prize, because that convinced me, at last, I did have some kind of method and technique and understood how to do something that other people did not know how to do. That was my moment of triumph in which I realized I really had succeeded in working out something worthwhile." What Feynman does not say in his written lecture is that he stood up at the end of Case's talk and said: "Your theorem must be wrong. I checked Slotnick's calculation last night and I agree with his results." In the days when calculations like Slotnick's could take as much as six months, this was the incident that put 'Feynman's diagrams' on the map.

The other piece of required reading for students of all disciplines is Feynman's article on 'Cargo Cult Science'. This was originally Feynman's commencement address to new CalTech graduates in 1974 and in it, Feynman discusses science, pseudoscience and learning how not to fool yourself. The unifying theme of the talk is Feynman's passionate belief in the necessity for "utter scientific integrity" — in not misleading funding agencies about likely applications of your research, in publishing results of experiments even if they do not support your pet theory, in giving government advice they may rather not hear, in designing unambiguous rat-running experiments and so on. As he says, "learning how to not fool ourselves is, I'm sorry to say, something that we haven't specifically included in any particular course that I know of. We just hope you've caught on by osmosis." He concludes with one wish for the new graduates: "the good luck to be somewhere where you are free to maintain the kind of integrity I have described, and where you do not feel forced by a need to maintain your position in the organization, or financial support, or so on, to lose your integrity." At the risk of sounding pompous, I think the world owes a vote of thanks to CalTech for providing just such an environment for Richard Feynman. Feynman was never restricted to research in any one particular field: it is to the exercise of just this freedom that we owe these *Feynman Lectures on Computation.*

It seems appropriate to end these reminiscences with two more 'Feynman stories'. The first story harks back to his safecracking days at Los Alamos. At a Conference in Irvine in 1971 Feynman agreed to be on a discussion panel at the end of the conference. He was asked if he thought that physicists were getting anywhere with answering the 'big questions'. Feynman replied: "You ask, are we getting anywhere. I'm reminded of a situation when I was asked the same question. I was trying to pick a safe. Somebody asked me how are you doing? Are you getting anywhere? You can't tell until you open it. But you

have tried a lot of numbers that you know don't work!" The second story is the last Feynman story of all. Gweneth was by his bedside in the hospital and Feynman was in a coma. She noticed that his hand was moving as if he wanted to hold hands with Gweneth. She asked the doctor if this was possible but was told that the motion was automatic and did not mean anything. At which point, Feynman, who had been in a coma for a day and a half or so, picked up his hands, shook out his sleeves and folded his hands behind his head. It was Feynman's way of telling the doctor that even in a coma he could hear and think - and that you should always distrust what so-called 'experts' tell you!

The final word deserves to be given to James Gleick, author of a biography of Feynman. Gleick memorably summed up Feynman's philosophy towards science with the following words:

> "He believed in the primacy of doubt, not as a blemish upon on our ability to know but as the essence of knowing."

Tony Hey

Southampton
March 1996

Suggested Reading

Chapters 1, 2 and 3

The Nature of Computation: An Introduction to Computer Science by Ira Pohl and Alan Shaw
Computer Science Press (1981)

Algorithmics by David Harel
Addison-Wesley Publishing Company, 2nd edition (1992)

Computer Organization and Design: The Hardware/Software Interface by John L. Hennessy and David A. Patterson
Morgan Kaufmann Publishers (1993)

Structured Computer Organization by Andrew S. Tanenbaum
Prentice-Hall, 2nd Edition (1984)

Computation: Finite and Infinite Machines by Marvin L. Minsky
Prentice-Hall (1967)

Turing's World 3.0: an Introduction to Computability Theory by John Barwise and John Etchemendy
CSLI Lecture Notes 35, Stanford CA

Introduction to Automata Theory, Languages, and Computation by John E. Hopcroft and Jeffrey D. Ullman
Addison-Wesley (1979)

'Operating Systems' by P.J. Denning and R.L. Brown
Scientific American, September 1984, 96

'The Problem of Integration in Finite Terms' by R.H. Risch
Transactions of the American Mathematical Society **139** 167 (1969)

'Integration of Elementary Functions' by M.Bronstein
Journal of Symbolic Computation **9** (2) 117 (1990)

Chapter 4

Mathematical Theory of Communication by Claude E. Shannon
University of Illinois Press (1963)

Coding and Information Theory by Richard W. Hamming
Prentice-Hall (1980)

Principles and Practice of Information Theory by R.E. Blahut
Addison-Wesley (1987)

Communication Systems by A.B. Carlson
McGraw-Hill (1986)

Chapter 5

'Logical Reversibility of Computation' by Charles H. Bennett
IBM Journal of Research and Development **17**, 525 (1973)

'Thermodynamics of Computation — A Review' by Charles H. Bennett
International Journal of Theoretical Physics **21**, 905 (1982)

'Notes on the History of Reversible Computation' by Charles H. Bennett
IBM Journal of Research and Development **32** (1) 16 (1988)

'Zig-Zag Path to Understanding' by Rolf Landauer
Reprint from *Proceedings of the Workshop on Physics and Computation: Physcomp '94*
IEEE Computer Society Press (1994)

Maxwell's Demon: Entropy, Information, Computing edited by Harvey S. Leff and Andrew F. Rex
Adam Hilger (1990)

Chapter 6

'Quantum Mechanical Models of Turing Machines that Dissipate No Energy' by Paul Benioff
Physical Review Letters **48**, 1581 (1982)

'Conservative Logic' by E. Fredkin and T. Toffoli
International Journal of Theoretical Physics **21**, 219 (1982)

'Bicontinuous Extensions of Invertible Combinatorial Functions' by T. Toffoli
Mathematical Systems Theory **14**, 13 (1981)

'On a Simple Combinatorial Structure Sufficient for Sublying Non-Trivial Self Reproduction' by L. Priese
Journal of Cybernetics **6**, 101 (1976)

Chapter 7

Introduction to VLSI Systems by Carver A. Mead and Lyn Conway
Addison-Wesley, (1980)

The Art of Electronics by Paul Horowitz and Winfield Hill
Cambridge University Press, 2nd edition (1989)

Physics of Semiconductor Devices by S.M. Sze
Wiley, 2nd Edition (1981)

Principles of CMOS VLSI Design: A Systems Perspective by Neil H.E. Weste and Kamran Eshraghian
Addison-Wesley, 2nd Edition (1993)

Introductory Semiconductor Device Physics by Greg Parker
Prentice-Hall (1994)

'Hot-Clock nMOS' by C.L. Seitz, A.H. Frey, S. Mattisson, S.D. Rabin, D.A. Speck and J.L.A. van de Snepscheut
Chapel Hill Conference on VLSI, 1 (1985)

'Scaling of MOS Technology to Submicrometer Feature Sizes' by Carver A. Mead
Journal of VLSI Signal Processing, **8**, 9 (1994)

'The CMOS End-point and Related Topics in Computing' by Maurice V. Wilkes
IEE Computing and Control Journal **7**, 101 (1996)

INDEX

Adder, binary
  asynchronous clocking and, 276
  feedback and, 44
  from reversible gates, 172, 189
  full, 33, 38, 189
  half, 22, 27
  "pairwise", 60
  Turing machine implementation, 73
Aliasing, 136
Alphabet, source, 116, 128
  efficiency of, 116
Aluminium (as dopant), 217
Amino acids, 164
Amplifier, *see* transistor
Analogue signal transmission, 129
AND
  operator, 11
  plane, in PLA, 270
AND gate, 22 *et seq.*
  in billiard ball computer, 177
  as finite state machine, 63
  irreversibility of, 35, 153
  multiple, 27
  realized by reversible gates, 172, 189
  realized by transistors, 29, 234
  relation to OR and NOT gates, 25

Annihilation
  electron-hole, in semiconductors, 218
  operator, in quantum computer, 193
Architecture, computer, 4, 19, 94
  wires in, 277
Arithmetic
  binary, 20
Assembly language, 18

Band gap energy, 214
Band theory of conduction 213
Benioff, P., x, 182
Bennett, C.H., viii, x, 36, 146, 148, 150, 151, 154, 155, 160, 164, 166, 173, 185, 187, 209, 211
Bernoulli numbers, 16
Billiard ball computer, 176
Bipolar junction transistor, 221
Boolean algebra, 11, 22, 42
Boron (as dopant), 217
Breakdown
  voltage, in diode, 221
  in miniturization of transistor, 237
Brownian computers, 166
Brown, R.L., 4
Butting contact, in VLSI, 262

Capacitance, gate, in MOSFET, 231
  and switching time, 236
"Cargo Cult Science", 292
Carnot engine, 152, 154
Case's Theorem, 291
Central processing unit (CPU), 19
Church, A., 54
Ciphers
  Baconian, 2
  prime factoring, 90
Clocking
  asynchronous, 276
  "hot", 247
  in PLA, 269
  in shift registers, 46
  synchronous, 276

Clock skew, 274
Cocke, J., viii
Codes
error correcting and detecting, 95 *et. seq.*
Morse, 124
perfect, 105
Coding, 95 *et seq.*
analogue signals, 129
Hamming, 98
Huffman, 124
predictive, 127
Collision
computation, in billiard ball computer, 177
gate, 178
Communication theory, 95 *et seq.*
Compilers, 18
Complementary Metal Oxide Semiconductor (CMOS) technology, 212, 230, 238 *et seq.*
energy dissipation in, 238, 244
logic gates in, 239, 243
Computation
mathematical limitations on, 52 *et seq.*
reversible, 151 *et seq.*, 185
speed of, and energy cost, 167
thermodynamics of, 151 *et seq.*
Computability,viii, ix, x, xiii, 88
and Turing machines, 54
Computers
architecture, 4, 19, 94
ballistic, 167, 176
Billiard ball, 176
Brownian, 166
Component failure in, 94
energetics of, 137, 151 *et seq.*
file clerk model of, 5
instruction hierarchies in, 3, 18
memory, 10, 42, 104, 269
organization, 20 *et seq.*
and primitive elements, 186
quantum mechanical, ix, 182, 185 *et seq.*
reversible, 151 *et seq.*
CONTROLLED CONTROLLED NOT (CCN) gate, 36 *et seq.*
as complete operator, 38
in quantum computer, 192 *et seq.*
reversibility of, 36, 154, 188
CONTROLLED NOT (CN) gate, 36 *et seq.*
in quantum computer, 194, 206
realized by switches (in QMC), 203
reversibility of, 37, 188
Converter, unary to binary, 74
Conway, L., 212, 240, 262, 264, 295
Copernicus, N., 291
"Copy" computation, 155
dipole copier, 160
realized in Nature, 163, 170
Copying Turing machine, 79
"Cosmic Cube", computer, ix
Creation
electron-hole, in semiconductors, 218
operator, in quantum computer, 194
Crossover gate, in billiard ball computer, 179

D-type flip-flop, 49
Data compression, 115, 124
Decoder
binary, 30, 40
realized by logic gates, 31
Deoxyribonucleic acid (DNA), 120, 163, 170
Delay
finite state machine, 57
identity gate as, 24
de Morgan's Theorem, 25
Denning, P.J., 3, 294
Depletion region
in MOSFET, 223, 260
in pn junction, 218
Depletion mode, transistor, 224, 230
in VLSI, 259, 262
Design rules, in VLSI, 263
Diffusion
electron-hole, in pn junction, 218

Diode, 213
pn junction, 217
light emitting, 220
Dipole copier, 160
Dirac, P., 289, 290
Direct Load, instruction, 13
"Dissipated action", 252
Doping, in semiconductors, 215
Drift velocity
cursor, in quantum computer, 200
electrons in silicon, 225, 254

Effective procedures, 52, 89
in calculus, 53
in geometry, 53
Turing machines and, 55, 67, 80
Efficiency
of coding, 102
of Turing machines, 54
Electromigration, 264
Electron-proton scattering, 285, 287
Encoder
binary, 33, 40
predictive, 127
see also: Coding
Enhancement mode, transistor, 224, 234
in VLSI, 261
p-channel, 231
Entropy
and information theory, 123
and reversible computation, 174
in thermodynamics, 140
Errors, correction and detection, 95 *et seq.*
component failure and, 94
multiple, 98
single, 96
EXCHANGE gate, 34 *et seq.*
controlled *see* Fredkin gate
as reversible primitive, 188
in VLSI, 273

FANOUT gate, 34 *et seq.*
constructed from CN gates
"double", in billiard ball computer, 178
as reversible primitive, 188
Feynman, R.P., viii, ix, x, xi, 1, 2, 95, 123, 212, 217, 238, 252, 283, 284 *et seq.*
and "Cargo Cult Science", 292
diagrams, 291
and Mayan hieroglyphics, 288
"notebooks" of, 286
parton model of, 285
File clerk model, of computer operation, 4
Finite state machines, 55
composite, 62
delay, 58
general, 64
and grammars, 91
limitations of, 60
logic gates as, 63
parity, 59, 96
and Turing machines, 66, 80
"Firing Squad" problem, 65
Flip-flops, 42
and clocking, 47, 250
and computer memory, 42
D-type, 49
master-slave, 48
RS, 47
Fluidic analogy
MOSFET operation, 225
energy dissipation in switching, 244, 248
Follower circuit, 235
Forward bias
in diode, 220
in MOSFET, 226
Fourier Transforms, 133
Fox, G., ix
Fredkin, E., 36, 39,176, 183, 185, 209, 211
Fredkin gate, 39
realized by billiard ball gates, 180
Free energy, 140
and information, 143
loss, in quantum computer, 199
and reversible computation, 155
and Shannon's Theorem, 150

Gallium Arsenide (as dopant), 220
Gang disks, 105

as finite state machines, 63
malfunctions in, 94
reversible, 34, 153, 187
see also AND, CONTROLLED NOT, CONTROLLED CONTROLLED NOT, EXCHANGE, FANOUT, FREDKIN, NAND, NOT, OR, XOR
Gell-Mann, M., 284 *et seq.*
General recursive propositions, 54
Gleick, J., 293
Gödel, K., 52
Gödel's Theorem, 52
Goldbach's conjecture, 52
Grammars
machine implementation of, 91
and machine translation, xiii

Halting Problem, 80 *et seq.*
Hamiltonian, in quantum computer, 185, 191 *et seq.*
Hamming
code, 98
distance, 112
and gang disks, 104
Hamming, R.W., 99
Hillis, D., viii
Hot clocking, 247
Hopfield, J., viii, ix
Huffman coding, 124
Huffman, D.A., 124

Imperfections, in quantum computer, 199
Inductance,
and energy dissipation during switching, 187, 244
in VLSI, 247
Information, viii, 115 *et seq.*
as fuel, 146
Information theory, viii, ix, 115 *et seq.*
Instruction sets, 3, 8
in general PLA, 268
Insulators, 214
Inversion layer, in MOSFET, 224
Inverter, 28
amplification in, 235, 242
chains of, 235, 266
seq.
in hot clocking process, 248
nMOS, 233
switching time of, 252
in VLSI, 262
see also Transistor, MOSFET

Jump, instruction, 13

Kepler, J., 291
Kleene, S.C., 54

Landauer, R., x, 148, 151
LASERs, 220
Law of Mass Action, 216
Leff, H.S., 148, 295
Leighton, R., 287, 289
Light emitting diode (LED), 220
Locating Turing machine, 75

Majority logic decisions, 104
Martin, A.J., 252
Mask, *see* Planar process fabrication
Master-slave flip-flop, 48
Maxwell's Demon, 148
Maxwell, J.C., 148, 149
Mead, C.A., viii, ix, 160, 212, 240, 262, 264, 296
Mean free path, 169, 200, 253
Memory, computer, 10, 42, 104, 269
Message space, 110
Metal Oxide Semiconductor Field Effect Transistor (MOSFET), 222
as amplifier, 235
in CMOS devices, 239
depletion mode, 230
energy use in, 238
enhancement mode, 230
fluid analogy for operation, 225
gate capacitance of, 231, 236
in hot clocking, 249
and logic gates, 233 *et seq.*
saturation, 225
threshold voltage, 224, 230
timing in, 236
Minsky, M.L., viii, 75, 82, 85, 295

Multiplexer, 32
Multiplier
binary, 73
unary, 72

NAND gate, 29 *et seq.*
irreversibility of, 35
realized in CMOS, 243
realized by transistors, 29, 186, 234
relation to AND and NOT gates, 35, 186
stick figure for, 266
in VLSI, 263
Newton, I., 291
nMOS technology, x, 29, 212, 223 *et seq.*
energy use in, 238
NOR gate, 30 *et seq.*
in flip-flops, 45
realized by transistors, 30, 234
NOT gate, 24 *et seq.*
and clocking, 48, 65
as finite state machine, 63
in quantum computer, 194, 210
realized by transistors, 28, 233, 262
relation to AND and OR gates, 25
as reversible primitive, 153, 187
see also inverter
Nucleotides, and protein synthesis, 164

Opcode, 14
Operators, complete sets of, 25, 38, 39, 186
Oppenheimer, J.R., 291
OR
operation, 12
plane, in PLA, 270
OR gate, 23 *et seq.*
in flip-flops, 44
irreversibility of, 35, 153
in predictive encoder, 128
realized by transistors, 30
relation to AND and NOT gates, 25

Parallel processing, 4, 18, 104, 255
error correction in, 167
and MOSFET gate electrons, 256
Parity, 59, 96
Parity counting finite state machine, 59
Parity checking, 96, 113
Partons, viii, 285, 286
Pass transistor, 267
modeling chain of, 275
Periodic table, 215
Phosphates, in protein synthesis, 164, 170
Phosphorus (as dopant), 223, 260
Planar process fabrication (VLSI), 258
pMOS technology, 223
Polysilicon, 223
path construction in VLSI, 260
role in VLSI, 261
signal propagation in, 275
Post, E., 54, 92
Post machine, 93
Priese, L., 202, 211
Prime numbers, factorization of, 90
Program counter, 8
in quantum computer, 196, 202
Programmable logic arrays (PLAs), 42, 267
Protein synthesis
as "copy" process, 164, 170
energy dissipation in, 166, 187
Pseudotape, in Turing machine, 68

Quantum mechanical computer, ix, 182, 185 *et seq.*
conditional operations in, 206
CONTROLLED NOT in, 203
effects of imperfections, 199
Hamiltonian in, 185, 191 *et seq.*
incremental binary counter in, 210
switching in, 202
Quantum theory, xiv, 4, 181
and computing, 181, 185 *et seq.*
and electrical conduction, 213
see also Uncertainty Principle
Quark picture of matter, 285
Quintuples, Turing machine, 67

Read only memory (ROM) control system, 269
Redirection gates, in billiard ball computer, 178

in file clerk model, 9
in quantum computer, 192
shift, 46, 50, 267
transfer language, 9
Rent, E., 277
Rent's Rule, 277
Resist, *see* Planar process fabrication
Resistance
and energy dissipation in inverter, 243
implementations in VLSI, 29, 230, 262
MOSFET as, 225, 230, 240
Reverse-bias
in diode, 220
in transistor, 226, 228, 243
Reversible computation, ix, 151 *et seq.*, 185
and thermodynamics of computation, 151
general reversible computer, 172
Rex, A.F., 148
Ribonucleic acid (RNA), 164
polymerase, 164, 170
Risch, R.H., 53
RS flip-flop, 47

Sampling Theorem, 133
Satellite communication, 103, 110
Saturation, in MOSFET, 225
fluidic analogy of, 227
"Scheme", programming language, ix
Second Law of Thermodynamics, 141, 148
Seitz, C.L., ix, 252
Semiconductors, ix, 28, 213
doping of, 215
electrons and holes in, 215
n-type, 216
p-type, 217
see also Silicon
Shannon, C., 106, 110, 123, 132, 294
Shannon's Theorem, ix, 106
energy and, 150
and message space, 110
and predictive encoding, 129
Shift registers, 46, 50
chip construction, *see* Planar process fabrication
dioxide, use in VLSI, 258
doping, 215
n-type, 216, 222, 230, 239, 258
p-type, 217, 222, 230, 239, 258
see also Polysilicon
Smith, W., 1
State diagrams
finite state machines, 56
Turing machines, 70
States, availability of, 170
Stick figures, in VLSI, 266
Stirling's formula, 108
Subroutines, in reversible computer, 175
Switching
device, in billiard ball machine, 179
functions, 23
"one electron switch", 253
in quantum computer, 202

Taylor, R., 52
Thermal excitation, of electrons, 214
Thermodynamics, ix, xi, 139
and information, 123
and measurement, 148
Threshold voltage, in MOSFET, 224
Timing
and shift registers, 46
in finite state machines, 64
in inverter, 236
in ROM control system, 269
controlled by components, 276
in quantum computer, 196
see also Clocking
Toffoli, T., 176, 185, 187, 209, 211
Transfer operations, 7
Transistor, x, 28, 213
as amplifier, 221
and AND gate, 29, 234
depletion mode,224, 230, 259, 262
energy dissipation in, 137, 152, 187, 238
enhancement mode, 224, 234, 261
and NAND gate, 29, 186, 234
and NOR gate, 30, 234

pass, 267, 275
in PLA, 271
reliability of, 94
in VLSI, 259
see also Inverter, MOSFET
Turing, A.M., 54,55, 66, 88
Turing machines, ix, x, 54, 66
and computability, 54, 80
copying machine, 79
countability of, 89
and Halting Problem, 80 *et seq.*
locating machine, 75
parenthesis checker, 71
parity counter, 68
Universal, 54, 67, 80 *et seq.*
Turing computability, 80
and effective procedures, 55

Uncertainty Principle, 181
limitations due to, 185, 201
"dissipated action" and, 257
Universal Turing machines (UTMs), 54, 67, 80 *et seq.*
and Halting Problem, 80
Universality, 2

Valence band, 213
V-A model of weak interactions, 287
Very Large Scale Integration (VLSI), viii, xi, 257 *et seq.*
circuit construction, *see* planar process fabrication
energetics of, 243
inductance in, 247
path conventions in, 258
timing in, 236
resistance in, 29, 230, 262
transistors in, 221, 261
Von Neumann, J., 4, 18,19, 123
Von Neumann, architecture, 4, 19, 94
Voyager satellite, 103

Wiles, A., 52
Wire
and clock skew, 274
packing, 277

XOR operation, 12
XOR gate, 23 *et seq.*
and computer memory, 43
constructed from AND and OR gates, 26
irreversibility of, 35, 188
relation to CN gate, 37

Zweig, G., 285, 288