Feynman

Hey and
Allen, Eds.

FEYNMAN LECTURES ON COMPUTATION

FEYNMAN LECTURES ON COMPUTATION

RICHARD P. FEYNMAN

EDITED BY
ANTHONY J.G. HEY AND 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

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

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

123456789—MA—0099989796
First printing, August 1996

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 others I have taken the liberty of correcting 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 'real'

---

[1] William Smith was the father of modern geology; in his work as a canal and mining engineer he 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 the superposition principle in which rocks are successively laid down upon older layers. Prior to Smith's great contribution, geology was more akin to armchair philosophy than an empirical science. [Editors]

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]

CONTENTS

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]