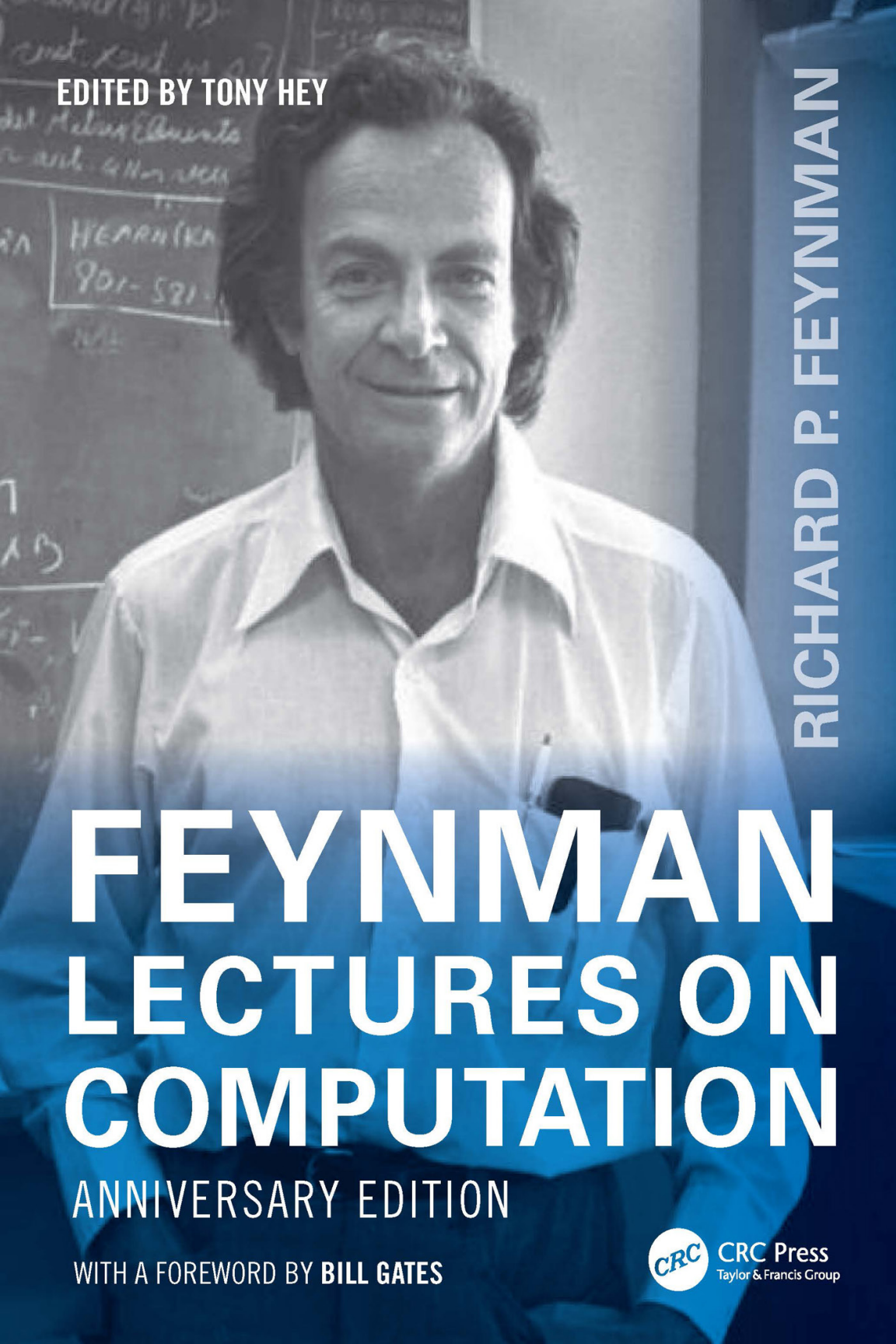


EDITED BY TONY HEY

RICHARD P. FEYNMAN

FEYNMAN LECTURES ON COMPUTATION

ANNIVERSARY EDITION

WITH A FOREWORD BY **BILL GATES**

CRC Press
Taylor & Francis Group

Feynman Lectures on Computation

The last lecture course that Nobel Prize winner Richard P. Feynman gave to students at Caltech from 1983 to 1986 was not on physics but on computer science. The first edition of the *Feynman Lectures on Computation*, published in 1996, provided an overview of standard and not-so-standard topics in computer science given in Feynman's inimitable style. Although now over 20 years old, most of the material is still relevant and interesting, and Feynman's unique philosophy of learning and discovery shines through.

For this new edition, Tony Hey has updated the lectures with an invited chapter from Professor John Preskill on "Quantum Computing 40 Years Later". This contribution captures the progress made toward building a quantum computer since Feynman's original suggestions in 1981. The last 25 years have also seen the "Moore's law" roadmap for the IT industry coming to an end. To reflect this transition, John Shalf, Senior Scientist at Lawrence Berkeley National Laboratory, has contributed a chapter on "The Future of Computing beyond Moore's Law". The final update for this edition is an attempt to capture Feynman's interest in artificial intelligence and artificial neural networks. Eric Mjolsness, now a Professor of Computer Science at the University of California Irvine, was a Teaching Assistant for Feynman's original lecture course and his research interests are now the application of artificial intelligence and machine learning for multi-scale science. He has contributed a chapter called "Feynman on Artificial Intelligence and Machine Learning" that captures the early discussions with Feynman and also looks toward future developments.

This exciting and important work provides key reading for students and scholars in the fields of computer science and computational physics.

Richard P. Feynman. Photograph courtesy of Michelle Feynman and Carl R. Feynman

Feynman Lectures on Computation

Anniversary Edition

Richard P. Feynman

Edited by

Tony Hey

With a Foreword by Bill Gates

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

Anniversary edition published 2023
by CRC Press
6000 Broken Sound Parkway NW, Suite 300, Boca Raton, FL 33487-2742

and by CRC Press
4 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

CRC Press is an imprint of Taylor & Francis Group, LLC

First published 1996 by Westview Press

First edition published by CRC Press 2018

Library of Congress Cataloging-in-Publication Data

Names: Feynman, Richard P. (Richard Phillips), 1918-1988, author. | Hey, Anthony J. G., editor.
Title: Feynman lectures on computation / Richard P. Feynman ; edited by Tony Hey.
Description: Anniversary edition. | Boca Raton : CRC Press, 2023. | Series: Frontiers in physics | Includes Feynman's lectures with some new supplementary essays. | Includes bibliographical references and index.
Identifiers: LCCN 2022046479 (print) | LCCN 2022046480 (ebook) | ISBN 9781032415888 (hardback) | ISBN 9780367857332 (paperback) | ISBN 9781003358817 (ebook)
Subjects: LCSH: Electronic data processing. | Computer science. | Feynman, Richard P. (Richard Phillips), 1918-1988.
Classification: LCC QA76 .F45 2023 (print) | LCC QA76 (ebook) | DDC 004--dc23/eng/20221122
LC record available at https://lccn.loc.gov/2022046479
LC ebook record available at https://lccn.loc.gov/2022046480

ISBN: 978-1-032-41588-8 (hbk)
ISBN: 978-0-367-85733-2 (pbk)
ISBN: 978-1-003-35881-7 (ebk)

DOI: 10.1201/9781003358817

Typeset in Palatino
by Deanta Global Publishing Services, Chennai, India

Contents

Reminiscences

Afterword

Foreword by Bill Gates

I was working hard at Microsoft when Richard Feynman gave the lectures in this book. But it wasn't his thoughts about computing that first got my attention – it was his Messenger Lectures on physics.

In the mid 1980s, I was planning a trip with a friend and decided to add some learning to our relaxation. I found one of the Messenger Lectures in a local university's film collection and checked it out. I loved it so much that I ended up watching it twice and immediately went looking for more! Years later I bought the rights to those lectures and worked with Microsoft to get them posted online for free.

Feynman had an amazing knack for making physics clear and fun at the same time. He took obvious delight in knowledge – during the lectures, you could see his face light up when he got excited about something. And he made the ideas so clear that anyone could understand them. I didn't read his lectures on computation until some time later, but they have all the same joyful qualities.

Given how quickly the computing industry has developed, you may be wondering how a series of thirty-year-old lectures could still be relevant today. After all, as I write this foreword, anyone with an Internet connection has access to an artificial intelligence agent – what Feynman called "advanced applications" – that can hold its own in a conversation and will soon be able to do even more.

There are several reasons. For one thing, some of the original chapters are as useful today as they were three decades ago. The first three, for example, are as clear and concise an explanation of how computers work as I have seen. Anyone who wants to do research, write code, or just understand the workings of their laptop machine should read them.

In addition, my friend and former colleague Tony Hey has done an admirable job of updating this edition with new material on neural networks, quantum computing, and the end of Moore's Law. It's a testament to Feynman's thinking that these subjects fit so naturally into this book. He not only foresaw developments like robotics, computer vision, and speech that were still decades away; he also created an intellectual framework that was powerful and flexible enough to accommodate them.

Finally, all of us can learn from the high standards that Feynman held himself to. He insisted that he didn't really understand something until he had figured it out for himself, checked his work in multiple ways, and

explained it in simple terms to someone else. And he hated the idea that he might be fooling himself – a quality that I've been trying to emulate for years, in my business career and now in my philanthropy. Even if you don't care about computing, you should care about testing your own assumptions, and no one questioned his own assumptions more rigorously than Richard Feynman.

Editor's Preface

The original printing of the *Feynman Lectures on Computation* was in 1996. Although the published lectures are now nearly 25 years old, much of the material is Feynman's treatment of many standard topics in computer science, such as computability and information theory, as well as some not-so-standard topics, such as reversible computing and quantum computers. In May 2019, I was pleasantly surprised to be contacted by a commissioning editor, Carolina Antunes, from Taylor and Francis about the possibility of publishing a new edition of the lectures to mark their 25th anniversary. Due to the Covid-19 pandemic, publication of the new edition was delayed.

Forty years ago, in his talk at a conference at MIT in May 1981, Feynman raised the possibility of building a new type of computer made up of intrinsically quantum mechanical elements that could be used to simulate large quantum systems that could not be simulated on a classical computer. Remarkably, Feynman stated explicitly that such a computer was "not a Turing machine, but a machine of a different kind". In the *Lectures on Computation*, I included his later, more detailed, analysis of how one might design such a quantum computer. With the present excitement about progress in quantum computing, there has been a continuing and growing interest in Feynman's lectures.

In the 25 years since the original publication of the lectures, what else has changed? After a brief introduction to logic gates and computers, Feynman examines fundamental limits to computation – from mathematics, from information theory, from thermodynamics, from quantum mechanics, and from semiconductor technology. Two major changes stand out. First, the progress made toward actually building quantum computers capable of performing useful calculations. Second, Moore's Law, as embodied in the IT industry's technological roadmap that has historically charted the course for the industry to make digital systems faster, smaller, and cheaper roughly every two years, is coming to an end. It therefore seems a good idea to include new material that reflects the significant progress and change in these two areas.

I am therefore delighted to include two new chapters written by expert guest authors on these advances. John Preskill, the Richard P. Feynman Professor of Theoretical Physics at Caltech, has contributed a new chapter on "Quantum Computing 40 Years Later", which details the progress made since Feynman's original 1981 talk at MIT. John Shalf, Senior Scientist at the Lawrence Berkeley National Laboratory, has contributed a chapter on "The Future of Computing beyond Moore's Law" that looks at

the different opportunities and strategies to continue computing performance improvements given the breakdown of the historical technology drivers.

I have also looked through the transcripts of the audio recordings and my original notes on the lectures to see if there was any significant additional content that could be included. Figure 0.1 shows Feynman's original

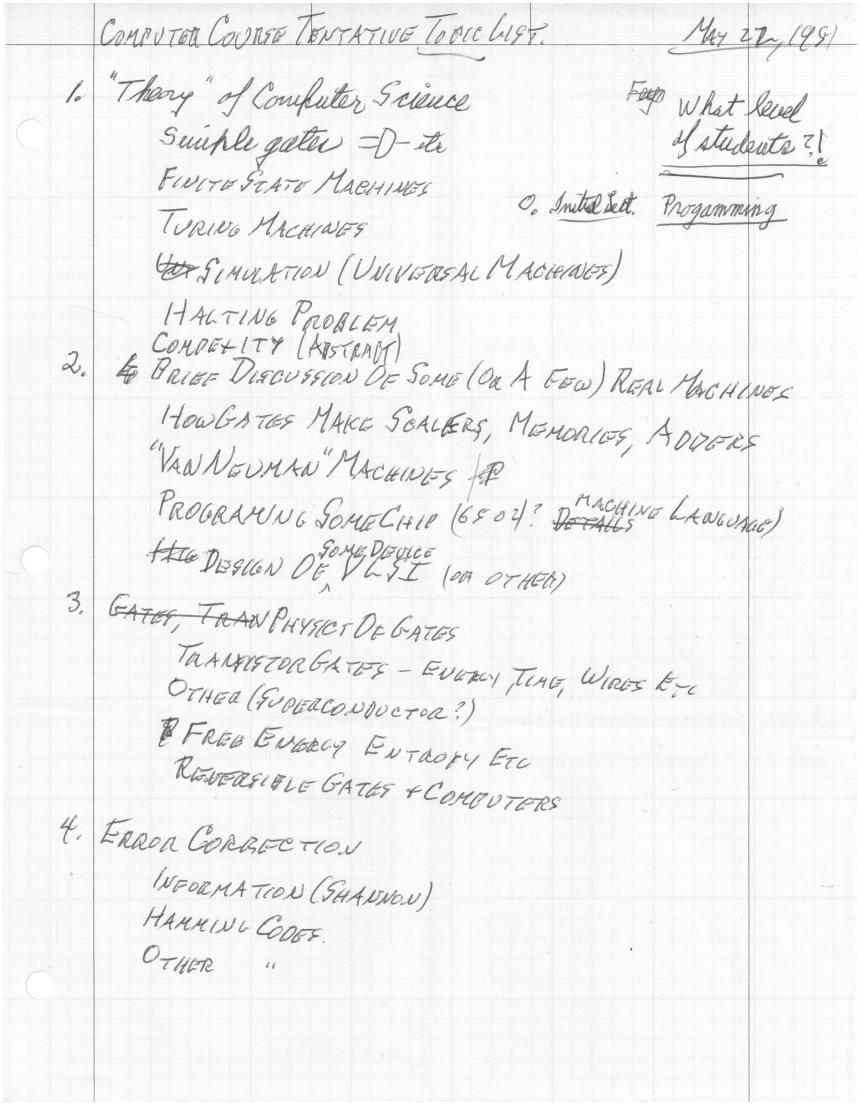


FIGURE 0.1
Feynman's 1981 draft course outline. Images are reproduced by kind permission of the Caltech archive.

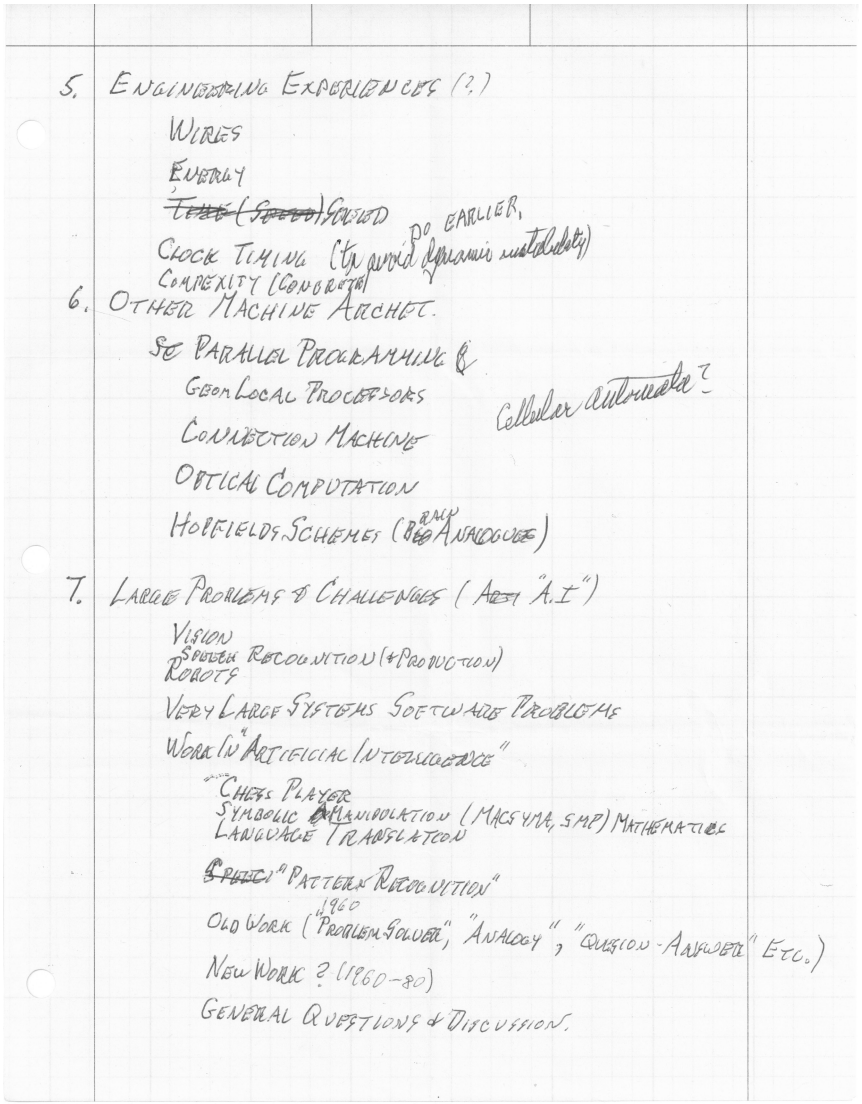
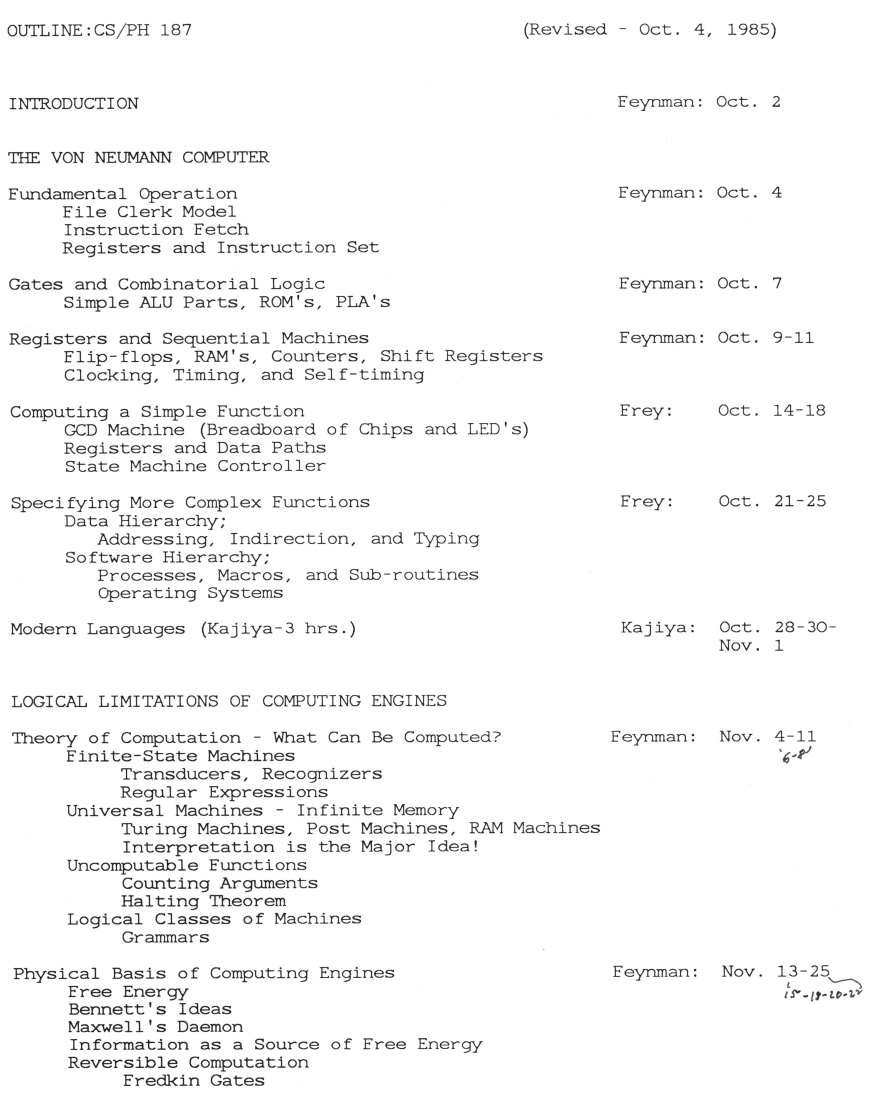


FIGURE 0.1 (CONTINUED)
Feynman's 1981 draft course outline. Images are reproduced by kind permission of the Caltech archive.

1981 notes for the topics he wanted to cover in the computation course. Over the next few years, he made several versions of such topic lists and he also made week-by-week schedules of which lectures he would give, as well as those that would be given by guest lecturers such as Gerry Sussman and Sandy Frey. See Figure 0.2 for the allocations for the last

version of the course in 1985/1986. Looking at these lecture allocations, I think that the published "Lectures on Computation" give a pretty accurate account of the major topics that Feynman himself lectured on in the course. It is intriguing to note that in his 1985/1986 schedule, Feynman devotes only one lecture to the topic of "Quantum Mechanical Computer".



OUTLINE:CS/PH 187 (Revised - Oct. 4, 1985)

INTRODUCTION — Feynman: Oct. 2

THE VON NEUMANN COMPUTER

Fundamental Operation — Feynman: Oct. 4
- File Clerk Model
- Instruction Fetch
- Registers and Instruction Set

Gates and Combinatorial Logic — Feynman: Oct. 7
- Simple ALU Parts, ROM's, PLA's

Registers and Sequential Machines — Feynman: Oct. 9-11
- Flip-flops, RAM's, Counters, Shift Registers
- Clocking, Timing, and Self-timing

Computing a Simple Function — Frey: Oct. 14-18
- GCD Machine (Breadboard of Chips and LED's)
- Registers and Data Paths
- State Machine Controller

Specifying More Complex Functions — Frey: Oct. 21-25
- Data Hierarchy;
  - Addressing, Indirection, and Typing
- Software Hierarchy;
  - Processes, Macros, and Sub-routines
  - Operating Systems

Modern Languages (Kajiya-3 hrs.) — Kajiya: Oct. 28-30-Nov. 1

LOGICAL LIMITATIONS OF COMPUTING ENGINES

Theory of Computation - What Can Be Computed? — Feynman: Nov. 4-11
- Finite-State Machines
  - Transducers, Recognizers
  - Regular Expressions
- Universal Machines - Infinite Memory
  - Turing Machines, Post Machines, RAM Machines
  - Interpretation is the Major Idea!
- Uncomputable Functions
  - Counting Arguments
  - Halting Theorem
- Logical Classes of Machines
  - Grammars

Physical Basis of Computing Engines — Feynman: Nov. 13-25
- Free Energy
- Bennett's Ideas
- Maxwell's Daemon
- Information as a Source of Free Energy
- Reversible Computation
  - Fredkin Gates

FIGURE 0.2
Feynman's 1985 course outline and lecture allocations. Images are reproduced by kind permission of the Caltech archive.

Billiard-Ball Computer		
Free Energy vs. Speed		
Information and Communications		
More Information and Entropy	Feynman:	Nov. 27
Noise and Communications		
Shannon's Theorem		
Data Compression	Feynman:	Dec. 2
Hamming Codes		
More Error Correcting Codes	Frey:	Dec. 4
Algorithms and Complexity		
Time and Space	Frey:	Dec. 6
Some Examples		
NP-Completeness	Frey:	Dec. 9
Digital Signalling	Frey:	Dec. 11-13
Sampling Theorem		
Spectral Analysis		
PHYSICAL ASPECTS OF COMPUTATION		
Semiconductor Physics	Feynman:	Jan. 8
PN, NPN		
CMOS Inverter	Feynman:	Jan. 10
Timing		
Energetics	Feynman:	Jan. 13
Pass Transistors		
Hot Clocking	Feynman:	Jan. 15
Use of Inductance		
Integrated Circuit Technology (VLSI example)		
Planar Process Fabrication Technology	Feynman:	Jan. 17
Stick figures		
Layout and design rules		
Examples - Registers, PLA,...		
2-Phase Clocks and Synchronous Design	Feynman:	Jan. 20
Practical Limitations	Feynman:	Jan. 22
Wires - Rence's rule		
Cray Design Ideas (other than algorithmic)	Feynman:	Jan. 24
Measured terminated transmission lines		
Getting the heat out		
Power supplies		
Other Fundamental Technologies		
Bipolar	Feynman:	Jan. 27
Gallium Arsinide		
Three-dimensional VLSI	Feynman:	Jan. 29
Superconductors - Josephson Junction		
Optical Computing	Feynman:	Jan. 31 -Feb. 3
Two-dimensional parallel		
Linear		
Non-linear amplifiers		
Memory storage		
Optical fibers		
Electro-optical transducers		
Quantum Mechanical Computer	Feynman:	Feb. 5

FIGURE 0.2 (CONTINUED)
Feynman's 1985 course outline and lecture allocations. Images are reproduced by kind permission of the Caltech archive.

ORGANIZATIONAL ASPECTS OF COMPUTING: Architectures of Computing Engines		
The Von Neumann Architecture	Frey:	Feb. 7
The memory interface bottleneck		-10-12
Cache memory		
RISC machines		
Pipelines and look-ahead		
I/O, interrupts, and instruction sequencing		
Wide horizontal microcode		
Specialized processing units		
SCHEME chip		
Floating Point Processing Element		
Concurrent Processor Architectures		
Microscopic parallelism		
Local communication only		
Systolic arrays of Kung and Leiserson	Douglas:	Feb. 14
Cellular automata	Feynman:	Feb. 17
Connection Machine	Feynman:	Feb. 19-21
Memory		
Processors		
Routing communications		
Hopfield type machines	Feynman:	Feb. 24
Process concurrency		
Special purpose hardware	Douglas:	Feb. 26
Orrery example		
QCD Machines		
Cosmic cube	Frey:	Feb. 28
Fine grain		-Mar. 3
Large grain		
Ultra and the IBM RP3	Frey:	Mar. 5
Application/process decomposition	Frey:	Mar. 7
Macroscopic parallelism	Frey:	Mar. 10
Networks of large computers		
Communication problems and limits	Frey:	Mar. 12-14
Global communication schemes		
Batcher sorting nets		
Shuffles		
Hypercubes		
Assynchronous arbiters		
Networks		
General properties of array machines		
Error detection and correction		
MORE DIFFICULT COMPUTER APPLICATIONS		
Computer Intelligence		
Example: Chess Playing Machines		
Robotics		
Closed-loop, Real-time Control Systems		

FIGURE 0.2 (CONTINUED)
Feynman's 1985 course outline and lecture allocations. Images are reproduced by kind permission of the Caltech archive.

```
Vision
     Creating Scenes
     Recognition
     Stereopsis
     Examples:
          Fingerprint Identification
          Grabbing Errant Satellites for Repair
          Selecting Parts from Bins

Speech
     Creation
     Recognition

Natural Language Understanding
     Syntax
     Semantics
     Translation

Searching Algorithms
     Monte Carlo
     Spin-Glass Annealing
     Alpha-Beta (Example, Chess playing machines)

Pattern Directed Search
     Representation of Knowledge
          Semantic networks
          Frames
          Mathematical logic as a representational system
               Planner, Prolog
          Analogical Representation
               Example, Blocks world
          Heuristic Search Rules
          Examples; Eurisco, Wilkins chess player

Useful Applications
     Sumbolic Manipulation (MACSYMA, SMP)
     Computer Aided Design and Manufacture (CAD-CAM)
     Expert Systems
          Example, MYCIN, why not so useful
     Chip and Board Layout Machines
     Spelling Correctors, Indexers, Translators

Summary and Conclusions
     What the big problems are today
     How we might go about solving them
```

FIGURE 0.2 (CONTINUED)
Feynman's 1985 course outline and lecture allocations. Images are reproduced by kind permission of the Caltech archive.

The new chapter by John Preskill fills in the gaps between Feynman's proposal in 1981, his lecture course in the early 1980s, and where we are now.

Feynman had worked as a consultant for Thinking Machines Corporation in the summer of 1983. Danny Hillis's ambitious goal for the company was to turn his PhD thesis ideas on massively parallel

computer architectures into a successful commercial product called the "Connection Machine". The early 1980s was a time of great experimentation in parallel computer architectures and Figure 0.2 shows a section headed "Concurrent Processor Architectures". Besides including lectures from Feynman on the Connection Machine, there were lectures on Caltech's own Cosmic Cube distributed memory parallel computer as well as on IBM's RP3 project, an experimental shared memory parallel computer. Nearly 40 years later, it is clear that Caltech's David beat IBM's Goliath in the sense that the Cosmic Cube's style of distributed memory, message passing, parallel architecture now forms the basis for almost all present-day supercomputers. There seems little point in including details of the many bold, but ultimately commercially unsuccessful, explorations of parallel computer architecture. Instead, the new chapter by John Shalf gives us an overview of possible post-Moore's law computer architectures and technologies. Figure 0.3 shows Jack Dongarra's famous collection of failed parallel computer company mugs.

In his original preface to the lectures, Feynman talks briefly about the field of "artificial intelligence" or AI – which he preferred to call "advanced applications". Figure 0.2 shows a list of "More difficult computer applications" – ranging from computer vision and robotics to speech and natural language understanding. In the original preparation of these lectures, I benefited greatly from previous work on some of the course material done by one of his teaching assistants for the course, Eric Mjolsness. Eric is now a Professor in Computer Science and Mathematics at the University of California in Irvine and his research interests are

FIGURE 0.3
Jack Dongarra's collection of failed computer company mugs. Reproduced here by kind permission of Jack Dongarra.

neural networks, machine learning, and symbolic AI, with applications to multi-scale scientific problems. Eric was a PhD student researching neural networks with John Hopfield at the time of the lectures and was an observer of debates between Feynman, advocating a neural network approach to AI, and Gerry Sussman, defending the more established practice of symbolic AI.

I suggested that the two major changes in the last 25 years were in quantum computing and the approaching end of Moore's law. In fact, I think there is a third major change that should be included. This is what Terry Sejnowski, one of the early pioneers of neural networks, calls "The Deep Learning Revolution" [1]. This revolution was triggered by Fei-Fei Li's ImageNet computer vision competition, and by AlexNet, the winning "deep neural network" entry from Geoffrey Hinton's team from Toronto, in 2012. I am very pleased that Eric agreed to write a chapter about "Feynman on Artificial Intelligence and Machine Learning" in which he would include some of his first-hand experiences and discussions with Feynman on his lectures.

Finally, I have included two new short reminiscences of life with Feynman at Caltech. One is from John Preskill with some of his memories of Feynman as a colleague at Caltech. Curiously, in his article for this new edition of the lectures, John reveals his regret that he never took the opportunity to discuss quantum computation with Feynman. The second is from Michael Douglas, who worked with Gerry Sussman on his Digital Orrery project and was a student on Feynman's original course on the "Potentialities and Limitations of Computing Machines". Michael was a teaching assistant on later versions of the course and is now a distinguished practitioner of string theory in the Simons Center for Geometry and Physics at Stony Brook University.

Acknowledgements

I would like to include belated acknowledgements to two individuals who helped correct a number of errors and misprints in the 1996 edition of the Lectures. The first is to Professor Yasuo Hara, then at Heisei Teikyo University in Japan, who translated Feynman's Lectures for a Japanese language edition. The second is to Joel Chavas, then a neurobiology student at the Max Planck Institute for Biophysical Chemistry in Goettingen, who corrected the UTM specification in Chapter 3.

Dedication

As a conclusion to the Editor's Preface for this new edition of Feynman's *Lectures on Computation*, I would like to dedicate this edition to the memory of Helen Tuck. Helen started as the secretary for the two Caltech Nobel Prize winners, Richard Feynman and Murray Gell-Mann, in 1981. She took over this interesting but intimidating job after the retirement of their previous secretary, the legendary Julie Curcio. I was on sabbatical leave at Caltech in 1981 and was fortunate to get to know Helen quite well as she settled into her new role. Her role in the computing book project was that she knew that Feynman really wanted his lectures written up and published. However, several people had tried to do this and then given up on the task. She recommended me as a possible editor in November 1987. When I visited Caltech for the Hypercube computing conference in January 1988, I met with Feynman and we agreed that I would edit his lecture notes for publication. I knew Feynman was ill with cancer but I was still shocked when he died so soon after our agreement. It was Helen who made sure that I received all the relevant material for Feynman's computing course – mainly the course notes and audio tapes. Although it took me a long time to sort out and edit all this into a form suitable for publication, I was determined to justify Helen's belief that I would finish the task. That Feynman's lectures on computing finally appeared in book form is certainly largely due to Helen's loyalty and persistence.

Tony Hey
Southampton

Reference

1. Terrence J. Sejnowski, *The Deep Learning Revolution*, The MIT Press, Cambridge, MA and London, 2018.

Feynman's Preface

When I produced the *Lectures on Physics*, some 30 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.* 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' intelligence and mind. Of course, we might get useful ideas from studying how the brain works, but we must

* 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. This led to his formulation of 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. [Editor]

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.*

* A companion volume to these lectures called "Feynman and Computation" was published in 1999. This contains articles from many of the same experts who contributed to Feynman's course as well reprints of Feynman's classic articles "There's Plenty of Room at the Bottom" and "Simulating Physics with Computers". [Editor]

Author and Editor Biographies

The late **Richard P. Feynman** was Richard Chace Tolman Professor of Theoretical Physics at the California Institute of Technology. He was awarded the Nobel Prize in 1965 for his work on the development of quantum electrodynamics and made many other fundamental contributions to physics. What is less well-known is his contribution to computer science with his ideas about quantum computing. He was one of the most famous and beloved figures of the 20th century, both in physics and in the public arena.

Tony Hey is Chief Data Scientist at the UK's Rutherford Appleton Laboratory at Harwell. After an academic career including Dean of Engineering at the University of Southampton in the UK, he became Director of the UK's pioneering eScience initiative. After ten years as a Vice President in Microsoft Research in Redmond in the United States, he returned to the UK and now leads a group applying Deep Learning neural networks to the analysis of experimental scientific data. He is also co-author of *The Computing Universe: A Journey through a Revolution*, a popular introduction to the development of computer science.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Contributors

John Preskill
Richard P. Feynman Professor of Theoretical Physics
California Institute of Technology
Pasadena, California, USA

John Shalf
Senior Scientist
Lawrence Berkeley National Laboratory
Berkeley, California, USA

Eric Mjolsness
Professor of Computer Science and Mathematics
University of California, Irvine
Irvine, California, USA

Michael Douglas
Professor in the Simons Center for Geometry and Physics
Stony Brook University
Stony Brook, New York, USA

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

1

Introduction to Computers

CONTENTS

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 bongos. 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 about 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: ls and 0s.

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

DOI: 10.1201/9781003358817-1

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 Great 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 whatnot. 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 that have not evolved a word for "automobile", and speakers of such tongues would have to invent some, possibly long and complex, descriptions 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*** that 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 13 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 ls and 0s on level one are transformed, by the time we get to 13, into commands to land aircraft in a simulation or check whether a 40-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.

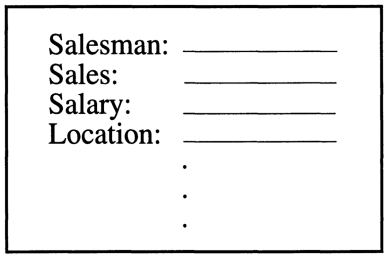
Also, our discussion will be restricted to computers with the so-called "Von Neumann architecture". Don't be put off by the word "architecture"; it's just a big word for how we arrange things, only we're arranging electronic components rather than bricks and columns. Von Neumann was a famous mathematician 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. 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.

1.1 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.



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 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.* 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

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

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

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:

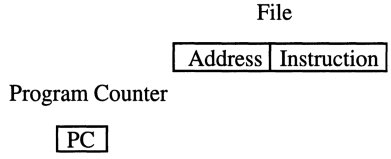
0000 0000 0000 0000 0000 1111 0000 0000 0000 0000...

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 ls, 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. First, there is a core "instruction set" of simple procedures that comes with the pad – add, transfer, and so on. 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 that 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 for what he is to do next. So he gets the instruction and stores it on his pad in an area that 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:*

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

* 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. [Editor]

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.

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*:

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

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

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

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

$$\text{Transfer (B) to M or M} \leftarrow \text{(B)}$$

$$\text{Transfer (M) to B or B} \leftarrow \text{(M)}$$

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

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

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:

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

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:

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

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:

$$\text{Shiftleft A and Shiftright A}$$

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 Chapter 2, 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:

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

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).

TABLE 1.1

The Truth Table for the AND Operator

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

$X = A \wedge B$

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 addition of 1 and 1 is 10, which is zero if you forget the carry. We can introduce the relevant logical symbols:

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

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

The actions of OR and XOR can also be summarized with truth tables (Table 1.2).

TABLE 1.2

The Truth Tables for the OR and XOR Operators

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

OR

$X = A \vee B$

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

XOR

$X = A \oplus B$

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

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

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

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:

$$\text{Direct Load: B} \leftarrow \text{N},$$

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 touched on above:

Jump to (Z) or PC ← (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 jumps 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 ← (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 100%, 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 n^{th} 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, or 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.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com